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.

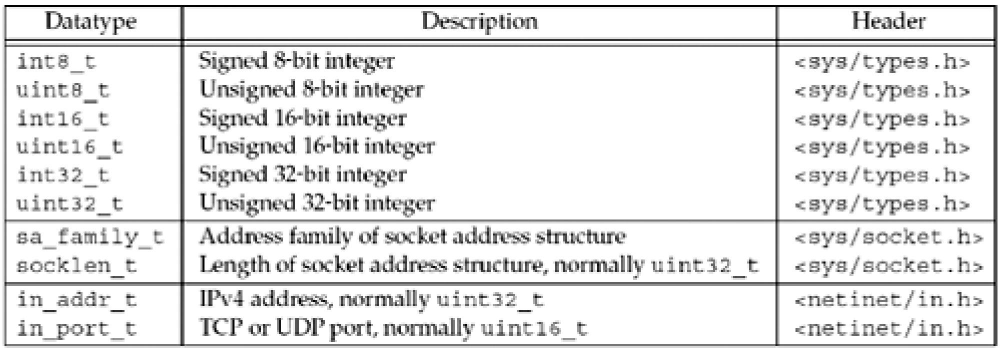
Figure 3.1 The Internet (IPv4) socket address structure: sockaddr_in.

```
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 high-order 12 bits are reserved

  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:

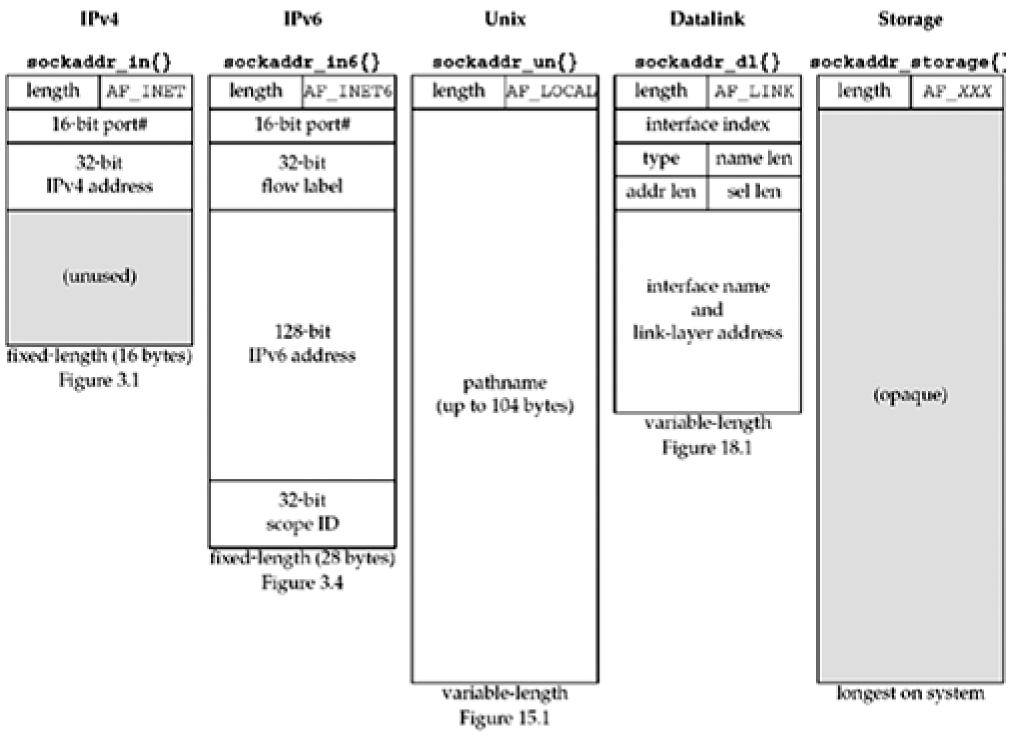
a. If any socket address structures that the system supports have alignment requirements, the `sockaddr_storage` provides the strictest alignment requirement.
b. The `sockaddr_storage` is large enough to contain any socket address structure that the system supports.

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

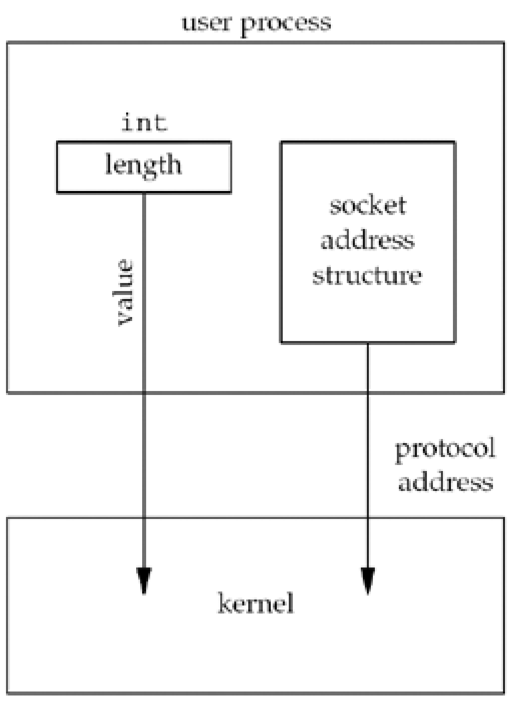
We mentioned that when a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure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



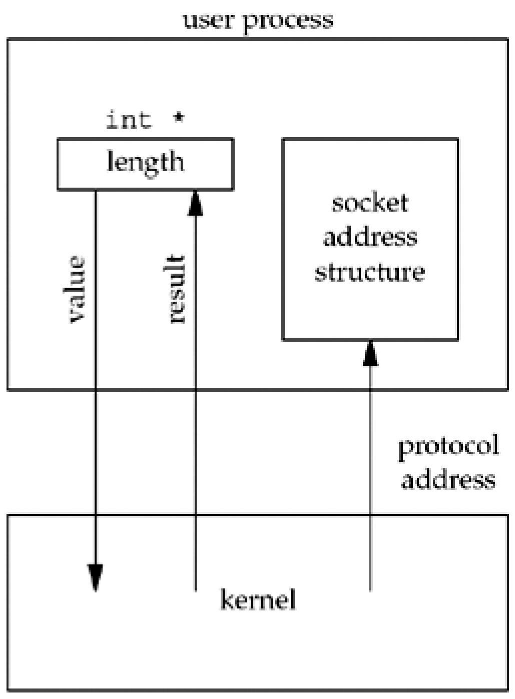
We will see in the next chapter that the datatype for the size of a socket address structure is actually `socklen_t` and not `int`, but the POSIX specification recommends that `socklen_t` be defined as `uint32_t`.

10. 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).

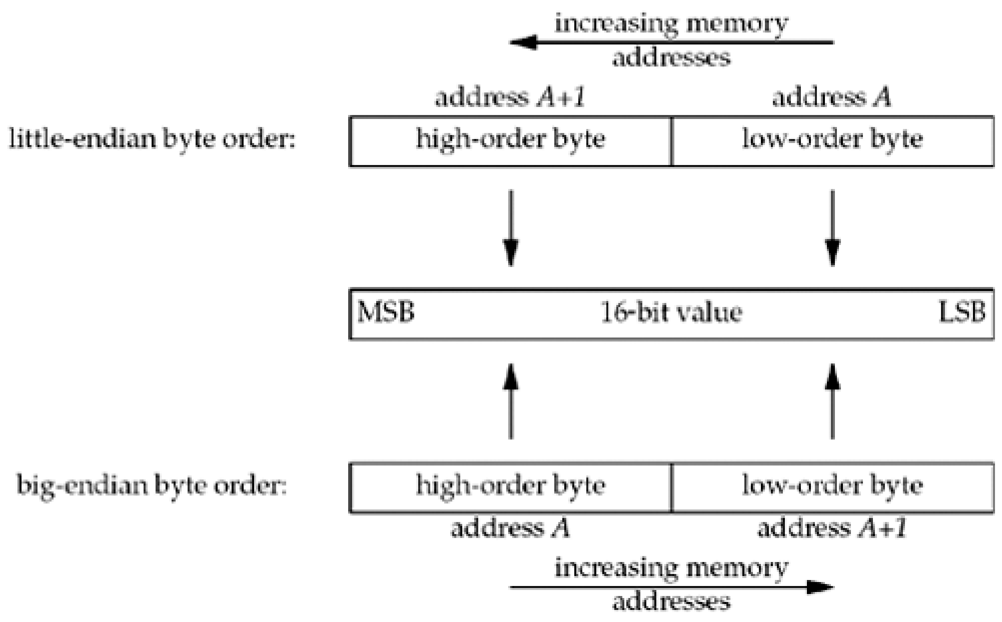
With network programming, the most common example of a value-result argument is the length of a returned socket address structure. But, we will encounter other value-result arguments in this text:

- 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.

3.7 'inet_pton' and 'inet_ntop' Functions

These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. We use these two functions throughout the text. The letters "`p`" and "`n`" stand for *presentation* and *numeric*. The presentation format for an address is often an ASCII string and the numeric format is the binary value that goes into a socket address structure.

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

int inet_pton(int family, const char *strptr, void *addrptr);
```

Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

```
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
```

Returns: pointer to result if OK, `NULL` on error

The *family* argument for both functions is either `AF_INET` or `AF_INET6`. If *family* is not supported, both functions return an error with `errno` set to `EAFNOSUPPORT`.

The first function tries to convert the string pointed to by *strptr*, storing the binary result through the pointer *addrptr*. If successful, the return value is 1. If the input string is not a valid presentation format for the specified *family*, 0 is returned.

`inet_ntop` does the reverse conversion, from numeric (*addrptr*) to presentation (*strptr*). The *len* argument is the size of the destination, to prevent the function from overflowing the caller's buffer. To help specify this size, the following two definitions are defined by including the `<netinet/in.h>` header:

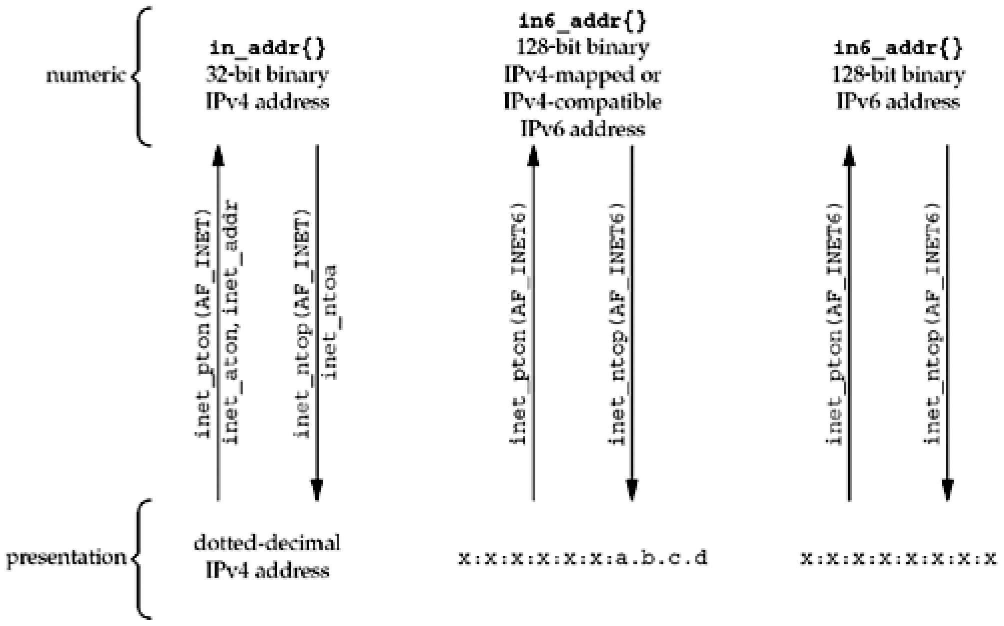
```
#define INET_ADDRSTRLEN      16      /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN     46      /* for IPv6 hex string */
```

If *len* is too small to hold the resulting presentation format, including the terminating null, a null pointer is returned and `errno` is set to `ENOSPC`.

The *strptr* argument to `inet_ntop` cannot be a null pointer. The caller must allocate memory for the destination and specify its size. On success, this pointer is the return value of the function.

Figure 3.11 summarizes the five functions that we have described in this section and the previous section.

Figure 3.11. Summary of address conversion functions.



Example

Even if your system does not yet include support for IPv6, you can start using these newer functions by replacing calls of the form

```
foo.sin_addr.s_addr = inet_addr(cp);
```

with

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

and replacing calls of the form

```
ptr = inet_ntoa(foo.sin_addr);
```

with

```
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

Figure 3.12 shows a simple definition of `inet_pton` that supports only IPv4. Similarly, Figure 3.13 shows a simple version of `inet_ntop` that supports only IPv4.

Figure 3.12 Simple version of inet_pton that supports only IPv4.

libfree/inet_pton_ipv4.c

```
int
inet_pton(int family, const char *strptr, void *addrptr)
{
    if (family == AF_INET) {
        struct in_addr in_val;

        if (inet_aton(strptr, &in_val)) {
            memcpy(addrptr, &in_val, sizeof(struct in_addr));
            return (1);
        }
        return (0);
    }
    errno = EAFNOSUPPORT;
    return (-1);
}
```

Figure 3.13 Simple version of inet_ntop that supports only IPv4.

libfree/inet_ntop_ipv4.c

```
const char *
inet_ntop(int family, const void *addrptr, char *strptr, size_t len)
{
    const u_char *p = (const u_char *) addrptr;

    if (family == AF_INET) {
        char    temp[INET_ADDRSTRLEN];

        snprintf(temp, sizeof(temp), "%d.%d.%d.%d", p[0], p[1], p[2],
p[3]);
        if (strlen(temp) >= len) {
            errno = ENOSPC;
            return (NULL);
        }
        strcpy(strptr, temp);
        return (strptr);
    }
    errno = EAFNOSUPPORT;
    return (NULL);
}
```

3.8 'sock_ntop' and Related Functions

A basic problem with `inet_ntop` is that it requires the caller to pass a pointer to a binary address. This address is normally contained in a socket address structure, requiring the caller to know the format of the structure and the address family. That is, to use it, we must write code of the form

```
struct sockaddr_in   addr;

inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

for IPv4, or

```
struct sockaddr_in6   addr6;

inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

for IPv6. This makes our code protocol-dependent.

To solve this, we will write our own function named `sock_ntop` that takes a pointer to a socket address structure, looks inside the structure, and calls the appropriate function to return the presentation format of the address.

```
#include "unp.h"

char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);

                              Returns: non-null pointer if OK, NULL on error
```

This is the notation we use for functions of our own (nonstandard system functions) that we use throughout the book: the box around the function prototype and return value is dashed. The header is included at the beginning is usually our `unp.h` header.

sockaddr points to a socket address structure whose length is *addrlen*. The function uses its own static buffer to hold the result and a pointer to this buffer is the return value.

Notice that using static storage for the result prevents the function from being *re-entrant* or *thread-safe*. We will talk more about this in Section 11.18. We made this design decision for this function to allow us to easily call it from the simple examples in the book.

The presentation format is the dotted-decimal form of an IPv4 address or the hex string form of an IPv6 address surrounded by brackets, followed by a terminator (we use a colon, similar to URL syntax), followed by the decimal port number, followed by a null character. Hence, the buffer size must be at least `INET_ADDRSTRLEN` plus 6 bytes for IPv4 (16 + 6 = 22), or `INET6_ADDRSTRLEN` plus 8 bytes for IPv6 (46 + 8 = 54).

We show the source code for only the `AF_INET case` in Figure 3.14.

Figure 3.14 Our sock_ntop function.

lib/sock_ntop.c

```
char *
sock_ntop(const struct sockaddr *sa, socklen_t salen)
{
    char    portstr[8];
    static char str[128];       /* Unix domain is largest */

    switch (sa->sa_family) {
    case AF_INET:{
            struct sockaddr_in *sin = (struct sockaddr_in *) sa;

            if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
                return (NULL);
            if (ntohs(sin->sin_port) != 0) {
                snprintf(portstr, sizeof(portstr), ":%d",
                         ntohs(sin->sin_port));
                strcat(str, portstr);
            }
            return (str);
        }
```

There are a few other functions that we define to operate on socket address structures, and these will simplify the portability of our code between IPv4 and IPv6.

```
#include "unp.h"

int sock_bind_wild(int sockfd, int family);
```

Returns: 0 if OK, -1 on error

```
int sock_cmp_addr(const struct sockaddr *sockaddr1,
          const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Returns: 0 if addresses are of the same family and ports are equal, else nonzero

```
int sock_cmp_port(const struct sockaddr *sockaddr1,
          const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Returns: 0 if addresses are of the same family and ports are equal, else nonzero

```
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);
```

```
#include "unp.h"
```

Returns: non-negative port number for IPv4 or IPv6 address, else -1

`char *sock_ntop_host(const struct sockaddr *`*sockaddr*`, socklen_t `*addrlen*`);`

Returns: non-null pointer if OK, `NULL` on error

`void sock_set_addr(const struct sockaddr *`*sockaddr*`, socklen_t `*addrlen*`, void *`*ptr*`);`

`void sock_set_port(const struct sockaddr *`*sockaddr*`, socklen_t `*addrlen*`, int `*port*`);`

`void sock_set_wild(struct sockaddr *`*sockaddr*`, socklen_t `*addrlen*`);`

`sock_bind_wild` binds the wildcard address and an ephemeral port to a socket. `sock_cmp_addr` compares the address portion of two socket address structures, and `sock_cmp_port` compares the port number of two socket address structures. `sock_get_port` returns just the port number, and `sock_ntop_host` converts just the host portion of a socket address structure to presentation format (not the port number). `sock_set_addr` sets just the address portion of a socket address structure to the value pointed to by *ptr*, and `sock_set_port` sets just the port number of a socket address structure. `sock_set_wild` sets the address portion of a socket address structure to the wildcard. As with all the functions in the text, we provide a wrapper function whose name begins with "`S`" for all of these functions that return values other than `void` and normally call the wrapper function from our programs. We do not show the source code for all these functions, but it is freely available (see the Preface).

3.9 'readn', 'writen', and 'readline' Functions

Stream sockets (e.g., TCP sockets) exhibit a behavior with the `read` and `write` functions that differs from normal file I/O. A `read` or `write` on a stream socket might input or output fewer bytes than requested, but this is not an error condition. The reason is that buffer limits might be reached for the socket in the kernel. All that is required to input or output the remaining bytes is for the caller to invoke the `read` or `write` function again. Some versions of Unix also exhibit this behavior when writing more than 4,096 bytes to a pipe. This scenario is always a possibility on a stream socket with `read`, but is normally seen with `write` only if the socket is nonblocking. Nevertheless, we always call our `writen` function instead of `write`, in case the implementation returns a short count.

We provide the following three functions that we use whenever we read from or write to a stream socket:

```
#include "unp.h"

ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t writen(int filedes, const void *buff, size_t nbytes);

ssize_t readline(int filedes, void *buff, size_t maxlen);
```

All return: number of bytes read or written, ☐1 on error

Figure 3.15 shows the `readn` function, Figure 3.16 shows the `writen` function, and Figure 3.17 shows the `readline` function.

Figure 3.15 readn function: Read *n* bytes from a descriptor.

lib/readn.c

```
#include    "unp.h"

ssize_t                      /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char   *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;      /* and call read() again */
            else
                return (-1);
        } else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

Figure 3.16 writen function: Write *n* bytes to a descriptor.

lib/writen.c

```
#include    "unp.h"

ssize_t                         /* Write "n" bytes to a descriptor. */
writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;   /* and call write() again */
            else
                return (-1);    /* error */
         }

         nleft -= nwritten;
         ptr += nwritten;
    }
    return (n);
}
```

Figure 3.17 readline function: Read a text line from a descriptor, one byte at a time.

test/readline1.c

```
#include     "unp.h"

/* PAINFULLY SLOW VERSION -- example only */
ssize_t
readline(int fd, void *vptr, size_t maxlen)
{
    ssize_t n, rc;
    char    c, *ptr;

    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
     again:
       if ( (rc = read(fd, &c, 1)) == 1) {
           *ptr++ = c;
           if (c == '\n')
               break;        /* newline is stored, like fgets() */
       } else if (rc == 0) {
           *ptr = 0;
           return (n - 1);     /* EOF, n - 1 bytes were read */
       } else {
           if (errno == EINTR)
               goto again;
           return (-1);       /* error, errno set by read() */
       }
   }

   *ptr = 0;                  /* null terminate like fgets() */
   return (n);
}
```

Our three functions look for the error `EINTR` (the system call was interrupted by a caught signal, which we will discuss in more detail in Section 5.9) and continue reading or writing if the error occurs. We handle the error here, instead of forcing the caller to call `readn` or `writen` again, since the purpose of these three functions is to prevent the caller from having to handle a short count.

In Section 14.3, we will mention that the `MSG_WAITALL` flag can be used with the `recv` function to replace the need for a separate `readn` function.

Note that our `readline` function calls the system's `read` function once for every byte of data. This is very inefficient, and why we've commented the code to state it is "PAINFULLY SLOW." When faced with the desire to read lines from a socket, it is quite tempting to turn to the standard I/O library (referred to as "stdio"). We will discuss this approach at length in Section 14.8, but it can be a dangerous path. The same stdio buffering that solves this performance problem creates numerous logistical problems that can lead to well-hidden bugs in your application. The reason is that the state of the stdio buffers is not exposed. To explain this further, consider a line-based protocol between a client and a server, where several clients and servers using that protocol may be implemented over time (really quite common; for example, there are many Web browsers and Web servers independently written to the HTTP specification). Good "defensive programming" techniques require these programs to not only expect their counterparts to follow the network protocol, but to check for unexpected network traffic as well. Such protocol violations should be reported as errors so that bugs are noticed and fixed (and malicious attempts are detected as well), and also so that network applications can recover from problem traffic and continue working if possible. Using stdio to buffer data for performance flies in the face of these goals since the application has no way to tell if unexpected data is being held in the stdio buffers at any given time.

There are many line-based network protocols such as SMTP, HTTP, the FTP control connection protocol, and finger. So, the desire to operate on lines comes up again and again. But our advice is to think in terms of buffers and not lines. Write your code to read buffers of data, and if a line is expected, check the buffer to see if it contains that line.

Figure 3.18 shows a faster version of the `readline` function, which uses its own buffering rather than stdio buffering. Most importantly, the state of `readline`'s internal buffer is exposed, so callers have visibility into exactly what has been received. Even with this feature, `readline` can be problematic, as we'll see in Section 6.3. System functions like `select` still won't know about `readline`'s internal buffer, so a carelessly written program could easily find itself waiting in `select` for data already received and stored in `readline`'s buffers. For that matter, mixing `readn` and `readline` calls will not work as expected unless `readn` is modified to check the internal buffer as well.

Figure 3.18 Better version of readline function.

lib/readline.c

```
#include    "unp.h"

static int read_cnt;
static char *read_ptr;
static char read_buf[MAXLINE];

static ssize_t
my_read(int fd, char *ptr)
{

    if (read_cnt <= 0) {
      again:
        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR)
                goto again;
            return (-1);
        } else if (read_cnt == 0)
            return (0);
        read_ptr = read_buf;
    }
18     read_cnt--;
19     *ptr = *read_ptr++;
20     return (1);
21 }
22 ssize_t
23 readline(int fd, void *vptr, size_t maxlen)
24 {
25     ssize_t n, rc;
26     char    c, *ptr;

27     ptr = vptr;
28     for (n = 1; n < maxlen; n++) {
29         if ( (rc = my_read(fd, &c)) == 1) {
30             *ptr++ = c;
31             if (c  == '\n')
32                 break;          /* newline is stored, like fgets() */
33         } else if (rc == 0) {
34             *ptr = 0;
35             return (n - 1);     /* EOF, n - 1 bytes were read */
36         } else
37             return (-1);       /* error, errno set by read() */
38     }

39     *ptr  = 0;                  /* null terminate like fgets() */
40     return (n);
41 }

42 ssize_t
43 readlinebuf(void **vptrptr)
44 {
45     if (read_cnt)
46         *vptrptr = read_ptr;
47     return (read_cnt);
48 }
```

2–21 The internal function `my_read` reads up to `MAXLINE` characters at a time and then returns them, one at a time.

29 The only change to the `readline` function itself is to call `my_read` instead of `read`.

42–48 A new function, `readlinebuf`, exposes the internal buffer state so that callers can check and see if more data was received beyond a single line.

Unfortunately, by using `static` variables in `readline.c` to maintain the state information across successive calls, the functions are not *re-entrant* or *thread-safe*.

We will discuss this in Sections 11.18 and 26.5. We will develop a thread-safe version using thread-specific data in Figure 26.11.

3.10 Summary

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to various socket functions. Sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents. We always pass these structures by reference (that is, we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument. When a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function. We call these value-result arguments.

Socket address structures are self-defining because they always begin with a field (the "family") that identifies the address family contained in the structure. Newer implementations that support variable-length socket address structures also contain a length field at the beginning, which contains the length of the entire structure.

The two functions that convert IP addresses between presentation format (what we write, such as ASCII characters) and numeric format (what goes into a socket address structure) are `inet_pton` and `inet_ntop`. Although we will use these two functions in the coming chapters, they are protocol-dependent. A better technique is to manipulate socket address structures as opaque objects, knowing just the pointer to the structure and its size. We used this method to develop a set of `sock_` functions that helped to make our programs protocol-independent. We will complete the development of our protocol-independent tools in Chapter 11 with the `getaddrinfo` and `getnameinfo` functions.

TCP sockets provide a byte stream to an application: There are no record markers. The return value from a `read` can be less than what we asked for, but this does not indicate an error. To help read and write a byte stream, we developed three functions, `readn`, `writen`, and `readline`, which we will use throughout the text. However, network programs should be written to act on buffers rather than lines.

Exercises

3.1 Why must value-result arguments such as the length of a socket address structure be passed by reference?

3.2 Why do both the `readn` and `writen` functions copy the `void*` pointer into a `char*` pointer?

3.3 The `inet_aton` and `inet_addr` functions have traditionally been liberal in what they accept as a dotted-decimal IPv4 address string: allowing from one to four numbers separated by decimal points, and allowing a leading `0x` to specify a hexadecimal number, or a leading 0 to specify an octal number. (Try `telnet 0xe` to see this behavior.) `inet_pton` is much stricter with IPv4 address and requires exactly four numbers separated by three decimal points, with each number being a decimal number between 0 and 255. `inet_pton` does not allow a dotted-decimal number to be specified when the address family is `AF_INET6`, although one could argue that these should be allowed and the return value should be the IPv4-mapped IPv6 address for the dotted-decimal string (Figure A.10).

Write a new function named `inet_pton_loose` that handles these scenarios: If the address family is `AF_INET` and `inet_pton` returns 0, call `inet_aton` and see if it succeeds. Similarly, if the address family is `AF_INET6` and `inet_pton` returns 0, call `inet_aton` and if it succeeds, return the IPv4-mapped IPv6 address.

Chapter 4. Elementary TCP Sockets

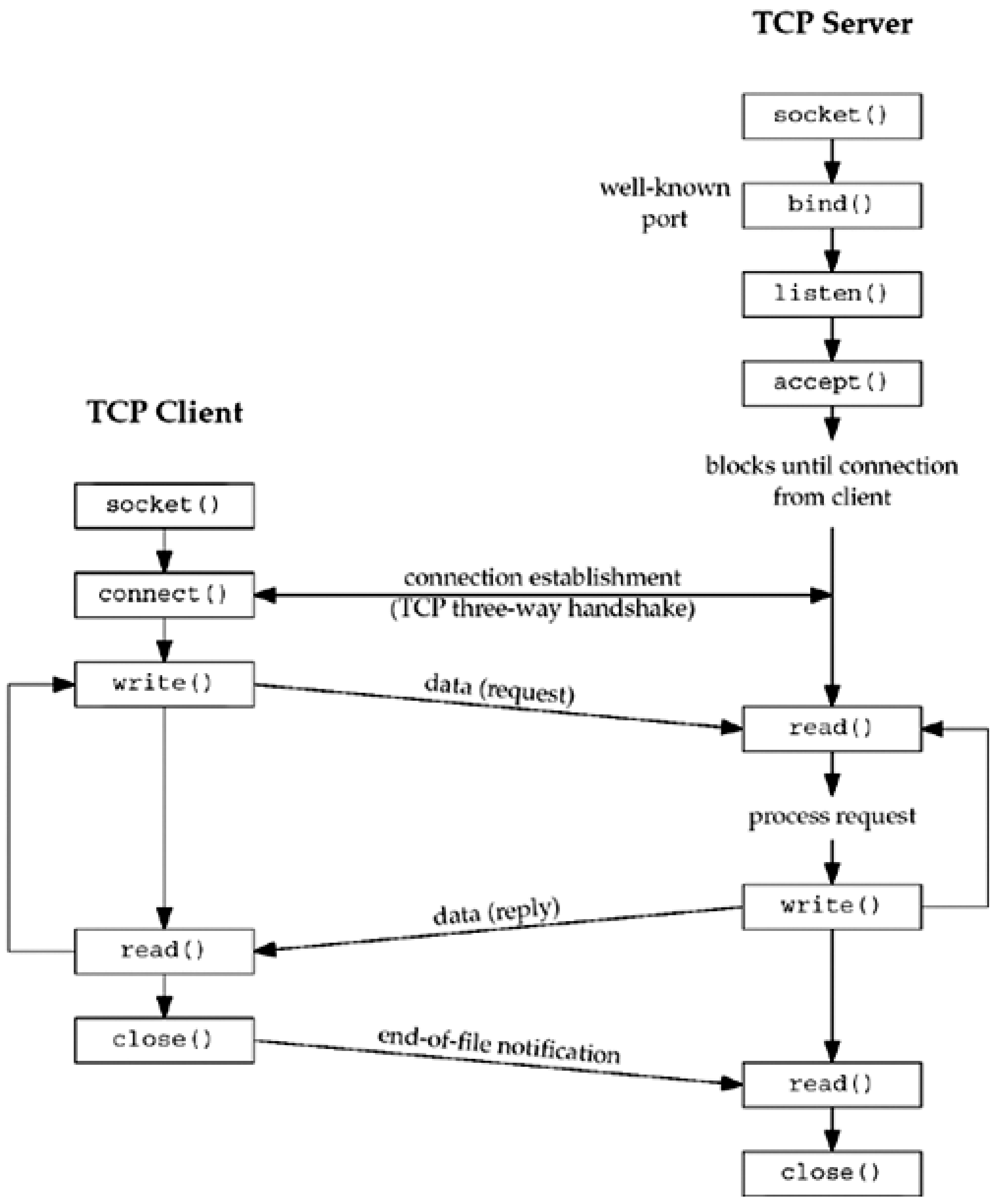
4.1 Introduction

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

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

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

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.

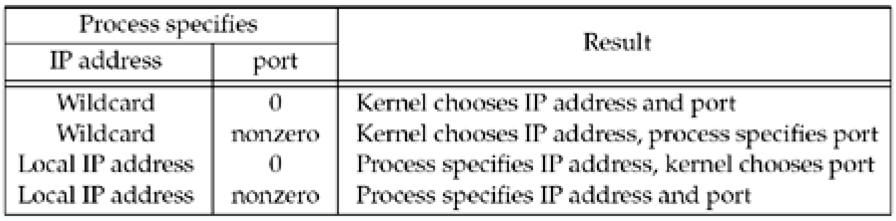
- A process can `bind` a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. For a TCP server, this restricts the socket to receive incoming client connections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

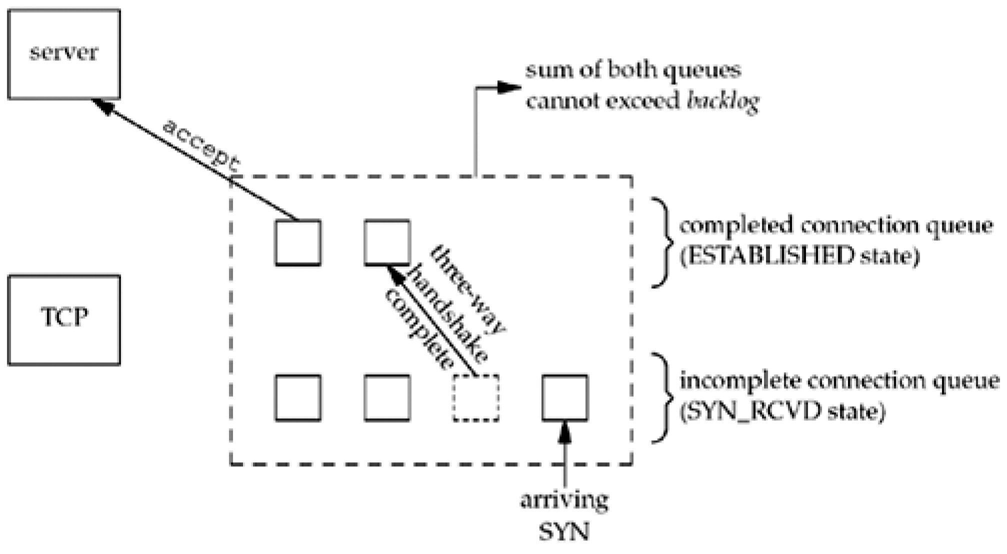
This function is normally called after both the `socket` and `bind` functions and must be called before calling the `accept` function.

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).

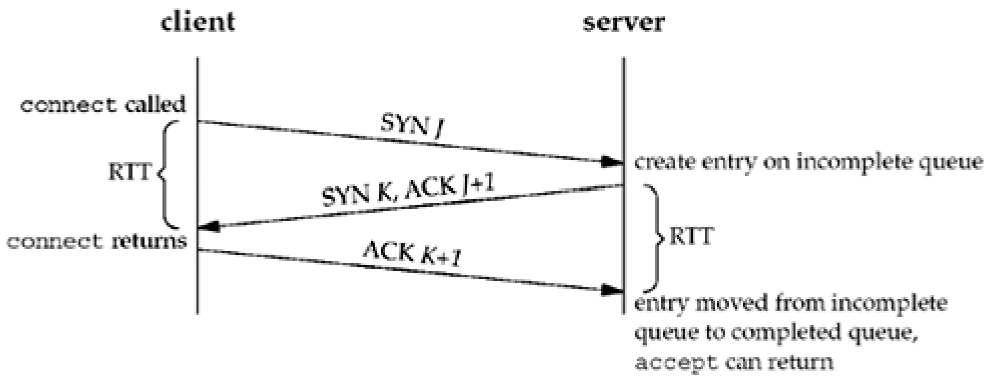
Figure 4.7 depicts these two queues for a given listening socket.

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



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");
}
```

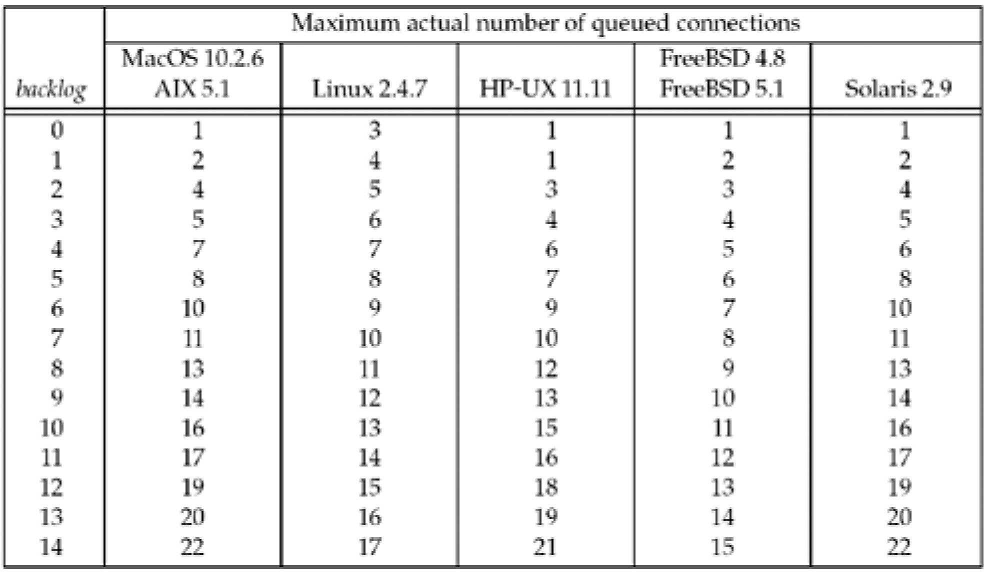
- Manuals and books have historically said that the reason for queuing a fixed number of connections is to handle the case of the server process being busy between successive calls to `accept`. This implies that of the two queues, the completed queue should normally have more entries than the incomplete 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"
#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));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(13);  /* daytime server */

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

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        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
```

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.

The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the six `exec` functions. (We will often refer generically to "the `exec` function" when it does not matter which of the six is called.) `exec` replaces the current process image with the new program file, and this new program normally starts at the `main` function. The process ID does not change. We refer to the process that calls `exec` as the *calling process* and the newly executed program as the *new program*.

Older manuals and books incorrectly refer to the new program as the *new process*, which is wrong, because a new process is not created.

The differences in the six `exec` functions are: (a) whether the program file to execute is specified by a *filename* or a *pathname*; (b) whether the arguments to the new program are listed one by one or referenced through an array of pointers; and (c) whether the environment of the calling process is passed to the new program or whether a new environment is specified.

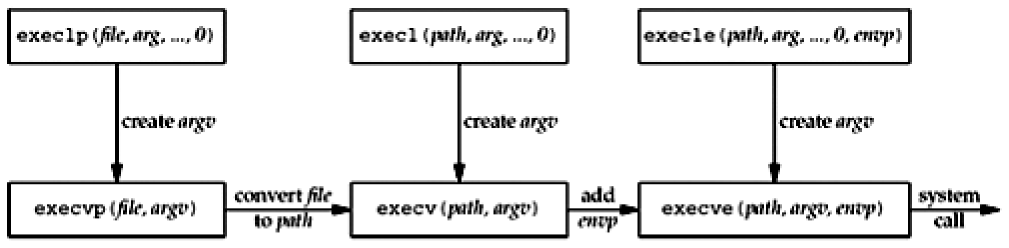
```
#include <unistd.h>
```

```
#include <unistd.h>

int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv (const char *pathname, char *const argv[]);

int execle (const char *pathname, const char *arg0, ...
                 /* (char *) 0, char *const envp[] */ );

int execve (const char *pathname, char *const argv[], char *const envp[]);

int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp (const char *filename, char *const argv[]);

                              All six return: -1 on error, no return on success
```

These functions return to the caller only if an error occurs. Otherwise, control passes to the start of the new program, normally the `main` function.

The relationship among these six functions is shown in Figure 4.12. Normally, only `execve` is a system call within the kernel and the other five are library functions that call `execve`.

Figure 4.12. Relationship among the six exec functions.



Note the following differences among these six functions:

1. The three functions in the top row specify each argument string as a separate argument to the `exec` function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an *argv* array, containing pointers to the argument strings. This *argv* array must contain a null pointer to specify its end, since a count is not specified.
2. The two functions in the left column specify a *filename* argument. This is converted into a *pathname* using the current `PATH` environment variable. If the *filename* argument to `execlp` or `execvp` contains a slash (/) anywhere in the string, the `PATH` variable is not used. The four functions in the right two columns specify a fully qualified *pathname* argument.

3. The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable `environ` is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The *envp* array of pointers must be terminated by a null pointer.

Descriptors open in the process before calling `exec` normally remain open across the `exec`. We use the qualifier "normally" because this can be disabled using `fcntl` to set the `FD_CLOEXEC` descriptor flag. The `inetd` server uses this feature, as we will describe in Section 13.5.

4.8 Concurrent Servers

The server in Figure 4.11 is an *iterative server*. For something as simple as a daytime server, this is fine. But when a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. The simplest way to write a *concurrent server* under Unix is to `fork` a child process to handle each client. Figure 4.13 shows the outline for a typical concurrent server.

Figure 4.13 Outline for typical concurrent server.

```
pid_t pid;
int   listenfd,  connfd;

listenfd = Socket( ... );

   /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
   connfd = Accept (listenfd, ... );    /* probably blocks */

   if( (pid = Fork()) == 0) {
     Close(listenfd);    /* child closes listening socket */
     doit(connfd);       /* process the request */
     Close(connfd);      /* done with this client */
     exit(0);            /* child terminates */
   }
```

```
    Close(connfd);          /* parent closes connected socket */
}
```

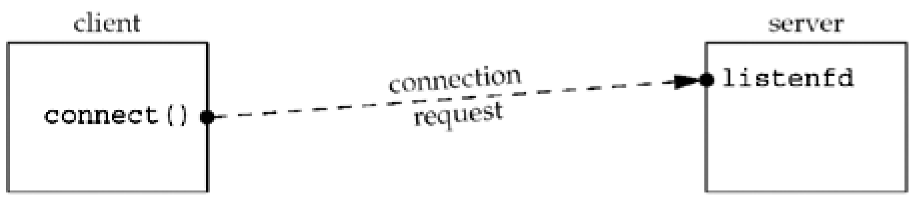
When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket). The parent closes the connected socket since the child handles the new client.

In Figure 4.13, we assume that the function doit does whatever is required to service the client. When this function returns, we explicitly close the connected socket in the child. This is not required since the next statement calls exit, and part of process termination is to close all open descriptors by the kernel. Whether to include this explicit call to close or not is a matter of personal programming taste.

We said in Section 2.6 that calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. Why doesn't the close of connfd in Figure 4.13 by the parent terminate its connection with the client? To understand what's happening, we must understand that every file or socket has a reference count. The reference count is maintained in the file table entry (pp. 57☐60 of APUE). This is a count of the number of descriptors that are currently open that refer to this file or socket. In Figure 4.13, after socket returns, the file table entry associated with listenfd has a reference count of 1. After accept returns, the file table entry associated with connfd has a reference count of 1. But, after fork returns, both descriptors are shared (i.e., duplicated) between the parent and child, so the file table entries associated with both sockets now have a reference count of 2. Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all. The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0. This will occur at some time later when the child closes connfd.

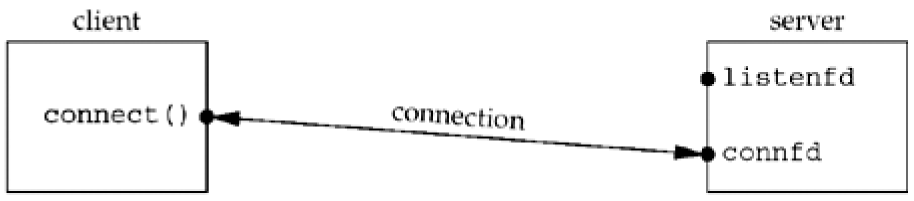
We can also visualize the sockets and connection that occur in Figure 4.13 as follows. First, Figure 4.14 shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

Figure 4.14. Status of client/server before call to accept returns.



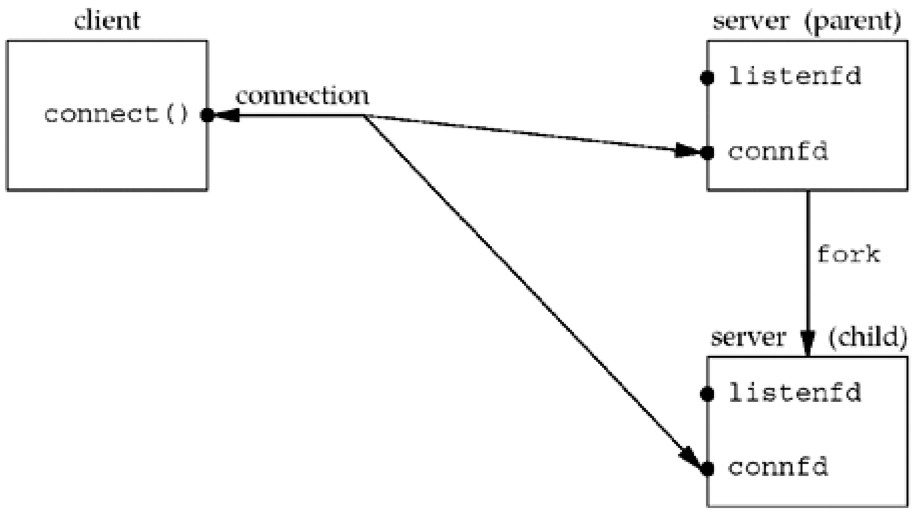
Immediately after accept returns, we have the scenario shown in Figure 4.15. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

Figure 4.15. Status of client/server after return from accept.



The next step in the concurrent server is to call `fork`. Figure 4.16 shows the status after `fork` returns.

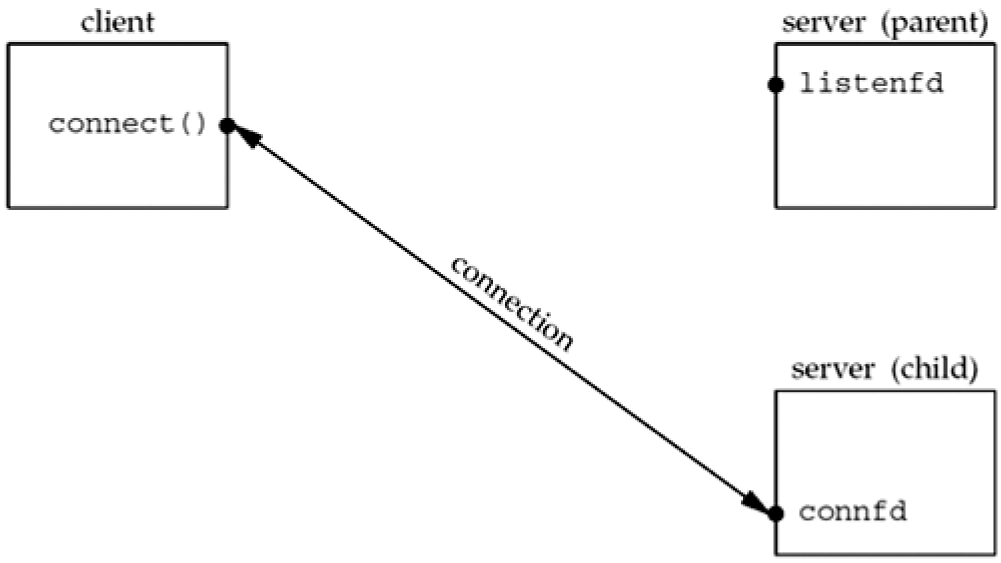
Figure 4.16. Status of client/server after fork returns.



Notice that both descriptors, `listenfd` and `connfd`, are shared (duplicated) between the parent and child.

The next step is for the parent to close the connected socket and the child to close the listening socket. This is shown in Figure 4.17.

Figure 4.17. Status of client/server after parent and child close appropriate sockets.



This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call `accept` again on the listening socket, to handle the next client connection.

4.9 'close' Function

The normal Unix `close` function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>

int close (int sockfd);
```

Returns: 0 if OK, -1 on error

The default action of `close` with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: It cannot be used as an argument to `read` or `write`. But, TCP will try to send any data that is already queued to be sent to the other end, and after this occurs, the normal TCP connection termination sequence takes place (Section 2.6).

In Section 7.5, we will describe the `SO_LINGER` socket option, which lets us change this default action with a TCP socket. In that section, we will also describe what a TCP application must do to be guaranteed that the peer application has received any outstanding data.

Descriptor Reference Counts

At the end of Section 4.8, we mentioned that when the parent process in our concurrent server `closes` the connected socket, this just decrements the reference count for the descriptor. Since the reference count was still greater than 0, this call to `close` did not initiate TCP's four-packet connection termination sequence. This is the behavior we want with our concurrent server with the connected socket that is shared between the parent and child.

If we really want to send a FIN on a TCP connection, the `shutdown` function can be used (Section 6.6) instead of `close`. We will describe the motivation for this in Section 6.5.

We must also be aware of what happens in our concurrent server if the parent does not call `close` for each connected socket returned by `accept`. First, the parent will eventually run out of descriptors, as there is usually a limit to the number of descriptors that any process can have open at any time. But more importantly, none of the client connections will be terminated. When the child closes the connected socket, its reference count will go from 2 to 1 and it will remain at 1 since the parent never `closes` the connected socket. This will prevent TCP's connection termination sequence from occurring, and the connection will remain open.

4.10 'getsockname' and 'getpeername' Functions

These two functions return either the local protocol address associated with a socket (`getsockname`) or the foreign protocol address associated with a socket (`getpeername`).

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

int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);

int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);

                                         Both return: 0 if OK, -1 on error
```

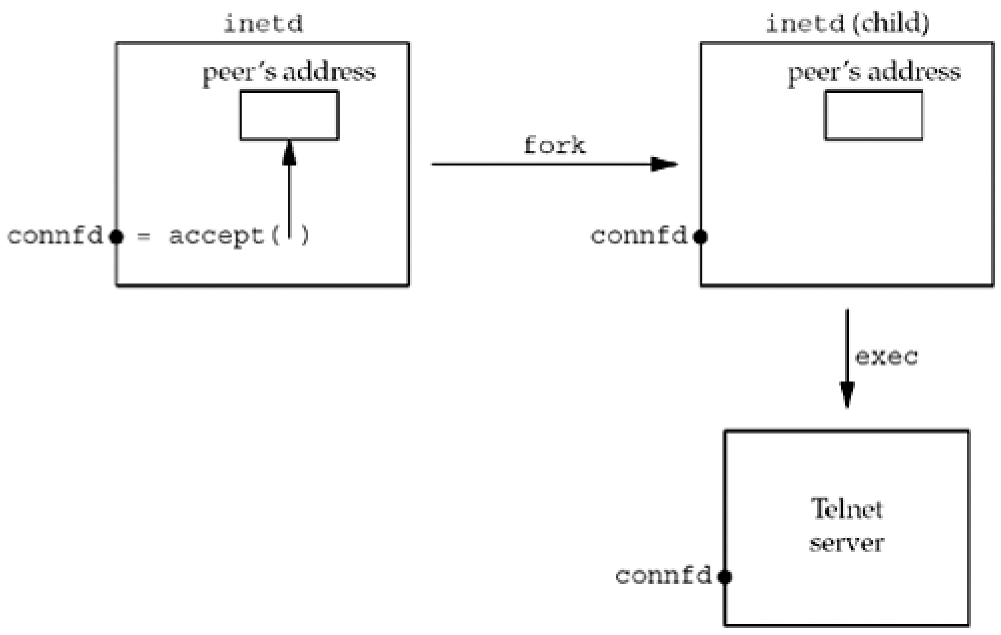
Notice that the final argument for both functions is a value-result argument. That is, both functions fill in the socket address structure pointed to by *localaddr* or *peeraddr*.

We mentioned in our discussion of `bind` that the term "name" is misleading. These two functions return the protocol address associated with one of the two ends of a network connection, which for IPV4 and IPV6 is the combination of an IP address and port number. These functions have nothing to do with domain names (Chapter 11).

These two functions are required for the following reasons:

- After `connect` successfully returns in a TCP client that does not call `bind`, `getsockname` returns the local IP address and local port number assigned to the connection by the kernel.
- After calling `bind` with a port number of 0 (telling the kernel to choose the local port number), `getsockname` returns the local port number that was assigned.
- `getsockname` can be called to obtain the address family of a socket, as we show in Figure 4.19.
- In a TCP server that `bind`s the wildcard IP address (Figure 1.9), once a connection is established with a client (`accept` returns successfully), the server can call `getsockname` to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is `exec`ed by the process that calls `accept`, the only way the server can obtain the identity of the client is to call `getpeername`. This is what happens whenever `inetd` (Section 13.5) `fork`s and `exec`s a TCP server. Figure 4.18 shows this scenario. `inetd` calls `accept` (top left box) and two values are returned: the connected socket descriptor, `connfd`, is the return value of the function, and the small box we label "peer's address" (an Internet socket address structure) contains the IP address and port number of the client. `fork` is called and a child of `inetd` is created. Since the child starts with a copy of the parent's memory image, the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child). But when the child `exec`s the real server (say the Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (i.e., the socket address structure containing the peer's address is lost), and the connected socket descriptor remains open across the `exec`. One of the first function calls performed by the Telnet server is `getpeername` to obtain the IP address and port number of the client.

Figure 4.18. Example of inetd spawning a server.



Obviously the Telnet server in this final example must know the value of `connfd` when it starts. There are two common ways to do this. First, the process calling `exec` can format the descriptor number as a character string and pass it as a command-line argument to the newly `exec`ed program. Alternately, a convention can be established that a certain descriptor is always set to the connected socket before calling `exec`. The latter is what `inetd` does, always setting descriptors 0, 1, and 2 to be the connected socket.

Example: Obtaining the Address Family of a Socket

The `sockfd_to_family` function shown in Figure 4.19 returns the address family of a socket.

Figure 4.19 Return the address family of a socket.

lib/sockfd_to_family.c

```
#include    "unp.h"
int
sockfd_to_family(int sockfd)
{
    struct sockaddr_storage ss;
    socklen_t len;

    len = sizeof(ss);
    if (getsockname(sockfd, (SA *) &ss, &len) < 0)
        return (-1);
10     return (ss.ss_family);
11 }
```

Allocate room for largest socket address structure

5 Since we do not know what type of socket address structure to allocate, we use a `sockaddr_storage` value, since it can hold any socket address structure supported by the system.

Call getsockname

7□10 We call `getsockname` and return the address family.

Since the POSIX specification allows a call to `getsockname` on an unbound socket, this function should work for any open socket descriptor.

4.11 Summary

All clients and servers begin with a call to `socket`, returning a socket descriptor. Clients then call `connect`, while servers call `bind`, `listen`, and `accept`. Sockets are normally closed with the standard `close` function, although we will see another way to do this with the `shutdown` function (Section 6.6), and we will also examine the effect of the `SO_LINGER` socket option (Section 7.5).

Most TCP servers are concurrent, with the server calling `fork` for every client connection that it handles. We will see that most UDP servers are iterative. While these two models have been used successfully for many years, in Chapter 30 we will look at other server design options that use threads and processes.

Exercises

4.1 In Section 4.4, we stated that the `INADDR_` constants defined by the `<netinet/in.h>` header are in host byte order. How can we tell this?

4.2 Modify Figure 1.5 to call `getsockname` after `connect` returns successfully. Print the local IP address and local port assigned to the TCP socket using `sock_ntop`. In what range (Figure 2.10) are your system's ephemeral ports?

4.3 In a concurrent server, assume the child runs first after the call to `fork`. The child then completes the service of the client before the call to `fork` returns to the parent. What happens in the two calls to `close` in Figure 4.13?

4.4 In Figure 4.11, first change the server's port from 13 to 9999 (so that we do not need super-user privileges to start the program). Remove the call to `listen`. What happens?

4.5 Continue the previous exercise. Remove the call to `bind`, but allow the call to `listen`. What happens?

Chapter 5. TCP Client/Server Example

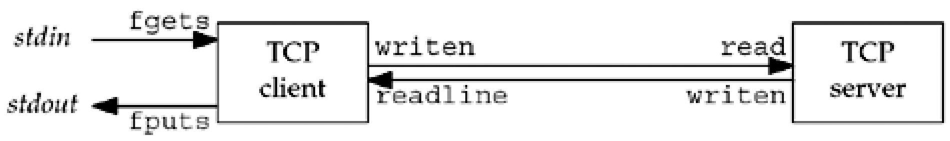
5.1 Introduction

We will now use the elementary functions from the previous chapter to write a complete TCP client/server example. Our simple example is an echo server that performs the following steps:

1. The client reads a line of text from its standard input and writes the line to the server.
2. The server reads the line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.

Figure 5.1 depicts this simple client/server along with the functions used for input and output.

Figure 5.1. Simple echo client and server.



We show two arrows between the client and server, but this is really one full-duplex TCP connection. The `fgets` and `fputs` functions are from the standard I/O library and the `writen` and `readline` functions were shown in Section 3.9.

While we will develop our own implementation of an echo server, most TCP/IP implementations provide such a server, using both TCP and UDP (Section 2.12). We will also use this server with our own client.

A client/server that echoes input lines is a valid, yet simple, example of a network application. All the basic steps required to implement any client/server are illustrated by this example. To expand this example into your own application, all you need to do is change what the server does with the input it receives from its clients.

Besides running our client and server in their normal mode (type in a line and watch it echo), we examine lots of boundary conditions for this example: what happens when the client and server are started; what happens when the client terminates normally; what happens to the client if the server process terminates before the client is done; what happens to the client if the server host crashes; and so on. By looking at all these scenarios and understanding what happens at the network level, and how this appears to the sockets API, we will understand more about what goes on at these levels and how to code our applications to handle these scenarios.

In all these examples, we have "hard-coded" protocol-specific constants such as addresses and ports. There are two reasons for this. First, we must understand exactly what needs to be stored in the protocol-specific address structures. Second, we have not yet covered the library functions that can make this more portable. These functions will be covered in Chapter 11.

We note now that we will make many changes to both the client and server in successive chapters as we learn more about network programming (Figures 1.12 and 1.13).

5.2 TCP Echo Server: 'main' Function

Our TCP client and server follow the flow of functions that we diagrammed in Figure 4.1. We show the concurrent server program in Figure 5.2.

Create socket, bind server's well-known port

9–15 A TCP socket is created. An Internet socket address structure is filled in with the wildcard address (`INADDR_ANY`) and the server's well-known port (`SERV_PORT`, which is defined as 9877 in our `unp.h` header). Binding the wildcard address tells the system that we will accept a connection destined for any local interface, in case the system is multihomed. Our choice of the TCP port number is based on Figure 2.10. It should be greater than 1023 (we do not need a reserved port), greater than 5000 (to avoid conflict with the ephemeral ports allocated by many Berkeley-derived implementations), less than 49152 (to avoid conflict with the "correct" range of ephemeral ports), and it should not conflict with any registered port. The socket is converted into a listening socket by `listen`.

Wait for client connection to complete

17–18 The server blocks in the call to `accept`, waiting for a client connection to complete.

Concurrent server

19–24 For each client, `fork` spawns a child, and the child handles the new client. As we discussed in Section 4.8, the child closes the listening socket and the parent closes the connected socket. The child then calls `str_echo` (Figure 5.3) to handle the client.

Figure 5.2 TCP echo server (improved in Figure 5.12).

tcpdiserv/tcpserv01.c

```
#include     "unp.h"

int
main(int argc, char **argv)
{
   int     listenfd, connfd;
   pid_t   childpid;
   socklen_t clilen;
   struct sockaddr_in cliaddr, servaddr;

   listenfd = Socket (AF_INET, SOCK_STREAM, 0);

   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
   servaddr.sin_port = htons (SERV_PORT);

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

   Listen(listenfd, LISTENQ);

   for ( ; ; ) {
       clilen = sizeof(cliaddr);
       connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

       if ( (childpid = Fork()) == 0) { /* child process */
           Close(listenfd);    /* close listening socket */
           str_echo(connfd);   /* process the request */
           exit (0);
       }
       Close(connfd);         /* parent closes connected socket */
   }
}
```

5.3 TCP Echo Server: 'str_echo' Function

The function `str_echo`, shown in Figure 5.3, performs the server processing for each client: It reads data from the client and echoes it back to the client.

Read a buffer and echo the buffer

8□9 `read` reads data from the socket and the line is echoed back to the client by `writen`. If the client closes the connection (the normal scenario), the receipt of the client's FIN causes the child's `read` to return 0. This causes the `str_echo` function to return, which terminates the child in Figure 5.2

5.4 TCP Echo Client: 'main' Function

Figure 5.4 shows the TCP client `main` function.

Figure 5.3 str_echo function: echoes data on a socket.

lib/str_echo.c

```
 1 #include    "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5     ssize_t n;
 6     char    buf[MAXLINE];

 7   again:
 8     while ( (n = read(sockfd, buf, MAXLINE)) > 0)
 9         Writen(sockfd, buf, n);

10     if (n < 0 && errno == EINTR)
11         goto again;
12     else if (n < 0)
13         err_sys("str_echo: read error");
14 }
```

Figure 5.4 TCP echo client.

tcpcliserv/tcpli01.c

```
#include    "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd;
    struct sockaddr_in servaddr;
```

```
 7     if (argc != 2)
 8         err_quit("usage: tcpcli <IPaddress>");

 9     sockfd = Socket(AF_INET, SOCK_STREAM, 0);

10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_port = htons(SERV_PORT);
13     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

14     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

15     str_cli(stdin, sockfd);     /* do it all */

16     exit(0);
17 }
```

Create socket, fill in Internet socket address structure

9–13 A TCP socket is created and an Internet socket address structure is filled in with the server's IP address and port number. We take the server's IP address from the command-line argument and the server's well-known port (`SERV_PORT`) is from our `unp.h` header.

Connect to server

14–15 `connect` establishes the connection with the server. The function `str_cli` (Figure 5.5) handles the rest of the client processing.

5.5 TCP Echo Client: 'str_cli' Function

This function, shown in Figure 5.5, handles the client processing loop: It reads a line of text from standard input, writes it to the server, reads back the server's echo of the line, and outputs the echoed line to standard output.

Figure 5.5 str_cli function: client processing loop.

lib/str_cli.c

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     char    sendline[MAXLINE], recvline[MAXLINE];

 6     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 7         Writen(sockfd, sendline, strlen (sendline));

 8         if (Readline(sockfd, recvline, MAXLINE) == 0)
 9             err_quit("str_cli: server terminated prematurely");

10         Fputs(recvline, stdout);
11     }
12 }
```

Read a line, write to server

6□7 `fgets` reads a line of text and `writen` sends the line to the server.

Read echoed line from server, write to standard output

8□10 `readline` reads the line echoed back from the server and `fputs` writes it to standard output.

Return to main

11□12 The loop terminates when `fgets` returns a null pointer, which occurs when it encounters either an end-of-file (EOF) or an error. Our `Fgets` wrapper function checks for an error and aborts if one occurs, so `Fgets` returns a null pointer only when an end-of-file is encountered.

5.6 Normal Startup

Although our TCP example is small (about 150 lines of code for the two `main` functions, `str_echo`, `str_cli`, `readline`, and `writen`), it is essential that we understand how the client and server start, how they end, and most importantly, what happens when something goes wrong: the client host crashes, the client process crashes, network connectivity is lost, and so on. Only by understanding these boundary conditions, and their interaction with the TCP/IP protocols, can we write robust clients and servers that can handle these conditions.

We first start the server in the background on the host `linux`.

```
linux % tcpserv01 &
[1] 17870
```

When the server starts, it calls `socket`, `bind`, `listen`, and `accept`, blocking in the call to `accept`. (We have not started the client yet.) Before starting the client, we run the `netstat` program to verify the state of the server's listening socket.

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address      State
tcp        0      0 *:9877             *:*                   LISTEN
```

Here we show only the first line of output (the heading), plus the line that we are interested in. This command shows the status of *all* sockets on the system, which can be lots of output. We must specify the `-a` flag to see listening sockets.

The output is what we expect. A socket is in the LISTEN state with a wildcard for the local IP address and a local port of 9877. `netstat` prints an asterisk for an IP address of 0 (`INADDR_ANY`, the wildcard) or for a port of 0.

We then start the client on the same host, specifying the server's IP address of 127.0.0.1 (the loopback address). We could have also specified the server's normal (nonloopback) IP address.

```
linux % tcpcli01 127.0.0.1
```

The client calls `socket` and `connect`, the latter causing TCP's three-way handshake to take place. When the three-way handshake completes, `connect` returns in the client and `accept` returns in the server. The connection is established. The following steps then take place:

1. The client calls `str_cli`, which will block in the call to `fgets`, because we have not typed a line of input yet.

2. When `accept` returns in the server, it calls `fork` and the child calls `str_echo`. This function calls `readline`, which calls `read`, which blocks while waiting for a line to be sent from the client.
3. The server parent, on the other hand, calls `accept` again, and blocks while waiting for the next client connection.

We have three processes, and all three are asleep (blocked): client, server parent, and server child.

When the three-way handshake completes, we purposely list the client step first, and then the server steps. The reason can be seen in Figure 2.5: `connect` returns when the second segment of the handshake is received by the client, but `accept` does not return until the third segment of the handshake is received by the server, one-half of the RTT after `connect` returns.

We purposely run the client and server on the same host because this is the easiest way to experiment with client/server applications. Since we are running the client and server on the same host, `netstat` now shows two additional lines of output, corresponding to the TCP connection:

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 local host:9877         localhost:42758
ESTABLISHED
tcp        0      0 local host:42758        localhost:9877
ESTABLISHED
tcp        0      0 *:9877                  *:*                     LISTEN
```

The first of the ESTABLISHED lines corresponds to the server child's socket, since the local port is 9877. The second of the ESTABLISHED lines is the client's socket, since the local port is 42758. If we were running the client and server on different hosts, the client host would display only the client's socket, and the server host would display only the two server sockets.

We can also use the `ps` command to check the status and relationship of these processes.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND         WCHAN
22038 22036 pts/6    S    -bash           wait4
17870 22038 pts/6    S    ./tcpserv01      wait_for_connect
19315 17870 pts/6    S    ./tcpserv01      tcp_data_wait
19314 22038 pts/6    S    ./tcpcli01 127.0 read_chan
```

(We have used very specific arguments to `ps` to only show us the information that pertains to this discussion.) In this output, we ran the client and server from the same window (`pts/6`, which stands for pseudo-terminal number 6). The PID and PPID columns show the parent and child relationships. We can tell that the first `tcpserv01` line is the parent and the second `tcpserv01` line is the child since the PPID of the child is the parent's PID. Also, the PPID of the parent is the shell (`bash`).

The STAT column for all three of our network processes is "S," meaning the process is sleeping (waiting for something). When a process is asleep, the WCHAN column specifies the condition. Linux prints `wait_for_connect` when a process is blocked in either `accept` or `connect`, `tcp_data_wait` when a process is blocked on socket input or output, or `read_chan` when a process is blocked on terminal I/O. The WCHAN values for our three network processes therefore make sense.

5.7 Normal Termination

At this point, the connection is established and whatever we type to the client is echoed back.

```
linux % tcpcli01 127.0.0.1        we showed this line earlier
hello, world                      we now type this
hello, world                      and the line is echoed
good bye
good bye
^D                                Control-D is our terminal EOF character
```

We type in two lines, each one is echoed, and then we type our terminal EOF character (Control-D), which terminates the client. If we immediately execute `netstat`, we have

```
linux % netstat -a | grep 9877
tcp        0      0 *:9877               *:*                LISTEN
tcp        0      0 localhost:42758      localhost:9877     TIME_WAIT
```

The client's side of the connection (since the local port is 42758) enters the TIME_WAIT state (Section 2.7), and the listening server is still waiting for another client connection. (This time we pipe the output of `netstat` into `grep`, printing only the lines with our server's well-known port. Doing this also removes the heading line.)

We can follow through the steps involved in the normal termination of our client and server:

1. When we type our EOF character, `fgets` returns a null pointer and the function `str_cli` (Figure 5.5) returns.
2. When `str_cli` returns to the client `main` function (Figure 5.4), the latter terminates by calling `exit`.
3. Part of process termination is the closing of all open descriptors, so the client socket is closed by the kernel. This sends a FIN to the server, to which the server TCP responds with an ACK. This is the first half of the TCP connection termination sequence. At this point, the server socket is in the CLOSE_WAIT state and the client socket is in the FIN_WAIT_2 state (Figures 2.4 and 2.5).
4. When the server TCP receives the FIN, the server child is blocked in a call to `readline` (Figure 5.3), and `readline` then returns 0. This causes the `str_echo` function to return to the server child `main`.
5. The server child terminates by calling `exit` (Figure 5.2).
6. All open descriptors in the server child are closed. The closing of the connected socket by the child causes the final two segments of the TCP connection termination to take place: a FIN from the server to the client, and an ACK from the client (Figure 2.5). At this point, the connection is completely terminated. The client socket enters the TIME_WAIT state.
7. Finally, the `SIGCHLD` signal is sent to the parent when the server child terminates. This occurs in this example, but we do not catch the signal in our code, and the default action of the signal is to be ignored. Thus, the child enters the zombie state. We can verify this with the `ps` command.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND          WCHAN
22038 22036 pts/6    S    -bash            read_chan
17870 22038 pts/6    S    ./tcpserv01      wait_for_connect
19315 17870 pts/6    Z    [tcpserv01 <defu do_exit
```

The STAT of the child is now `Z` (for zombie).

We need to clean up our zombie processes and doing this requires dealing with Unix signals. In the next section, we will give an overview of signal handling.

5.8 POSIX Signal Handling

A *signal* is a notification to a process that an event has occurred. Signals are sometimes called *software interrupts*. Signals usually occur *asynchronously*. By this we mean that a process doesn't know ahead of time exactly when a signal will occur.

Signals can be sent

- By one process to another process (or to itself)
- By the kernel to a process

The `SIGCHLD` signal that we described at the end of the previous section is one that is sent by the kernel whenever a process terminates, to the parent of the terminating process.

Every signal has a *disposition*, which is also called the *action* associated with the signal. We set the disposition of a signal by calling the `sigaction` function (described shortly) and we have three choices for the disposition:

1. We can provide a function that is called whenever a specific signal occurs. This function is called a *signal handler* and this action is called *catching* a signal. The two signals `SIGKILL` and `SIGSTOP` cannot be caught. Our function is called with a single integer argument that is the signal number and the function returns nothing. Its function prototype is therefore
2.
3.
4.
5. `void handler (int signo);`
6.

   For most signals, calling `sigaction` and specifying a function to be called when the signal occurs is all that is required to catch a signal. But we will see later that a few signals, `SIGIO`, `SIGPOLL`, and `SIGURG`, all require additional actions on the part of the process to catch the signal.

7. We can *ignore* a signal by setting its disposition to `SIG_IGN`. The two signals `SIGKILL` and `SIGSTOP` cannot be ignored.

8. We can set the *default* disposition for a signal by setting its disposition to `SIG_DFL`. The default is normally to terminate a process on receipt of a signal, with certain signals also generating a core image of the process in its current working directory. There are a few signals whose default disposition is to be ignored: `SIGCHLD` and `SIGURG` (sent on the arrival of out-of-band data, Chapter 24) are two that we will encounter in this text.

signal Function

The POSIX way to establish the disposition of a signal is to call the `sigaction` function. This gets complicated, however, as one argument to the function is a structure that we must allocate and fill in. An easier way to set the disposition of a signal is to call the `signal` function. The first argument is the signal name and the second argument is either a pointer to a function or one of the constants `SIG_IGN` or `SIG_DFL`. But, `signal` is an historical function that predates POSIX. Different implementations provide different signal semantics when it is called, providing backward compatibility, whereas POSIX explicitly spells out the semantics when `sigaction` is called. The solution is to define our own function named `signal` that just calls the POSIX `sigaction` function. This provides a simple interface with the desired POSIX semantics. We include this function in our own library, along with our `err_`*XXX* functions and our wrapper functions, for example, that we specify when building any of our programs in this text. This function is shown in Figure 5.6 (the corresponding wrapper function, `Signal`, is not shown here as it would be the same whether it called our function or a vendor-supplied `signal` function).

Figure 5.6 signal function that calls the POSIX sigaction function.

lib/signal.c

```
#include    "unp.h"

Sigfunc *
signal (int signo, Sigfunc *func)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset (&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;     /* SunOS 4.x */
#endif
    } else {
```

```
14 #ifdef  SA_RESTART
15         act.sa_flags |= SA_RESTART; /* SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction (signo, &act, &oact) < 0)
19         return (SIG_ERR);
20     return (oact.sa_handler);
21 }
```

Simplify function prototype using typedef

2□3 The normal function prototype for `signal` is complicated by the level of nested parentheses.

```
void (*signal (int signo, void (*func) (int))) (int);
```

To simplify this, we define the `Sigfunc` type in our `unp.h` header as

```
typedef    void    Sigfunc(int);
```

stating that signal handlers are functions with an integer argument and the function returns nothing (`void`). The function prototype then becomes

```
Sigfunc *signal (int signo, Sigfunc *func);
```

A pointer to a signal handling function is the second argument to the function, as well as the return value from the function.

Set handler

6 The `sa_handler` member of the `sigaction` structure is set to the *func* argument.

Set signal mask for handler

7 POSIX allows us to specify a set of signals that will be *blocked* when our signal handler is called. Any signal that is blocked cannot be *delivered* to a process. We set the `sa_mask` member to the empty set, which means that no additional signals will be blocked while our signal handler is running. POSIX guarantees that the signal being caught is always blocked while its handler is executing.

Set SA_RESTART flag

8□17 `SA_RESTART` is an optional flag. When the flag is set, a system call interrupted by this signal will be automatically restarted by the kernel. (We will talk more about interrupted system calls in the next section when we continue our example.) If the signal being caught is not `SIGALRM`, we specify the `SA_RESTART` flag, if defined. (The reason for making a special case for `SIGALRM` is that the purpose of generating this signal is normally to place a timeout on an I/O operation, as we will show in Section 14.2, in which case, we want the blocked system call to be interrupted by the signal.) Some older systems, notably SunOS 4.x, automatically restart an interrupted system call by default and then define the complement of this flag as `SA_INTERRUPT`. If this flag is defined, we set it if the signal being caught is `SIGALRM`.

Call sigaction

18□20 We call `sigaction` and then return the old action for the signal as the return value of the `signal` function.

Throughout this text, we will use the `signal` function from Figure 5.6.

POSIX Signal Semantics

We summarize the following points about signal handling on a POSIX-compliant system:

- Once a signal handler is installed, it remains installed. (Older systems removed the signal handler each time it was executed.)
- While a signal handler is executing, the signal being delivered is blocked. Furthermore, any additional signals that were specified in the `sa_mask` signal set passed to `sigaction` when the handler was installed are also blocked. In Figure 5.6, we set `sa_mask` to the empty set, meaning no additional signals are blocked other than the signal being caught.
- If a signal is generated one or more times while it is blocked, it is normally delivered only one time after the signal is unblocked. That is, by default, Unix signals are not *queued*. We will see an example of this in the next section. The POSIX real-time standard, 1003.1b, defines some reliable signals that are queued, but we do not use them in this text.
- It is possible to selectively block and unblock a set of signals using the `sigprocmask` function. This lets us protect a critical region of code by

preventing certain signals from being caught while that region of code is executing.

5.9 Handling 'SIGCHLD' Signals

The purpose of the zombie state is to maintain information about the child for the parent to fetch at some later time. This information includes the process ID of the child, its termination status, and information on the resource utilization of the child (CPU time, memory, etc.). If a process terminates, and that process has children in the zombie state, the parent process ID of all the zombie children is set to 1 (the `init` process), which will inherit the children and clean them up (i.e., `init` will `wait` for them, which removes the zombie). Some Unix systems show the COMMAND column for a zombie process as `<defunct>`.

Handling Zombies

Obviously we do not want to leave zombies around. They take up space in the kernel and eventually we can run out of processes. Whenever we `fork` children, we must `wait` for them to prevent them from becoming zombies. To do this, we establish a signal handler to catch `SIGCHLD`, and within the handler, we call `wait`. (We will describe the `wait` and `waitpid` functions in Section 5.10.) We establish the signal handler by adding the function call

```
Signal (SIGCHLD, sig_chld);
```

in Figure 5.2, after the call to `listen`. (It must be done sometime before we `fork` the first child and needs to be done only once.) We then define the signal handler, the function `sig_chld`, which we show in Figure 5.7.

Figure 5.7 Version of SIGCHLD signal handler that calls wait (improved in Figure 5.11).

tcpcliserv/sigchldwait.c

```
#include     "unp.h"

void
sig_chld(int signo)
{
    pid_t   pid;
   int     stat;

   pid = wait(&stat);
   printf("child %d terminated\", pid);
   return;
}
```

Warning: Calling standard I/O functions such as `printf` in a signal handler is not recommended, for reasons that we will discuss in Section 11.18. We call `printf` here as a diagnostic tool to see when the child terminates.

Under System V and Unix 98, the child of a process does not become a zombie if the process sets the disposition of `SIGCHLD` to `SIG_IGN`. Unfortunately, this works only under System V and Unix 98. POSIX explicitly states that this behavior is unspecified. The portable way to handle zombies is to catch `SIGCHLD` and call `wait` or `waitpid`.

If we compile this program□Figure 5.2, with the call to `Signal`, with our `sig_chld` handler□under Solaris 9 and use the `signal` function from the system library (not our version from Figure 5.6), we have the following:

```
solaris % tcpserv02 &                    start server in background
[2] 16939
solaris % tcpcli01 127.0.0.1             then start client in foreground
hi there                                 we type this
hi there                                 and this is echoed
^D                                       we type our EOF character
child 16942 terminated                   output by printf in signal handler
accept error: Interrupted system call    main function aborts
```

The sequence of steps is as follows:

1. We terminate the client by typing our EOF character. The client TCP sends a FIN to the server and the server responds with an ACK.
2. The receipt of the FIN delivers an EOF to the child's pending `readline`. The child terminates.
3. The parent is blocked in its call to `accept` when the `SIGCHLD` signal is delivered. The `sig_chld` function executes (our signal handler), `wait` fetches the child's PID and termination status, and `printf` is called from the signal handler. The signal handler returns.
4. Since the signal was caught by the parent while the parent was blocked in a slow system call (`accept`), the kernel causes the `accept` to return an error of

EINTR (interrupted system call). The parent does not handle this error (Figure 5.2), so it aborts.

The purpose of this example is to show that when writing network programs that catch signals, we must be cognizant of interrupted system calls, and we must handle them. In this specific example, running under Solaris 9, the signal function provided in the standard C library does not cause an interrupted system call to be automatically restarted by the kernel. That is, the SA_RESTART flag that we set in Figure 5.6 is not set by the signal function in the system library. Some other systems automatically restart the interrupted system call. If we run the same example under 4.4BSD, using its library version of the signal function, the kernel restarts the interrupted system call and accept does not return an error. To handle this potential problem between different operating systems is one reason we define our own version of the signal function that we use throughout the text (Figure 5.6).

As part of the coding conventions used in this text, we always code an explicit return in our signal handlers (Figure 5.7), even though falling off the end of the function does the same thing for a function returning void. When reading the code, the unnecessary return statement acts as a reminder that the return may interrupt a system call.

Handling Interrupted System Calls

We used the term "slow system call" to describe accept, and we use this term for any system call that can block forever. That is, the system call need never return. Most networking functions fall into this category. For example, there is no guarantee that a server's call to accept will ever return, if there are no clients that will connect to the server. Similarly, our server's call to read in Figure 5.3 will never return if the client never sends a line for the server to echo. Other examples of slow system calls are reads and writes of pipes and terminal devices. A notable exception is disk I/O, which usually returns to the caller (assuming no catastrophic hardware failure).

The basic rule that applies here is that when a process is blocked in a slow system call *and* the process catches a signal *and* the signal handler returns, the system call *can* return an error of EINTR. *Some* kernels automatically restart *some* interrupted system calls. For portability, when we write a program that catches signals (most concurrent servers catch SIGCHLD), we must be prepared for slow system calls to return EINTR. Portability problems are caused by the qualifiers "can" and "some," which were used earlier, and the fact that support for the POSIX SA_RESTART flag is optional. Even if an implementation supports the SA_RESTART flag, not all interrupted system calls may automatically be restarted. Most Berkeley-derived implementations, for example, never automatically restart select, and some of these implementations never restart accept or recvfrom.

To handle an interrupted `accept`, we change the call to `accept` in Figure 5.2, the beginning of the `for` loop, to the following:

```
    for ( ; ; ) {
        clilen = sizeof (cliaddr);
        if ( (connfd = accept (listenfd, (SA *) &cliaddr, &clilen)) < 0)
{
            if (errno == EINTR)
                continue;         /* back to for () */
            else
                err_sys ("accept error");
        }
```

Notice that we call `accept` and not our wrapper function `Accept`, since we must handle the failure of the function ourselves.

What we are doing in this piece of code is restarting the interrupted system call. This is fine for `accept`, along with functions such as `read`, `write`, `select`, and `open`. But there is one function that we cannot restart: `connect`. If this function returns `EINTR`, we cannot call it again, as doing so will return an immediate error. When `connect` is interrupted by a caught signal and is not automatically restarted, we must call `select` to wait for the connection to complete, as we will describe in Section 16.3.

5.10 'wait' and 'waitpid' Functions

In Figure 5.7, we called the `wait` function to handle the terminated child.

`#include <sys/wait.h>`
`pid_t wait (int` **statloc*);
`pid_t waitpid (pid_t` *pid*, `int` **statloc*, `int` *options*);
Both return: process ID if OK, 0 or□1 on error

`wait` and `waitpid` both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (an integer) is returned through the *statloc* pointer. There are three macros that we can call that examine the termination status and tell us if the child terminated normally, was killed by a signal, or was just stopped by job control. Additional macros let us then fetch the exit status of the child, or the value of the signal that killed the child, or the value of the job-control signal that stopped the child. We will use the `WIFEXITED` and `WEXITSTATUS` macros in Figure 15.10 for this purpose.

If there are no terminated children for the process calling `wait`, but the process has one or more children that are still executing, then `wait` blocks until the first of the existing children terminates.

`waitpid` gives us more control over which process to wait for and whether or not to block. First, the *pid* argument lets us specify the process ID that we want to wait for. A value of -1 says to wait for the first of our children to terminate. (There are other options, dealing with process group IDs, but we do not need them in this text.) The *options* argument lets us specify additional options. The most common option is `WNOHANG`. This option tells the kernel not to block if there are no terminated children.

Difference between wait and waitpid

We now illustrate the difference between the `wait` and `waitpid` functions when used to clean up terminated children. To do this, we modify our TCP client as shown in Figure 5.9. The client establishes five connections with the server and then uses only the first one (`sockfd[0]`) in the call to `str_cli`. The purpose of establishing multiple connections is to spawn multiple children from the concurrent server, as shown in Figure 5.8.

Figure 5.8. Client with five established connections to same concurrent server.

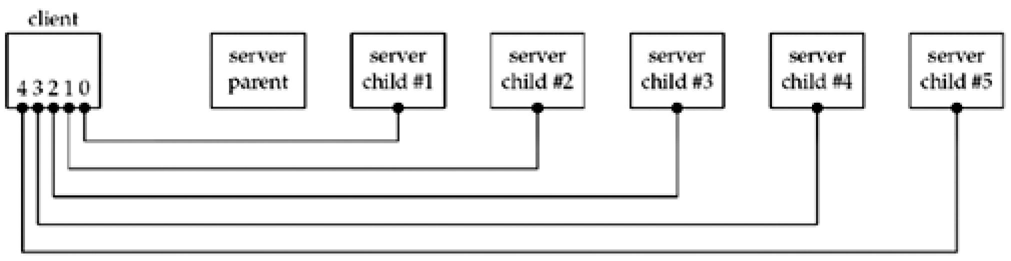


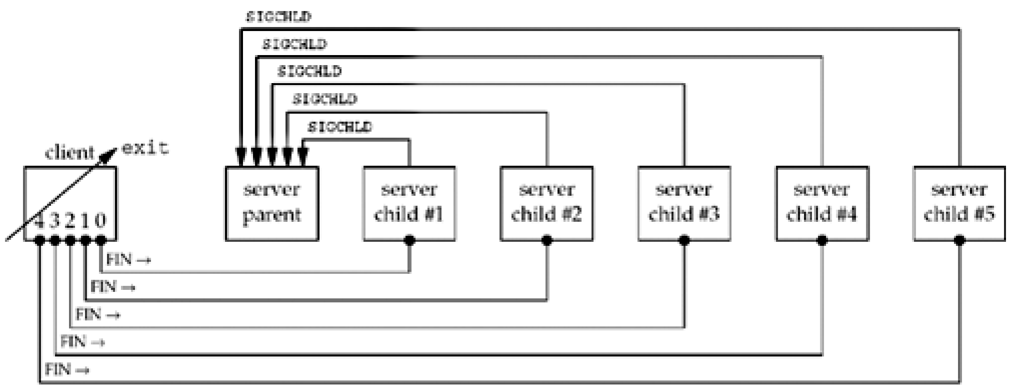
Figure 5.9 TCP client that establishes five connections with server.

tcpcliserv/tcpcli04.c

```
#include     "unp.h"

int
main (int argc, char **argv)
{
    int     i, sockfd[5];
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit ("usage: tcpcli <IPaddress>";

    for (i = 0; i < 5; i++) {
        sockfd[i] = Socket (AF_INET, SOCK_STREAM, 0);

        bzero (&servaddr, sizeof (servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons (SERV_PORT);
        Inet_pton (AF_INET, argv[1], &servaddr.sin_addr);

        Connect (sockfd[i], (SA *) &servaddr, sizeof (servaddr));
   }

   str_cli (stdin, sockfd[0]);  /* do it all */

   exit(0);
}
```

When the client terminates, all open descriptors are closed automatically by the kernel (we do not call `close`, only `exit`), and all five connections are terminated at about the same time. This causes five FINs to be sent, one on each connection, which in turn causes all five server children to terminate at about the same time. This causes five `SIGCHLD` signals to be delivered to the parent at about the same time, which we show in Figure 5.10.

Figure 5.10. Client terminates, closing all five connections, terminating all five children.



It is this delivery of multiple occurrences of the same signal that causes the problem we are about to see.

We first run the server in the background and then our new client. Our server is Figure 5.2, modified to call `signal` to establish Figure 5.7 as a signal handler for `SIGCHLD`.

```
linux % tcpserv03 &
[1] 20419
linux % tcpcli04 127.0.0.1
hello                                    we type this
hello                                    and it is echoed
^D                                       we then type our EOF character
child 20426 terminated                   output by server
```

The first thing we notice is that only one `printf` is output, when we expect all five children to have terminated. If we execute `ps`, we see that the other four children still exist as zombies.

```
  PID TTY          TIME CMD
20419 pts/6    00:00:00 tcpserv03
20421 pts/6    00:00:00 tcpserv03 <defunct>
20422 pts/6    00:00:00 tcpserv03 <defunct>
20423 pts/6    00:00:00 tcpserv03 <defunct>
```

Establishing a signal handler and calling `wait` from that handler are insufficient for preventing zombies. The problem is that all five signals are generated before the signal handler is executed, and the signal handler is executed only one time because Unix signals are normally not *queued*. Furthermore, this problem is nondeterministic. In the example we just ran, with the client and server on the same host, the signal handler is executed once, leaving four zombies. But if we run the client and server on different hosts, the signal handler is normally executed two times: once as a result of the first signal being generated, and since the other four signals occur while the signal handler is executing, the handler is called only one more time. This leaves three zombies. But sometimes, probably dependent on the timing of the FINs arriving at the server host, the signal handler is executed three or even four times.

The correct solution is to call `waitpid` instead of `wait`. Figure 5.11 shows the version of our `sig_chld` function that handles `SIGCHLD` correctly. This version works because we call `waitpid` within a loop, fetching the status of any of our children that have terminated. We must specify the `WNOHANG` option: This tells `waitpid` not to block if there are running children that have not yet terminated. In Figure 5.7, we cannot call `wait` in a loop, because there is no way to prevent `wait` from blocking if there are running children that have not yet terminated.

Figure 5.12 shows the final version of our server. It correctly handles a return of `EINTR` from `accept` and it establishes a signal handler (Figure 5.11) that calls `waitpid` for all terminated children.

Figure 5.11 Final (correct) version of sig_chld function that calls waitpid.

tcpcliserv/sigchldwaitpid.c

```
#include    "unp.h"

void
sig_chld(int signo)
{
   pid_t    pid;
   int      stat;

   while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
       printf("child %d terminated\n", pid);
   return;
}
```

Figure 5.12 Final (correct) version of TCP server that handles an error of EINTR from accept.

tcpcliserv/tcpserv04.c

```
#include     "unp.h"
int
main(int argc, char **argv)
{
   int     listenfd, connfd;
   pid_t   childpid;
   socklen_t clilen;
   struct sockaddr_in cliaddr, servaddr;
   void    sig_chld(int);

   listenfd = Socket (AF_INET, SOCK_STREAM, 0);

   bzero (&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servaddr.sin_port = htons(SERV_PORT);
```

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

    Listen(listenfd, LISTENQ);

    Signal (SIGCHLD, sig_chld); /* must call waitpid() */

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if ( (connfd = accept (listenfd, (SA *) &cliaddr, &clilen)) < 0) {
            if (errno == EINTR)
                continue;          /* back to for() */
            else
                err_sys("accept error");
        }

        if ( (childpid = Fork()) == 0) { /* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        Close (connfd);         /* parent closes connected socket */
    }
}
```

The purpose of this section has been to demonstrate three scenarios that we can encounter with network programming:

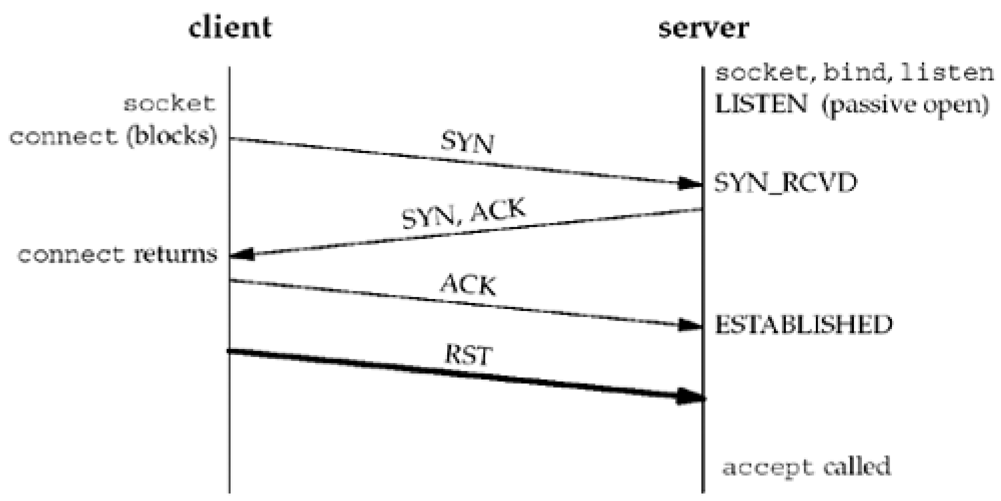
1. We must catch the `SIGCHLD` signal when `forking` child processes.
2. We must handle interrupted system calls when we catch signals.
3. A `SIGCHLD` handler must be coded correctly using `waitpid` to prevent any zombies from being left around.

The final version of our TCP server (Figure 5.12), along with the `SIGCHLD` handler in Figure 5.11, handles all three scenarios.

5.11 Connection Abort before 'accept' Returns

There is another condition similar to the interrupted system call example in the previous section that can cause `accept` to return a nonfatal error, in which case we should just call accept again. The sequence of packets shown in Figure 5.13 has been seen on busy servers (typically busy Web servers).

Figure 5.13. Receiving an RST for an ESTABLISHED connection before accept is called.



Here, the three-way handshake completes, the connection is established, and then the client TCP sends an RST (reset). On the server side, the connection is queued by its TCP, waiting for the server process to call accept when the RST arrives. Sometime later, the server process calls accept.

An easy way to simulate this scenario is to start the server, have it call socket, bind, and listen, and then go to sleep for a short period of time before calling accept. While the server process is asleep, start the client and have it call socket and connect. As soon as connect returns, set the SO_LINGER socket option to generate the RST (which we will describe in Section 7.5 and show an example of in Figure 16.21) and terminate.

Unfortunately, what happens to the aborted connection is implementation-dependent. Berkeley-derived implementations handle the aborted connection completely within the kernel, and the server process never sees it. Most SVR4 implementations, however, return an error to the process as the return from accept, and the error depends on the implementation. These SVR4 implementations return an errno of EPROTO ("protocol error"), but POSIX specifies that the return must be ECONNABORTED ("software caused connection abort") instead. The reason for the POSIX change is that EPROTO is also returned when some fatal protocol-related events occur on the streams subsystem. Returning the same error for the nonfatal abort of an established connection by the client makes it impossible for the server to know whether to call accept again or not. In the case of the ECONNABORTED error, the server can ignore the error and just call accept again.

The steps involved in Berkeley-derived kernels that never pass this error to the process can be followed in TCPv2. The RST is processed on p. 964, causing tcp_close to be called. This function calls `in_pcbdetach` on p. 897, which in turn calls `sofree` on p. 719. `sofree` (p. 473) finds that the socket being aborted is still on the listening socket's completed connection queue and removes the socket from the queue and frees the socket. When the server gets around to calling `accept`, it will never know that a connection that was completed has since been removed from the queue.

We will return to these aborted connections in Section 16.6 and see how they can present a problem when combined with `select` and a listening socket in the normal blocking mode.

5.12 Termination of Server Process

We will now start our client/server and then kill the server child process. This simulates the crashing of the server process, so we can see what happens to the client. (We must be careful to distinguish between the crashing of the server *process*, which we are about to describe, and the crashing of the server *host*, which we will describe in Section 5.14.) The following steps take place:

1. We start the server and client and type one line to the client to verify that all is okay. That line is echoed normally by the server child.
2. We find the process ID of the server child and `kill` it. As part of process termination, all open descriptors in the child are closed. This causes a FIN to be sent to the client, and the client TCP responds with an ACK. This is the first half of the TCP connection termination.
3. The `SIGCHLD` signal is sent to the server parent and handled correctly (Figure 5.12).
4. Nothing happens at the client. The client TCP receives the FIN from the server TCP and responds with an ACK, but the problem is that the client process is blocked in the call to `fgets` waiting for a line from the terminal.
5. Running `netstat` at this point shows the state of the sockets.

```



linux % netstat -a | grep 9877
 tcp       0      0 *:9877              *:*                LISTEN
 tcp       0      0 localhost:9877      localhost:43604
FIN_WAIT2
 tcp       1      0 localhost:43604     localhost:9877
CLOSE_WAIT

```

From Figure 2.4, we see that half of the TCP connection termination sequence has taken place.

14. We can still type a line of input to the client. Here is what happens at the client starting from Step 1:

```
linux % tcpcli01 127.0.0.1          start client
hello                               the first line that we type
hello                               is echoed correctly here we kill the server
                                    child on the server host
another line                        we then type a second line to the client
str_cli : server terminated
prematurely
```

15. When we type "another line," `str_cli` calls `writen` and the client TCP sends the data to the server. This is allowed by TCP because the receipt of the FIN by the client TCP only indicates that the server process has closed its end of the connection and will not be sending any more data. The receipt of the FIN does *not* tell the client TCP that the server process has terminated (which in this case, it has). We will cover this again in Section 6.6 when we talk about TCP's half-close.
16. When the server TCP receives the data from the client, it responds with an RST since the process that had that socket open has terminated. We can verify that the RST was sent by watching the packets with `tcpdump`.
17. The client process will not see the RST because it calls `readline` immediately after the call to `writen` and `readline` returns 0 (EOF) immediately because of the FIN that was received in Step 2. Our client is not expecting to receive an EOF at this point (Figure 5.5) so it quits with the error message "server terminated prematurely."
18. When the client terminates (by calling `err_quit` in Figure 5.5), all its open descriptors are closed.

What we have described also depends on the timing of the example. The client's call to `readline` may happen before the server's RST is received by the client, or it may happen after. If the `readline` happens before the RST is received, as we've shown in our example, the result is an unexpected EOF in the client. But if the RST arrives first, the result is an `ECONNRESET` ("Connection reset by peer") error return from `readline`.

The problem in this example is that the client is blocked in the call to `fgets` when the FIN arrives on the socket. The client is really working with two descriptors□the socket and the user input□and instead of blocking on input from only one of the two sources (as `str_cli` is currently coded), it should block on input from either source. Indeed, this is one purpose of the `select` and `poll` functions, which we will describe in

Chapter 6. When we recode the `str_cli` function in Section 6.4, as soon as we `kill` the server child, the client is notified of the received FIN.

5.13 'SIGPIPE' Signal

What happens if the client ignores the error return from `readline` and writes more data to the server? This can happen, for example, if the client needs to perform two writes to the server before reading anything back, with the first write eliciting the RST.

The rule that applies is: When a process writes to a socket that has received an RST, the `SIGPIPE` signal is sent to the process. The default action of this signal is to terminate the process, so the process must catch the signal to avoid being involuntarily terminated.

If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns `EPIPE`.

A frequently asked question (FAQ) on Usenet is how to obtain this signal on the first write, and not the second. This is not possible. Following our discussion above, the first write elicits the RST and the second write elicits the signal. It is okay to write to a socket that has received a FIN, but it is an error to write to a socket that has received an RST.

To see what happens with `SIGPIPE`, we modify our client as shown in Figure 5.14.

Figure 5.14 str_cli that calls writen twice.

tcpcliserv/str_cli11.c

```
#include     "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    char    sendline [MAXLINE], recvline [MAXLINE];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Writen(sockfd, sendline, 1);
        sleep(1);
        Writen(sockfd, sendline + 1, strlen(sendline) - 1);
10        if (Readline(sockfd, recvline, MAXLINE) == 0)
11            err_quit("str_cli: server terminated prematurely");

12        Fputs(recvline, stdout);
13    }
14 }
```

7□9 All we have changed is to call `writen` two times: the first time the first byte of data is written to the socket, followed by a pause of one second, followed by the remainder of the line. The intent is for the first `writen` to elicit the RST and then for the second `writen` to generate `SIGPIPE`.

If we run the client on our Linux host, we get:

```
linux % tcpclill 127.0.0.1
hi there                 we type this line
hi there                 this is echoed by the server
                         here we kill the server child
bye                      then we type this line
Broken pipe              this is printed by the shell
```

We start the client, type in one line, see that line echoed correctly, and then terminate the server child on the server host. We then type another line ("bye") and the shell tells us the process died with a `SIGPIPE` signal (some shells do not print anything when a process dies without dumping core, but the shell we're using for this example, `bash`, tells us what we want to know).

The recommended way to handle `SIGPIPE` depends on what the application wants to do when this occurs. If there is nothing special to do, then setting the signal disposition to `SIG_IGN` is easy, assuming that subsequent output operations will catch the error of `EPIPE` and terminate. If special actions are needed when the signal occurs (writing to a log file perhaps), then the signal should be caught and any desired actions can be performed in the signal handler. Be aware, however, that if multiple sockets are in use, the delivery of the signal will not tell us which socket encountered the error. If we need to know which `write` caused the error, then we must either ignore the signal or return from the signal handler and handle `EPIPE` from the `write`.

5.14 Crashing of Server Host

This scenario will test to see what happens when the server host crashes. To simulate this, we must run the client and server on different hosts. We then start the server, start the client, type in a line to the client to verify that the connection is up, disconnect the server host from the network, and type in another line at the client. This also covers the scenario of the server host being unreachable when the client sends data (i.e., some intermediate router goes down after the connection has been established).

The following steps take place:

1. When the server host crashes, nothing is sent out on the existing network connections. That is, we are assuming the host crashes and is not shut down by an operator (which we will cover in Section 5.16).
2. We type a line of input to the client, it is written by `writen` (Figure 5.5), and is sent by the client TCP as a data segment. The client then blocks in the call to `readline`, waiting for the echoed reply.
3. If we watch the network with `tcpdump`, we will see the client TCP continually retransmitting the data segment, trying to receive an ACK from the server. Section 25.11 of TCPv2 shows a typical pattern for TCP retransmissions: Berkeley-derived implementations retransmit the data segment 12 times, waiting for around 9 minutes before giving up. When the client TCP finally gives up (assuming the server host has not been rebooted during this time, or if the server host has not crashed but was unreachable on the network, assuming the host was still unreachable), an error is returned to the client process. Since the client is blocked in the call to `readline`, it returns an error. Assuming the server host crashed and there were no responses at all to the client's data segments, the error is `ETIMEDOUT`. But if some intermediate router determined that the server host was unreachable and responded with an ICMP "destination unreachable' message, the error is either `EHOSTUNREACH` or `ENETUNREACH`.

Although our client discovers (eventually) that the peer is down or unreachable, there are times when we want to detect this quicker than having to wait nine minutes. The solution is to place a timeout on the call to `readline`, which we will discuss in Section 14.2.

The scenario that we just discussed detects that the server host has crashed only when we send data to that host. If we want to detect the crashing of the server host even if we are not actively sending it data, another technique is required. We will discuss the `SO_KEEPALIVE` socket option in Section 7.5.

5.15 Crashing and Rebooting of Server Host

In this scenario, we will establish a connection between the client and server and then assume the server host crashes and reboots. In the previous section, the server host was still down when we sent it data. Here, we will let the server host reboot before sending it data. The easiest way to simulate this is to establish the connection, disconnect the server from the network, shut down the server host and then reboot it, and then reconnect the server host to the network. We do not want the client to see the server host shut down (which we will cover in Section 5.16).

As stated in the previous section, if the client is not actively sending data to the server when the server host crashes, the client is not aware that the server host has crashed. (This assumes we are not using the `SO_KEEPALIVE` socket option.) The following steps take place:

1. We start the server and then the client. We type a line to verify that the connection is established.
2. The server host crashes and reboots.
3. We type a line of input to the client, which is sent as a TCP data segment to the server host.
4. When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP responds to the received data segment from the client with an RST.
5. Our client is blocked in the call to `readline` when the RST is received, causing `readline` to return the error `ECONNRESET`.

If it is important for our client to detect the crashing of the server host, even if the client is not actively sending data, then some other technique (such as the `SO_KEEPALIVE` socket option or some client/server heartbeat function) is required.

5.16 Shutdown of Server Host

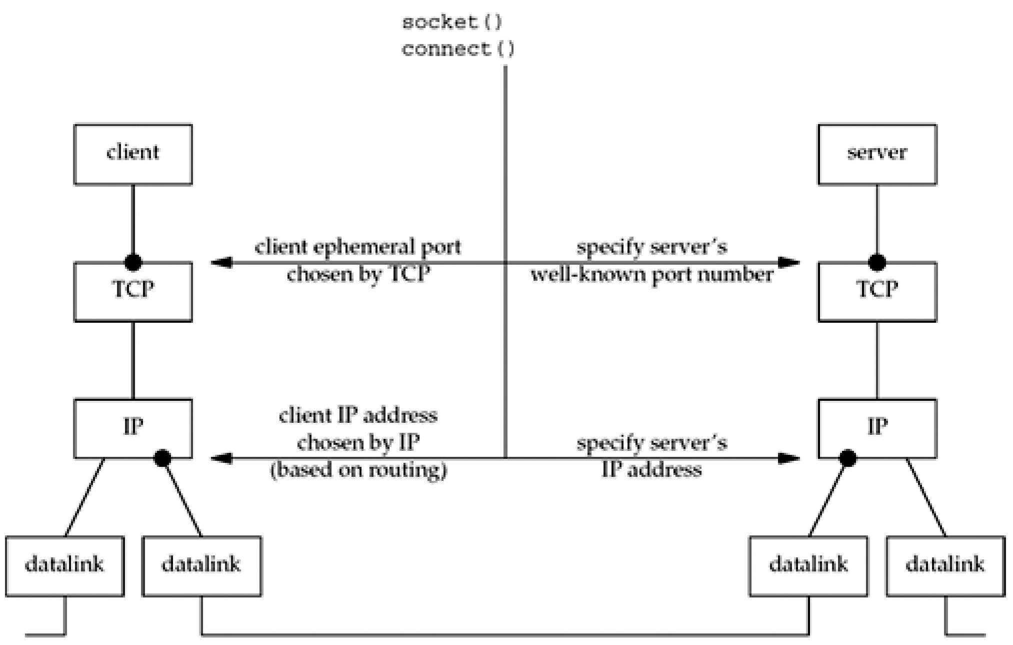
The previous two sections discussed the crashing of the server host, or the server host being unreachable across the network. We now consider what happens if the server host is shut down by an operator while our server process is running on that host.

When a Unix system is shut down, the `init` process normally sends the `SIGTERM` signal to all processes (we can catch this signal), waits some fixed amount of time (often between 5 and 20 seconds), and then sends the `SIGKILL` signal (which we cannot catch) to any processes still running. This gives all running processes a short amount of time to clean up and terminate. If we do not catch `SIGTERM` and terminate, our server will be terminated by the `SIGKILL` signal. When the process terminates, all open descriptors are closed, and we then follow the same sequence of steps discussed in Section 5.12. As stated there, we must use the `select` or `poll` function in our client to have the client detect the termination of the server process as soon as it occurs.

5.17 Summary of TCP Example

Before any TCP client and server can communicate with each other, each end must specify the socket pair for the connection: the local IP address, local port, foreign IP address, and foreign port. In Figure 5.15, we show these four values as bullets. This figure is from the client's perspective. The foreign IP address and foreign port must be specified by the client in the call to `connect`. The two local values are normally chosen by the kernel as part of the `connect` function. The client has the option of specifying either or both of the local values, by calling `bind` before `connect`, but this is not common.

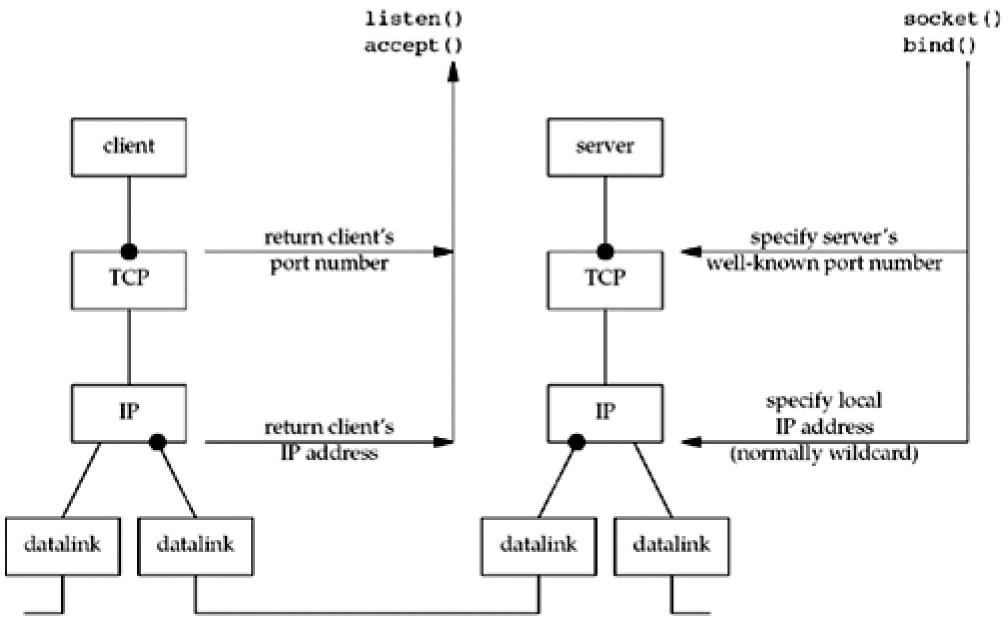
Figure 5.15. Summary of TCP client/server from client's perspective.


socket()
connect()
client
server
client ephemeral port chosen by TCP
specify server's well-known port number
TCP
TCP
client IP address chosen by IP (based on routing)
specify server's IP address
IP
IP
datalink
datalink
datalink
datalink

As we mentioned in Section 4.10, the client can obtain the two local values chosen by the kernel by calling `getsockname` after the connection is established.

Figure 5.16 shows the same four values, but from the server's perspective.

Figure 5.16. Summary of TCP client/server from server's perspective.


listen()
accept()
socket()
bind()
client
server
TCP
IP
datalink
return client's port number
return client's IP address
specify server's well-known port number
specify local IP address (normally wildcard)

The local port (the server's well-known port) is specified by bind. Normally, the server also specifies the wildcard IP address in this call. If the server binds the wildcard IP address on a multihomed host, it can determine the local IP address by calling getsockname after the connection is established (Section 4.10). The two foreign values are returned to the server by accept. As we mentioned in Section 4.10, if another program is execed by the server that calls accept, that program can call getpeername to determine the client's IP address and port, if necessary.

5.18 Data Format

In our example, the server never examines the request that it receives from the client. The server just reads all the data up through and including the newline and sends it back to the client, looking for only the newline. This is an exception, not the rule, and normally we must worry about the format of the data exchanged between the client and server.

Example: Passing Text Strings between Client and Server

Let's modify our server so that it still reads a line of text from the client, but the server now expects that line to contain two integers separated by white space, and the server returns the sum of those two integers. Our client and server main functions remain the same, as does our str_cli function. All that changes is our str_echo function, which we show in Figure 5.17.

Figure 5.17 str_echo function that adds two numbers.

tcpcliserv/str_ech08.c

```
 1 #include    "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5     long    arg1,    arg2;
 6     ssize_t n;
 7     char    line[MAXLINE];

 8     for ( ; ; ) {
 9         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
10             return;            /* connection closed by other end */

11         if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
12             snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
13         else
14             snprintf(line, sizeof(line), "input error\n");

15         n = strlen(line);
16         Writen(sockfd, line, n);
17     }
18 }
```

11□14 We call `sscanf` to convert the two arguments from text strings to long integers, and then `snprintf` is called to convert the result into a text string.

This new client and server work fine, regardless of the byte ordering of the client and server hosts.

Example: Passing Binary Structures between Client and Server

We now modify our client and server to pass binary values across the socket, instead of text strings. We will see that this does not work when the client and server are run on hosts with different byte orders, or on hosts that do not agree on the size of a long integer (Figure 1.17).

Our client and server `main` functions do not change. We define one structure for the two arguments, another structure for the result, and place both definitions in our `sum.h` header, shown in Figure 5.18. Figure 5.19 shows the `str_cli` function.

Figure 5.18 sum.h header.

tcpcliserv/sum.h

```
1 struct args {
2     long    arg1;
3     long    arg2;
4 };

5 struct result {
6     long    sum;
7 };
```

Figure 5.19 str_cli function which sends two binary integers to server.

tcpcliserv/str_cli09.c

```
 1 #include     "unp.h"
 2 #include     "sum.h"

 3 void
 4 str_cli(FILE *fp, int sockfd)
 5 {
 6     char     sendline[MAXLINE];
 7     struct args args;
 8     struct result result;

 9     while  (Fgets(sendline, MAXLINE, fp) != NULL) {

10         if  (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
11             printf("invalid input: %s", sendline);
12             continue;
13         }
14         Writen(sockfd, &args, sizeof(args));

15         if (Readn(sockfd, &result, sizeof(result)) == 0)
16             err_quit("str_cli: server terminated prematurely");

17         printf("%ld\n", result.sum);
18     }
19 }
```

10–14 `sscanf` converts the two arguments from text strings to binary, and we call `writen` to send the structure to the server.

15–17 We call `readn` to read the reply, and print the result using `printf`.

Figure 5.20 shows our `str_echo` function.

Figure 5.20 str_echo function that adds two binary integers.

tcpcliserv/str_ech09.c

```
 1 #include    "unp.h"
 2 #include    "sum.h"

 3 void
 4 str_echo(int sockfd)
 5 {
 6     ssize_t n;
 7     struct args args;
 8     struct result result;

 9     for ( ; ; ) {
10         if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
11             return;            /* connection closed by other end */

12         result.sum = args.arg1 + args.arg2;
13         Writen(sockfd, &result, sizeof (result));
14     }
15 }
```

9□14 We read the arguments by calling `readn`, calculate and store the sum, and call `writen` to send back the result structure.

If we run the client and server on two machines of the same architecture, say two SPARC machines, everything works fine. Here is the client interaction:

```
solaris % :tcpcli09 12.106.32.254
11 22                                         we type this
33                                            this is the server's reply
-11 -44
-55
```

But when the client and server are on two machines of different architectures (say the server is on the big-endian SPARC system `freebsd` and the client is on the little endian Intel system `linux`), it does not work.

```
linux % tcpcli09 206.168.112.96
1 2                         we type this
3                           and it works
-22 -77                     then we type this
-16777314                   and it does not work
```

The problem is that the two binary integers are sent across the socket in little-endian format by the client, but interpreted as big-endian integers by the server. We see that it appears to work for positive integers but fails for negative integers (see Exercise 5.8). There are really three potential problems with this example:

1. Different implementations store binary numbers in different formats. The most common formats are big-endian and little-endian, as we described in Section 3.4.
2. Different implementations can store the same C datatype differently. For example, most 32-bit Unix systems use 32 bits for a `long` but 64-bit systems typically use 64 bits for the same datatype (Figure 1.17). There is no guarantee that a `short`, `int`, or `long` is of any certain size.
3. Different implementations pack structures differently, depending on the number of bits used for the various datatypes and the alignment restrictions of the machine. Therefore, it is never wise to send binary structures across a socket.

There are two common solutions to this data format problem:

1. Pass all numeric data as text strings. This is what we did in Figure 5.17. This assumes that both hosts have the same character set.
2. Explicitly define the binary formats of the supported datatypes (number of bits, big- or little-endian) and pass all data between the client and server in this format. RPC packages normally use this technique. RFC 1832 [Srinivasan 1995] describes the *External Data Representation* (XDR) standard that is used with the Sun RPC package.

5.19 Summary

The first version of our echo client/server totaled about 150 lines (including the `readline` and `writen` functions), yet provided lots of details to examine. The first problem we encountered was zombie children and we caught the `SIGCHLD` signal to handle this. Our signal handler then called `waitpid` and we demonstrated that we must call this function instead of the older `wait` function, since Unix signals are not queued. This led us into some of the details of POSIX signal handling (additional information on this topic is provided in Chapter 10 of APUE).

The next problem we encountered was the client not being notified when the server process terminated. We saw that our client's TCP was notified, but we did not receive that notification since we were blocked, waiting for user input. We will use the `select` or `poll` function in Chapter 6 to handle this scenario, by waiting for any one of multiple descriptors to be ready, instead of blocking on a single descriptor.

We also discovered that if the server host crashes, we do not detect this until the client sends data to the server. Some applications must be made aware of this fact sooner; in Section 7.5, we will look at the `SO_KEEPALIVE` socket option.

Our simple example exchanged lines of text, which was okay since the server never looked at the lines it echoed. Sending numeric data between the client and server can lead to a new set of problems, as shown.

Exercises

5.1 Build the TCP server from Figures 5.2 and 5.3 and the TCP client from Figures 5.4 and 5.5. Start the server and then start the client. Type in a few lines to verify that the client and server work. Terminate the client by typing your EOF character and note the time. Use `netstat` on the client host to verify that the client's end of the connection goes through the TIME_WAIT state. Execute `netstat` every five seconds or so to see when the TIME_WAIT state ends. What is the MSL for this implementation?

5.2 What happens with our echo client/server if we run the client and redirect standard input to a binary file?

5.3 What is the difference between our echo client/server and using the Telnet client to communicate with our echo server?

5.4 In our example in Section 5.12, we verified that the first two segments of the connection termination are sent (the FIN from the server that is then ACKed by the client) by looking at the socket states using `netstat`. Are the final two segments exchanged (a FIN from the client that is ACKed by the server)? If so, when, and if not, why?

5.5 What happens in the example outlined in Section 5.14 if between Steps 2 and 3 we restart our server application on the server host?

5.6 To verify what we claimed happens with `SIGPIPE` in Section 5.13, modify Figure 5.4 as follows: Write a signal handler for `SIGPIPE` that just prints a message and returns. Establish this signal handler before calling `connect`.

Change the server's port number to 13, the daytime server. When the connection is established, sleep for two seconds, write a few bytes to the socket, sleep for another two seconds, and write a few more bytes to the socket. Run the program. What happens?

5.7 What happens in Figure 5.15 if the IP address of the server host that is specified by the client in its call to connect is the IP address associated with the rightmost datalink on the server, instead of the IP address associated with the leftmost datalink on the server?

5.8 In our example output from Figure 5.20, when the client and server were on different endian systems, the example worked for small positive numbers, but not for small negative numbers. Why? (*Hint*: Draw a picture of the values exchanged across the socket, similar to Figure 3.9.)

5.9 In our example in Figures 5.19 and 5.20, can we solve the byte ordering problem by having the client convert the two arguments into network byte order using htonl, having the server then call ntohl on each argument before doing the addition, and then doing a similar conversion on the result?

5.10 What happens in Figures 5.19 and 5.20 if the client is on a SPARC that stores a long in 32 bits, but the server is on a Digital Alpha that stores a long in 64 bits? Does this change if the client and server are swapped between these two hosts?

5.11 In Figure 5.15, we say that the client IP address is chosen by IP based on routing. What does this mean?

Chapter 6. I/O Multiplexing: The 'select' and 'poll' Functions

6.1 Introduction

In Section 5.12, we saw our TCP client handling two inputs at the same time: standard input and a TCP socket. We encountered a problem when the client was blocked in a call to `fgets` (on standard input) and the server process was killed. The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input, it never saw the EOF until it read from the socket (possibly much later). What we need is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output). This capability is called *I/O multiplexing* and is provided by the `select` and `poll` functions. We will also cover a newer POSIX variation of the former, called `pselect`.

Some systems provide more advanced ways for processes to wait for a list of events. A *poll device* is one mechanism provided in different forms by different vendors. This mechanism will be described in Chapter 14.

I/O multiplexing is typically used in networking applications in the following scenarios:

- When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used. This is the scenario we described previously.
- It is possible, but rare, for a client to handle multiple sockets at the same time. We will show an example of this using `select` in Section 16.5 in the context of a Web client.
- If a TCP server handles both a listening socket and its connected sockets, I/O multiplexing is normally used, as we will show in Section 6.8.
- If a server handles both TCP and UDP, I/O multiplexing is normally used. We will show an example of this in Section 8.15.

- If a server handles multiple services and perhaps multiple protocols (e.g., the `inetd` daemon that we will describe in Section 13.5), I/O multiplexing is normally used.

I/O multiplexing is not limited to network programming. Many nontrivial applications find a need for these techniques.

6.2 I/O Models

Before describing `select` and `poll`, we need to step back and look at the bigger picture, examining the basic differences in the five I/O models that are available to us under Unix:

- blocking I/O
- nonblocking I/O
- I/O multiplexing (`select` and `poll`)
- signal driven I/O (`SIGIO`)
- asynchronous I/O (the POSIX `aio_`functions)

You may want to skim this section on your first reading and then refer back to it as you encounter the different I/O models described in more detail in later chapters.

As we show in all the examples in this section, there are normally two distinct phases for an input operation:

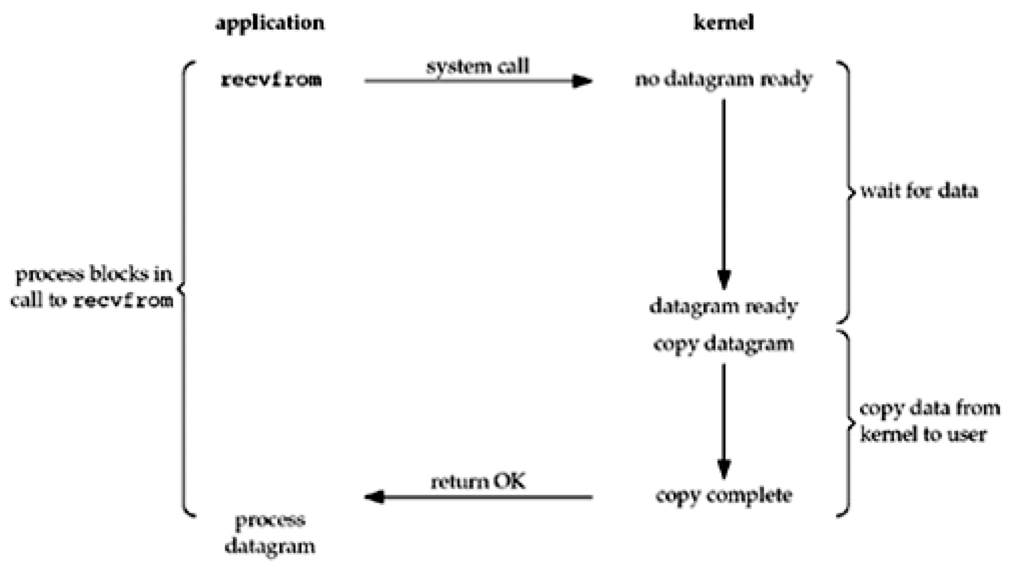
1. Waiting for the data to be ready
2. Copying the data from the kernel to the process

For an input operation on a socket, the first step normally involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel. The second step is copying this data from the kernel's buffer into our application buffer.

Blocking I/O Model

The most prevalent model for I/O is the *blocking I/O model*, which we have used for all our examples so far in the text. By default, all sockets are blocking. Using a datagram socket for our examples, we have the scenario shown in Figure 6.1.

Figure 6.1. Blocking I/O model.



We use UDP for this example instead of TCP because with UDP, the concept of data being "ready" to read is simple: either an entire datagram has been received or it has not. With TCP it gets more complicated, as additional variables such as the socket's low-water mark come into play.

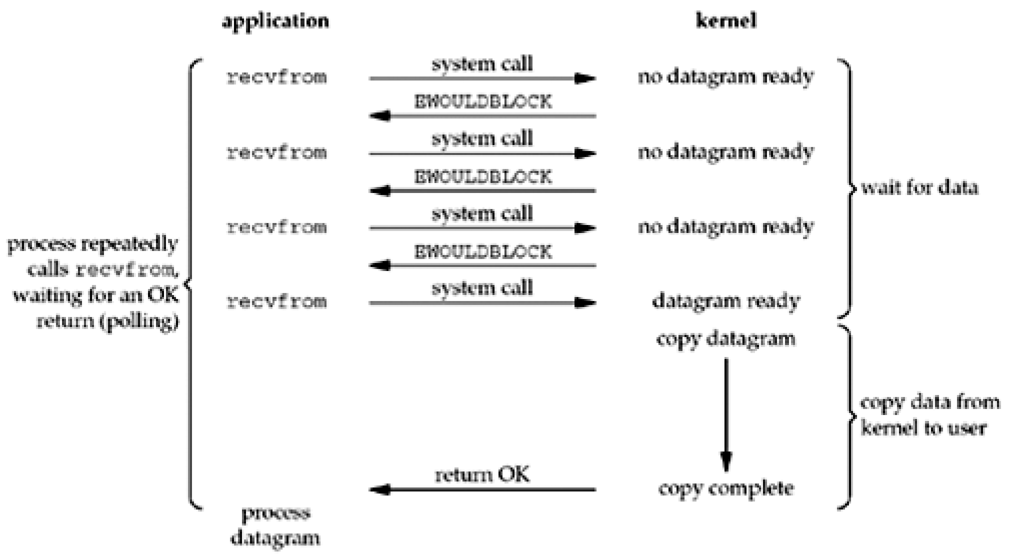
In the examples in this section, we also refer to `recvfrom` as a system call because we are differentiating between our application and the kernel. Regardless of how `recvfrom` is implemented (as a system call on a Berkeley-derived kernel or as a function that invokes the `getmsg` system call on a System V kernel), there is normally a switch from running in the application to running in the kernel, followed at some time later by a return to the application.

In Figure 6.1, the process calls `recvfrom` and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call being interrupted by a signal, as we described in Section 5.9. We say that our process is *blocked* the entire time from when it calls `recvfrom` until it returns. When `recvfrom` returns successfully, our application processes the datagram.

Nonblocking I/O Model

When we set a socket to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead." We will describe nonblocking I/O in Chapter 16, but Figure 6.2 shows a summary of the example we are considering.

Figure 6.2. Nonblocking I/O model.



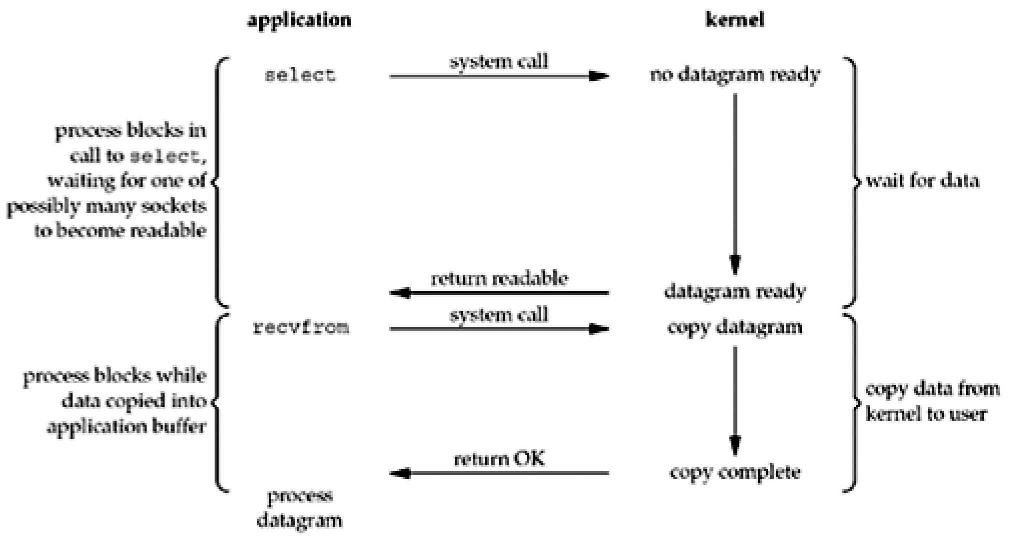
The first three times that we call `recvfrom`, there is no data to return, so the kernel immediately returns an error of `EWOULDBLOCK` instead. The fourth time we call `recvfrom`, a datagram is ready, it is copied into our application buffer, and `recvfrom` returns successfully. We then process the data.

When an application sits in a loop calling `recvfrom` on a nonblocking descriptor like this, it is called *polling*. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing Model

With *I/O multiplexing*, we call `select` or `poll` and block in one of these two system calls, instead of blocking in the actual I/O system call. Figure 6.3 is a summary of the I/O multiplexing model.

Figure 6.3. I/O multiplexing model.


application
kernel
select
system call
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

We block in a call to `select`, waiting for the datagram socket to be readable. When `select` returns that the socket is readable, we then call `recvfrom` to copy the datagram into our application buffer.

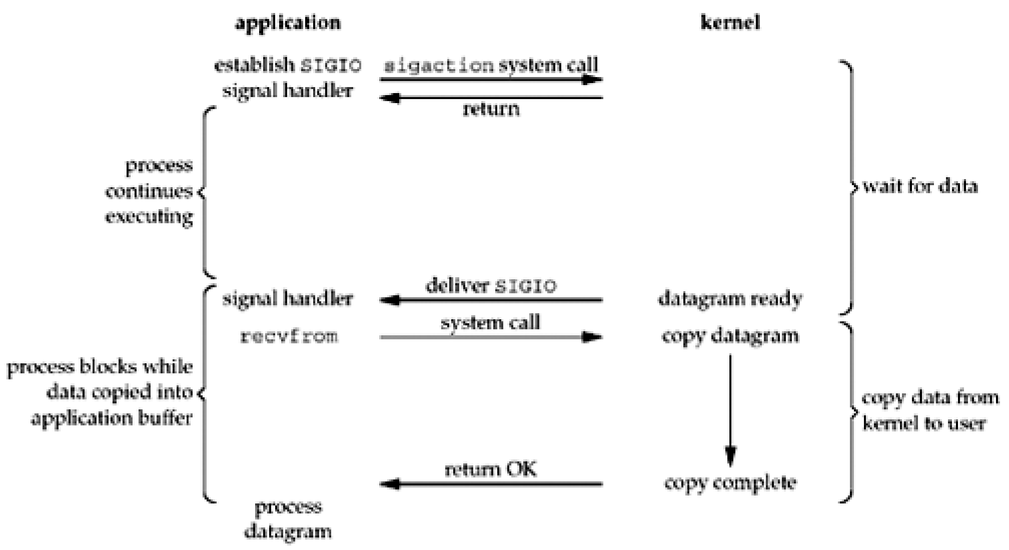
Comparing Figure 6.3 to Figure 6.1, there does not appear to be any advantage, and in fact, there is a slight disadvantage because using `select` requires two system calls instead of one. But the advantage in using `select`, which we will see later in this chapter, is that we can wait for more than one descriptor to be ready.

Another closely related I/O model is to use multithreading with blocking I/O. That model very closely resembles the model described above, except that instead of using `select` to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like `recvfrom`.

Signal-Driven I/O Model

We can also use signals, telling the kernel to notify us with the `SIGIO` signal when the descriptor is ready. We call this *signal-driven I/O* and show a summary of it in Figure 6.4.

Figure 6.4. Signal-Driven I/O model.



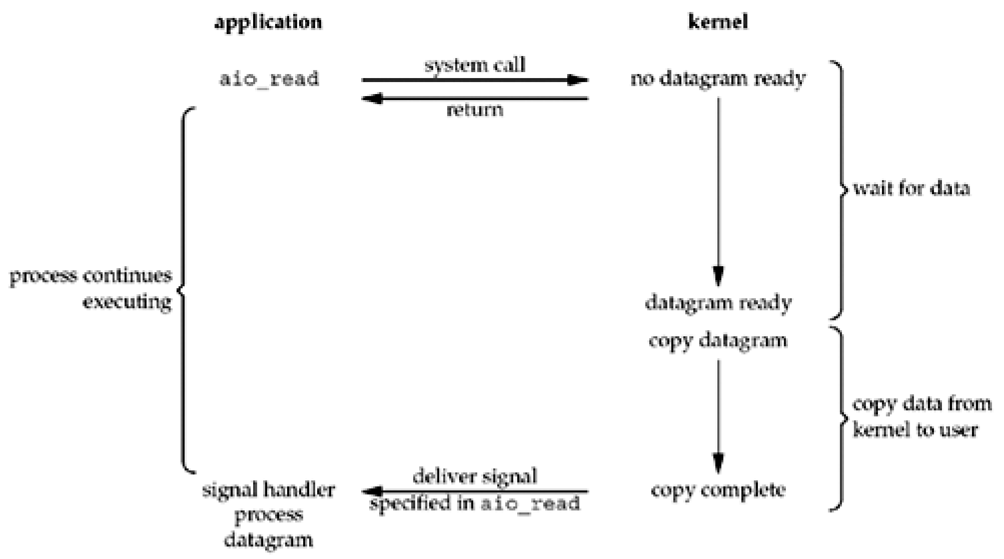
We first enable the socket for signal-driven I/O (as we will describe in Section 25.2) and install a signal handler using the `sigaction` system call. The return from this system call is immediate and our process continues; it is not blocked. When the datagram is ready to be read, the `SIGIO` signal is generated for our process. We can either read the datagram from the signal handler by calling `recvfrom` and then notify the main loop that the data is ready to be processed (this is what we will do in Section 25.3), or we can notify the main loop and let it read the datagram.

Regardless of how we handle the signal, the advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

Asynchronous I/O Model

Asynchronous I/O is defined by the POSIX specification, and various differences in the *real-time* functions that appeared in the various standards which came together to form the current POSIX specification have been reconciled. In general, these functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete. The main difference between this model and the signal-driven I/O model in the previous section is that with signal-driven I/O, the kernel tells us when an I/O operation can be *initiated*, but with asynchronous I/O, the kernel tells us when an I/O operation is *complete*. We show an example in Figure 6.5.

Figure 6.5. Asynchronous I/O model.


application
kernel
aio_read
system call
return
no datagram ready
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
signal handler process datagram
deliver signal specified in aio_read
copy complete

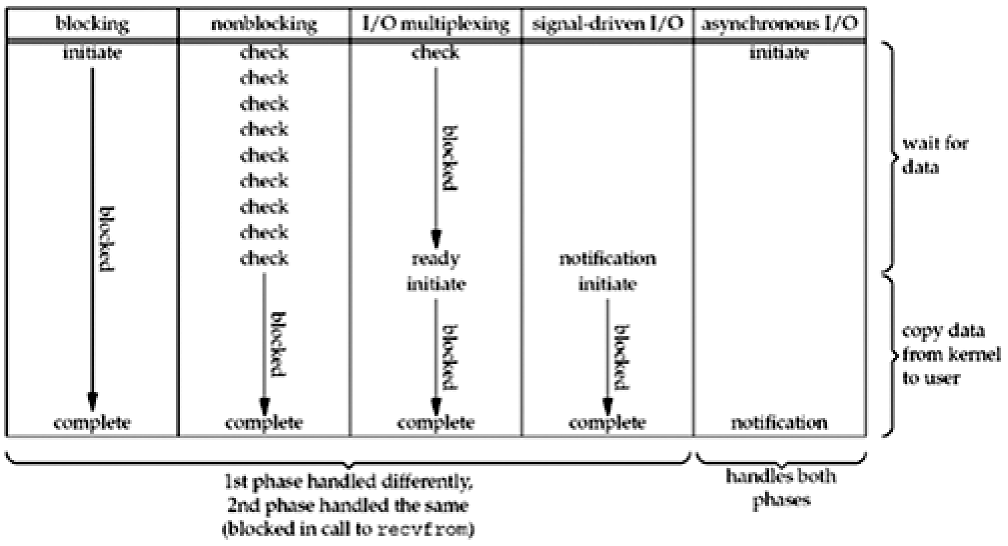
We call `aio_read` (the POSIX asynchronous I/O functions begin with `aio_` or `lio_`) and pass the kernel the descriptor, buffer pointer, buffer size (the same three arguments for `read`), file offset (similar to `lseek`), and how to notify us when the entire operation is complete. This system call returns immediately and our process is not blocked while waiting for the I/O to complete. We assume in this example that we ask the kernel to generate some signal when the operation is complete. This signal is not generated until the data has been copied into our application buffer, which is different from the signal-driven I/O model.

As of this writing, few systems support POSIX asynchronous I/O. We are not certain, for example, if systems will support it for sockets. Our use of it here is as an example to compare against the signal-driven I/O model.

Comparison of the I/O Models

Figure 6.6 is a comparison of the five different I/O models. It shows that the main difference between the first four models is the first phase, as the second phase in the first four models is the same: the process is blocked in a call to `recvfrom` while the data is copied from the kernel to the caller's buffer. Asynchronous I/O, however, handles both phases and is different from the first four.

Figure 6.6. Comparison of the five I/O models.



Synchronous I/O versus Asynchronous I/O

POSIX defines these two terms as follows:

- A *synchronous I/O operation* causes the requesting process to be blocked until that I/O operation completes.
- An *asynchronous I/O operation* does not cause the requesting process to be blocked.

Using these definitions, the first four I/O models☐blocking, nonblocking, I/O multiplexing, and signal-driven I/O☐are all synchronous because the actual I/O operation (`recvfrom`) blocks the process. Only the asynchronous I/O model matches the asynchronous I/O definition.

6.3 'select' Function

This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

As an example, we can call `select` and tell the kernel to return only when:

- Any of the descriptors in the set {1, 4, 5} are ready for reading
- Any of the descriptors in the set {2, 7} are ready for writing
- Any of the descriptors in the set {1, 4} have an exception condition pending
- 10.2 seconds have elapsed

That is, we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets; any descriptor can be tested using `select`.

Berkeley-derived implementations have always allowed I/O multiplexing with any descriptor. SVR3 originally limited I/O multiplexing to descriptors that were STREAMS devices (Chapter 31), but this limitation was removed with SVR4.

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

#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
const struct timeval *timeout);
```

Returns: positive count of ready descriptors, 0 on timeout, ☐1 on error

We start our description of this function with its final argument, which tells the kernel how long to wait for one of the specified descriptors to become ready. A `timeval` structure specifies the number of seconds and microseconds.

```
struct timeval  {
  long   tv_sec;          /* seconds */
  long   tv_usec;         /* microseconds */
};
```

There are three possibilities:

1. Wait forever☐ Return only when one of the specified descriptors is ready for I/O. For this, we specify the *timeout* argument as a null pointer.
2. Wait up to a fixed amount of time☐ Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the `timeval` structure pointed to by the *timeout* argument.
3. Do not wait at all☐ Return immediately after checking the descriptors. This is called *polling*. To specify this, the *timeout* argument must point to a `timeval` structure and the timer value (the number of seconds and microseconds specified by the structure) must be 0.

The wait in the first two scenarios is normally interrupted if the process catches a signal and returns from the signal handler.

Berkeley-derived kernels never automatically restart `select` (p. 527 of TCPv2), while SVR4 will if the `SA_RESTART` flag is specified when the signal handler is installed. This means that for portability, we must be prepared for `select` to return an error of `EINTR` if we are catching signals.

Although the `timeval` structure lets us specify a resolution in microseconds, the actual resolution supported by the kernel is often more coarse. For example, many Unix kernels round the timeout value up to a multiple of 10 ms. There is also a scheduling latency involved, meaning it takes some time after the timer expires before the kernel schedules this process to run.

On some systems, `select` will fail with `EINVAL` if the `tv_sec` field in the timeout is over 100 million seconds. Of course, that's a very large timeout (over three years) and likely not very useful, but the point is that the `timeval` structure can represent values that are not supported by `select`.

The `const` qualifier on the *timeout* argument means it is not modified by `select` on return. For example, if we specify a time limit of 10 seconds, and `select` returns before the timer expires with one or more of the descriptors ready or with an error of `EINTR`, the `timeval` structure is not updated with the number of seconds remaining when the function returns. If we wish to know this value, we must obtain the system time before calling `select`, and then again when it returns, and subtract the two (any robust program will take into account that the system time may be adjusted by either the administrator or by a daemon like `ntpd` occasionally).

Some Linux versions modify the `timeval` structure. Therefore, for portability, assume the `timeval` structure is undefined upon return, and initialize it before each call to `select`. POSIX specifies the `const` qualifier.

The three middle arguments, *readset*, *writeset*, and *exceptset*, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions. There are only two exception conditions currently supported:

1. The arrival of out-of-band data for a socket. We will describe this in more detail in Chapter 24.
2. The presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. We do not talk about pseudo-terminals in this book.

A design problem is how to specify one or more descriptor values for each of these three arguments. `select` uses *descriptor sets*, typically an array of integers, with each bit in each integer corresponding to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All the implementation details are irrelevant to the application and are hidden in the `fd_set` datatype and the following four macros:

```
void FD_ZERO(fd_set *fdset);          /* clear all bits in fdset */
void FD_SET(int fd, fd_set *fdset);   /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset);   /* turn off the bit for fd in fdset */
int FD_ISSET(int fd, fd_set *fdset);  /* is the bit for fd on in fdset ? */
```

We allocate a descriptor set of the `fd_set` datatype, we set and test the bits in the set using these macros, and we can also assign it to another descriptor set across an equals sign (=) in C.

What we are describing, an array of integers using one bit per descriptor, is just one possible way to implement `select`. Nevertheless, it is common to refer to the individual descriptors within a descriptor set as *bits*, as in "turn on the bit for the listening descriptor in the read set."

We will see in Section 6.10 that the `poll` function uses a completely different representation: a variable-length array of structures with one structure per descriptor.

For example, to define a variable of type `fd_set` and then turn on the bits for descriptors 1, 4, and 5, we write

```
fd_set rset;

FD_ZERO(&rset);          /* initialize the set: all bits off */
FD_SET(1, &rset);        /* turn on bit for fd 1 */
FD_SET(4, &rset);        /* turn on bit for fd 4 */
FD_SET(5, &rset);        /* turn on bit for fd 5 */
```

It is important to initialize the set, since unpredictable results can occur if the set is allocated as an automatic variable and not initialized.

Any of the middle three arguments to `select`, *readset*, *writeset*, or *exceptset*, can be specified as a null pointer if we are not interested in that condition. Indeed, if all three pointers are null, then we have a higher precision timer than the normal Unix `sleep` function (which sleeps for multiples of a second). The `poll` function provides similar functionality. Figures C.9 and C.10 of APUE show a `sleep_us` function implemented using both `select` and `poll` that sleeps for multiples of a microsecond.

The *maxfdp1* argument specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested plus one (hence our name of *maxfdp1*). The descriptors 0, 1, 2, up through and including *maxfdp1□1* are tested.

The constant `FD_SETSIZE`, defined by including `<sys/select.h>`, is the number of descriptors in the `fd_set` datatype. Its value is often 1024, but few programs use that many descriptors. The *maxfdp1* argument forces us to calculate the largest descriptor that we are interested in and then tell the kernel this value. For example, given the previous code that turns on the indicators for descriptors 1, 4, and 5, the *maxfdp1* value is 6. The reason it is 6 and not 5 is that we are specifying the number of descriptors, not the largest value, and descriptors start at 0.

The reason this argument exists, along with the burden of calculating its value, is purely for efficiency. Although each `fd_set` has room for many descriptors, typically 1,024, this is much more than the number used by a typical process. The kernel gains efficiency by not copying unneeded portions of the descriptor set between the process and the kernel, and by not testing bits that are always 0 (Section 16.13 of TCPv2).

`select` modifies the descriptor sets pointed to by the *readset*, *writeset*, and *exceptset* pointers. These three arguments are value-result arguments. When we call the function, we specify the values of the descriptors that we are interested in, and on return, the result indicates which descriptors are ready. We use the `FD_ISSET` macro on return to test a specific descriptor in an `fd_set` structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this, we turn on all the bits in which we are interested in all the descriptor sets each time we call `select`.

The two most common programming errors when using `select` are to forget to add one to the largest descriptor number and to forget that the descriptor sets are value-result arguments. The second error results in `select` being called with a bit set to 0 in the descriptor set, when we think that bit is 1.

The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of □1 indicates an error (which can happen, for example, if the function is interrupted by a caught signal).

Early releases of SVR4 had a bug in their implementation of `select:` If the same bit was on in multiple sets, say a descriptor was ready for both reading and writing, it was counted only once. Current releases fix this bug.

Under What Conditions Is a Descriptor Ready?

We have been talking about waiting for a descriptor to become ready for I/O (reading or writing) or to have an exception condition pending on it (out-of-band data). While readability and writability are obvious for descriptors such as regular files, we must be more specific about the conditions that cause `select` to return "ready" for sockets (Figure 16.52 of TCPv2).

1. A socket is ready for reading if any of the following four conditions is true:
    a. The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. A read operation on the socket will not block and will return a value greater than 0 (i.e., the data that is ready to be read). We can set this low-water mark using the `SO_RCVLOWAT` socket option. It defaults to 1 for TCP and UDP sockets.
    b. The read half of the connection is closed (i.e., a TCP connection that has received a FIN). A read operation on the socket will not block and will return 0 (i.e., EOF).
    c. The socket is a listening socket and the number of completed connections is nonzero. An `accept` on the listening socket will normally not block, although we will describe a timing condition in Section 16.6 under which the `accept` can block.
    d. A socket error is pending. A read operation on the socket will not block and will return an error (☐1) with `errno` set to the specific error condition. These *pending errors* can also be fetched and cleared by calling `getsockopt` and specifying the `SO_ERROR` socket option.
2. A socket is ready for writing if any of the following four conditions is true:
    a. The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer *and* either: (i) the socket is connected, or (ii) the socket does not require a connection (e.g., UDP). This means that if we set the socket to nonblocking (Chapter 16), a write operation will not block and will return a positive value (e.g., the number of bytes accepted by the transport layer). We can set this low-water mark using the `SO_SNDLOWAT` socket option. This low-water mark normally defaults to 2048 for TCP and UDP sockets.
    b. The write half of the connection is closed. A write operation on the socket will generate `SIGPIPE` (Section 5.12).
    c. A socket using a non-blocking `connect` has completed the connection, or the `connect` has failed.
    d. A socket error is pending. A write operation on the socket will not block and will return an error (☐1) with `errno` set to the specific error condition. These *pending errors* can also be fetched and cleared by calling `getsockopt` with the `SO_ERROR` socket option.
3. A socket has an exception condition pending if there is out-of-band data for the socket or the socket is still at the out-of-band mark. (We will describe out-of-band data in Chapter 24.)

> Our definitions of "readable" and "writable" are taken directly from the kernel's `soreadable` and `sowriteable` macros on pp. 530□531 of TCPv2. Similarly, our definition of the "exception condition" for a socket is from the `soo_select` function on these same pages.

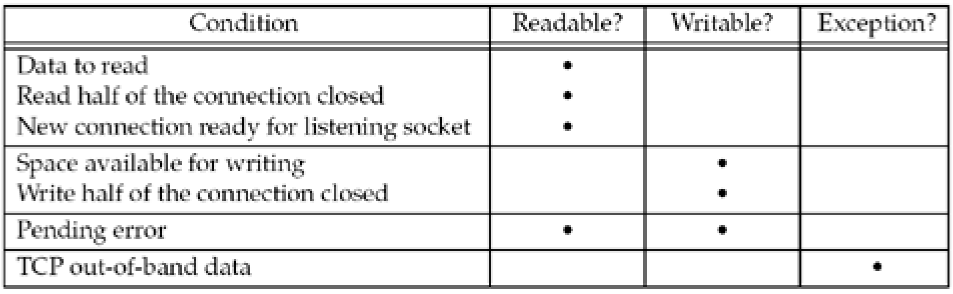
Notice that when an error occurs on a socket, it is marked as both readable and writable by `select`.

The purpose of the receive and send low-water marks is to give the application control over how much data must be available for reading or how much space must be available for writing before `select` returns a readable or writable status. For example, if we know that our application has nothing productive to do unless at least 64 bytes of data are present, we can set the receive low-water mark to 64 to prevent `select` from waking us up if less than 64 bytes are ready for reading.

As long as the send low-water mark for a UDP socket is less than the send buffer size (which should always be the default relationship), the UDP socket is always writable, since a connection is not required.

Figure 6.7 summarizes the conditions just described that cause a socket to be ready for `select`.

Figure 6.7. Summary of conditions that cause a socket to be ready for select.



Condition	Readable?	Writable?	Exception?
Data to read	•		
Read half of the connection closed	•		
New connection ready for listening socket	•		
Space available for writing		•	
Write half of the connection closed		•	
Pending error	•	•	
TCP out-of-band data			•

Maximum Number of Descriptors for select

We said earlier that most applications do not use lots of descriptors. It is rare, for example, to find an application that uses hundreds of descriptors. But, such applications do exist, and they often use `select` to multiplex the descriptors. When `select` was originally designed, the OS normally had an upper limit on the maximum number of descriptors per process (the 4.2BSD limit was 31), and `select` just used this same limit. But, current versions of Unix allow for a virtually unlimited number of descriptors per process (often limited only by the amount of memory and any administrative limits), so the question is: How does this affect `select?`

Many implementations have declarations similar to the following, which are taken from the 4.4BSD `<sys/types.h>` header:

```
/*
 * Select uses bitmasks of file descriptors in longs. These macros
 * manipulate such bit fields (the filesystem macros use chars).
 * FD_SETSIZE may be defined by the user, but the default here should
 * be enough for most uses.
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE     256
#endif
```

This makes us think that we can just `#define FD_SETSIZE` to some larger value before including this header to increase the size of the descriptor sets used by `select`. Unfortunately, this normally does not work.

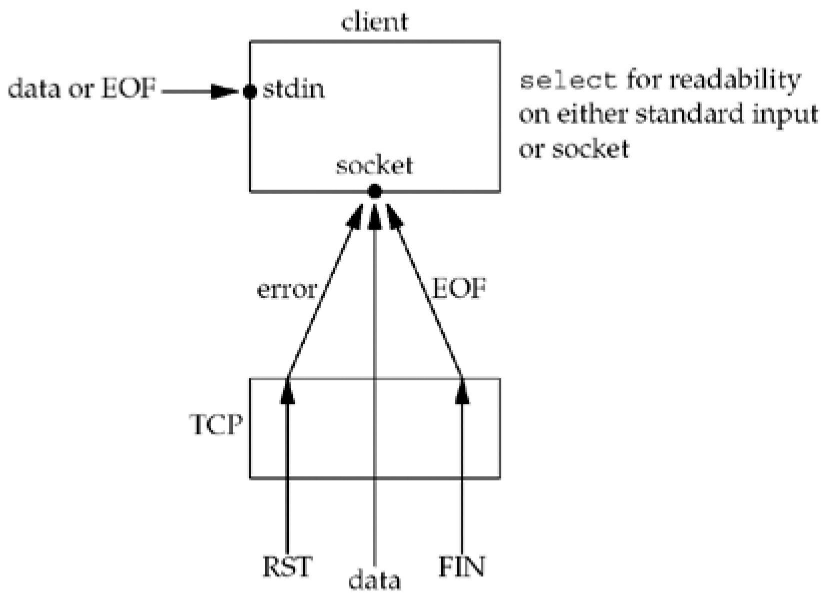
To see what is wrong, notice that Figure 16.53 of TCPv2 declares three descriptor sets within the kernel and also uses the kernel's definition of `FD_SETSIZE` as the upper limit. The only way to increase the size of the descriptor sets is to increase the value of `FD_SETSIZE` and then recompile the kernel. Changing the value without recompiling the kernel is inadequate.

Some vendors are changing their implementation of `select` to allow the process to define `FD_SETSIZE` to a larger value than the default. BSD/OS has changed the kernel implementation to allow larger descriptor sets, and it also provides four new `FD_`*xxx* macros to dynamically allocate and manipulate these larger sets. From a portability standpoint, however, beware of using large descriptor sets.

6.4 'str_cli' Function (Revisited)

We can now rewrite our `str_cli` function from Section 5.5, this time using `select`, so we are notified as soon as the server process terminates. The problem with that earlier version was that we could be blocked in the call to `fgets` when something happened on the socket. Our new version blocks in a call to `select` instead, waiting for either standard input or the socket to be readable. Figure 6.8 shows the various conditions that are handled by our call to `select`.

Figure 6.8. Conditions handled by select in str_cli.



Three conditions are handled with the socket:

1. If the peer TCP sends data, the socket becomes readable and `read` returns greater than 0 (i.e., the number of bytes of data).
2. If the peer TCP sends a FIN (the peer process terminates), the socket becomes readable and `read` returns 0 (EOF).
3. If the peer TCP sends an RST (the peer host has crashed and rebooted), the socket becomes readable, `read` returns □1, and `errno` contains the specific error code.

Figure 6.9 shows the source code for this new version.

Figure 6.9 Implementation of str_cli function using select (improved in Figure 6.13).

select/strcliselect01.c

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int     maxfdp1;
 6     fd_set  rset;
 7     char    sendline[MAXLINE], recvline[MAXLINE];

 8     FD_ZERO(&rset);
 9     for ( ; ; ) {
10         FD_SET(fileno(fp), &rset);
11         FD_SET(sockfd, &rset);
12         maxfdp1 = max(fileno(fp), sockfd) + 1;
13         Select(maxfdp1, &rset, NULL, NULL, NULL);

14         if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
15             if (Readline(sockfd, recvline, MAXLINE) == 0)
16                 err_quit("str_cli: server terminated prematurely");
17             Fputs(recvline, stdout);
18         }

19         if (FD_ISSET(fileno(fp), &rset)) { /* input is readable */
20             if (Fgets(sendline, MAXLINE, fp) == NULL)
21                 return;         /* all done */
22             Writen(sockfd, sendline, strlen(sendline));
23         }
24     }
25 }
```

Call select

8□13 We only need one descriptor set□to check for readability. This set is initialized by `FD_ZERO` and then two bits are turned on using `FD_SET:` the bit corresponding to the standard I/O file pointer, `fp`, and the bit corresponding to the socket, `sockfd`. The function `fileno` converts a standard I/O file pointer into its corresponding descriptor. `select` (and `poll`) work only with descriptors.

`select` is called after calculating the maximum of the two descriptors. In the call, the write-set pointer and the exception-set pointer are both null pointers. The final argument (the time limit) is also a null pointer since we want the call to block until something is ready.

Handle readable socket

14□18 If, on return from `select`, the socket is readable, the echoed line is read with `readline` and output by `fputs`.

Handle readable input

19□23 If the standard input is readable, a line is read by `fgets` and written to the socket using `writen`.

Notice that the same four I/O functions are used as in Figure 5.5, `fgets`, `writen`, `readline`, and `fputs`, but the order of flow within the function has changed. Instead of the function flow being driven by the call to `fgets`, it is now driven by the call to `select`. With only a few additional lines of code in Figure 6.9, compared to Figure 5.5, we have added greatly to the robustness of our client.

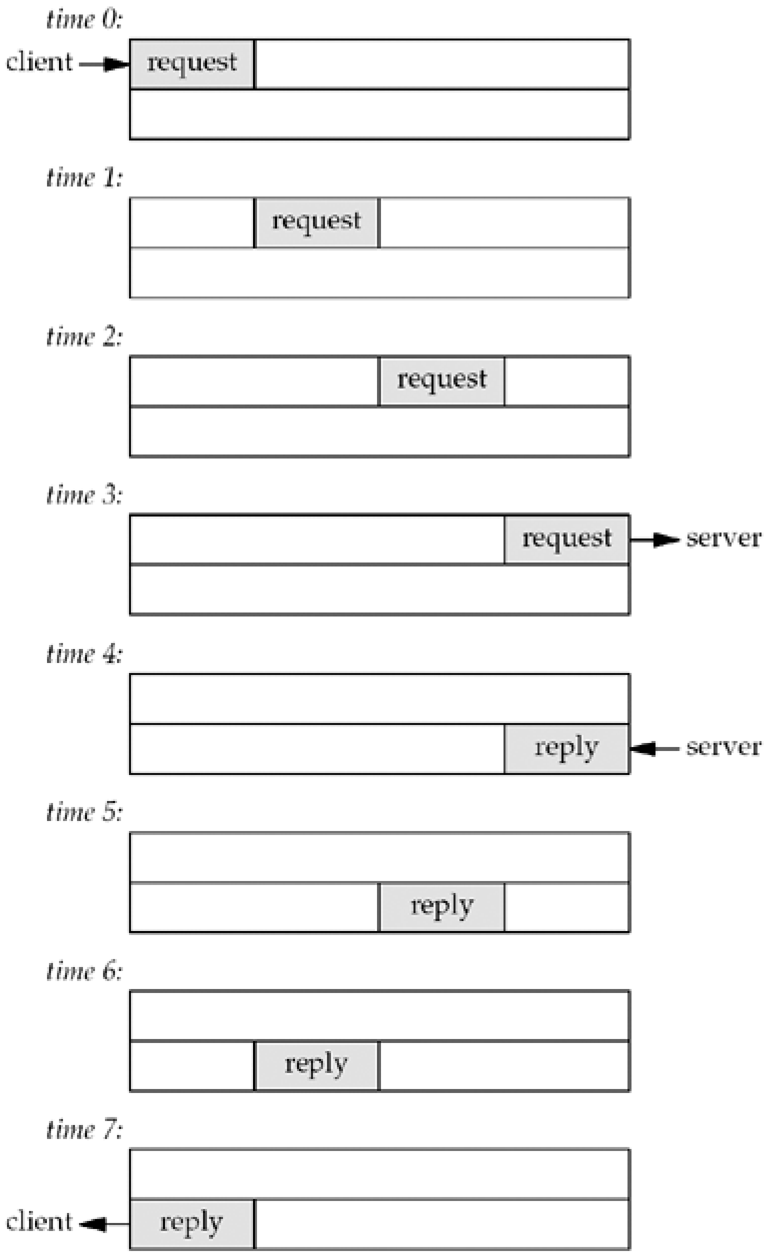
6.5 Batch Input and Buffering

Unfortunately, our `str_cli` function is still not correct. First, let's go back to our original version, Figure 5.5. It operates in a stop-and-wait mode, which is fine for interactive use: It sends a line to the server and then waits for the reply. This amount of time is one RTT plus the server's processing time (which is close to 0 for a simple echo server). We can therefore estimate how long it will take for a given number of lines to be echoed if we know the RTT between the client and server.

The `ping` program is an easy way to measure RTTs. If we run `ping` to the host `connix.com` from our host `solaris`, the average RTT over 30 measurements is 175 ms. Page 89 of TCPv1 shows that these `ping` measurements are for an IP datagram whose length is 84 bytes. If we take the first 2,000 lines of the Solaris `termcap` file, the resulting file size is 98,349 bytes, for an average of 49 bytes per line. If we add the sizes of the IP header (20 bytes) and the TCP header (20), the average TCP segment will be about 89 bytes, nearly the same as the `ping` packet sizes. We can therefore estimate that the total clock time will be around 350 seconds for 2,000 lines (2,000x0.175*sec*). If we run our TCP echo client from Chapter 5, the actual time is about 354 seconds, which is very close to our estimate.

If we consider the network between the client and server as a full-duplex pipe, with requests going from the client to the server and replies in the reverse direction, then Figure 6.10 shows our stop-and-wait mode.

Figure 6.10. Time line of stop-and-wait mode: interactive input.

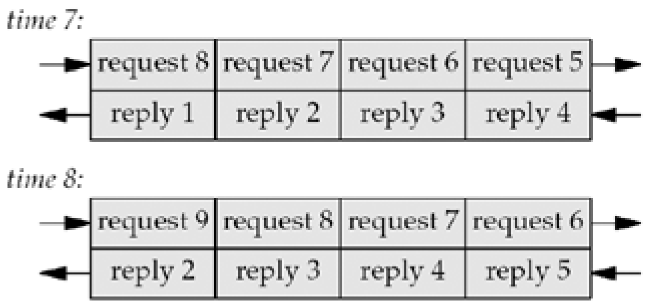


A request is sent by the client at time 0 and we assume an RTT of 8 units of time. The reply sent at time 4 is received at time 7. We also assume that there is no server processing time and that the size of the request is the same as the reply. We show only the data packets between the client and server, ignoring the TCP acknowledgments that are also going across the network.

Since there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full-duplex, in this example, we are only using one-eighth of the pipe's capacity. This stop-and-wait mode is fine for interactive input, but since our client reads from standard input and writes to standard output, and since it is trivial under the Unix shells to redirect the input and output, we can easily run our client in a batch mode. When we redirect the input and output, however, the resulting output file is always smaller than the input file (and they should be identical for an echo server).

To see what's happening, realize that in a batch mode, we can keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate. This leads to the full pipe at time 7, as shown in Figure 6.11.

Figure 6.11. Filling the pipe between the client and server: batch mode.



Here we assume that after sending the first request, we immediately send another, and then another. We also assume that we can keep sending requests as fast as the network can accept them, along with processing replies as fast as the network supplies them.

There are numerous subtleties dealing with TCP's bulk data flow that we are ignoring here, such as its slow-start algorithm, which limits the rate at which data is sent on a new or idle connection, and the returning ACKs. These are all covered in Chapter 20 of TCPv1.

To see the problem with our revised `str_cli` function in Figure 6.9, assume that the input file contains only nine lines. The last line is sent at time 8, as shown in Figure 6.11. But we cannot close the connection after writing this request because there are still other requests and replies in the pipe. The cause of the problem is our handling of an EOF on input: The function returns to the `main` function, which then terminates. But in a batch mode, an EOF on input does not imply that we have finished reading from the socket; there might still be requests on the way to the server, or replies on the way back from the server.

What we need is a way to close one-half of the TCP connection. That is, we want to send a FIN to the server, telling it we have finished sending data, but leave the socket descriptor open for reading. This is done with the `shutdown` function, which is described in the next section.

In general, buffering for performance adds complexity to a network application, and the code in Figure 6.9 suffers from this complexity. Consider the case when several lines of input are available from the standard input. `select` will cause the code at line 20 to read the input using `fgets` and that, in turn, will read the available lines into a buffer used by stdio. But, `fgets` only returns a single line and leaves any remaining data sitting in the stdio buffer. The code at line 22 of Figure 6.9 writes that single line to the server and then `select` is called again to wait for more work, even if there are additional lines to consume in the stdio buffer. The reason for this is that `select` knows nothing of the buffers used by stdio☐it will only show readability from the viewpoint of the `read` system call, not calls like `fgets`. For this reason, mixing stdio and `select` is considered very error-prone and should only be done with great care.

The same problem exists with the call to `readline` in the example in Figure 6.9. Instead of data being hidden from `select` in a stdio buffer, it is hidden in `readline`'s buffer. Recall that in Section 3.9 we provided a function that gives visibility into `readline`'s buffer, so one possible solution is to modify our code to use that function *before* calling `select` to see if data has already been read but not consumed. But again, the complexity grows out of hand quickly when we have to handle the case where the `readline` buffer contains a partial line (meaning we still need to read more) as well as when it contains one or more complete lines (which we can consume).

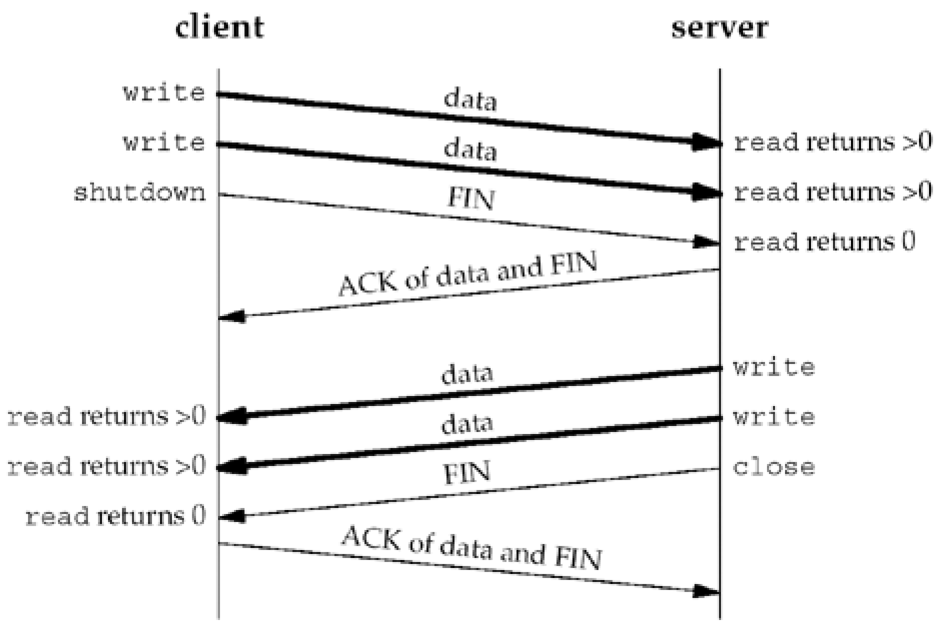
We will address these buffering concerns in the improved version of `str_cli` shown in Section 6.7.

6.6 'shutdown' Function

The normal way to terminate a network connection is to call the `close` function. But, there are two limitations with `close` that can be avoided with `shutdown:`

1. `close` decrements the descriptor's reference count and closes the socket only if the count reaches 0. We talked about this in Section 4.8. With `shutdown`, we can initiate TCP's normal connection termination sequence (the four segments beginning with a FIN in Figure 2.5), regardless of the reference count.
2. `close` terminates both directions of data transfer, reading and writing. Since a TCP connection is full-duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us. This is the scenario we encountered in the previous section with batch input to our `str_cli` function. Figure 6.12 shows the typical function calls in this scenario.

Figure 6.12. Calling shutdown to close half of a TCP connection.



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

int shutdown(int sockfd, int howto);
```

Returns: 0 if OK, □1 on error

The action of the function depends on the value of the *howto* argument.

`SHUT_RD` The read half of the connection is closed□ No more data can be received on the socket and any data currently in the socket receive buffer is discarded. The process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP socket is acknowledged and then silently discarded.

By default, everything written to a routing socket (Chapter 18) loops back as possible input to all routing sockets on the host. Some programs call `shutdown` with a second argument of `SHUT_RD` to prevent the loopback copy. An alternative way to prevent this loopback copy is to clear the `SO_USELOOPBACK` socket option.

`SHUT_WR` The write half of the connection is closed□ In the case of TCP, this is called a *half-close* (Section 18.5 of TCPv1). Any data currently in the socket send buffer will be sent, followed by TCP's normal connection termination sequence. As we mentioned earlier, this closing of the write half is done regardless of whether or not the socket descriptor's reference count is currently greater than 0. The process can no longer issue any of the write functions on the socket.

`SHUT_RDWR` The read half and the write half of the connection are both closed□ This is equivalent to calling `shutdown` twice: first with `SHUT_RD` and then with

SHUT_RD The read half of the connection is closed☐ No more data can be received on the socket and any data currently in the socket receive buffer is discarded. The process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP socket is acknowledged and then silently discarded.

By default, everything written to a routing socket (Chapter 18) loops back as possible input to all routing sockets on the host. Some programs call shutdown with a second argument of SHUT_RD to prevent the loopback copy. An alternative way to prevent this loopback copy is to clear the SO_USELOOPBACK socket option.

SHUT_WR.

Figure 7.12 will summarize the different possibilities available to the process by calling shutdown and close. The operation of close depends on the value of the SO_LINGER socket option.

The three SHUT_*xxx* names are defined by the POSIX specification. Typical values for the *howto* argument that you will encounter will be 0 (close the read half), 1 (close the write half), and 2 (close the read half and the write half).

6.7 'str_cli' Function (Revisited Again)

Figure 6.13 shows our revised (and correct) version of the str_cli function. This version uses select and shutdown. The former notifies us as soon as the server closes its end of the connection and the latter lets us handle batch input correctly. This version also does away with line-centric code and operates instead on buffers, eliminating the complexity concerns raised in Section 6.5.

Figure 6.13 str_cli function using select that handles EOF correctly.

select/strcliselect02.c

```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int     maxfdp1, stdineof;
    fd_set  rset;
    char    buf[MAXLINE];
```

```
 8     int     n;

 9     stdineof = 0;
10     FD_ZERO(&rset);
11     for ( ; ; ) {
12         if (stdineof == 0)
13             FD_SET(fileno(fp), &rset);
14         FD_SET(sockfd, &rset);
15         maxfdp1 = max(fileno(fp), sockfd) + 1;
16         Select(maxfdp1, &rset, NULL, NULL, NULL);

17         if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
18             if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
19                 if (stdineof == 1)
20                     return;       /* normal termination */
21                 else
22                     err_quit("str_cli: server terminated prematurely");
23                 }
24                 Write(fileno(stdout), buf, n);
25         }
26         if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
27             if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
28                 stdineof = 1;
29                 Shutdown(sockfd, SHUT_WR);  /* send FIN */
30                 FD_CLR(fileno(fp), &rset);
31                 continue;
32             }
33             Writen(sockfd, buf, n);
34         }
35     }
36 }
```

5□8 `stdineof` is a new flag that is initialized to 0. As long as this flag is 0, each time around the main loop, we `select` on standard input for readability.

17□25 When we read the EOF on the socket, if we have already encountered an EOF on standard input, this is normal termination and the function returns. But if we have not yet encountered an EOF on standard input, the server process has prematurely terminated. We now call `read` and `write` to operate on buffers instead of lines and allow `select` to work for us as expected.

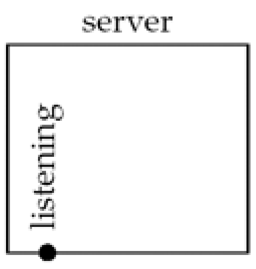
26□34 When we encounter the EOF on standard input, our new flag, `stdineof`, is set and we call `shutdown` with a second argument of `SHUT_WR` to send the FIN. Here also, we've changed to operating on buffers instead of lines, using `read` and `writen`.

We are not finished with our `str_cli` function. We will develop a version using nonblocking I/O in Section 16.2 and a version using threads in Section 26.3.

6.8 TCP Echo Server (Revisited)

We can revisit our TCP echo server from Sections 5.2 and 5.3 and rewrite the server as a single process that uses `select` to handle any number of clients, instead of `forking` one child per client. Before showing the code, let's look at the data structures that we will use to keep track of the clients. Figure 6.14 shows the state of the server before the first client has established a connection.

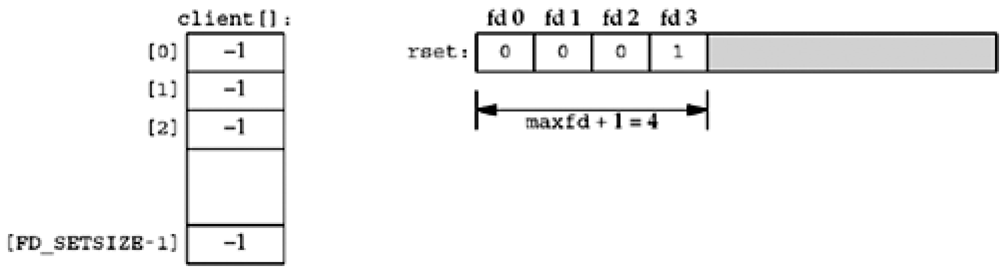
Figure 6.14. TCP server before first client has established a connection.



The server has a single listening descriptor, which we show as a bullet.

The server maintains only a read descriptor set, which we show in Figure 6.15. We assume that the server is started in the foreground, so descriptors 0, 1, and 2 are set to standard input, output, and error. Therefore, the first available descriptor for the listening socket is 3. We also show an array of integers named `client` that contains the connected socket descriptor for each client. All elements in this array are initialized to □1.

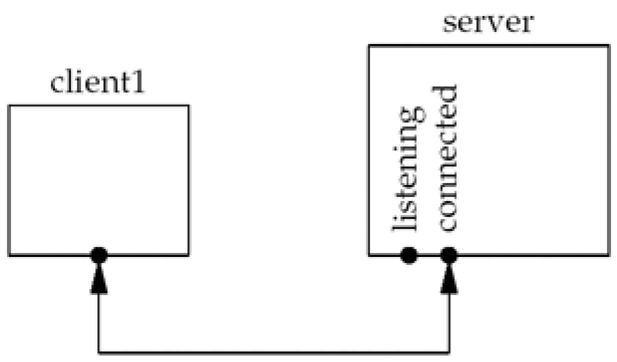
Figure 6.15. Data structures for TCP server with just a listening socket.



The only nonzero entry in the descriptor set is the entry for the listening sockets and the first argument to `select` will be 4.

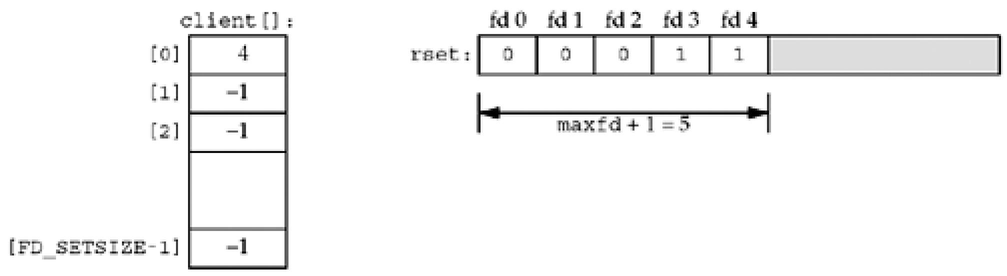
When the first client establishes a connection with our server, the listening descriptor becomes readable and our server calls `accept`. The new connected descriptor returned by `accept` will be 4, given the assumptions of this example. Figure 6.16 shows the connection from the client to the server.

Figure 6.16. TCP server after first client establishes connection.



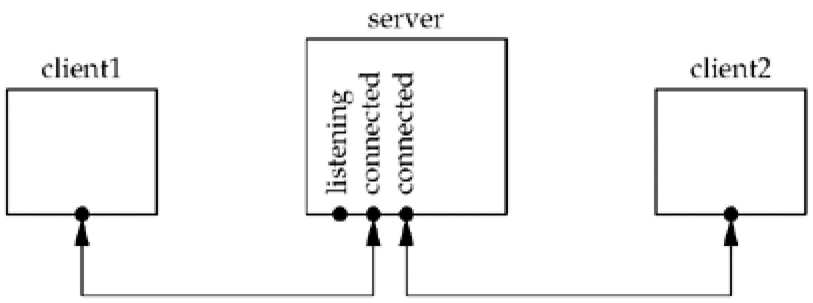
From this point on, our server must remember the new connected socket in its `client` array, and the connected socket must be added to the descriptor set. These updated data structures are shown in Figure 6.17.

Figure 6.17. Data structures after first client connection is established.



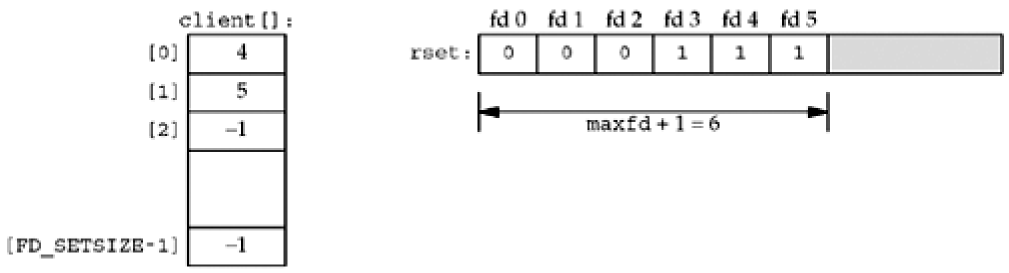
Sometime later a second client establishes a connection and we have the scenario shown in Figure 6.18.

Figure 6.18. TCP server after second client connection is established.



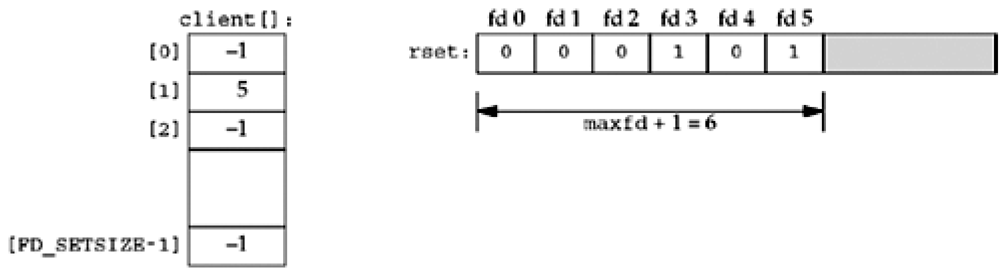
The new connected socket (which we assume is 5) must be remembered, giving the data structures shown in Figure 6.19.

Figure 6.19. Data structures after second client connection is established.



Next, we assume the first client terminates its connection. The client TCP sends a FIN, which makes descriptor 4 in the server readable. When our server reads this connected socket, `read` returns 0. We then close this socket and update our data structures accordingly. The value of `client [0]` is set to ☐1 and descriptor 4 in the descriptor set is set to 0. This is shown in Figure 6.20. Notice that the value of `maxfd` does not change.

Figure 6.20. Data structures after first client terminates its connection.



In summary, as clients arrive, we record their connected socket descriptor in the first available entry in the `client` array (i.e., the first entry with a value of ☐1). We must also add the connected socket to the read descriptor set. The variable `maxi` is the highest index in the `client` array that is currently in use and the variable `maxfd` (plus one) is the current value of the first argument to `select`. The only limit on the number of clients that this server can handle is the minimum of the two values `FD_SETSIZE` and the maximum number of descriptors allowed for this process by the kernel (which we talked about at the end of Section 6.3).

Figure 6.21 shows the first half of this version of the server.

Figure 6.21 TCP server using a single process and select: initialization.

tcpcliserv/tcpservselect01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, maxi, maxfd, listenfd, connfd, sockfd;
 6     int     nready, client[FD_SETSIZE];
 7     ssize_t n;
 8     fd_set  rset, allset;
 9     char    buf[MAXLINE];
10     socklen_t  clilen;
11     struct sockaddr_in cliaddr, servaddr;

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

13     bzero(&servaddr, sizeof(servaddr));
14     servaddr.sin_family = AF_INET;
15     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16     servaddr.sin_port = htons(SERV_PORT);

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

18     Listen(listenfd, LISTENQ);

19     maxfd = listenfd;           /* initialize */
20     maxi = -1;                  /* index into client[] array */
21     for (i = 0; i < FD_SETSIZE;  i++)
22         client[i] = -1;         /* -1 indicates available entry */
23     FD_ZERO(&allset);
24     FD_SET(listenfd, &allset);
```

Create listening socket and initialize for select

12□24 The steps to create the listening socket are the same as seen earlier: `socket`, `bind`, and `listen`. We initialize our data structures assuming that the only descriptor that we will `select` on initially is the listening socket.

The last half of the function is shown in Figure 6.22

Figure 6.22 TCP server using a single process and select loop.

tcpcliserv/tcpservselect01.c

```
25      for ( ; ; ) {
26          rset = allset;          /* structure assignment */
```

```
        nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &rset)) {      /* new client connection */
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

            for (i = 0; i < FD_SETSIZE; i++)
                if (client[i] < 0) {
                    client[i] = connfd; /* save descriptor */
                    break;
                }
            if (i == FD_SETSIZE)
                err_quit("too many clients");
            FD_SET(connfd, &allset);      /* add new descriptor to set */
            if (connfd > maxfd)
                maxfd = connfd; /* for select */
            if (i > maxi)
                maxi = i;         /* max index in client[] array */

            if (--nready <= 0)
                continue;         /* no more readable descriptors */
        }
        for (i = 0; i <= maxi; i++) {      /* check all clients for data */
            if ( (sockfd = client[i]) < 0)
                continue;
            if (FD_ISSET(sockfd, &rset)) {
                if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
                        /* connection closed by client */
                    Close(sockfd);
                    FD_CLR(sockfd, &allset);
                    client[i] = -1;
                } else
                    Writen(sockfd, buf, n);

                if (--nready <= 0)
                    break;        /* no more readable descriptors */
            }
        }
    }
}
```

Block in select

26-27 `select` waits for something to happen: either the establishment of a new client connection or the arrival of data, a FIN, or an RST on an existing connection.

accept new connections

28-45 If the listening socket is readable, a new connection has been established. We call `accept` and update our data structures accordingly. We use the first unused entry in the `client` array to record the connected socket. The number of ready descriptors is decremented, and if it is 0, we can avoid the next `for` loop. This lets us use the return value from `select` to avoid checking descriptors that are not ready.

Check existing connections

46-60 A test is made for each existing client connection as to whether or not its descriptor is in the descriptor set returned by `select`. If so, a line is read from the client and echoed back to the client. If the client closes the connection, `read` returns 0 and we update our data structures accordingly.

We never decrement the value of `maxi`, but we could check for this possibility each time a client closes its connection.

This server is more complicated than the one shown in Figures 5.2 and 5.3, but it avoids all the overhead of creating a new process for each client and it is a nice example of `select`. Nevertheless, in Section 16.6, we will describe a problem with this server that is easily fixed by making the listening socket nonblocking and then checking for, and ignoring, a few errors from `accept`.

Denial-of-Service Attacks

Unfortunately, there is a problem with the server that we just showed. Consider what happens if a malicious client connects to the server, sends one byte of data (other than a newline), and then goes to sleep. The server will call `read`, which will read the single byte of data from the client and then block in the next call to `read`, waiting for more data from this client. The server is then blocked ("hung" may be a better term) by this one client and will not service any other clients (either new client connections or existing clients' data) until the malicious client either sends a newline or terminates.

The basic concept here is that when a server is handling multiple clients, the server can *never* block in a function call related to a single client. Doing so can hang the server and deny service to all other clients. This is called a *denial-of-service* attack. It does something to the server that prevents it from servicing other legitimate clients. Possible solutions are to: (i) use nonblocking I/O (Chapter 16), (ii) have each client serviced by a separate thread of control (e.g., either spawn a process or a thread to service each client), or (iii) place a timeout on the I/O operations (Section 14.2).

6.9 'pselect' Function

The `pselect` function was invented by POSIX and is now supported by many of the Unix variants.

`#include <sys/select.h>`
`#include <signal.h>`
`#include <time.h>`
`int pselect (int` *maxfdp1*, `fd_set *`*readset*, `fd_set *`*writeset*, `fd_set *`*exceptset*, `const struct timespec *`*timeout*, `const sigset_t *`*sigmask*`)`;
Returns: count of ready descriptors, 0 on timeout, □1 on error

`pselect` contains two changes from the normal `select` function:

1. `pselect` uses the `timespec` structure, another POSIX invention, instead of the `timeval` structure.

```
struct timespec {
  time_t tv_sec;       /* seconds */
  long   tv_nsec;      /* nanoseconds */
};
```

   The difference in these two structures is with the second member: The `tv_nsec` member of the newer structure specifies nanoseconds, whereas the `tv_usec` member of the older structure specifies microseconds.

10. `pselect` adds a sixth argument: a pointer to a signal mask. This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call `pselect`, telling it to reset the signal mask.

With regard to the second point, consider the following example (discussed on pp. 308□309 of APUE). Our program's signal handler for `SIGINT` just sets the global `intr_flag` and returns. If our process is blocked in a call to `select`, the return from the signal handler causes the function to return with `errno` set to `EINTR`. But when `select` is called, the code looks like the following:

```
if (intr_flag)
    handle_intr();       /* handle the signal */
if ( (nready = select( ... )) < 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

The problem is that between the test of `intr_flag` and the call to `select`, if the signal occurs, it will be lost if `select` blocks forever. With `pselect`, we can now code this example reliably as

```
sigset_t newmask, oldmask, zeromask;

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */
if (intr_flag)
    handle_intr();     /* handle the signal */
if ( (nready = pselect ( ... , &zeromask)) < 0) {
    if (errno == EINTR)  {
        if (intr_flag)
            handle_intr ();
    }
    ...
}
```

Before testing the `intr_flag` variable, we block `SIGINT`. When `pselect` is called, it replaces the signal mask of the process with an empty set (i.e., `zeromask`) and then checks the descriptors, possibly going to sleep. But when `pselect` returns, the signal mask of the process is reset to its value before `pselect` was called (i.e., `SIGINT` is blocked).

We will say more about `pselect` and show an example of it in Section 20.5. We will use `pselect` in Figure 20.7 and show a simple, albeit incorrect, implementation of `pselect` in Figure 20.8.

There is one other slight difference between the two `select` functions. The first member of the `timeval` structure is a signed long integer, while the first member of the `timespec` structure is a `time_t`. The signed long in the former should also be a `time_t`, but was not changed retroactively to avoid breaking existing code. The brand new function, however, could make this change.

6.10 'poll' Function

The `poll` function originated with SVR3 and was originally limited to STREAMS devices (Chapter 31). SVR4 removed this limitation, allowing `poll` to work with any descriptor. `poll` provides functionality that is similar to `select`, but `poll` provides additional information when dealing with STREAMS devices.

```
#include <poll.h>
```

```
int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);
```

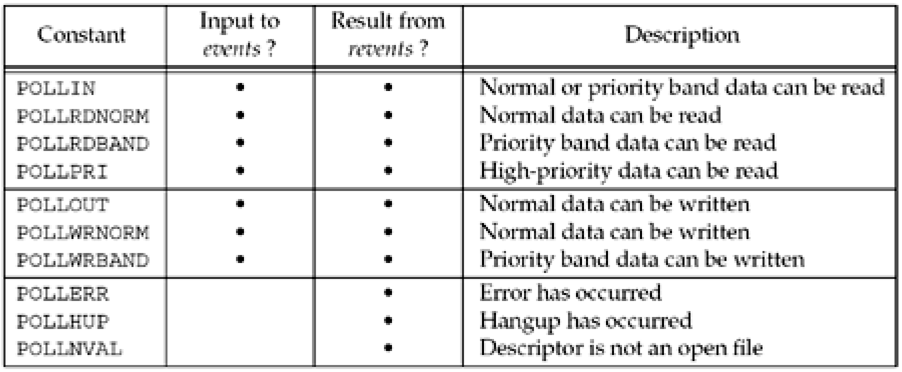
Returns: count of ready descriptors, 0 on timeout, □1 on error

The first argument is a pointer to the first element of an array of structures. Each element of the array is a `pollfd` structure that specifies the conditions to be tested for a given descriptor, `fd`.

```
struct pollfd {
  int     fd;       /* descriptor to check */
  short   events;   /* events of interest on fd */
  short   revents;  /* events that occurred on fd */
};
```

The conditions to be tested are specified by the `events` member, and the function returns the status for that descriptor in the corresponding `revents` member. (Having two variables per descriptor, one a value and one a result, avoids value-result arguments. Recall that the middle three arguments for `select` are value-result.) Each of these two members is composed of one or more bits that specify a certain condition. Figure 6.23 shows the constants used to specify the `events` flag and to test the `revents` flag against.

Figure 6.23. Input *events* and returned *revents* for poll.



Constant	Input to *events* ?	Result from *revents* ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

We have divided this figure into three sections: The first four constants deal with input, the next three deal with output, and the final three deal with errors. Notice that the final three cannot be set in `events`, but are always returned in `revents` when the corresponding condition exists.

There are three classes of data identified by `poll:` *normal*, *priority band*, and *high-priority*. These terms come from the STREAMS-based implementations (Figure 31.5).

`POLLIN` can be defined as the logical OR of `POLLRDNORM` and `POLLRDBAND`. The `POLLIN` constant exists from SVR3 implementations that predated the priority bands in SVR4, so the constant remains for backward compatibility. Similarly, `POLLOUT` is equivalent to `POLLWRNORM`, with the former predating the latter.

With regard to TCP and UDP sockets, the following conditions cause `poll` to return the specified *revent*. Unfortunately, POSIX leaves many holes (i.e., optional ways to return the same condition) in its definition of `poll`.

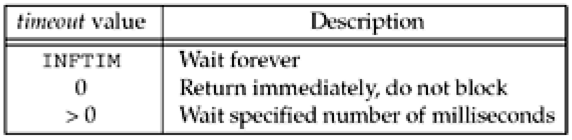
- All regular TCP data and all UDP data is considered normal.
- TCP's out-of-band data (Chapter 24) is considered priority band.
- When the read half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
- The presence of an error for a TCP connection can be considered either normal data or an error (`POLLERR`). In either case, a subsequent `read` will return □1 with `errno` set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
- The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
- The completion of a nonblocking `connect` is considered to make a socket writable.

The number of elements in the array of structures is specified by the *nfds* argument.

Historically, this argument has been an `unsigned long`, which seems excessive. An `unsigned int` would be adequate. Unix 98 defines a new datatype for this argument: `nfds_t`.

The *timeout* argument specifies how long the function is to wait before returning. A positive value specifies the number of milliseconds to wait. Figure 6.24 shows the possible values for the *timeout* argument.

Figure 6.24. *timeout* values for poll.



timeout value	Description
`INFTIM`	Wait forever
0	Return immediately, do not block
> 0	Wait specified number of milliseconds

The constant `INFTIM` is defined to be a negative value. If the system does not provide a timer with millisecond accuracy, the value is rounded up to the nearest supported value.

The POSIX specification requires that `INFTIM` be defined by including `<poll.h>`, but many systems still define it in `<sys/stropts.h>`.

As with `select`, any timeout set for `poll` is limited by the implementation's clock resolution (often 10 ms).

The return value from `poll` is □1 if an error occurred, 0 if no descriptors are ready before the timer expires, otherwise it is the number of descriptors that have a nonzero `revents` member.

If we are no longer interested in a particular descriptor, we just set the `fd` member of the `pollfd` structure to a negative value. Then the `events` member is ignored and the `revents` member is set to 0 on return.

Recall our discussion at the end of Section 6.3 about `FD_SETSIZE` and the maximum number of descriptors per descriptor set versus the maximum number of descriptors per process. We do not have that problem with `poll` since it is the caller's responsibility to allocate an array of `pollfd` structures and then tell the kernel the number of elements in the array. There is no fixed-size datatype similar to `fd_set` that the kernel knows about.

The POSIX specification requires both `select` and `poll`. But, from a portability perspective today, more systems support `select` than `poll`. Also, POSIX defines `pselect`, an enhanced version of `select` that handles signal blocking and provides increased time resolution. Nothing similar is defined for `poll`.

6.11 TCP Echo Server (Revisited Again)

We now redo our TCP echo server from Section 6.8 using `poll` instead of `select`. In the previous version using `select`, we had to allocate a `client` array along with a descriptor set named `rset` (Figure 6.15). With `poll`, we must allocate an array of `pollfd` structures to maintain the client information instead of allocating another array. We handle the `fd` member of this array the same way we handled the `client` array in Figure 6.15: a value of ☐1 means the entry is not in use; otherwise, it is the descriptor value. Recall from the previous section that any entry in the array of `pollfd` structures passed to `poll` with a negative value for the `fd` member is just ignored.

Figure 6.25 shows the first half of our server.

Figure 6.25 First half of TCP server using poll.

tcpcliserv/tcpservpoll01.c

```
#include     "unp.h"
#include     <limits.h>        /* for OPEN_MAX */

int
main(int argc, char **argv)
{
    int     i, maxi, listenfd, connfd, sockfd;
    int     nready;
    ssize_t n;
    char    buf[MAXLINE];
    socklen_t clilen;
    struct pollfd client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

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

    Listen(listenfd, LISTENQ);
```

```
20     client[0].fd = listenfd;
21     client[0].events = POLLRDNORM;
22     for (i = 1; i < OPEN_MAX; i++)
23         client[i].fd = -1;      /* -1 indicates available entry */
24     maxi = 0;                  /* max index into client[] array */
```

Allocate array of pollfd structures

11 We declare `OPEN_MAX` elements in our array of `pollfd` structures. Determining the maximum number of descriptors that a process can have open at any one time is difficult. We will encounter this problem again in Figure 13.4. One way is to call the POSIX `sysconf` function with an argument of `_SC_OPEN_MAX` (as described on pp. 42□44 of APUE) and then dynamically allocate an array of the appropriate size. But one of the possible returns from `sysconf` is "indeterminate," meaning we still have to guess a value. Here, we just use the POSIX `OPEN_MAX` constant.

Initialize

20□24 We use the first entry in the `client` array for the listening socket and set the descriptor for the remaining entries to □1. We also set the `POLLRDNORM` event for this descriptor, to be notified by `poll` when a new connection is ready to be accepted. The variable `maxi` contains the largest index of the `client` array currently in use.

The second half of our function is shown in Figure 6.26.

Figure 6.26 Second half of TCP server using poll.

tcpcliserv/tcpservpoll01.c

```
25     for ( ; ; ) {
26         nready = Poll(client, maxi + 1, INFTIM);

27         if (client[0].revents & POLLRDNORM) {  /* new client connection
*/
28             clilen = sizeof(cliaddr);
29             connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

30             for (i = 1; i < OPEN_MAX; i++)
31                 if (client[i].fd < 0) {
32                     client[i].fd = connfd;  /* save descriptor */
33                     break;
34                 }
35             if (i == OPEN_MAX)
36                 err_quit("too many clients");
37             client[i].events = POLLRDNORM;
38            if (i > maxi)
39                maxi = i;       /* max index in client[] array */

40            if (--nready <= 0)
41                continue;       /* no more readable descriptors */
42        }

43        for (i = 1; i <= maxi; i++) {       /* check all clients for data
*/
44            if ( (sockfd = client[i].fd) < 0)
45                continue;
46            if (client[i].revents & (POLLRDNORM | POLLERR)) {
47                if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
48                    if (errno == ECONNRESET) {
49                            /* connection reset by client */
50                        Close(sockfd);
51                        client[i].fd = -1;
52                    } else
53                        err_sys("read error");
54                } else if (n == 0) {
55                        /* connection closed by client */
56                    Close(sockfd);
57                    client[i].fd = -1;
58                } else
59                   Writen(sockfd, buf, n);
60                if (--nready <= 0)
61                        break;              /* no more readable descriptors
*/
62            }
63        }
64    }
65 }
```

Call poll, check for new connection

26□42 We call `poll` to wait for either a new connection or data on existing connection. When a new connection is accepted, we find the first available entry in the `client` array by looking for the first one with a negative descriptor. Notice that we start the search with the index of 1, since `client[0]` is used for the listening socket. When an available entry is found, we save the descriptor and set the `POLLRDNORM` event.

Check for data on an existing connection

43□63 The two return events that we check for are `POLLRDNORM` and `POLLERR`. The second of these we did not set in the `events` member because it is always returned when the condition is true. The reason we check for `POLLERR` is because some implementations return this event when an RST is received for a connection, while others just return POLLRDNORM. In either case, we call read and if an error has occurred, it will return an error. When an existing connection is terminated by the client, we just set the fd member to □1.

6.12 Summary

There are five different models for I/O provided by Unix:

- Blocking
- Nonblocking
- I/O multiplexing
- Signal-driven I/O
- Asynchronous I/O

The default is blocking I/O, which is also the most commonly used. We will cover nonblocking I/O and signal-driven I/O in later chapters and have covered I/O multiplexing in this chapter. True asynchronous I/O is defined by the POSIX specification, but few implementations exist.

The most commonly used function for I/O multiplexing is select. We tell the select function what descriptors we are interested in (for reading, writing, and exceptions), the maximum amount of time to wait, and the maximum descriptor number (plus one). Most calls to select specify readability, and we noted that the only exception condition when dealing with sockets is the arrival of out-of-band data (Chapter 24). Since select provides a time limit on how long a function blocks, we will use this feature in Figure 14.3 to place a time limit on an input operation.

We used our echo client in a batch mode using select and discovered that even though the end of the user input is encountered, data can still be in the pipe to or from the server. To handle this scenario requires the shutdown function, and it lets us take advantage of TCP's half-close feature.

The dangers of mixing stdio buffering (as well as our own readline buffering) with select caused us to produce versions of the echo client and server that operated on buffers instead of lines.

POSIX defines the function pselect, which increases the time precision from microseconds to nanoseconds and takes a new argument that is a pointer to a signal set. This lets us avoid race conditions when signals are being caught and we talk more about this in Section 20.5.

The `poll` function from System V provides functionality similar to `select` and provides additional information on STREAMS devices. POSIX requires both `select` and `poll`, but the former is used more often.

Exercises

6.1 We said that a descriptor set can be assigned to another descriptor set across an equals sign in C. How is this done if a descriptor set is an array of integers? (*Hint*: Look at your system's `<sys/select.h>` or `<sys/types.h>` header.)

6.2 When describing the conditions for which `select` returns "writable" in Section 6.3, why did we need the qualifier that the socket had to be nonblocking for a write operation to return a positive value?

6.3 What happens in Figure 6.9 if we prepend the word `"else"` before the word `"if"` on line 19?

6.4 In our example in Figure 6.21 add code to allow the server to be able to use as many descriptors as currently allowed by the kernel. (*Hint*: Look at the `setrlimit` function.)

6.5 Let's see what happens when the second argument to `shutdown` is `SHUT_RD`. Start with the TCP client in Figure 5.4 and make the following changes: Change the port number from `SERV_PORT` to 19, the `chargen` server (Figure 2.18); then, replace the call to `str_cli` with a call to the `pause` function. Run this program specifying the IP address of a local host that runs the `chargen` server. Watch the packets with a tool such as `tcpdump` (Section C.5). What happens?

6.6 Why would an application call `shutdown` with an argument of `SHUT_RDWR` instead of just calling `close`?

6.7 What happens in Figure 6.22 when the client sends an RST to terminate the connection?

6.8 Recode Figure 6.25 to call `sysconf` to determine the maximum number of descriptors and allocate the `client` array accordingly.

Chapter 7. Socket Options

7.1 Introduction

There are various ways to get and set the options that affect a socket:

- The `getsockopt` and `setsockopt` functions
- The `fcntl` function
- The `ioctl` function

This chapter starts by covering the `setsockopt` and `getsockopt` functions, followed by an example that prints the default value of all the options, and then a detailed description of all the socket options. We divide the detailed descriptions into the following categories: generic, IPv4, IPv6, TCP, and SCTP. This detailed coverage can be skipped during a first reading of this chapter, and the individual sections referred to when needed. A few options are discussed in detail in a later chapter, such as the IPv4 and IPv6 multicasting options, which we will describe with multicasting in Section 21.6.

We also describe the `fcntl` function, because it is the POSIX way to set a socket for nonblocking I/O, signal-driven I/O, and to set the owner of a socket. We save the `ioctl` function for Chapter 17.

7.2 'getsockopt' and 'setsockopt' Functions

These two functions apply only to sockets.

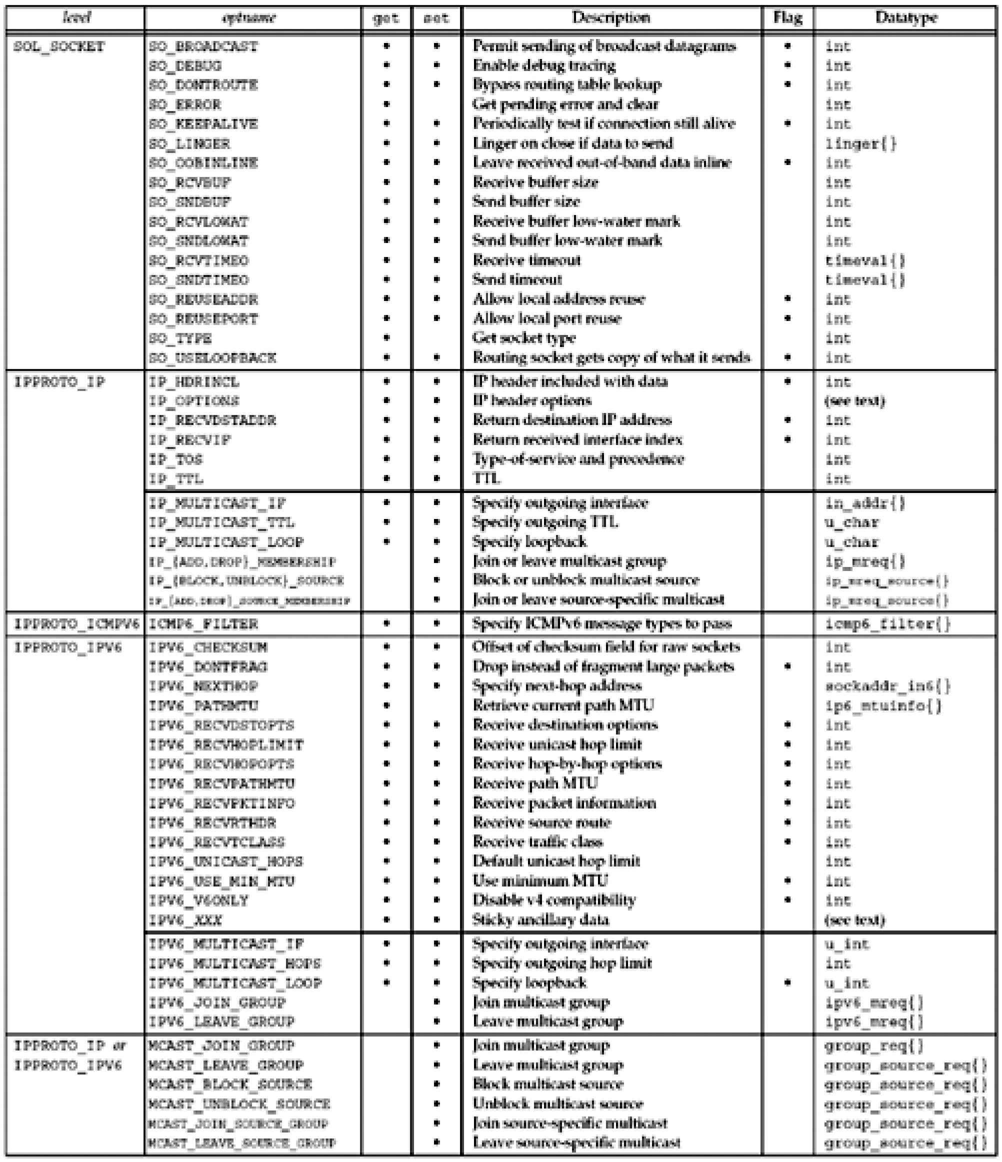
`#include <sys/socket.h>`
`int getsockopt(int` *sockfd*, `int` *level*, `int` *optname*, `void *`*optval*, `socklen_t *`*optlen*);
`int setsockopt(int` *sockfd*, `int` *level*, `int` *optname*, `const void *`*optval* `socklen_t` *optlen*);
Both return: 0 if OK,☐1 on error

sockfd must refer to an open socket descriptor. *level* specifies the code in the system that interprets the option: the general socket code or some protocol-specific code (e.g., IPv4, IPv6, TCP, or SCTP).

optval is a pointer to a variable from which the new value of the option is fetched by `setsockopt`, or into which the current value of the option is stored by `getsockopt`. The size of this variable is specified by the final argument, as a value for `setsockopt` and as a value-result for `getsockopt`.

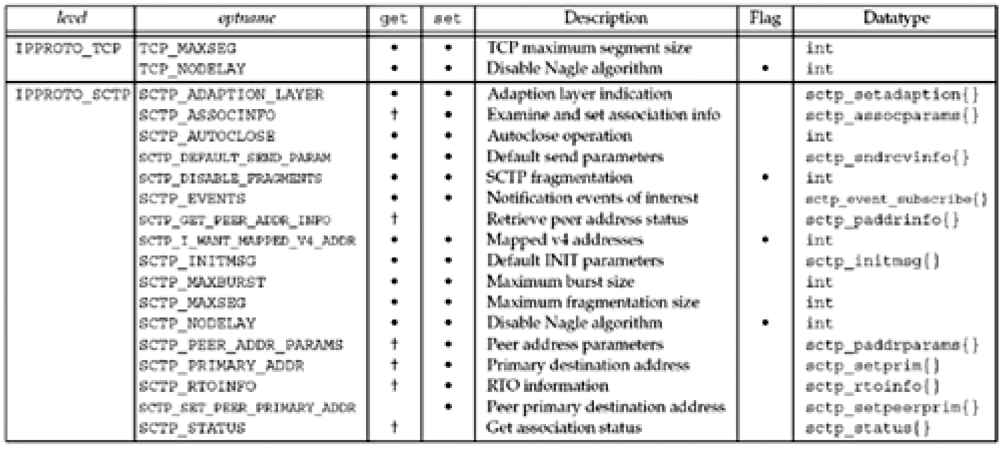
Figures 7.1 and 7.2 summarize the options that can be queried by `getsockopt` or set by `setsockopt`. The "Datatype" column shows the datatype of what the *optval* pointer must point to for each option. We use the notation of two braces to indicate a structure, as in `linger{}` to mean a `struct linger`.

Figure 7.1. Summary of socket and IP-layer socket options for getsockopt and setsockopt.



level	*optname*	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	Permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	Enable debug tracing	•	int
	SO_DONTROUTE	•	•	Bypass routing table lookup	•	int
	SO_ERROR	•		Get pending error and clear		int
	SO_KEEPALIVE	•	•	Periodically test if connection still alive	•	int
	SO_LINGER	•	•	Linger on close if data to send		linger{}
	SO_OOBINLINE	•	•	Leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	Receive buffer size		int
	SO_SNDBUF	•	•	Send buffer size		int
	SO_RCVLOWAT	•	•	Receive buffer low-water mark		int
	SO_SNDLOWAT	•	•	Send buffer low-water mark		int
	SO_RCVTIMEO	•	•	Receive timeout		timeval{}
	SO_SNDTIMEO	•	•	Send timeout		timeval{}
	SO_REUSEADDR	•	•	Allow local address reuse	•	int
	SO_REUSEPORT	•	•	Allow local port reuse	•	int
	SO_TYPE	•		Get socket type		int
	SO_USELOOPBACK	•	•	Routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options		(see text)
	IP_RECVDSTADDR	•	•	Return destination IP address	•	int
	IP_RECVIF	•	•	Return received interface index	•	int
	IP_TOS	•	•	Type-of-service and precedence		int
	IP_TTL	•	•	TTL		int
	IP_MULTICAST_IF	•	•	Specify outgoing interface		in_addr{}
	IP_MULTICAST_TTL	•	•	Specify outgoing TTL		u_char
	IP_MULTICAST_LOOP	•	•	Specify loopback		u_char
	IP_{ADD,DROP}_MEMBERSHIP		•	Join or leave multicast group		ip_mreq{}
	IP_{BLOCK,UNBLOCK}_SOURCE		•	Block or unblock multicast source		ip_mreq_source{}
	IP_{ADD,DROP}_SOURCE_MEMBERSHIP		•	Join or leave source-specific multicast		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Specify ICMPv6 message types to pass		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Offset of checksum field for raw sockets		int
	IPV6_DONTFRAG	•	•	Drop instead of fragment large packets	•	int
	IPV6_NEXTHOP	•	•	Specify next-hop address		sockaddr_in6{}
	IPV6_PATHMTU	•		Retrieve current path MTU		ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	•	•	Receive destination options	•	int
	IPV6_RECVHOPLIMIT	•	•	Receive unicast hop limit	•	int
	IPV6_RECVHOPOPTS	•	•	Receive hop-by-hop options	•	int
	IPV6_RECVPATHMTU	•	•	Receive path MTU	•	int
	IPV6_RECVPKTINFO	•	•	Receive packet information	•	int
	IPV6_RECVRTHDR	•	•	Receive source route	•	int
	IPV6_RECVTCLASS	•	•	Receive traffic class	•	int
	IPV6_UNICAST_HOPS	•	•	Default unicast hop limit		int
	IPV6_USE_MIN_MTU	•	•	Use minimum MTU	•	int
	IPV6_V6ONLY	•	•	Disable v4 compatibility	•	int
	IPV6_XXX	•	•	Sticky ancillary data		(see text)
	IPV6_MULTICAST_IF	•	•	Specify outgoing interface		u_int
	IPV6_MULTICAST_HOPS	•	•	Specify outgoing hop limit		int
	IPV6_MULTICAST_LOOP	•	•	Specify loopback	•	u_int
	IPV6_JOIN_GROUP		•	Join multicast group		ipv6_mreq{}
	IPV6_LEAVE_GROUP		•	Leave multicast group		ipv6_mreq{}
IPPROTO_IP *or* IPPROTO_IPV6	MCAST_JOIN_GROUP		•	Join multicast group		group_req{}
	MCAST_LEAVE_GROUP		•	Leave multicast group		group_source_req{}
	MCAST_BLOCK_SOURCE		•	Block multicast source		group_source_req{}
	MCAST_UNBLOCK_SOURCE		•	Unblock multicast source		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP		•	Join source-specific multicast		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP		•	Leave source-specific multicast		group_source_req{}

Figure 7.2. Summary of transport-layer socket options.



level	*optname*	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	Disable Nagle algorithm	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Adaption layer indication		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	Examine and set association info		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Autoclose operation		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Default send parameters		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP fragmentation	•	int
	SCTP_EVENTS	•	•	Notification events of interest		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		Retrieve peer address status		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Mapped v4 addresses	•	int
	SCTP_INITMSG	•	•	Default INIT parameters		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Maximum burst size		int
	SCTP_MAXSEG	•	•	Maximum fragmentation size		int
	SCTP_NODELAY	•	•	Disable Nagle algorithm	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	Peer address parameters		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	Primary destination address		sctp_setprim{}
	SCTP_RTOINFO	†	•	RTO information		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	Peer primary destination address		sctp_setpeerprim{}
	SCTP_STATUS	†		Get association status		sctp_status{}

There are two basic types of options: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values). The column labeled "Flag" specifies if the option is a flag option. When calling `getsockopt` for these flag options, `*`*optval* is an integer. The value returned in `*`*optval* is zero if the option is disabled, or nonzero if the option is enabled. Similarly, `setsockopt` requires a nonzero `*`*optval* to turn the option on, and a zero value to turn the option off. If the "Flag" column does not contain a "•," then the option is used to pass a value of the specified datatype between the user process and the system.

Subsequent sections of this chapter will give additional details on the options that affect a socket.

7.3 Checking if an Option Is Supported and Obtaining the Default

We now write a program to check whether most of the options defined in Figures 7.1 and 7.2 are supported, and if so, print their default value. Figure 7.3 contains the declarations for our program.

Declare union of possible values

3–8 Our `union` contains one member for each possible return value from `getsockopt`.

Define function prototypes

9–12 We define function prototypes for four functions that are called to print the value for a given socket option.

Define structure and initialize array

13–52 Our `sock_opts` structure contains all the information necessary to call `getsockopt` for each socket option and then print its current value. The final member, `opt_val_str`, is a pointer to one of our four functions that will print the option value. We allocate and initialize an array of these structures, one element for each socket option.

Not all implementations support all socket options. The way to determine if a given option is supported is to use an `#ifdef` or a `#if defined`, as we show for `SO_REUSEPORT`. For completeness, *every* element of the array should be compiled similarly to what we show for `SO_REUSEPORT`, but we omit these because the `#ifdefs` just lengthen the code that we show and add nothing to the discussion.

Figure 7.3 Declarations for our program to check the socket options.

sockopt/checkopts.c

```
#include    "unp.h"
#include    <netinet/tcp.h>      /* for TCP_xxx defines */
union val {
  int               i_val;
  long              l_val;
  struct linger     linger_val;
  struct timeval    timeval_val;
} val;
static char *sock_str_flag(union val *, int);
static char *sock_str_int(union val *, int);
static char *sock_str_linger(union val *, int);
static char *sock_str_timeval(union val *, int);
struct sock_opts {
  const char      *opt_str;
  int       opt_level;
  int       opt_name;
  char   *(*opt_val_str) (union val *, int);
} sock_opts[] = {
    { "SO_BROADCAST",       SOL_SOCKET, SO_BROADCAST,   sock_str_flag },
    { "SO_DEBUG",           SOL_SOCKET, SO_DEBUG,       sock_str_flag },
    { "SO_DONTROUTE",       SOL_SOCKET, SO_DONTROUTE,   sock_str_flag },
    { "SO_ERROR",           SOL_SOCKET, SO_ERROR,       sock_str_int },
    { "SO_KEEPALIVE",       SOL_SOCKET, SO_KEEPALIVE,   sock_str_flag },
    { "SO_LINGER",          SOL_SOCKET, SO_LINGER,      sock_str_linger },
    { "SO_OOBINLINE",       SOL_SOCKET, SO_OOBINLINE,   sock_str_flag },
    { "SO_RCVBUF",          SOL_SOCKET, SO_RCVBUF,      sock_str_int },
    { "SO_SNDBUF",          SOL_SOCKET, SO_SNDBUF,      sock_str_int },
    { "SO_RCVLOWAT",        SOL_SOCKET, SO_RCVLOWAT,    sock_str_int },
    { "SO_SNDLOWAT",        SOL_SOCKET, SO_SNDLOWAT,    sock_str_int },
    { "SO_RCVTIMEO",        SOL_SOCKET, SO_RCVTIMEO,    sock_str_timeval },
    { "SO_SNDTIMEO",        SOL_SOCKET, SO_SNDTIMEO,    sock_str_timeval },
    { "SO_REUSEADDR",       SOL_SOCKET, SO_REUSEADDR,   sock_str_flag },
#ifdef SO_REUSEPORT
    { "SO_REUSEPORT",       SOL_SOCKET, SO_REUSEPORT,   sock_str_flag },
#else
```

```
    { "SO_REUSEPORT",       0,         0,             NULL },
#endif
    { "SO_TYPE",           SOL_SOCKET, SO_TYPE,       sock_str_int },
    { "SO_USELOOPBACK",     SOL_SOCKET, SO_USELOOPBACK,
sock_str_flag },
    { "IP_TOS",            IPPROTO_IP, IP_TOS,        sock_str_int },
    { "IP_TTL",            IPPROTO_IP, IP_TTL,        sock_str_int },
    { "IPV6_DONTFRAG",      IPPROTO_IPV6,IPV6_DONTFRAG,
sock_str_flag },
    { "IPV6_UNICAST_HOPS",
IPPROTO_IPV6,IPV6_UNICAST_HOPS,sock_str_int },
    { "IPV6_V6ONLY",        IPPROTO_IPV6,IPV6_V6ONLY,
sock_str_flag },
    {"TCP_MAXSEG",         IPPROTO_TCP,TCP_MAXSEG,     sock_str_int},
    { "TCP_NODELAY",        IPPROTO_TCP,TCP_NODELAY,
sock_str_flag },
    { "SCTP_AUTOCLOSE",
IPPROTO_SCTP,SCTP_AUTOCLOSE,sock_str_int },
    { "SCTP_MAXBURST",      IPPROTO_SCTP,SCTP_MAXBURST,
sock_str_int },
    { "SCTP_MAXSEG",        IPPROTO_SCTP,SCTP_MAXSEG,
sock_str_int },
    { "SCTP_NODELAY",       IPPROTO_SCTP,SCTP_NODELAY,
sock_str_flag },
    { NULL,                0,         0,             NULL }
};
```

Figure 7.4 shows our `main` function.

Figure 7.4 main function to check all socket options.

sockopt/checkopts.c

```
int
main(int argc, char **argv)
{
    int     fd;
    socklen_t len;
    struct sock_opts *ptr;

    for (ptr = sock_opts; ptr->opt_str != NULL; ptr++) {
        printf("%s: ", ptr->opt_str);
        if (ptr->opt_val_str == NULL)
            printf("(undefined)\n");
63        else {
64            switch (ptr->opt_level) {
65            case SOL_SOCKET:
66            case IPPROTO_IP:
67            case IPPROTO_TCP:
68                fd = Socket(AF_INET, SOCK_STREAM, 0);
69                break;
70 #ifdef  IPV6
71            case IPPROTO_IPV6:
72                fd = Socket(AF_INET6, SOCK_STREAM, 0);
73                break;
74 #endif
75 #ifdef  IPPROTO_SCTP
76            case IPPROTO_SCTP:
77                fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
78                break;
79 #endif
80            default:
81                err_quit("Can't create fd for level %d\n",
ptr->opt_level);
82            }
83            len = sizeof(val);
84            if (getsockopt(fd, ptr->opt_level, ptr->opt_name,
85                      &val, &len) == -1) {
86                err_ret("getsockopt error");
87            } else {
88                printf("default = %s\n", (*ptr->opt_val_str)  (&val,
len));
89            }
90            close(fd);
91        }
92    }
93    exit(0);
94 }
```

Go through all options

59–63 We go through all elements in our array. If the `opt_val_str` pointer is null, the option is not defined by the implementation (which we showed for `SO_REUSEPORT`).

Create socket

63–82 We create a socket on which to try the option. To try socket, IPv4, and TCP layer socket options, we use an IPv4 TCP socket. To try IPv6 layer socket options, we use an IPv6 TCP socket, and to try SCTP layer socket options, we use an IPv4 SCTP socket.

Call getsockopt

83□87 We call `getsockopt` but do not terminate if an error is returned. Many implementations define some of the socket option names even though they do not support the option. Unsupported options should elicit an error of `ENOPROTOOPT`.

Print option's default value

88□89 If `getsockopt` returns success, we call our function to convert the option value to a string and print the string.

In Figure 7.3, we showed four function prototypes, one for each type of option value that is returned. Figure 7.5 shows one of these four functions, `sock_str_flag`, which prints the value of a flag option. The other three functions are similar.

Figure 7.5 sock_str_flag function: convert flag option to a string.

sockopt/checkopts.c

```
 95 static char strres[128];

 96 static char *
 97 sock_str_flag(union val *ptr, int len)
 98 {
 99    if (len != sizeof(int))
100        snprintf(strres, sizeof(strres), "size (%d) not sizeof(int)",
len);
101    else
102        snprintf(strres, sizeof(strres),
103                 "%s", (ptr->i_val == 0) ? "off" : "on");
104    return(strres);
105 }
```

99□104 Recall that the final argument to `getsockopt` is a value-result argument. The first check we make is that the size of the value returned by `getsockopt` is the expected size. The string returned is `off` or `on`, depending on whether the value of the flag option is zero or nonzero, respectively.

Running this program under FreeBSD 4.8 with KAME SCTP patches gives the following output:

```
freebsd % checkopts
```

```
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = l_onoff = 0, l_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 57344
SO_SNDBUF: default = 32768
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 2048
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: default = off
SO_TYPE: default = 1
SO_USELOOPBACK: default = off
IP_TOS: default = 0
IP_TTL: default = 64
IPV6_DONTFRAG: default = off
IPV6_UNICAST_HOPS: default = -1
IPV6_V6ONLY: default = off
TCP_MAXSEG: default = 512
TCP_NODELAY: default = off
SCTP_AUTOCLOSE: default = 0
SCTP_MAXBURST: default = 4
SCTP_MAXSEG: default = 1408
SCTP_NODELAY: default = off
```

The value of 1 returned for the `SO_TYPE` option corresponds to `SOCK_STREAM` for this implementation.

7.4 Socket States

Some socket options have timing considerations about when to set or fetch the option versus the state of the socket. We mention these with the affected options.

The following socket options are inherited by a connected TCP socket from the listening socket (pp. 462□463 of TCPv2): `SO_DEBUG`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_LINGER`, `SO_OOBINLINE`, `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF`, `SO_SNDLOWAT`, `TCP_MAXSEG`, and `TCP_NODELAY`. This is important with TCP because the connected socket is not returned to a server by `accept` until the three-way handshake is completed by the TCP layer. To ensure that one of these socket options is set for the connected socket when the three-way handshake completes, we must set that option for the listening socket.

7.5 Generic Socket Options

We start with a discussion of the generic socket options. These options are protocol-independent (that is, they are handled by the protocol-independent code within the kernel, not by one particular protocol module such as IPv4), but some of the options apply to only certain types of sockets. For example, even though the `SO_BROADCAST` socket option is called "generic," it applies only to datagram sockets.

SO_BROADCAST Socket Option

This option enables or disables the ability of the process to send broadcast messages. Broadcasting is supported for only datagram sockets and only on networks that support the concept of a broadcast message (e.g., Ethernet, token ring, etc.). You cannot broadcast on a point-to-point link or any connection-based transport protocol such as SCTP or TCP. We will talk more about broadcasting in Chapter 20.

Since an application must set this socket option before sending a broadcast datagram, it prevents a process from sending a broadcast when the application was never designed to broadcast. For example, a UDP application might take the destination IP address as a command-line argument, but the application never intended for a user to type in a broadcast address. Rather than forcing the application to try to determine if a given address is a broadcast address or not, the test is in the kernel: If the destination address is a broadcast address and this socket option is not set, `EACCES` is returned (p. 233 of TCPv2).

SO_DEBUG Socket Option

This option is supported only by TCP. When enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for the socket. These are kept in a circular buffer within the kernel that can be examined with the `trpt` program. Pages 916□920 of TCPv2 provide additional details and an example that uses this option.

SO_DONTROUTE Socket Option

This option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol. For example, with IPv4, the packet is directed to the appropriate local interface, as specified by the network and subnet portions of the destination address. If the local interface cannot be determined from the destination address (e.g., the destination is not on the other end of a point-to-point link, or is not on a shared network), `ENETUNREACH` is returned.

The equivalent of this option can also be applied to individual datagrams using the MSG_DONTROUTE flag with the send, sendto, or sendmsg functions.

This option is often used by routing daemons (e.g., routed and gated) to bypass the routing table and force a packet to be sent out a particular interface.

SO_ERROR Socket Option

When an error occurs on a socket, the protocol module in a Berkeley-derived kernel sets a variable named so_error for that socket to one of the standard Unix E*xxx* values. This is called the *pending error* for the socket. The process can be immediately notified of the error in one of two ways:

1. If the process is blocked in a call to select on the socket (Section 6.3), for either readability or writability, select returns with either or both conditions set.
2. If the process is using signal-driven I/O (Chapter 25), the SIGIO signal is generated for either the process or the process group.

The process can then obtain the value of so_error by fetching the SO_ERROR socket option. The integer value returned by getsockopt is the pending error for the socket. The value of so_error is then reset to 0 by the kernel (p. 547 of TCPv2).

If so_error is nonzero when the process calls read and there is no data to return, read returns□1 with errno set to the value of so_error (p. 516 of TCPv2). The value of so_error is then reset to 0. If there is data queued for the socket, that data is returned by read instead of the error condition. If so_error is nonzero when the process calls write, □1 is returned with errno set to the value of so_error (p. 495 of TCPv2) and so_error is reset to 0.

There is a bug in the code shown on p. 495 of TCPv2 in that so_error is not reset to 0. This has been fixed in most modern releases. Anytime the pending error for a socket is returned, it must be reset to 0.

This is the first socket option that we have encountered that can be fetched but cannot be set.

SO_KEEPALIVE Socket Option

When the keep-alive option is set for a TCP socket and no data has been exchanged across the socket in either direction for two hours, TCP automatically sends a *keep-alive probe* to the peer. This probe is a TCP segment to which the peer must respond. One of three scenarios results:

1. The peer responds with the expected ACK. The application is not notified (since everything is okay). TCP will send another probe following another two hours of inactivity.
2. The peer responds with an RST, which tells the local TCP that the peer host has crashed and rebooted. The socket's pending error is set to `ECONNRESET` and the socket is closed.
3. There is no response from the peer to the keep-alive probe. Berkeley-derived TCPs send 8 additional probes, 75 seconds apart, trying to elicit a response. TCP will give up if there is no response within 11 minutes and 15 seconds after sending the first probe.

   HP-UX 11 treats the keep-alive probes in the same way as it would treat data, sending the second probe after a retransmission timeout and doubling the timeout for each packet until the configured maximum interval, with a default of 10 minutes.

   If there is no response at all to TCP's keep-alive probes, the socket's pending error is set to `ETIMEDOUT` and the socket is closed. But if the socket receives an ICMP error in response to one of the keep-alive probes, the corresponding error (Figures A.15 and A.16) is returned instead (and the socket is still closed). A common ICMP error in this scenario is "host unreachable," indicating that the peer host is unreachable, in which case, the pending error is set to `EHOSTUNREACH`. This can occur either because of a network failure or because the remote host has crashed and the last-hop router has detected the crash.

Chapter 23 of TCPv1 and pp. 828□831 of TCPv2 contain additional details on the keep-alive option.

Undoubtedly the most common question regarding this option is whether the timing parameters can be modified (usually to reduce the two-hour period of inactivity to some shorter value). Appendix E of TCPv1 discusses how to change these timing parameters for various kernels, but be aware that most kernels maintain these parameters on a per-kernel basis, not on a per-socket basis, so changing the inactivity period from 2 hours to 15 minutes, for example, will affect *all* sockets on the host that enable this option. However, such questions usually result from a misunderstanding of the purpose of this option.

The purpose of this option is to detect if the peer *host* crashes or becomes unreachable (e.g., dial-up modem connection drops, power fails, etc.). If the peer *process* crashes, its TCP will send a FIN across the connection, which we can easily detect with `select`. (This was why we used `select` in Section 6.4.) Also realize that if there is no response to any of the keep-alive probes (scenario 3), we are not guaranteed that the peer host has crashed, and TCP may well terminate a valid connection. It could be that some intermediate router has crashed for 15 minutes, and that period of time just happens to completely overlap our host's 11-minute and 15-second keep-alive probe period. In fact, this function might more properly be called "make-dead" rather than "keep-alive" since it can terminate live connections.

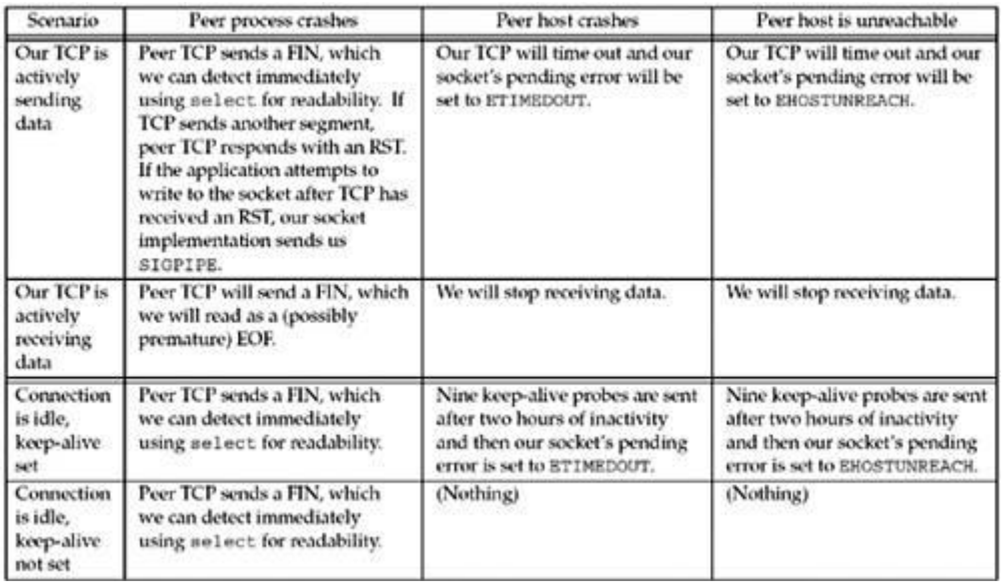
This option is normally used by servers, although clients can also use the option. Servers use the option because they spend most of their time blocked waiting for input across the TCP connection, that is, waiting for a client request. But if the client host's connection drops, is powered off, or crashes, the server process will never know about it, and the server will continually wait for input that can never arrive. This is called a *half-open connection*. The keep-alive option will detect these half-open connections and terminate them.

Some servers, notably FTP servers, provide an application timeout, often on the order of minutes. This is done by the application itself, normally around a call to `read`, reading the next client command. This timeout does not involve this socket option. This is often a better method of eliminating connections to missing clients, since the application has complete control if it implements the timeout itself.

SCTP has a *heartbeat* mechanism that is similar to TCP's "keep-alive" mechanism. The heartbeat mechanism is controlled through parameters of the `SCTP_SET_PEER_ADDR_PARAMS` socket option discussed later in this chapter, rather than the `SO_KEEPALIVE` socket option. The settings made by `SO_KEEPALIVE` on a SCTP socket are ignored and do not affect the SCTP heartbeat mechanism.

Figure 7.6 summarizes the various methods that we have to detect when something happens on the other end of a TCP connection. When we say "using `select` for readability," we mean calling `select` to test whether a socket is readable.

Figure 7.6. Ways to detect various TCP conditions.



Scenario	Peer process crashes	Peer host crashes	Peer host is unreachable
Our TCP is actively sending data	Peer TCP sends a FIN, which we can detect immediately using `select` for readability. If TCP sends another segment, peer TCP responds with an RST. If the application attempts to write to the socket after TCP has received an RST, our socket implementation sends us `SIGPIPE`.	Our TCP will time out and our socket's pending error will be set to `ETIMEDOUT`.	Our TCP will time out and our socket's pending error will be set to `EHOSTUNREACH`.
Our TCP is actively receiving data	Peer TCP will send a FIN, which we will read as a (possibly premature) EOF.	We will stop receiving data.	We will stop receiving data.
Connection is idle, keep-alive set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	Nine keep-alive probes are sent after two hours of inactivity and then our socket's pending error is set to `ETIMEDOUT`.	Nine keep-alive probes are sent after two hours of inactivity and then our socket's pending error is set to `EHOSTUNREACH`.
Connection is idle, keep-alive not set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	(Nothing)	(Nothing)

SO_LINGER Socket Option

This option specifies how the `close` function operates for a connection-oriented protocol (e.g., for TCP and SCTP, but not for UDP). By default, `close` returns immediately, but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer.

The `SO_LINGER` socket option lets us change this default. This option requires the following structure to be passed between the user process and the kernel. It is defined by including `<sys/socket.h>`.

```
struct linger {
  int   l_onoff;       /* 0=off, nonzero=on */
  int   l_linger;      /* linger time, POSIX specifies units as seconds */
};
```

Calling `setsockopt` leads to one of the following three scenarios, depending on the values of the two structure members:

1. If `l_onoff` is 0, the option is turned off. The value of `l_linger` is ignored and the previously discussed TCP default applies: `close` returns immediately.
2. If `l_onoff` is nonzero and `l_linger` is zero, TCP aborts the connection when it is closed (pp. 1019□1020 of TCPv2). That is, TCP discards any data still remaining in the socket send buffer and sends an RST to the peer, not the normal four-packet connection termination sequence (Section 2.6). We will show an example of this in Figure 16.21. This avoids TCP's TIME_WAIT state, but in doing so, leaves open the possibility of another incarnation of this connection being created within 2MSL seconds (Section 2.7) and having old duplicate segments from the just-terminated connection being incorrectly delivered to the new incarnation.

   SCTP will also do an abortive close of the socket by sending an ABORT chunk to the peer (see Section 9.2 of [Stewart and Xie 2001]) when `l_onoff` is nonzero and `l_linger` is zero.

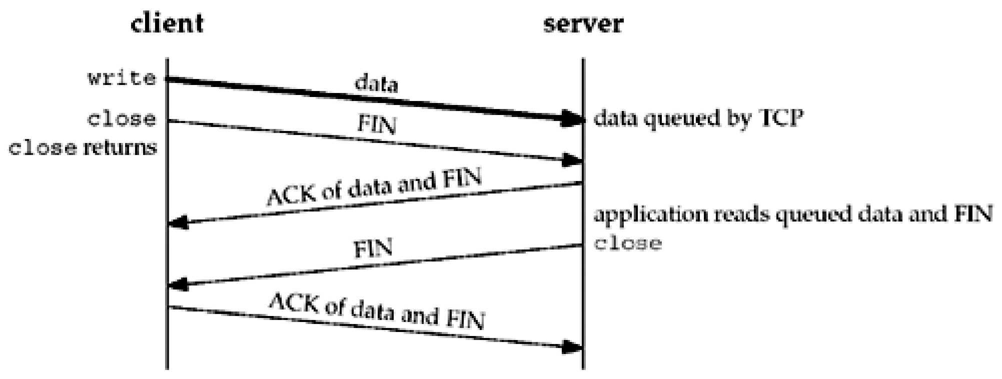
   Occasional USENET postings advocate the use of this feature just to avoid the TIME_WAIT state and to be able to restart a listening server even if connections are still in use with the server's well-known port. This should NOT be done and could lead to data corruption, as detailed in RFC 1337 [Braden 1992]. Instead, the `SO_REUSEADDR` socket option should always be used in the server before the call to `bind`, as we will describe shortly. The TIME_WAIT state is our friend and is there to help us (i.e., to let old duplicate segments expire in the network). Instead of trying to avoid the state, we should understand it (Section 2.7).

There are certain circumstances which warrant using this feature to send an abortive close. One example is an RS-232 terminal server, which might hang forever in CLOSE_WAIT trying to deliver data to a struck terminal port, but would properly reset the stuck port if it got an RST to discard the pending data.

3. If `l_onoff` is nonzero and `l_linger` is nonzero, then the kernel will *linger* when the socket is closed (p. 472 of TCPv2). That is, if there is any data still remaining in the socket send buffer, the process is put to sleep until either: (i) all the data is sent and acknowledged by the peer TCP, or (ii) the linger time expires. If the socket has been set to nonblocking (Chapter 16), it will not wait for the `close` to complete, even if the linger time is nonzero. When using this feature of the `SO_LINGER` option, it is important for the application to check the return value from `close`, because if the linger time expires before the remaining data is sent and acknowledged, `close` returns `EWOULDBLOCK` and any remaining data in the send buffer is discarded.

We now need to see exactly when a `close` on a socket returns given the various scenarios we looked at. We assume that the client writes data to the socket and then calls `close`. Figure 7.7 shows the default situation.

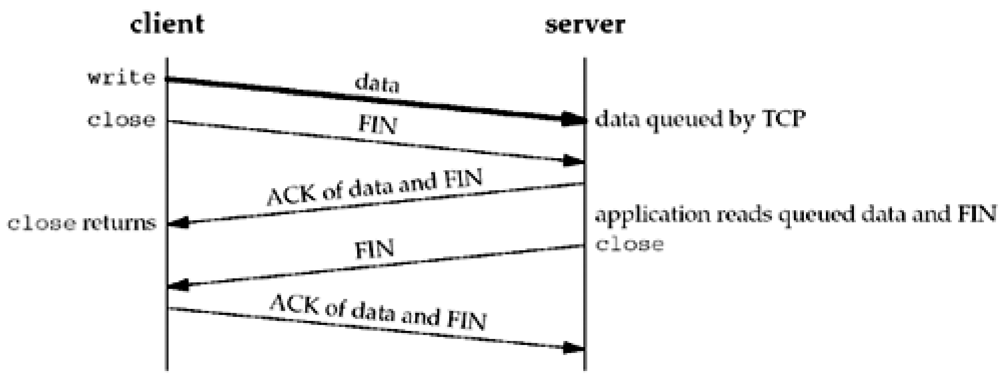
Figure 7.7. Default operation of close: it returns immediately.



We assume that when the client's data arrives, the server is temporarily busy, so the data is added to the socket receive buffer by its TCP. Similarly, the next segment, the client's FIN, is also added to the socket receive buffer (in whatever manner the implementation records that a FIN has been received on the connection). But by default, the client's `close` returns immediately. As we show in this scenario, the client's `close` can return before the server reads the remaining data in its socket receive buffer. Therefore, it is possible for the server host to crash before the server application reads this remaining data, and the client application will never know.

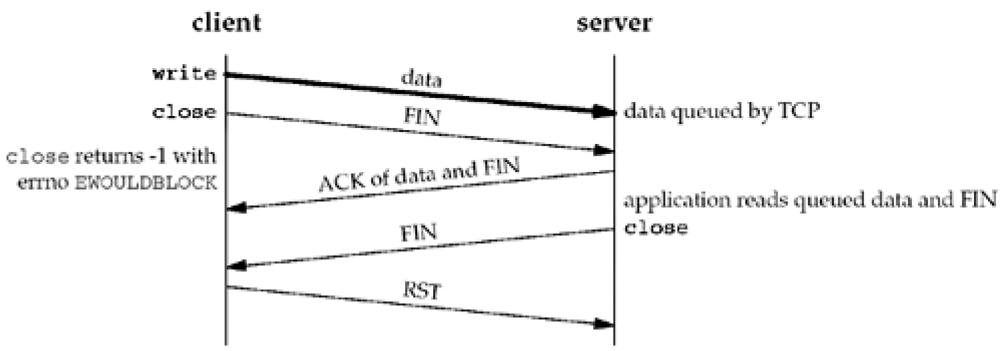
The client can set the SO_LINGER socket option, specifying some positive linger time. When this occurs, the client's close does not return until all the client's data and its FIN have been acknowledged by the server TCP. We show this in Figure 7.8.

Figure 7.8. close with SO_LINGER socket option set and l_linger a positive value.



But we still have the same problem as in Figure 7.7: The server host can crash before the server application reads its remaining data, and the client application will never know. Worse, Figure 7.9 shows what can happen when the SO_LINGER option is set to a value that is too low.

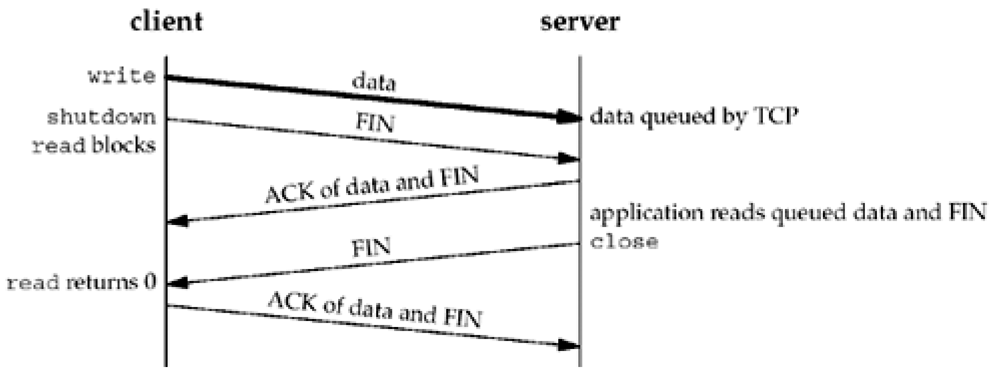
Figure 7.9. close with SO_LINGER socket option set and l_linger a small positive value.



The basic principle here is that a successful return from close, with the SO_LINGER socket option set, only tells us that the data we sent (and our FIN) have been acknowledged by the peer TCP. This does *not* tell us whether the peer application has read the data. If we do not set the SO_LINGER socket option, we do not know whether the peer TCP has acknowledged the data.

One way for the client to know that the server has read its data is to call shutdown (with a second argument of SHUT_WR) instead of close and wait for the peer to close its end of the connection. We show this scenario in Figure 7.10.

Figure 7.10. Using shutdown to know that peer has received our data.



Comparing this figure to Figures 7.7 and 7.8 we see that when we close our end of the connection, depending on the function called (`close` or `shutdown`) and whether the `SO_LINGER` socket option is set, the return can occur at three different times:

1. `close` returns immediately, without waiting at all (the default; Figure 7.7).
2. `close` lingers until the ACK of our FIN is received (Figure 7.7).
3. `shutdown` followed by a `read` waits until we receive the peer's FIN (Figure 7.10).

Another way to know that the peer application has read our data is to use an *application-level acknowledgment*, or *application ACK*. For example, in the following, the client sends its data to the server and then calls `read` for one byte of data:

```
char  ack;

Write(sockfd, data, nbytes);      /* data from client to server */
n = Read(sockfd, &ack, 1);        /* wait for application-level ACK */
```

The server reads the data from the client and then sends back the one-byte application-level ACK:

```
nbytes = Read(sockfd, buff, sizeof(buff)); /* data from client */
        /* server verifies it received correct
           amount of data from client */
Write(sockfd, "", 1);          /* server's ACK back to client */
```

We are guaranteed that when the `read` in the client returns, the server process has read the data we sent. (This assumes that either the server knows how much data the client is sending, or there is some application-defined end-of-record marker, which we do not show here.) Here, the application-level ACK is a byte of 0, but the contents of this byte could be used to signal other conditions from the server to the client. Figure 7.11 shows the possible packet exchange.

Figure 7.11. Application ACK.

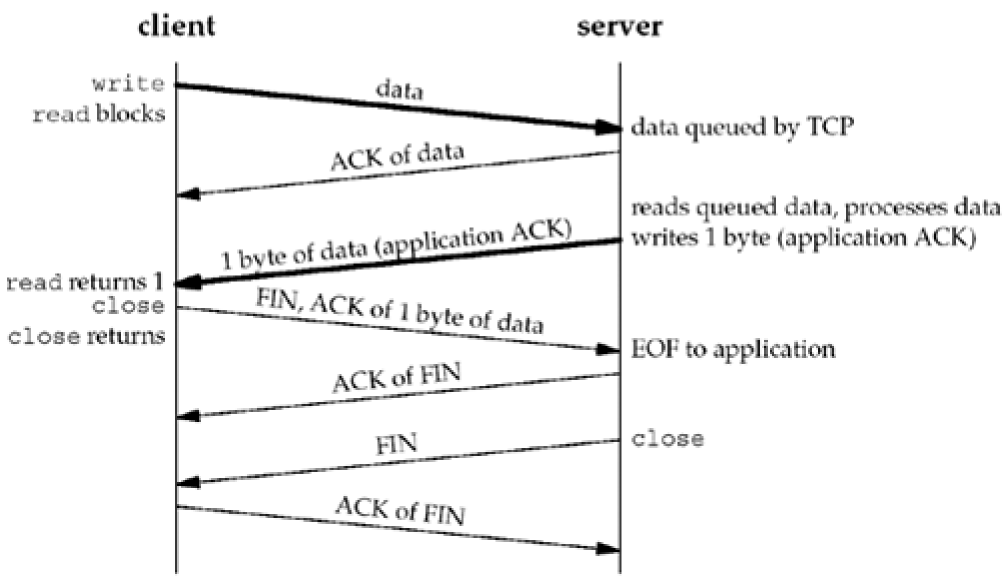
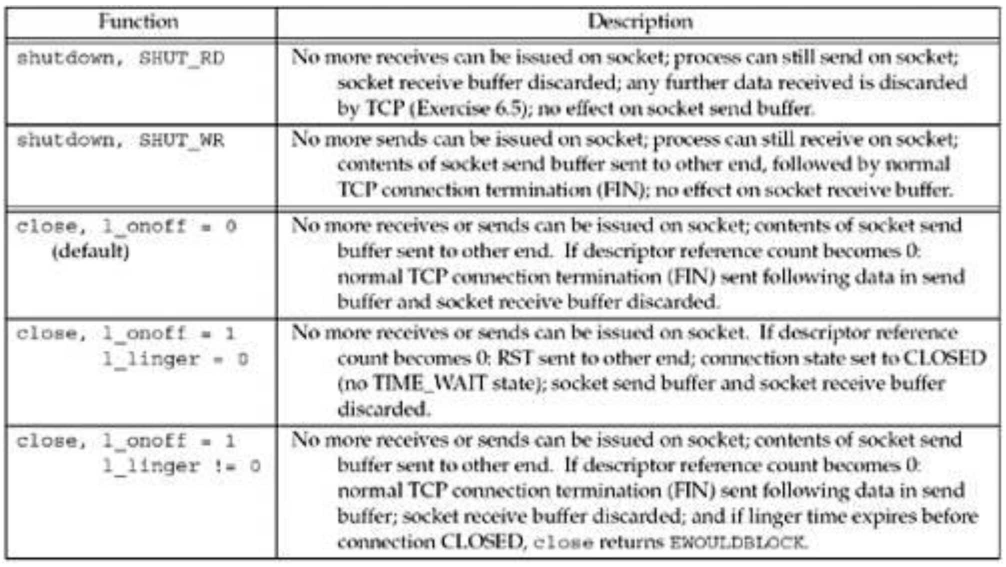


Figure 7.12 summarizes the two possible calls to `shutdown` and the three possible calls to `close`, and the effect on a TCP socket.

Figure 7.12. Summary of shutdown and SO_LINGER scenarios.



Function	Description
`shutdown, SHUT_RD`	No more receives can be issued on socket; process can still send on socket; socket receive buffer discarded; any further data received is discarded by TCP (Exercise 6.5); no effect on socket send buffer.
`shutdown, SHUT_WR`	No more sends can be issued on socket; process can still receive on socket; contents of socket send buffer sent to other end, followed by normal TCP connection termination (FIN); no effect on socket receive buffer.
`close, l_onoff = 0` (default)	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer and socket receive buffer discarded.
`close, l_onoff = 1` `l_linger = 0`	No more receives or sends can be issued on socket. If descriptor reference count becomes 0: RST sent to other end; connection state set to CLOSED (no TIME_WAIT state); socket send buffer and socket receive buffer discarded.
`close, l_onoff = 1` `l_linger != 0`	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer; socket receive buffer discarded; and if linger time expires before connection CLOSED, `close` returns `EWOULDBLOCK`.

SO_OOBINLINE Socket Option

When this option is set, out-of-band data will be placed in the normal input queue (i.e., inline). When this occurs, the `MSG_OOB` flag to the receive functions cannot be used to read the out-of-band data. We will discuss out-of-band data in more detail in Chapter 24.

SO_RCVBUF and SO_SNDBUF Socket Options

Every socket has a send buffer and a receive buffer. We described the operation of the send buffers with TCP, UDP, and SCTP in Figures 2.15, 2.16, and 2.17.

The receive buffers are used by TCP, UDP, and SCTP to hold received data until it is read by the application. With TCP, the available room in the socket receive buffer limits the window that TCP can advertise to the other end. The TCP socket receive buffer cannot overflow because the peer is not allowed to send data beyond the advertised window. This is TCP's flow control, and if the peer ignores the advertised window and sends data beyond the window, the receiving TCP discards it. With UDP, however, when a datagram arrives that will not fit in the socket receive buffer, that datagram is discarded. Recall that UDP has no flow control: It is easy for a fast sender to overwhelm a slower receiver, causing datagrams to be discarded by the receiver's UDP, as we will show in Section 8.13. In fact, a fast sender can overwhelm its own network interface, causing datagrams to be discarded by the sender itself.

These two socket options let us change the default sizes. The default values differ widely between implementations. Older Berkeley-derived implementations would default the TCP send and receive buffers to 4,096 bytes, but newer systems use larger values, anywhere from 8,192 to 61,440 bytes. The UDP send buffer size often defaults to a value around 9,000 bytes if the host supports NFS, and the UDP receive buffer size often defaults to a value around 40,000 bytes.

When setting the size of the TCP socket receive buffer, the ordering of the function calls is important. This is because of TCP's window scale option (Section 2.6), which is exchanged with the peer on the SYN segments when the connection is established. For a client, this means the `SO_RCVBUF` socket option must be set before calling `connect`. For a server, this means the socket option must be set for the listening socket before calling `listen`. Setting this option for the connected socket will have no effect whatsoever on the possible window scale option because `accept` does not return with the connected socket until TCP's three-way handshake is complete. That is why this option must be set for the listening socket. (The sizes of the socket buffers are always inherited from the listening socket by the newly created connected socket: pp. 462□463 of TCPv2.)

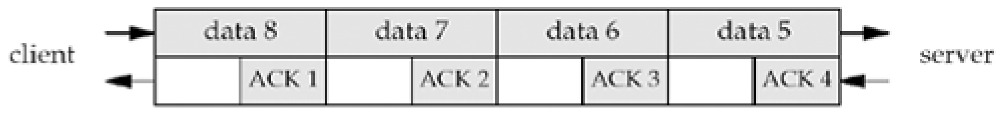
The TCP socket buffer sizes should be at least four times the MSS for the connection. If we are dealing with unidirectional data transfer, such as a file transfer in one direction, when we say "socket buffer sizes," we mean the socket send buffer size on the sending host and the socket receive buffer size on the receiving host. For bidirectional data transfer, we mean both socket buffer sizes on the sender and both socket buffer sizes on the receiver. With typical default buffer sizes of 8,192 bytes or larger, and a typical MSS of 512 or 1,460, this requirement is normally met.

The minimum MSS multiple of four is a result of the way that TCP's fast recovery algorithm works. The TCP sender uses three duplicate acknowledgments to detect that a packet was lost (RFC 2581 [Allman, Paxson, and Stevens 1999]). The receiver sends a duplicate acknowledgment for each segment it receives after a lost segment. If the window size is smaller than four segments, there cannot be three duplicate acknowledgments, so the fast recovery algorithm cannot be invoked.

To avoid wasting potential buffer space, the TCP socket buffer sizes should also be an even multiple of the MSS for the connection. Some implementations handle this detail for the application, rounding up the socket buffer size after the connection is established (p. 902 of TCPv2). This is another reason to set these two socket options before establishing a connection. For example, using the default 4.4BSD size of 8,192 and assuming an Ethernet with an MSS of 1,460, both socket buffers are rounded up to 8,760 (6 x 1,460) when the connection is established. This is not a crucial requirement; the additional space in the socket buffer above the multiple of the MSS is simply unused.

Another consideration in setting the socket buffer sizes deals with performance. Figure 7.13 shows a TCP connection between two endpoints (which we call a *pipe*) with a capacity of eight segments.

Figure 7.13. TCP connection (pipe) with a capacity of eight segments.



We show four data segments on the top and four ACKs on the bottom. Even though there are only four segments of data in the pipe, the client must have a send buffer capacity of at least eight segments, because the client TCP must keep a copy of each segment until the ACK is received from the server.

We are ignoring some details here. First, TCP's slow-start algorithm limits the rate at which segments are initially sent on an idle connection. Next, TCP often acknowledges every other segment, not every segment as we show. All these details are covered in Chapters 20 and 24 of TCPv1.

What is important to understand is the concept of the full-duplex pipe, its capacity, and how that relates to the socket buffer sizes on both ends of the connection. The capacity of the pipe is called the *bandwidth-delay product* and we calculate this by multiplying the bandwidth (in bits/sec) times the RTT (in seconds), converting the result from bits to bytes. The RTT is easily measured with the `ping` program.

The bandwidth is the value corresponding to the slowest link between two endpoints and must somehow be known. For example, a T1 line (1,536,000 bits/sec) with an RTT of 60 ms gives a bandwidth-delay product of 11,520 bytes. If the socket buffer sizes are less than this, the pipe will not stay full, and the performance will be less than expected. Large socket buffers are required when the bandwidth gets larger (e.g., T3 lines at 45 Mbits/sec) or when the RTT gets large (e.g., satellite links with an RTT around 500 ms). When the bandwidth-delay product exceeds TCP's maximum normal window size (65,535 bytes), both endpoints also need the TCP *long fat pipe* options that we mentioned in Section 2.6.

Most implementations have an upper limit for the sizes of the socket send and receive buffers, and sometimes this limit can be modified by the administrator. Older Berkeley-derived implementations had a hard upper limit of around 52,000 bytes, but newer implementations have a default limit of 256,000 bytes or more, and this can usually be increased by the administrator. Unfortunately, there is no simple way for an application to determine this limit. POSIX defines the `fpathconf` function, which most implementations support, and using the `_PC_SOCK_MAXBUF` constant as the second argument, we can retrieve the maximum size of the socket buffers. Alternately, an application can try setting the socket buffers to the desired value, and if that fails, cut the value in half and try again until it succeeds. Finally, an application should make sure that it's not actually making the socket buffer smaller when it sets it to a preconfigured "large" value; calling `getsockopt` first to retrieve the system's default and seeing if that's large enough is often a good start.

SO_RCVLOWAT and SO_SNDLOWAT Socket Options

Every socket also has a receive low-water mark and a send low-water mark. These are used by the `select` function, as we described in Section 6.3. These two socket options, `SO_RCVLOWAT` and `SO_SNDLOWAT`, let us change these two low-water marks.

The receive low-water mark is the amount of data that must be in the socket receive buffer for `select` to return "readable." It defaults to 1 for TCP, UDP, and SCTP sockets. The send low-water mark is the amount of available space that must exist in the socket send buffer for `select` to return "writable." This low-water mark normally defaults to 2,048 for TCP sockets. With UDP, the low-water mark is used, as we described in Section 6.3, but since the number of bytes of available space in the send buffer for a UDP socket never changes (since UDP does not keep a copy of the datagrams sent by the application), as long as the UDP socket send buffer size is greater than the socket's low-water mark, the UDP socket is always writable. Recall from Figure 2.16 that UDP does not have a send buffer; it has only a send buffer size.

SO_RCVTIMEO and SO_SNDTIMEO Socket Options

These two socket options allow us to place a timeout on socket receives and sends. Notice that the argument to the two `sockopt` functions is a pointer to a `timeval` structure, the same one used with `select` (Section 6.3). This lets us specify the timeouts in seconds and microseconds. We disable a timeout by setting its value to 0 seconds and 0 microseconds. Both timeouts are disabled by default.

The receive timeout affects the five input functions: `read`, `readv`, `recdv`, `recvfrom`, and `recvmsg`. The send timeout affects the five output functions: `write`, `writev`, `send`, `sendto`, and `sendmsg`. We will talk more about socket timeouts in Section 14.2.

These two socket options and the concept of inherent timeouts on socket receives and sends were added with 4.3BSD Reno.

In Berkeley-derived implementations, these two values really implement an inactivity timer and not an absolute timer on the read or write system call. Pages 496 and 516 of TCPv2 talk about this in more detail.

SO_REUSEADDR and SO_REUSEPORT Socket Options

The `SO_REUSEADDR` socket option serves four different purposes:

1. `SO_REUSEADDR` allows a listening server to start and `bind` its well-known port, even if previously established connections exist that use this port as their local port. This condition is typically encountered as follows:
    a. A listening server is started.
    b. A connection request arrives and a child process is spawned to handle that client.
    c. The listening server terminates, but the child continues to service the client on the existing connection.
    d. The listening server is restarted.

    By default, when the listening server is restarted in (d) by calling `socket`, `bind`, and `listen`, the call to `bind` fails because the listening server is trying to bind a port that is part of an existing connection (the one being handled by the previously spawned child). But if the server sets the `SO_REUSEADDR` socket option between the calls to `socket` and `bind`, the latter function will succeed. *All* TCP servers should specify this socket option to allow the server to be restarted in this situation.

2. This scenario is one of the most frequently asked questions on USENET.

3. `SO_REUSEADDR` allows a new server to be started on the same port as an existing server that is bound to the wildcard address, as long as each instance binds a different local IP address. This is common for a site hosting multiple HTTP servers using the IP alias technique (Section A.4). Assume the local host's primary IP address is 198.69.10.2 but it has two aliases: 198.69.10.128 and 198.69.10.129. Three HTTP servers are started. The first HTTP server would call `bind` with the wildcard as the local IP address and a local port of 80 (the well-known port for HTTP). The second server would call `bind` with a local IP address of 198.69.10.128 and a local port of 80. But, this second call to `bind` fails unless `SO_REUSEADDR` is set before the call. The third server would `bind` 198.69.10.129 and port 80. Again, `SO_REUSEADDR` is required for this final call to succeed. Assuming `SO_REUSEADDR` is set and the three servers are started, incoming TCP connection requests with a destination IP address of 198.69.10.128 and a destination port of 80 are delivered to the second server, incoming requests with a destination IP address of 198.69.10.129 and a destination port of 80 are delivered to the third server, and all other TCP connection requests with a destination port of 80 are delivered to the first server. This "default" server handles requests destined for 198.69.10.2 in addition to any other IP aliases that the host may have configured. The wildcard means "everything that doesn't have a better (more specific) match." Note that this scenario of allowing multiple servers for a given service is handled automatically if the server always sets the `SO_REUSEADDR` socket option (as we recommend).

   With TCP, we are never able to start multiple servers that `bind` the same IP address and the same port: a *completely duplicate binding*. That is, we cannot start one server that binds 198.69.10.2 port 80 and start another that also binds 198.69.10.2 port 80, even if we set the `SO_REUSEADDR` socket option for the second server.

   For security reasons, some operating systems prevent *any* "more specific" bind to a port that is already bound to the wildcard address, that is, the series of binds described here would not work with or without `SO_REUSEADDR`. On such a system, the server that performs the wildcard bind must be started last. This is to avoid the problem of a rogue server binding to an IP address and port that are being served already by a system service and intercepting legitimate requests. This is a particular problem for NFS, which generally does not use a privileged port.

4. `SO_REUSEADDR` allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address. This is common for UDP servers that need to know the destination IP address of client requests on systems that do not provide the `IP_RECVDSTADDR` socket option. This technique is normally not used with TCP servers since a TCP server can always determine the destination IP address by calling `getsockname` after the

connection is established. However, a TCP server wishing to serve connections to some, but not all, addresses belonging to a multihomed host should use this technique.

5. `SO_REUSEADDR` allows *completely duplicate bindings*: a `bind` of an IP address and port, when that same IP address and port are already bound to another socket, if the transport protocol supports it. Normally this feature is supported only for UDP sockets.

   This feature is used with multicasting to allow the same application to be run multiple times on the same host. When a UDP datagram is received for one of these multiply bound sockets, the rule is that if the datagram is destined for either a broadcast address or a multicast address, one copy of the datagram is delivered to each matching socket. But if the datagram is destined for a unicast address, the datagram is delivered to only one socket. If, in the case of a unicast datagram, there are multiple sockets that match the datagram, the choice of which socket receives the datagram is implementation-dependent. Pages *777☐779* of TCPv2 talk more about this feature. We will talk more about broadcasting and multicasting in Chapters 20 and 21.

Exercises 7.5 and 7.6 show some examples of this socket option.

4.4BSD introduced the `SO_REUSEPORT` socket option when support for multicasting was added. Instead of overloading `SO_REUSEADDR` with the desired multicast semantics that allow completely duplicate bindings, this new socket option was introduced with the following semantics:

1. This option allows completely duplicate bindings, but only if each socket that wants to bind the same IP address and port specify this socket option.
2. `SO_REUSEADDR` is considered equivalent to `SO_REUSEPORT` if the IP address being bound is a multicast address (p. 731 of TCPv2).

The problem with this socket option is that not all systems support it, and on those that do not support the option but do support multicasting, `SO_REUSEADDR` is used instead of `SO_REUSEPORT` to allow completely duplicate bindings when it makes sense (i.e., a UDP server that can be run multiple times on the same host at the same time and that expects to receive either broadcast or multicast datagrams).

We can summarize our discussion of these socket options with the following recommendations:

1. Set the `SO_REUSEADDR` socket option before calling `bind` in all TCP servers.
2. When writing a multicast application that can be run multiple times on the same host at the same time, set the `SO_REUSEADDR` socket option and bind the group's multicast address as the local IP address.

Chapter 22 of TCPv2 talks about these two socket options in more detail.

There is a potential security problem with `SO_REUSEADDR`. If a socket exists that is bound to, say, the wildcard address and port 5555, if we specify `SO_REUSEADDR`, we can bind that same port to a different IP address, say the primary IP address of the host. Any future datagrams that arrive destined to port 5555 and the IP address that we bound to our socket are delivered to our socket, not to the other socket bound to the wildcard address. These could be TCP SYN segments, SCTP INIT chunks, or UDP datagrams. (Exercises 11.9 shows this feature with UDP.) For most well-known services, HTTP, FTP, and Telnet, for example, this is not a problem because these servers all bind a reserved port. Hence, any process that comes along later and tries to bind a more specific instance of that port (i.e., steal the port) requires superuser privileges. NFS, however, can be a problem since its normal port (2049) is not reserved.

One underlying problem with the sockets API is that the setting of the socket pair is done with two function calls (`bind` and `connect`) instead of one. [Torek 1994] proposes a single function that solves this problem.

```
int bind_connect_listen(int sockfd, const struct sockaddr *laddr, int
laddrlen, const struct sockaddr *faddr, int faddrlen, int listen);
```

laddr specifies the local IP address and local port, *faddr* specifies the foreign IP address and foreign port, and *listen* specifies a client (zero) or a server (nonzero; same as the backlog argument to `listen`). Then, `bind` would be a library function that calls this function with *faddr* a null pointer and *faddrlen* 0, and `connect` would be a library function that calls this function with *laddr* a null pointer and *laddrlen* 0. There are a few applications, notably TFTP, that need to specify both the local pair and the foreign pair, and they could call `bind_connect_listen` directly. With such a function, the need for `SO_REUSEADDR` disappears, other than for multicast UDP servers that explicitly need to allow completely duplicate bindings of the same IP address and port. Another benefit of this new function is that a TCP server could restrict itself to servicing connection requests that arrive from one specific IP address and port, something which RFC 793 [Postel 1981c] specifies but is impossible to implement with the existing sockets API.

SO_TYPE Socket Option

This option returns the socket type. The integer value returned is a value such as `SOCK_STREAM` or `SOCK_DGRAM`. This option is typically used by a process that inherits a socket when it is started.

SO_USELOOPBACK Socket Option

This option applies only to sockets in the routing domain (AF_ROUTE). This option defaults to ON for these sockets (the only one of the SO_*xxx* socket options that defaults to ON instead of OFF). When this option is enabled, the socket receives a copy of everything sent on the socket.

Another way to disable these loopback copies is to call shutdown with a second argument of SHUT_RD.

7.6 IPv4 Socket Options

These socket options are processed by IPv4 and have a *level* of IPPROTO_IP. We defer discussion of the multicasting socket options until Section 21.6.

IP_HDRINCL Socket Option

If this option is set for a raw IP socket (Chapter 28), we must build our own IP header for all the datagrams we send on the raw socket. Normally, the kernel builds the IP header for datagrams sent on a raw socket, but there are some applications (notably traceroute) that build their own IP header to override values that IP would place into certain header fields.

When this option is set, we build a complete IP header, with the following exceptions:

- IP always calculates and stores the IP header checksum.
- If we set the IP identification field to 0, the kernel will set the field.
- If the source IP address is INADDR_ANY, IP sets it to the primary IP address of the outgoing interface.
- Setting IP options is implementation-dependent. Some implementations take any IP options that were set using the IP_OPTIONS socket option and append these to the header that we build, while others require our header to also contain any desired IP options.
- Some fields must be in host byte order, and some in network byte order. This is implementation-dependent, which makes writing raw packets with IP_HDRINCL not as portable as we'd like.

We show an example of this option in Section 29.7. Pages 1056□1057 of TCPv2 provide additional details on this socket option.

IP_OPTIONS Socket Option

Setting this option allows us to set IP options in the IPv4 header. This requires intimate knowledge of the format of the IP options in the IP header. We will discuss this option with regard to IPv4 source routes in Section 27.3.

IP_RECVDSTADDR Socket Option

This socket option causes the destination IP address of a received UDP datagram to be returned as ancillary data by recvmsg. We will show an example of this option in Section 22.2.

IP_RECVIF Socket Option

This socket option causes the index of the interface on which a UDP datagram is received to be returned as ancillary data by recvmsg. We will show an example of this option in Section 22.2.

IP_TOS Socket Option

This option lets us set the type-of-service (TOS) field (which contains the DSCP and ECN fields, Figure A.1) in the IP header for a TCP, UDP, or SCTP socket. If we call getsockopt for this option, the current value that would be placed into the DSCP and ECN fields in the IP header (which defaults to 0) is returned. There is no way to fetch the value from a received IP datagram.

An application can set the DSCP to a value negotiated with the network service provider to receive prearranged services, e.g., low delay for IP telephony or higher throughput for bulk data transfer. The diffserv architecture, defined in RFC 2474 [Nichols et al. 1998], provides for only limited backward compatibility with the historical TOS field definition (from RFC 1349 [Almquist 1992]). Application that set IP_TOS to one of the contents from <netinet/ip.h>, for instance, IPTOS_LOWDELAY or IPTOS_THROUGHPUT, should instead use a user-specified DSCP value. The only TOS values that diffserv retains are precedence levels 6 ("internetwork control") and 7 ("network control"); this means that applications that set IP_TOS to IPTOS_PREC_NETCONTROL or IPTOS_PREC_INTERNETCONTROL *will* work in a diffserv network.

RFC 3168 [Ramakrishnan, Floyd, and Black 2001] contains the definition of the ECN field. Applications should generally leave the setting of the ECN field to the kernel, and should specify zero values in the low two bits of the value set with IP_TOS.

IP_TTL Socket Option

With this option, we can set and fetch the default TTL (Figure A.1) that the system will use for unicast packets sent on a given socket. (The multicast TTL is set using the IP_MULTICAST_TTL socket option, described in Section 21.6.) 4.4BSD, for example, uses the default of 64 for both TCP and UDP sockets (specified in the IANA's "IP Option Numbers" registry [IANA]) and 255 for raw sockets. As with the TOS field, calling getsockopt returns the default value of the field that the system will use in outgoing datagrams□there is no way to obtain the value from a received datagram. We will set this socket option with our traceroute program in Figure 28.19.

7.7 ICMPv6 Socket Option

This option lets us fetch and set an `icmp6_filter` structure that specifies which of the 256 possible ICMPv6 message types will be passed to the process on a raw socket. We will discuss this option in Section 28.4.

7.8 IPv6 Socket Options

These socket options are processed by IPv6 and have a *level* of `IPPROTO_IPV6`. We defer discussion of the multicasting socket options until Section 21.6. We note that many of these options make use of *ancillary data* with the `recvmsg` function, and we will describe this in Section 14.6. All the IPv6 socket options are defined in RFC 3493 [Gilligan et al. 2003] and RFC 3542 [Stevens et al. 2003].

IPV6_CHECKSUM Socket Option

This socket option specifies the byte offset into the user data where the checksum field is located. If this value is non-negative, the kernel will: (i) compute and store a checksum for all outgoing packets, and (ii) verify the received checksum on input, discarding packets with an invalid checksum. This option affects all IPv6 raw sockets, except ICMPv6 raw sockets. (The kernel always calculates and stores the checksum for ICMPv6 raw sockets.) If a value of -1 is specified (the default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

All protocols that use IPv6 should have a checksum in their own protocol header. These checksums include a pseudoheader (RFC 2460 [Deering and Hinden 1998]) that includes the source IPv6 address as part of the checksum (which differs from all the other protocols that are normally implemented using a raw socket with IPv4). Rather than forcing the application using the raw socket to perform source address selection, the kernel will do this and then calculate and store the checksum incorporating the standard IPv6 pseudoheader.

IPV6_DONTFRAG Socket Option

Setting this option disables the automatic insertion of a fragment header for UDP and raw sockets. When this option is set, output packets larger than the MTU of the outgoing interface will be dropped. No error needs to be returned from the system call that sends the packet, since the packet might exceed the path MTU en-route. Instead, the application should enable the `IPV6_RECVPATHMTU` option (Section 22.9) to learn about path MTU changes.

IPV6_NEXTHOP Socket Option

This option specifies the next-hop address for a datagram as a socket address structure, and is a privileged operation. We will say more about this feature in Section 22.8.

IPV6_PATHMTU Socket Option

This option cannot be set, only retrieved. When this option is retrieved, the current MTU as determined by path-MTU discovery is returned (see Section 22.9).

IPV6_RECVDSTOPTS Socket Option

Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by `recvmsg`. This option defaults to OFF. We will describe the functions that are used to build and process these options in Section 27.5.

IPV6_RECVHOPLIMIT Socket Option

Setting this option specifies that the received hop limit field is to be returned as ancillary data by `recvmsg`. This option defaults to OFF. We will describe this option in Section 22.8.

There is no way with IPv4 to obtain the received TTL field.

IPV6_RECVHOPOPTS Socket Option

Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by `recvmsg`. This option defaults to OFF. We will describe the functions that are used to build and process these options in Section 27.5.

IPV6_RECVPATHMTU Socket Option

Setting this option specifies that the path MTU of a path is to be returned as ancillary data by `recvmsg` (without any accompanying data) when it changes. We will describe this option in Section 22.9.

IPV6_RECVPKTINFO Socket Option

Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by `recvmsg`: the destination IPv6 address and the arriving interface index. We will describe this option in Section 22.8.

IPV6_RECVRTHDR Socket Option

Setting this option specifies that a received IPv6 routing header is to be returned as ancillary data by `recvmsg`. This option defaults to OFF. We will describe the functions that are used to build and process an IPv6 routing header in Section 27.6.

IPV6_RECVTCLASS Socket Option

Setting this option specifies that the received traffic class (containing the DSCP and ECN fields) is to be returned as ancillary data by `recvmsg`. This option defaults to OFF. We will describe this option in Section 22.8.

IPV6_UNICAST_HOPS Socket Option

This IPv6 option is similar to the IPv4 `IP_TTL` socket option. Setting the socket option specifies the default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value for the hop limit that the kernel will use for the socket. The actual hop limit field from a received IPv6 datagram is obtained by using the `IPV6_RECVHOPLIMIT` socket option. We will set this socket option with our `traceroute` program in Figure 28.19.

IPV6_USE_MIN_MTU Socket Option

Setting this option to 1 specifies that path MTU discovery is not to be performed and that packets are sent using the minimum IPv6 MTU to avoid fragmentation. Setting it to 0 causes path MTU discovery to occur for all destinations. Setting it to☐1 specifies that path MTU discovery is performed for unicast destinations but the minimum MTU is used when sending to multicast destinations. This option defaults to ☐1. We will describe this option in Section 22.9.

IPV6_V6ONLY Socket Option

Setting this option on an `AF_INET6` socket restricts it to IPv6 communication only. This option defaults to OFF, although some systems have an option to turn it ON by default. We will describe IPv4 and IPv6 communication using `AF_INET6` sockets in Sections 12.2 and 12.3.

IPV6_XXX Socket Options

Most of the IPv6 options for header modification assume a UDP socket with information being passed between the kernel and the application using ancillary data with `recvmsg` and `sendmsg`. A TCP socket fetches and stores these values using `getsockopt` and `setsockopt` instead. The socket option is the same as the type of the ancillary data, and the buffer contains the same information as would be present in the ancillary data. We will describe this in Section 27.7.

7.9 TCP Socket Options

There are two socket options for TCP. We specify the *level* as `IPPROTO_TCP`.

TCP_MAXSEG Socket Option

This socket option allows us to fetch or set the MSS for a TCP connection. The value returned is the maximum amount of data that our TCP will send to the other end; often, it is the MSS announced by the other end with its SYN, unless our TCP chooses to use a smaller value than the peer's announced MSS. If this value is fetched before the socket is connected, the value returned is the default value that will be used if an MSS option is not received from the other end. Also be aware that a value smaller than the returned value can actually be used for the connection if the timestamp option, for example, is in use, because this option occupies 12 bytes of TCP options in each segment.

The maximum amount of data that our TCP will send per segment can also change during the life of a connection if TCP supports path MTU discovery. If the route to the peer changes, this value can go up or down.

We note in Figure 7.1 that this socket option can also be set by the application. This is not possible on all systems; it was originally a read-only option. 4.4BSD limits the application to *decreasing* the value: We cannot increase the value (p. 1023 of TCPv2). Since this option controls the amount of data that TCP sends per segment, it makes sense to forbid the application from increasing the value. Once the connection is established, this value is the MSS option announced by the peer, and we cannot exceed that value. Our TCP, however, can always send less than the peer's announced MSS.

TCP_NODELAY Socket Option

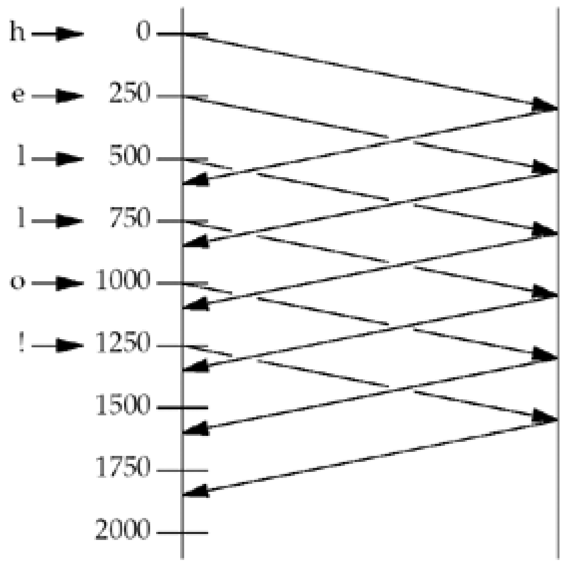
If set, this option disables TCP's *Nagle algorithm* (Section 19.4 of TCPv1 and pp. 858□859 of TCPv2). By default, this algorithm is enabled.

The purpose of the Nagle algorithm is to reduce the number of small packets on a WAN. The algorithm states that if a given connection has outstanding data (i.e., data that our TCP has sent, and for which it is currently awaiting an acknowledgment), then no small packets will be sent on the connection in response to a user write operation until the existing data is acknowledged. The definition of a "small" packet is any packet smaller than the MSS. TCP will always send a full-sized packet if possible; the purpose of the Nagle algorithm is to prevent a connection from having multiple small packets outstanding at any time.

The two common generators of small packets are the Rlogin and Telnet clients, since they normally send each keystroke as a separate packet. On a fast LAN, we normally do not notice the Nagle algorithm with these clients, because the time required for a small packet to be acknowledged is typically a few milliseconds□far less than the time between two successive characters that we type. But on a WAN, where it can take a second for a small packet to be acknowledged, we can notice a delay in the character echoing, and this delay is often exaggerated by the Nagle algorithm.

Consider the following example: We type the six-character string "hello!" to either an Rlogin or Telnet client, with exactly 250 ms between each character. The RTT to the server is 600 ms and the server immediately sends back the echo of each character. We assume the ACK of the client's character is sent back to the client along with the character echo and we ignore the ACKs that the client sends for the server's echo. (We will talk about delayed ACKs shortly.) Assuming the Nagle algorithm is disabled, we have the 12 packets shown in Figure 7.14.

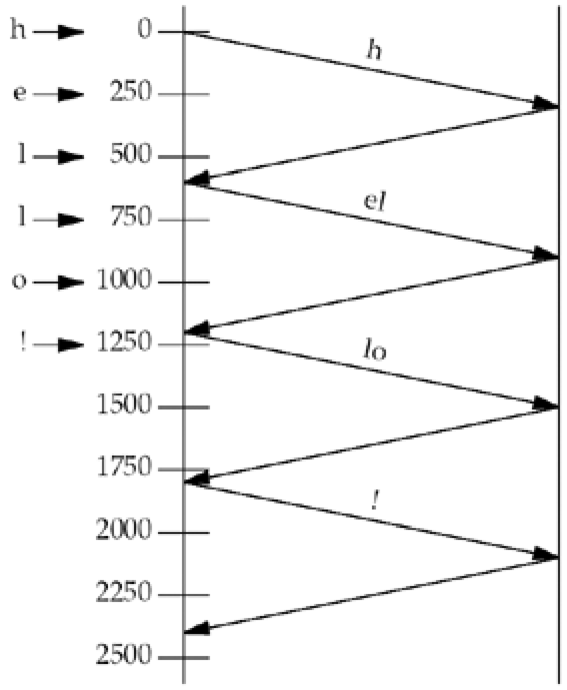
Figure 7.14. Six characters echoed by server with Nagle algorithm disabled.



Each character is sent in a packet by itself: the data segments from left to right, and the ACKs from right to left.

If the Nagle algorithm is enabled (the default), we have the eight packets shown in Figure 7.15. The first character is sent as a packet by itself, but the next two characters are not sent, since the connection has a small packet outstanding. At time 600, when the ACK of the first packet is received, along with the echo of the first character, these two characters are sent. Until this packet is ACKed at time 1200, no more small packets are sent.

Figure 7.15. Six characters echoed by server with Nagle algorithm enabled.



The Nagle algorithm often interacts with another TCP algorithm: the *delayed ACK* algorithm. This algorithm causes TCP to not send an ACK immediately when it receives data; instead, TCP will wait some small amount of time (typically 50□200 ms) and only then send the ACK. The hope is that in this small amount of time, there will be data to send back to the peer, and the ACK can piggyback with the data, saving one TCP segment. This is normally the case with the Rlogin and Telnet clients, because the servers typically echo each character sent by the client, so the ACK of the client's character piggybacks with the server's echo of that character.

The problem is with other clients whose servers do not generate traffic in the reverse direction on which ACKs can piggyback. These clients can detect noticeable delays because the client TCP will not send any data to the server until the server's delayed ACK timer expires. These clients need a way to disable the Nagle algorithm, hence the `TCP_NODELAY` option.

Another type of client that interacts badly with the Nagle algorithm and TCP's delayed ACKs is a client that sends a single logical request to its server in small pieces. For example, assume a client sends a 400-byte request to its server, but this is a 4-byte request type followed by 396 bytes of request data. If the client performs a 4-byte `write` followed by a 396-byte `write`, the second write will not be sent by the client TCP until the server TCP acknowledges the 4-byte write. Also, since the server application cannot operate on the 4 bytes of data until it receives the remaining 396 bytes of data, the server TCP will delay the ACK of the 4 bytes of data (i.e., there will not be any data from the server to the client on which to piggyback the ACK). There are three ways to fix this type of client:

1. Use writev (Section 14.4) instead of two calls to write. A single call to writev ends up with one call to TCP output instead of two calls, resulting in one TCP segment for our example. This is the preferred solution.
2. Copy the 4 bytes of data and the 396 bytes of data into a single buffer and call write once for this buffer.
3. Set the TCP_NODELAY socket option and continue to call write two times. This is the least desirable solution, and is harmful to the network, so it generally should not even be considered.

Exercises 7.8 and 7.9 continue this example.

7.10 SCTP Socket Options

The relatively large number of socket options for SCTP (17 at present writing) reflects the finer grain of control SCTP provides to the application developer. We specify the *level* as IPPROTO_SCTP.

Several options used to get information about SCTP require that data be passed into the kernel (e.g., association ID and/or peer address). While some implementations of getsockopt support passing data both into and out of the kernel, not all do. The SCTP API defines a sctp_opt_info function (Section 9.11) that hides this difference. On systems on which getsockopt does support this, it is simply a wrapper around getsockopt. Otherwise, it performs the required action, perhaps using a custom ioctl or a new system call. We recommend always using sctp_opt_info when retrieving these options for maximum portability. These options are marked with a dagger (†) in Figure 7.2 and include SCTP_ASSOCINFO, SCTP_GET_PEER_ADDR_INFO, SCTP_PEER_ADDR_PARAMS, SCTP_PRIMARY_ADDR, SCTP_RTOINFO, and SCTP_STATUS.

SCTP_ADAPTION_LAYER Socket Option

During association initialization, either endpoint may specify an adaption layer indication. This indication is a 32-bit unsigned integer that can be used by the two applications to coordinate any local application adaption layer. This option allows the caller to fetch or set the adaption layer indication that this endpoint will provide to peers.

When fetching this value, the caller will only retrieve the value the local socket will provide to all future peers. To retrieve the peer's adaption layer indication, an application must subscribe to adaption layer events.

SCTP_ASSOCINFO Socket Option

The SCTP_ASSOCINFO socket option can be used for three purposes: (i) to retrieve information about an existing association, (ii) to change the parameters of an existing association, and/or (iii) to set defaults for future associations. When retrieving information about an existing association, the `sctp_opt_info` function should be used instead of `getsockopt`. This option takes as input the `sctp_assocparams` structure.

```
struct sctp_assocparams {
  sctp_assoc_t sasoc_assoc_id;
  u_int16_t sasoc_asocmaxrxt;
  u_int16_t sasoc_number_peer_destinations;
  u_int32_t sasoc_peer_rwnd;
  u_int32_t sasoc_local_rwnd;
  u_int32_t sasoc_cookie_life;
};
```

These fields have the following meaning:

- `sasoc_assoc_id` holds the identification for the association of interest. If this value is set to 0 when calling the `setsockopt` function, then `sasoc_asocmaxrxt` and `sasoc_cookie_life` represent values that are to be set as defaults on the socket. Calling `getsockopt` will return association-specific information if the association ID is supplied; otherwise, if this field is 0, the default endpoint settings will be returned.
- `sasoc_asocmaxrxt` holds the maximum number of retransmissions an association will make without acknowledgment before giving up, reporting the peer unusable and closing the association.
- `sasoc_number_peer_destinations` holds the number of peer destination addresses. It cannot be set, only retrieved.
- `sasoc_peer_rwnd` holds the peer's current calculated receive window. This value represents the total number of data bytes that can yet be sent. This field is dynamic; as the local endpoint sends data, this value decreases. As the remote application reads data that has been received, this value increases. This value cannot be changed by this socket option call.
- `sasoc_local_rwnd` represents the local receive window the SCTP stack is currently reporting to the peer. This value is dynamic as well and is influenced by the `SO_SNDBUF` socket option. This value cannot be changed by this socket option call.
- `sasoc_cookie_life` represents the number of milliseconds for which a cookie, given to a remote peer, is valid. Each state cookie sent to a peer has a lifetime associated with it to prevent replay attacks. The default value of 60,000 milliseconds can be changed by setting this option with a `sasoc_assoc_id` value of 0.

We will provide advice on tuning the value of `sasoc_asocmaxrxt` for performance in Section 23.11. The `sasoc_cookie_life` can be reduced for greater protection against cookie replay attacks but less robustness to network delay during association initiation. The other values are useful for debugging.

SCTP_AUTOCLOSE Socket Option

This option allows us to fetch or set the autoclose time for an SCTP endpoint. The autoclose time is the number of seconds an SCTP association will remain open when idle. Idle is defined by the SCTP stack as neither endpoint sending or receiving user data. The default is for the autoclose function to be disabled.

The autoclose option is intended to be used in the one-to-many-style SCTP interface (Chapter 9). When this option is set, the integer passed to the option is the number of seconds before an idle connection should be closed; a value of 0 disables autoclose. Only future associations created by this endpoint will be affected by this option; existing associations retain their current setting.

Autoclose can be used by a server to force the closing of idle associations without the server needing to maintain additional state. A server using this feature needs to carefully assess the longest idle time expected on all its associations. Setting the autoclose value smaller than needed results in the premature closing of associations.

SCTP_DEFAULT_SEND_PARAM Socket Option

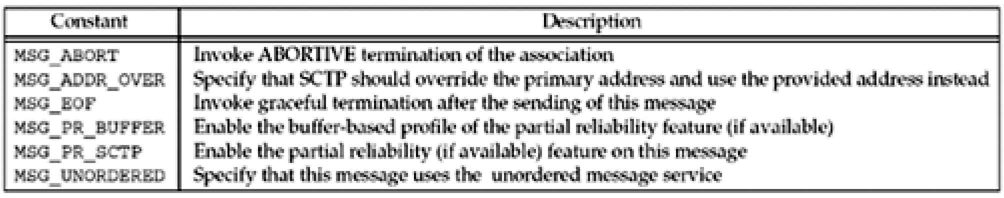
SCTP has many optional send parameters that are often passed as ancillary data or used with the `sctp_sendmsg` function call (which is often implemented as a library call that passes ancillary data for the user). An application that wishes to send a large number of messages, all with the same parameters, can use this option to set up the default parameters and thus avoid using ancillary data or the `sctp_sendmsg` call. This option takes as input the `sctp_sndrcvinfo` structure.

```
struct sctp_sndrcvinfo {
  u_int16_t sinfo_stream;
  u_int16_t sinfo_ssn;
  u_int16_t sinfo_flags;
  u_int32_t sinfo_ppid;
  u_int32_t sinfo_context;
  u_int32_t sinfo_timetolive;
  u_int32_t sinfo_tsn;
  u_int32_t sinfo_cumtsn;
  sctp_assoc_t sinfo_assoc_id;
};
```

These fields are defined as follows:

- `sinfo_stream` specifies the new default stream to which all messages will be sent.
- `sinfo_ssn` is ignored when setting the default options. When receiving a message with the `recvmsg` function or `sctp_recvmsg` function, this field will hold the value the peer placed in the stream sequence number (SSN) field in the SCTP DATA chunk.
- `sinfo_flags` dictates the default flags to apply to all future message sends. Allowable flag values can be found in Figure 7.16.

Figure 7.16. Allowable SCTP flag values for the sinfo_flags field.



Constant	Description
MSG_ABORT	Invoke ABORTIVE termination of the association
MSG_ADDR_OVER	Specify that SCTP should override the primary address and use the provided address instead
MSG_EOF	Invoke graceful termination after the sending of this message
MSG_PR_BUFFER	Enable the buffer-based profile of the partial reliability feature (if available)
MSG_PR_SCTP	Enable the partial reliability (if available) feature on this message
MSG_UNORDERED	Specify that this message uses the unordered message service

- `sinfo_pid` provides the default value to use when setting the SCTP payload protocol identifier in all data transmissions.
- `sinfo_context` specifies the default value to place in the `sinfo_context` field, which is provided as a local tag when messages that could not be sent to a peer are retrieved.
- `sinfo_timetolive` dictates the default lifetime that will be applied to all message sends. The lifetime field is used by SCTP stacks to know when to discard an outgoing message due to excessive delay (prior to its first transmission). If the two endpoints support the partial reliability option, then the lifetime is also used to specify how long a message is valid after its first transmission.
- `sinfo_tsn` is ignored when setting the default options. When receiving a message with the `recvmsg` function or `sctp_recvmsg` function, this field will hold the value the peer placed in the transport sequence number (TSN) field in the SCTP DATA chunk.
- `sinfo_cumtsn` is ignored when setting the default options. When receiving a message with the `recvmsg` function or `sctp_recvmsg` function, this field will hold the current cumulative TSN the local SCTP stack has associated with its remote peer.
- `sinfo_assoc_id` specifies the association identification that the requester wishes the default parameters to be set against. For one-to-one sockets, this field is ignored.

Note that all default settings will only affect messages sent without their own `sctp_sndrcvinfo` structure. Any send that provides this structure (e.g., `sctp_sendmsg` or `sendmsg` function with ancillary data) will override the default settings. Besides setting the default values, this option may be used to retrieve the current default parameters by using the `sctp_opt_info` function.

SCTP_DISABLE_FRAGMENTS Socket Option

SCTP normally fragments any user message that does not fit in a single SCTP packet into multiple DATA chunks. Setting this option disables this behavior on the sender. When disabled by this option, SCTP will return the error `EMSGSIZE` and not send the message. The default behavior is for this option to be disabled; SCTP will normally fragment user messages.

This option may be used by applications that wish to control message sizes, ensuring that every user application message will fit in a single IP packet. An application that enables this option must be prepared to handle the error case (i.e., its message was too big) by either providing application-layer fragmentation of the message or a smaller message.

SCTP_EVENTS Socket Option

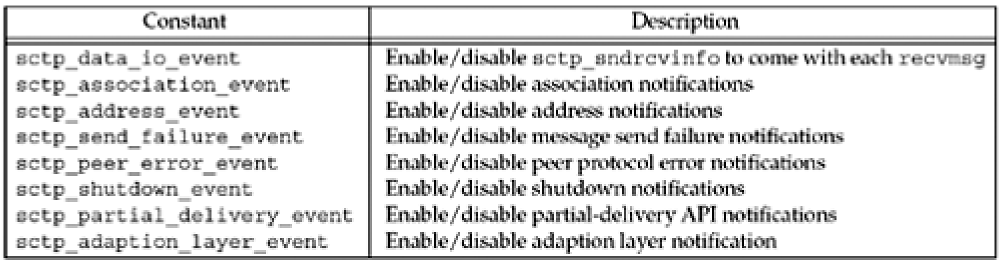
This socket option allows a caller to fetch, enable, or disable various SCTP notifications. An SCTP notification is a message that the SCTP stack will send to the application. The message is read as normal data, with the `msg_flags` field of the `recvmsg` function being set to `MSG_NOTIFICATION`. An application that is not prepared to use either `recvmsg` or `sctp_recvmsg` should not enable events. Eight different types of events can be subscribed to by using this option and passing an `sctp_event_subscribe` structure. Any value of 0 represents a non-subscription and a value of 1 represents a subscription.

The `sctp_event_subscribe` structure takes the following form:

```
struct sctp_event_subscribe {
  u_int8_t sctp_data_io_event;
  u_int8_t sctp_association_event;
  u_int8_t sctp_address_event;
  u_int8_t sctp_send_failure_event;
  u_int8_t sctp_peer_error_event;
  u_int8_t sctp_shutdown_event;
  u_int8_t sctp_partial_delivery_event;
  u_int8_t sctp_adaption_layer_event;
};
```

Figure 7.17 summarizes the various events. Further details on notifications can be found in Section 9.14.

Figure 7.17. SCTP event subscriptions.



Constant	Description
sctp_data_io_event	Enable/disable sctp_sndrcvinfo to come with each recvmsg
sctp_association_event	Enable/disable association notifications
sctp_address_event	Enable/disable address notifications
sctp_send_failure_event	Enable/disable message send failure notifications
sctp_peer_error_event	Enable/disable peer protocol error notifications
sctp_shutdown_event	Enable/disable shutdown notifications
sctp_partial_delivery_event	Enable/disable partial-delivery API notifications
sctp_adaption_layer_event	Enable/disable adaption layer notification

SCTP_GET_PEER_ADDR_INFO Socket Option

This option retrieves information about a peer address, including the congestion window, smoothed RTT and MTU. This option may only be used to retrieve information about a specific peer address. The caller provides a `sctp_paddrinfo` structure with the `spinfo_address` field filled in with the peer address of interest, and should use `sctp_opt_info` instead of `getsockopt` for maximum portability. The sctp_paddrinfo structure has the following format:

```
struct sctp_paddrinfo {
  sctp_assoc_t spinfo_assoc_id;
  struct sockaddr_storage spinfo_address;
  int32_t spinfo_state;
  u_int32_t spinfo_cwnd;
  u_int32_t spinfo_srtt;
  u_int32_t spinfo_rto;
  u_int32_t spinfo_mtu;
};
```

The data returned to the caller provides the following:

- `spinfo_assoc_id` contains association identification information, also provided in the "communication up" notification (`SCTP_COMM_UP`). This unique value can be used as a shorthand method to represent the association for almost all SCTP operations.

- `spinfo_address` is set by the caller to inform the SCTP socket on which address to return information. On return, its value should be unchanged.
- `spinfo_state` holds one or more of the values seen in Figure 7.18.

Figure 7.18. SCTP peer address states.

Constant	Description
`SCTP_ACTIVE`	Address is active and reachable
`SCTP_INACTIVE`	Address cannot currently be reached
`SCTP_ADDR_UNCONFIRMED`	No heartbeat or data has confirmed this address

An *unconfirmed address* is one that the peer had listed as a valid address, but the local SCTP endpoint has not been able to confirm that the peer holds that address. An SCTP endpoint confirms an address when a heartbeat or user data, sent to that address, is acknowledged. Note that an unconfirmed address will also not have a valid retransmission timeout (RTO) value. Active addresses represent addresses that are considered available for use.

- `spinfo_cwnd` represents the current congestion window recorded for the peer address. A description of how the the `cwnd` value is managed can be found on page 177 of [Stewart and Xie 2001].
- `spinfo_srtt` represents the current estimate of the smoothed RTT for this address.
- `spinfo_rto` represents the current retransmission timeout in use for this address.
- `spinfo_mtu` represents the current path MTU as discovered by path MTU discovery.

One interesting use for this option is to translate an IP address structure into an association identification that can be used in other calls. We will illustrate the use of this socket option in Chapter 23. Another possibility is for the application to track performance to each address of a multihomed peer and update the primary address of the association to the peer's best address. These values are also useful for logging and debugging.

SCTP_I_WANT_MAPPED_V4_ADDR Socket Option

This flag can be used to enable or disable IPv4-mapped addresses on an `AF_INET6`-type socket. Note that when enabled (which is the default behavior), all IPv4 addresses will be mapped to a IPv6 address before sending to the application. If this option is disabled, the SCTP socket will *not* map IPv4 addresses and will instead pass them as a `sockaddr_in` structure.

SCTP_INITMSG Socket Option

This option can be used to get or set the default initial parameters used on an SCTP socket when sending out the INIT message. The option uses the `sctp_initmsg` structure, which is defined as:

```
struct sctp_initmsg {
  uint16_t sinit_num_ostreams;
  uint16_t sinit_max_instreams;
  uint16_t sinit_max_attempts;
  uint16_t sinit_max_init_timeo;
};
```

These fields are defined as follows:

- `sinit_num_ostreams` represents the number of outbound SCTP streams an application would like to request. This value is not confirmed until after the association finishes the initial handshake, and may be negotiated downward via peer endpoint limitations.
- `sinit_max_instreams` represents the maximum number of inbound streams the application is prepared to allow. This value will be overridden by the SCTP stack if it is greater than the maximum allowable streams the SCTP stack supports.
- `sinit_max_attempts` expresses how many times the SCTP stack should send the initial INIT message before considering the peer endpoint unreachable.
- `sinit_max_init_timeo` represents the maximum RTO value for the INIT timer. During exponential backoff of the initial timer, this value replaces `RTO.max` as the ceiling for retransmissions. This value is expressed in milliseconds.

Note that when setting these fields, any value set to 0 will be ignored by the SCTP socket. A user of the one-to-many-style socket (described in Section 9.2) may also pass an `sctp_initmsg` structure in ancillary data during implicit association setup.

SCTP_MAXBURST Socket Option

This socket option allows the application to fetch or set the *maximum burst size* used when sending packets. When an SCTP implementation sends data to a peer, no more than `SCTP_MAXBURST` packets are sent at once to avoid flooding the network with packets. An implementation may apply this limit by either: (i) reducing its congestion window to the current flight size plus the maximum burst size times the path MTU, or (ii) using this value as a separate micro-control, sending at most maximum burst packets at any single send opportunity.

SCTP_MAXSEG Socket Option

This socket option allows the application to fetch or set the *maximum fragment size* used during SCTP fragmentation. This option is similar to the TCP option `TCP_MAXSEG` described in Section 7.9.

When an SCTP sender receives a message from an application that is larger than this value, the SCTP sender will break the message into multiple pieces for transport to the peer endpoint. The size that the SCTP sender normally uses is the smallest MTU of all addresses associated with the peer. This option overrides this value downward to the value specified. Note that the SCTP stack may fragment a message at a smaller boundary than requested by this option. This smaller fragmentation will occur when one of the paths to the peer endpoint has a smaller MTU than the value requested in the `SCTP_MAXSEG` option.

This value is an endpoint-wide setting and may affect more than one association in the one-to-many interface style.

SCTP_NODELAY Socket Option

If set, this option disables SCTP's *Nagle algorithm*. This option is OFF by default (i.e., the Nagle algorithm is ON by default). SCTP's Nagle algorithm works identically to TCP's except that it is trying to coalesce multiple DATA chunks as opposed to simply coalescing bytes on a stream. For a further discussion of the Nagle algorithm, see `TCP_MAXSEG`.

SCTP_PEER_ADDR_PARAMS Socket Option

This socket option allows an application to fetch or set various parameters on an association. The caller provides the `sctp_paddrparams` structure, filling in the association identification. The `sctp_paddrparams` structure has the following format:

```
struct sctp_paddrparams {
  sctp_assoc_t spp_assoc_id;
  struct sockaddr_storage spp_address;
  u_int32_t spp_hbinterval;
  u_int16_t spp_pathmaxrxt;
};
```

These fields are defined as follows:

- `spp_assoc_id` holds the association identification for the information being requested or set. If this value is set to 0, the endpoint default values are set or retrieved instead of the association-specific values.
- `spp_address` specifies the IP address for which these parameters are being requested or set. If the `spp_assoc_id` field is set to 0, then this field is ignored.
- `spp_hbinterval` is the interval between heartbeats. A value of `SCTP_NO_HB` disables heartbeats. A value of `SCTP_ISSUE_HB` requests an on-demand heartbeat. Any other value changes the heartbeat interval to this value in milliseconds. When setting the default parameters, the value of `SCTP_ISSUE_HB` is not allowed.
- `spp_hbpathmaxrxt` holds the number of retransmissions that will be attempted on this destination before it is declared INACTIVE. When an address is declared INACTIVE, if it is the primary address, an alternate address will be chosen as the primary.

SCTP_PRIMARY_ADDR Socket Option

This socket option fetches or sets the address that the local endpoint is using as primary. The primary address is used, by default, as the destination address for all messages sent to a peer. To set this value, the caller fills in the association identification and the peer's address that should be used as the primary address. The caller passes this information in a `sctp_setprim` structure, which is defined as:

```
struct sctp_setprim {
  sctp_assoc_t            ssp_assoc_id;
  struct sockaddr_storage ssp_addr;
};
```

These fields are defined as follows:

- `spp_assoc_id` specifies the association identification on which the requester wishes to set or retrieve the current primary address. For the one-to-one style, this field is ignored.
- `sspp_addr` specifies the primary address, which must be an address belonging to the peer. If the operation is a `setsockopt` function call, then the value in this field represents the new peer address the requester would like to be made into the primary destination address.

Note that retrieving the value of this option on a one-to-one socket that has only one local address associated with it is the same as calling `getsockname`.

SCTP_RTOINFO Socket Option

This socket option can be used to fetch or set various RTO information on a specific association or the default values used by this endpoint. When fetching, the caller should use `sctp_opt_info` instead of `getsockopt` for maximum portability. The caller provides a `sctp_rtoinfo` structure of the following form:

```
struct sctp_rtoinfo {
  sctp_assoc          srto_assoc_id;
  uint32_t            srto_initial;
  uint32_t            srto_max;
  uint32_t            srto_min;
};
```

These fields are defined as follows:

- `srto_assoc_id` holds either the specific association of interest or 0. If this field contains the value 0, then the system's default parameters are affected by the call.
- `srto_initial` contains the initial RTO value used for a peer address. The initial RTO is used when sending an INIT chunk to the peer. This value is in milliseconds and has a default value of 3,000.
- `srto_max` contains the maximum RTO value that will be used when an update is made to the retransmission timer. If the updated value is larger than the RTO maximum, then the RTO maximum is used as the RTO instead of the calculated value. The default value for this field is 60,000 milliseconds.
- `srto_min` contains the minimum RTO value that will be used when starting a retransmission timer. Anytime an update is made to the RTO timer, the RTO minimum value is checked against the new value. If the new value is smaller than the minimum, the minimum replaces the new value. The default value for this field is 1,000 milliseconds.

A value of 0 for `srto_initial`, `srto_max`, or `srto_min` indicates that the default value currently set should not be changed. All time values are expressed in milliseconds. We provide guidance on setting these timers for performance in Section 23.11.

SCTP_SET_PEER_PRIMARY_ADDR Socket Option

Setting this option causes a message to be sent that requests that the peer set the specified local address as its primary address. The caller provides an sctp_setpeerprim structure and must fill in both the association identification and a local address to request the peer mark as its primary. The address provided must be one of the local endpoint's bound addresses. The sctp_setpeerprim structure is defined as follows:

```
struct sctp_setpeerprim {
  sctp_assoc_t           sspp_assoc_id;
  struct sockaddr_storage sspp_addr;
};
```

These fields are defined as follows:

- sspp_assoc_id specifies the association identification on which the requester wishes to set the primary address. For the one-to-one style, this field is ignored.
- sspp_addr holds the local address that the requester wishes to ask the peer system to set as the primary address.

This feature is optional, and must be supported by both endpoints to operate. If the local endpoint does not support the feature, an error of EOPNOTSUPP will be returned to the caller. If the remote endpoint does not support the feature, an error of EINVAL will be returned to the caller. Note that this value may only be set and cannot be retrieved.

SCTP_STATUS Socket Option

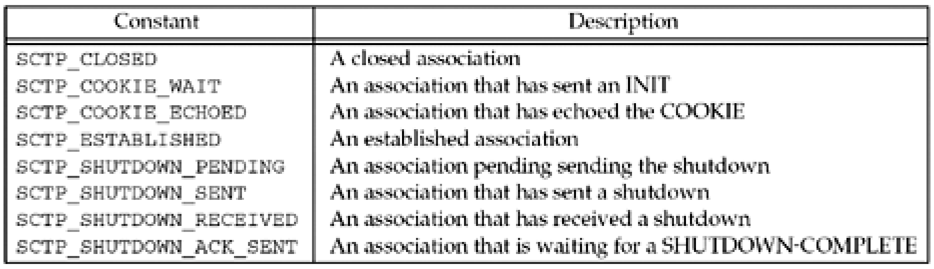
This socket option will retrieve the current state of an SCTP association. The caller should use sctp_opt_info instead of getaddrinfo for maximum portability. The caller provides an sctp_status structure, filling in the association identification field, sstat_assoc_id. The structure will be returned filled in with the information pertaining to the requested association. The sctp_status structure has the following format:

```
struct sctp_status {
  sctp_assoc_t sstat_assoc_id;
  int32_t sstat_state;
  u_int32_t sstat_rwnd;
  u_int16_t sstat_unackdata;
  u_int16_t sstat_penddata;
  u_int16_t sstat_instrms;
  u_int16_t sstat_outstrms;
  u_int32_t sstat_fragmentation_point;
  struct sctp_paddrinfo sstat_primary;
};
```

These fields are defined as follows:

- `sstat_assoc_id` holds the association identification.
- `sstat_state` holds one of the values found in Figure 7.19 and indicates the overall state of the association. A detailed depiction of the states an SCTP endpoint goes through during association setup or shutdown can be found in Figure 2.8.

Figure 7.19. SCTP states.



Constant	Description
SCTP_CLOSED	A closed association
SCTP_COOKIE_WAIT	An association that has sent an INIT
SCTP_COOKIE_ECHOED	An association that has echoed the COOKIE
SCTP_ESTABLISHED	An established association
SCTP_SHUTDOWN_PENDING	An association pending sending the shutdown
SCTP_SHUTDOWN_SENT	An association that has sent a shutdown
SCTP_SHUTDOWN_RECEIVED	An association that has received a shutdown
SCTP_SHUTDOWN_ACK_SENT	An association that is waiting for a SHUTDOWN-COMPLETE

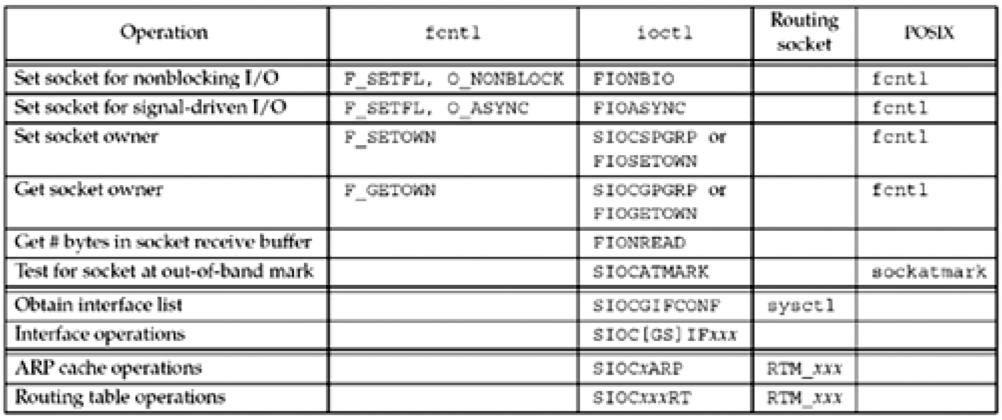
- `sstat_rwnd` holds our endpoint's current estimate of the peer's receive window.
- `sstat_unackdata` holds the number of unacknowledged DATA chunks pending for the peer.
- `sstat_penddata` holds the number of unread DATA chunks that the local endpoint is holding for the application to read.
- `sstat_instrms` holds the number of streams the peer is using to send data to this endpoint.
- `sstat_outstrms` holds the number of allowable streams that this endpoint can use to send data to the peer.
- `sstat_fragmentation_point` contains the current value the local SCTP endpoint is using as the fragmentation point for user messages. This value is normally the smallest MTU of all destinations, or possibly a smaller value set by the local application with `SCTP_MAXSEG`.
- `sstat_primary` holds the current primary address. The primary address is the default address used when sending data to the peer endpoint.

These values are useful for diagnostics and for determining the characteristics of the session; for example, the `sctp_get_no_strms` function in Section 10.2 will use the sstat_outstrms member to determine how many streams are available for outbound use. A low sstat_rwnd and/or a high sstat_unackdata value can be used to determine that the peer's receive socket buffer is becoming full, which can be used as a cue to the application to slow down transmission if possible. The sstat_fragmentation_point can be used by some applications to reduce the number of fragments that SCTP has to create, by sending smaller application messages.

7.11 'fcntl' Function

fcntl stands for "file control" and this function performs various descriptor control operations. Before describing the function and how it affects a socket, we need to look at the bigger picture. Figure 7.20 summarizes the different operations performed by fcntl, ioctl, and routing sockets.

Figure 7.20. Summary of fcntl, ioctl, and routing socket operations.



Operation	fcntl	ioctl	Routing socket	POSIX
Set socket for nonblocking I/O	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
Set socket for signal-driven I/O	F_SETFL, O_ASYNC	FIOASYNC		fcntl
Set socket owner	F_SETOWN	SIOCSPGRP or FIOSETOWN		fcntl
Get socket owner	F_GETOWN	SIOCGPGRP or FIOGETOWN		fcntl
Get # bytes in socket receive buffer		FIONREAD		
Test for socket at out-of-band mark		SIOCATMARK		sockatmark
Obtain interface list		SIOCGIFCONF	sysctl	
Interface operations		SIOC[GS]IF*xxx*		
ARP cache operations		SIOC*x*ARP	RTM_*xxx*	
Routing table operations		SIOC*xxx*RT	RTM_*xxx*	

The first six operations can be applied to sockets by any process; the second two (interface operations) are less common, but are still general-purpose; and the last two (ARP and routing table) are issued by administrative programs such as ifconfig and route. We will talk more about the various ioctl operations in Chapter 17 and routing sockets in Chapter 18.

There are multiple ways to perform the first four operations, but we note in the final column that POSIX specifies that fcntl is the preferred way. We also note that POSIX provides the sockatmark function (Section 24.3) as the preferred way to test for the out-of-band mark. The remaining operations, with a blank final column, have not been standardized by POSIX.

We also note that the first two operations, setting a socket for nonblocking I/O and for signal-driven I/O, have been set historically using the FNDELAY and FASYNC commands with fcntl. POSIX defines the O_*XXX* constants.

The fcntl function provides the following features related to network programming:

- Nonblocking I/O☐ We can set the O_NONBLOCK file status flag using the F_SETFL command to set a socket as nonblocking. We will describe nonblocking I/O in Chapter 16.
- Signal-driven I/O☐ We can set the O_ASYNC file status flag using the F_SETFL command, which causes the SIGIO signal to be generated when the status of a socket changes. We will discuss this in Chapter 25.
- The F_SETOWN command lets us set the socket owner (the process ID or process group ID) to receive the SIGIO and SIGURG signals. The former signal is generated when signal-driven I/O is enabled for a socket (Chapter 25) and the latter signal is generated when new out-of-band data arrives for a socket

  (Chapter 24). The F_GETOWN command returns the current owner of the socket.

The term "socket owner" is defined by POSIX. Historically, Berkeley-derived implementations have called this "the process group ID of the socket" because the variable that stores this ID is the so_pgid member of the socket structure (p. 438 of TCPv2).

#include <fcntl.h>
int fcntl(int*fd*, int *cmd*, ... /* int *arg* */);
Returns: depends on *cmd* if OK, -1 on error

Each descriptor (including a socket) has a set of file flags that is fetched with the F_GETFL command and set with the F_SETFL command. The two flags that affect a socket are

- O_NONBLOCK☐nonblocking I/O
- O_ASYNC☐signal-driven I/O

We will describe both of these features in more detail later. For now, we note that typical code to enable nonblocking I/O, using fcntl, would be:

```
int     flags;

     /* Set a socket as nonblocking */
if  ( (flags = fcntl (fd, F_GETFL, 0)) < 0)
    err_sys("F_GETFL error");
```

```
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

Beware of code that you may encounter that simply sets the desired flag.

```
    /* Wrong way to set a socket as nonblocking */
if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
    err_sys("F_SETFL error");
```

While this sets the nonblocking flag, it also clears all the other file status flags. The only correct way to set one of the file status flags is to fetch the current flags, logically OR in the new flag, and then set the flags.

The following code turns off the nonblocking flag, assuming `flags` was set by the call to `fcntl` shown above:

```
flags &= ~O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

The two signals `SIGIO` and `SIGURG` are different from other signals in that they are generated for a socket only if the socket has been assigned an owner with the `F_SETOWN` command. The integer *arg* value for the `F_SETOWN` command can be either a positive integer, specifying the process ID to receive the signal, or a negative integer whose absolute value is the process group ID to receive the signal. The `F_GETOWN` command returns the socket owner as the return value from the `fcntl` function, either the process ID (a positive return value) or the process group ID (a negative value other than □1). The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group (perhaps more than one) to receive the signal.

When a new socket is created by `socket`, it has no owner. But when a new socket is created from a listening socket, the socket owner is inherited from the listening socket by the connected socket (as are many socket options [pp. 462□463 of TCPv2]).

7.12 Summary

Socket options run the gamut from the very general (SO_ERROR) to the very specific (IP header options). The most commonly used options that we might encounter are SO_KEEPALIVE, SO_RCVBUF, SO_SNDBUF, and SO_REUSEADDR. The latter should always be set for a TCP server before it calls bind (Figure 11.12). The SO_BROADCAST option and the 10 multicast socket options are only for applications that broadcast or multicast, respectively.

The SO_KEEPALIVE socket option is set by many TCP servers and automatically terminates a half-open connection. The nice feature of this option is that it is handled by the TCP layer, without requiring an application-level inactivity timer; its downside is that it cannot tell the difference between a crashed client host and a temporary loss of connectivity to the client. SCTP provides 17 socket options that are used by the application to control the transport. SCTP_NODELAY and SCTP_MAXSEG are similar to TCP_NODELAY and TCP_MAXSEG and perform equivalent functions. The other 15 options give the application finer control of the SCTP stack; we will discuss the use of many of these socket options in Chapter 23.

The SO_LINGER socket option gives us more control over when close returns and also lets us force an RST to be sent instead of TCP's four-packet connection termination sequence. We must be careful sending RSTs, because this avoids TCP's TIME_WAIT state. Much of the time, this socket option does not provide the information that we need, in which case, an application-level ACK is required.

Every TCP and SCTP socket has a send buffer and a receive buffer, and every UDP socket has a receive buffer. The SO_SNDBUF and SO_RCVBUF socket options let us change the sizes of these buffers. The most common use of these options is for bulk data transfer across long fat pipes: TCP connections with either a high bandwidth or a long delay, often using the RFC 1323 extensions. UDP sockets, on the other hand, might want to increase the size of the receive buffer to allow the kernel to queue more datagrams if the application is busy.

Exercises

7.1 Write a program that prints the default TCP, UDP, and SCTP send and receive buffer sizes and run it on the systems to which you have access.

7.2 Modify Figure 1.5 as follows: Before calling `connect`, call `getsockopt` to obtain the socket receive buffer size and MSS. Print both values. After `connect` returns success, fetch these same two socket options and print their values. Have the values changed? Why? Run the program connecting to a server on your local network and also run the program connecting to a server on a remote network. Does the MSS change? Why? You should also run the program on any different hosts to which you have access.

7.3 Start with our TCP server from Figures 5.2 and 5.3 and our TCP client from Figures 5.4 and 5.5. Modify the client `main` function to set the `SO_LINGER` socket option before calling `exit`, setting `l_onoff` to 1 and `l_linger` to 0. Start the server and then start the client. Type in a line or two at the client to verify the operation, and then terminate the client by entering your EOF character. What happens? After you terminate the client, run `netstat` on the client host and see if the socket goes through the TIME_WAIT state.

7.4 Assume two TCP clients start at about the same time. Both set the `SO_REUSEADDR` socket option and then call `bind` with the same local IP address and the same local port (say 1500). But, one client `connects` to 198.69.10.2 port 7000 and the second `connects` to 198.69.10.2 (same peer IP address) but port 8000. Describe the race condition that occurs.

7.5 Obtain the source code for the examples in this book (see the Preface) and compile the `sock` program (Section C.3). First, classify your host as (a) no multicast support, (b) multicast support but `SO_REUSEPORT` not provided, or (c) multicast support and `SO_REUSEPORT` provided. Try to start multiple instances of the `sock` program as a TCP server (`-s` command-line option) on the same port, binding the wildcard address, one of your host's interface addresses, and the loopback address. Do you need to specify the `SO_REUSEADDR` option (the `-A` command-line option)? Use `netstat` to see the listening sockets.

7.6 Continue the previous example, but start a UDP server (`-u` command-line option) and try to start two instances, both binding the same local IP address and port. If your implementation supports `SO_REUSEPORT`, try using it (`-T` command-line option).

7.7 Many versions of the `ping` program have a `-d` flag to enable the `SO_DEBUG` socket option. What does this do?

7.8 Continuing the example at the end of our discussion of the `TCP_NODELAY` socket option, assume that a client performs two `writes`: the first of 4 bytes and the second of 396 bytes. Also assume that the server's delayed ACK time is 100 ms, the RTT between the client and server is 100 ms, and the server's processing time for the client's request is 50 ms. Draw a timeline that shows the interaction of the Nagle algorithm with delayed ACKs.

7.9 Redo the previous exercise, assuming the `TCP_NODELAY` socket option is set.

7.10 Redo Exercises 7.8 assuming the process calls `writev` one time, for both the 4-byte buffer and the 396-byte buffer.

7.11 Read RFC 1122 [Braden 1989] to determine the recommended interval for delayed ACKs.

7.12 Where does our server in Figures 5.2 and 5.3 spend most of its time? Assume the server sets the `SO_KEEPALIVE` socket option, there is no data being exchanged across the connection, and the client host crashes and does not reboot. What happens?

7.13 Where does our client in Figures 5.4 and 5.5 spend most of its time? Assume the client sets the `SO_KEEPALIVE` socket option, there is no data being exchanged across the connection, and the server host crashes and does not reboot. What happens?

7.14 Where does our client in Figures 5.4 and 6.13 spend most of its time? Assume the client sets the `SO_KEEPALIVE` socket option, there is no data being exchanged across the connection, and the server host crashes and does not reboot. What happens?

7.15 Assume both a client and server set the `SO_KEEPALIVE` socket option. Connectivity is maintained between the two peers, but there is no application data exchanged across the connection. When the keep-alive timer expires every two hours, how many TCP segments are exchanged across the connection?

7.16 Almost all implementations define the constant `SO_ACCEPTCONN` in the `<sys/socket.h>` header, but we have not described this option. Read [Lanciani 1996] to find out why this option exists.

Chapter 8. Elementary UDP Sockets

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.

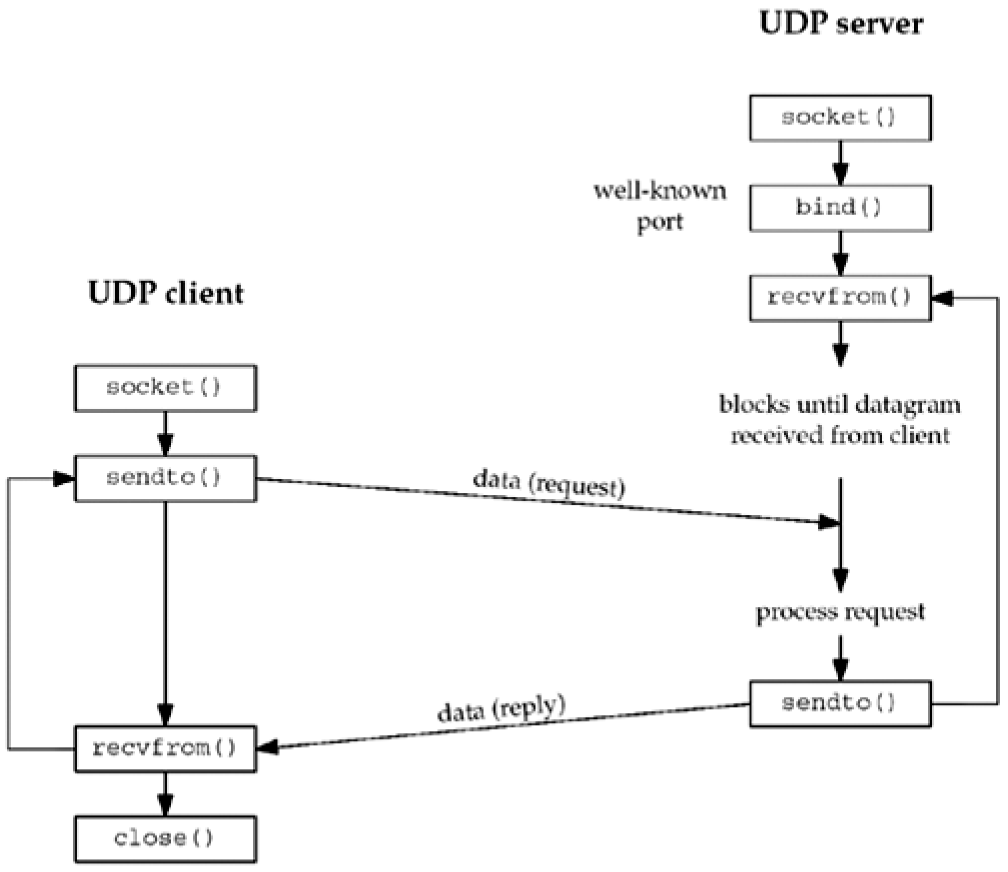


Figure 8.1 shows a timeline of the typical scenario that takes place for a UDP client/server exchange. We can compare this to the typical TCP exchange that was shown in Figure 4.1.

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.

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*`);`

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

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.

8.3 UDP Echo Server: 'main' Function

We will now redo our simple echo client/server from Chapter 5 using UDP. Our UDP client and server programs follow the function call flow that we diagrammed in Figure 8.1. Figure 8.2 depicts the functions that are used. Figure 8.3 shows the server `main` function.

Figure 8.2. Simple echo client/server using UDP.

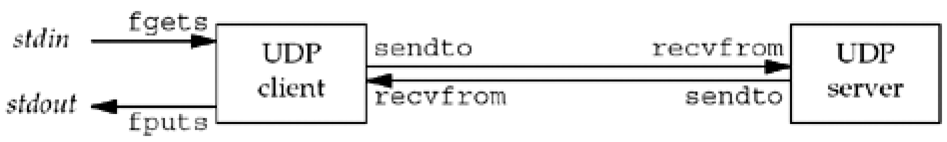


Figure 8.3 UDP echo server.

udpcliserv/udpserv01.c

```
#include     "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd;
    struct sockaddr_in servaddr, cliaddr;

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
13     dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

Create UDP socket, bind server's well-known port

7□12 We create a UDP socket by specifying the second argument to `socket` as `SOCK_DGRAM` (a datagram socket in the IPv4 protocol). As with the TCP server example, the IPv4 address for the `bind` is specified as `INADDR_ANY` and the server's well-known port is the constant `SERV_PORT` from the `unp.h` header.

13 The function `dg_echo` is called to perform server processing.

8.4 UDP Echo Server: 'dg_echo' Function

Figure 8.4 shows the `dg_echo` function.

Figure 8.4 dg_echo function: echo lines on a datagram socket.

lib/dg_echo.c

```
 1 #include     "unp.h"

 2 void
 3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4 {
 5     int     n;
 6     socklen_t len;
 7     char    mesg[MAXLINE];

 8     for ( ; ; ) {
 9         len = clilen;
10         n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11         Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12     }
13 }
```

Read datagram, echo back to sender

8□12 This function is a simple loop that reads the next datagram arriving at the server's port using `recvfrom` and sends it back using `sendto`.

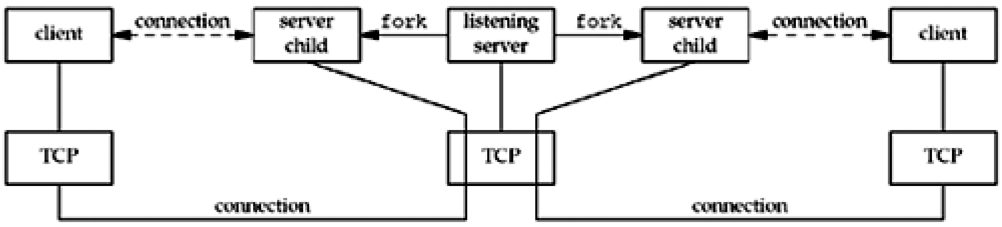
Despite the simplicity of this function, there are numerous details to consider. First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.

Next, this function provides an *iterative server*, not a concurrent server as we had with TCP. There is no call to `fork`, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

There is implied queuing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer. When the process calls `recvfrom`, the next datagram from the buffer is returned to the process in a first-in, first-out (FIFO) order. This way, if multiple datagrams arrive for the socket before the process can read what's already queued for the socket, the arriving datagrams are just added to the socket receive buffer. But, this buffer has a limited size. We discussed this size and how to increase it with the `SO_RCVBUF` socket option in Section 7.5.

Figure 8.5 summarizes our TCP client/server from Chapter 5 when two clients establish connections with the server.

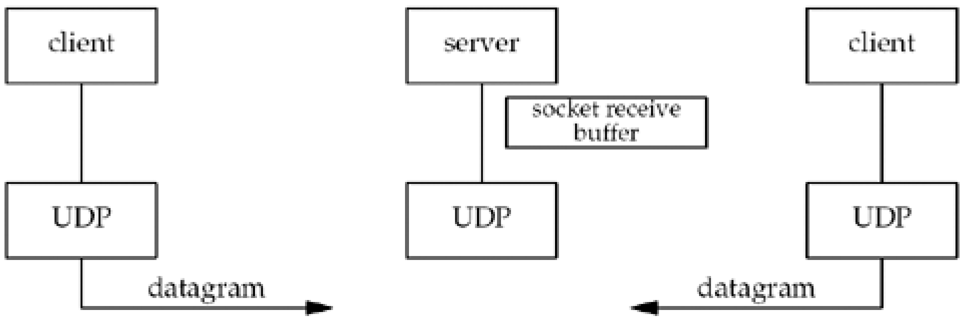
Figure 8.5. Summary of TCP client/server with two clients.



There are two connected sockets and each of the two connected sockets on the server host has its own socket receive buffer.

Figure 8.6 shows the scenario when two clients send datagrams to our UDP server.

Figure 8.6. Summary of UDP client/server with two clients.



There is only one server process and it has a single socket on which it receives all arriving datagrams and sends all responses. That socket has a receive buffer into which all arriving datagrams are placed.

The `main` function in Figure 8.3 is *protocol-dependent* (it creates a socket of protocol `AF_INET` and allocates and initializes an IPv4 socket address structure), but the `dg_echo` function is *protocol-independent*. The reason `dg_echo` is protocol-independent is because the caller (the `main` function in our case) must allocate a socket address structure of the correct size, and a pointer to this structure, along with its size, are passed as arguments to `dg_echo`. The function `dg_echo` never looks inside this protocol-dependent structure: It simply passes a pointer to the structure to `recvfrom` and `sendto`. `recvfrom` fills this structure with the IP address and port number of the client, and since the same pointer (`pcliaddr`) is then passed to `sendto` as the destination address, this is how the datagram is echoed back to the client that sent the datagram.

8.5 UDP Echo Client: 'main' Function

The UDP client `main` function is shown in Figure 8.7.

Figure 8.7 UDP echo client.

udpcliserv/udpcli01.c

```
#include     "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd;
    struct sockaddr_in servaddr;

    if(argc != 2)
       err_quit("usage: udpcli <IPaddress>");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

15     exit(0);
16 }
```

Fill in socket address structure with server's address

9□12 An IPv4 socket address structure is filled in with the IP address and port number of the server. This structure will be passed to `dg_cli`, specifying where to send datagrams.

13□14 A UDP socket is created and the function `dg_cli` is called.

8.6 UDP Echo Client: 'dg_cli' Function

Figure 8.8 shows the function `dg_cli`, which performs most of the client processing.

Figure 8.8 dg_cli function: client processing loop.

lib/dg_cli.c

```
#include    "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int     n;
    char    sendline[MAXLINE], recvline[MAXLINE + 1];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

        recvline[n] = 0;        /* null terminate */
        Fputs(recvline, stdout);
    }
}
```

7–12 There are four steps in the client processing loop: read a line from standard input using fgets, send the line to the server using sendto, read back the server's echo using recvfrom, and print the echoed line to standard output using fputs.

Our client has not asked the kernel to assign an ephemeral port to its socket. (With a TCP client, we said the call to connect is where this takes place.) With a UDP socket, the first time the process calls sendto, if the socket has not yet had a local port bound to it, that is when an ephemeral port is chosen by the kernel for the socket. As with TCP, the client can call bind explicitly, but this is rarely done.

Notice that the call to recvfrom specifies a null pointer as the fifth and sixth arguments. This tells the kernel that we are not interested in knowing who sent the reply. There is a risk that any process, on either the same host or some other host, can send a datagram to the client's IP address and port, and that datagram will be read by the client, who will think it is the server's reply. We will address this in Section 8.8.

As with the server function dg_echo, the client function dg_cli is protocol-independent, but the client main function is protocol-dependent. The main function allocates and initializes a socket address structure of some protocol type and then passes a pointer to this structure, along with its size, to dg_cli.

8.7 Lost Datagrams

Our UDP client/server example is not reliable. If a client datagram is lost (say it is discarded by some router between the client and server), the client will block forever in its call to recvfrom in the function dg_cli, waiting for a server reply that will never arrive. Similarly, if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to recvfrom. A typical way to prevent this is to place a timeout on the client's call to recvfrom. We will discuss this in Section 14.2.

Just placing a timeout on the recvfrom is not the entire solution. For example, if we do time out, we cannot tell whether our datagram never made it to the server, or if the server's reply never made it back. If the client's request was something like "transfer a certain amount of money from account A to account B" (instead of our simple echo server), it would make a big difference as to whether the request was lost or the reply was lost. We will talk more about adding reliability to a UDP client/server in Section 22.5.

8.8 Verifying Received Response

At the end of Section 8.6, we mentioned that any process that knows the client's ephemeral port number could send datagrams to our client, and these would be intermixed with the normal server replies. What we can do is change the call to `recvfrom` in Figure 8.8 to return the IP address and port of who sent the reply and ignore any received datagrams that are not from the server to whom we sent the datagram. There are a few pitfalls with this, however, as we will see.

First, we change the client `main` function (Figure 8.7) to use the standard echo server (Figure 2.18). We just replace the assignment

```
servaddr.sin_port = htons(SERV_PORT);
```

with

```
servaddr.sin_port = htons(7);
```

We do this so we can use any host running the standard echo server with our client.

We then recode the `dg_cli` function to allocate another socket address structure to hold the structure returned by `recvfrom`. We show this in Figure 8.9.

Allocate another socket address structure

9 We allocate another socket address structure by calling `malloc`. Notice that the `dg_cli` function is still protocol-independent; because we do not care what type of socket address structure we are dealing with, we use only its size in the call to `malloc`.

Compare returned address

*12*18* In the call to `recvfrom`, we tell the kernel to return the address of the sender of the datagram. We first compare the length returned by `recvfrom` in the value-result argument and then compare the socket address structures themselves using `memcmp`.

Section 3.2 says that even if the socket address structure contains a length field, we need never set it or examine it. However, `memcmp` compares every byte of data in the two socket address structures, and the length field is set in the socket address structure that the kernel returns; so in this case we must set it when constructing the `sockaddr`. If we don't, the `memcmp` will compare a *0* (since we didn't set it) with a *16* (assuming `sockaddr_in`) and will not match.

Figure 8.9 Version of dg_cli that verifies returned socket address.

udpcliserv/dgcliaddr.c

```
#include    "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int     n;
    char    sendline[MAXLINE], recvline[MAXLINE + 1];
    socklen_t len;
    struct sockaddr *preply_addr;

    preply_addr = Malloc(servlen);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        len = servlen;
        n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
        if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
            printf("reply from %s (ignored)\n", Sock_ntop(preply_addr, len));
            continue;
        }

        recvline[n] = 0;      /* null terminate */
        Fputs(recvline, stdout);
    }
}
```

This new version of our client works fine if the server is on a host with just a single IP address. But this program can fail if the server is multihomed. We run this program to our host `freebsd4`, which has two interfaces and two IP addresses.

```
macosx % **host freebsd4**
freebsd4.unpbook.com has address 172.24.37.94
freebsd4.unpbook.com has address 135.197.17.100
macosx % **udpcli02 135.197.17.100**
**hello**
reply from 172.24.37.94:7 (ignored)
**goodbye**
reply from 172.24.37.94:7 (ignored)
```

We specified the IP address that does not share the same subnet as the client.

This is normally allowed. Most IP implementations accept an arriving IP datagram that is destined for *any* of the host's IP addresses, regardless of the interface on which the datagram arrives (pp. 217□219 of TCPv2). RFC 1122 [Braden 1989] calls this the *weak end system model*. If a system implemented what this RFC calls the *strong end system model*, it would accept an arriving datagram only if that datagram arrived on the interface to which it was addressed.

The IP address returned by `recvfrom` (the source IP address of the UDP datagram) is not the IP address to which we sent the datagram. When the server sends its reply, the destination IP address is 172.24.37.78. The routing function within the kernel on `freebsd4` chooses 172.24.37.94 as the outgoing interface. Since the server has not bound an IP address to its socket (the server has bound the wildcard address to its socket, which is something we can verify by running `netstat` on `freebsd`), the kernel chooses the source address for the IP datagram. It is chosen to be the primary IP address of the outgoing interface (pp. 232□233 of TCPv2). Also, since it is the primary IP address of the interface, if we send our datagram to a nonprimary IP address of the interface (i.e., an alias), this will also cause our test in Figure 8.9 to fail.

One solution is for the client to verify the responding host's domain name instead of its IP address by looking up the server's name in the DNS (Chapter 11), given the IP address returned by `recvfrom`. Another solution is for the UDP server to create one socket for every IP address that is configured on the host, `bind` that IP address to the socket, use `select` across all these sockets (waiting for any one to become readable), and then reply from the socket that is readable. Since the socket used for the reply was bound to the IP address that was the destination address of the client's request (or the datagram would not have been delivered to the socket), this guaranteed that the source address of the reply was the same as the destination address of the request. We will show an example of this in Section 22.6.

On a multihomed Solaris system, the source IP address for the server's reply is the destination IP address of the client's request. The scenario described in this section is for Berkeley-derived implementations that choose the source IP address based on the outgoing interface.

8.9 Server Not Running

The next scenario to examine is starting the client without starting the server. If we do so and type in a single line to the client, nothing happens. The client blocks forever in its call to `recvfrom`, waiting for a server reply that will never appear. But, this is an example where we need to understand more about the underlying protocols to understand what is happening to our networking application.

First we start `tcpdump` on the host `macosx`, and then we start the client on the same host, specifying the host `freebsd4` as the server host. We then type a single line, but the line is not echoed.

```
macosx % udpcli01 172.24.37.94

hello, world
                                we type this line but nothing is echoed back
```

Figure 8.10 shows the `tcpdump` output.

Figure 8.10 tcpdump output when server process not started on server host.

```
1 0.0                    arp who-has freebsd4 tell macosx
2 0.003576 ( 0.0036)     arp reply freebsd4 is-at 0:40:5:42:d6:de

3 0.003601 ( 0.0000)     macosx.51139 > freebsd4.9877: udp 13
4 0.009781 ( 0.0062)     freebsd4 > macosx: icmp: freebsd4 udp port 9877
unreachable
```

First we notice that an ARP request and reply are needed before the client host can send the UDP datagram to the server host. (We left this exchange in the output to reiterate the potential for an ARP request-reply before an IP datagram can be sent to another host or router on the local network.)

In line 3, we see the client datagram sent but the server host responds in line 4 with an ICMP "port unreachable." (The length of 13 accounts for the 12 characters and the newline.) This ICMP error, however, is not returned to the client process, for reasons that we will describe shortly. Instead, the client blocks forever in the call to `recvfrom` in Figure 8.8. We also note that ICMPv6 has a "port unreachable" error, similar to ICMPv4 (Figures A.15 and A.16), so the results described here are similar for IPv6.

We call this ICMP error an *asynchronous error*. The error was caused by `sendto`, but `sendto` returned successfully. Recall from Section 2.11 that a successful return from a UDP output operation only means there was room for the resulting IP datagram on the interface output queue. The ICMP error is not returned until later (4 ms later in Figure 8.10), which is why it is called asynchronous.

The basic rule is that an asynchronous error is not returned for a UDP socket unless the socket has been connected. We will describe how to call `connect` for a UDP socket in Section 8.11. Why this design decision was made when sockets were first implemented is rarely understood. (The implementation implications are discussed on pp. 748□749 of TCPv2.)

Consider a UDP client that sends three datagrams in a row to three different servers (i.e., three different IP addresses) on a single UDP socket. The client then enters a loop that calls `recvfrom` to read the replies. Two of the datagrams are correctly delivered (that is, the server was running on two of the three hosts) but the third host was not running the server. This third host responds with an ICMP port unreachable. This ICMP error message contains the IP header and UDP header of the datagram that caused the error. (ICMPv4 and ICMPv6 error messages always contain the IP header and all of the UDP header or part of the TCP header to allow the receiver of the ICMP error to determine which socket caused the error. We will show this in Figures 28.21 and 28.22.) The client that sent the three datagrams needs to know the destination of the datagram that caused the error to distinguish which of the three datagrams caused the error. But how can the kernel return this information to the process? The only piece of information that `recvfrom` can return is an `errno` value; `recvfrom` has no way of returning the destination IP address and destination UDP port number of the datagram in error. The decision was made, therefore, to return these asynchronous errors to the process only if the process connected the UDP socket to exactly one peer.

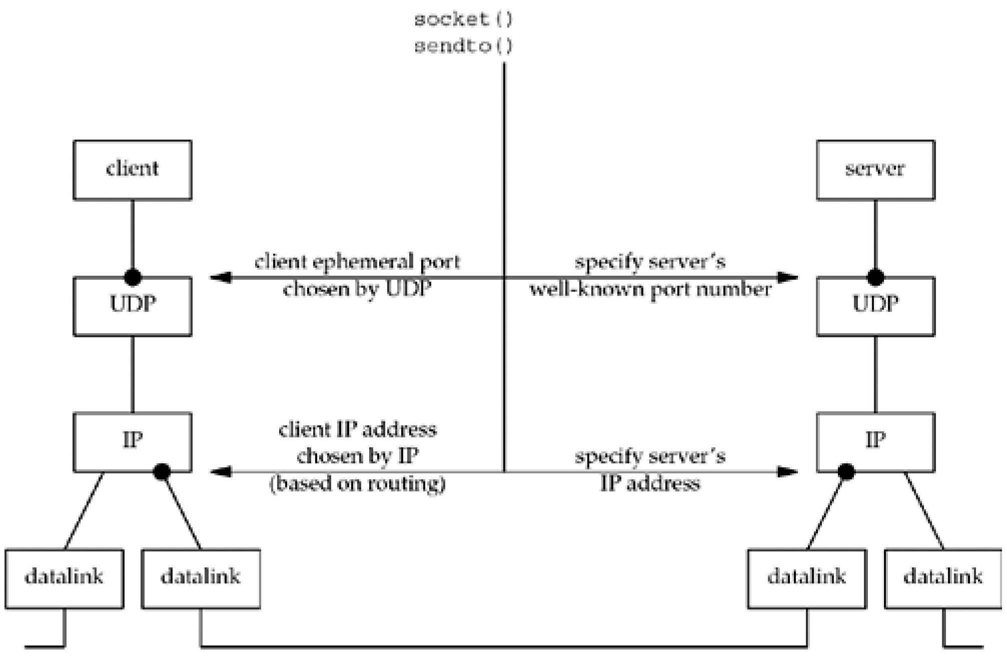
Linux returns most ICMP "destination unreachable" errors even for unconnected sockets, as long as the `SO_BSDCOMPAT` socket option is not enabled. All the ICMP "destination unreachable" errors from Figure A.15 are returned, except codes 0, 1, 4, 5, 11, and 12.

We return to this problem of asynchronous errors with UDP sockets in Section 28.7 and show an easy way to obtain these errors on unconnected sockets using a daemon of our own.

8.10 Summary of UDP Example

Figure 8.11 shows as bullets the four values that must be specified or chosen when the client sends a UDP datagram.

Figure 8.11. Summary of UDP client/server from client's perspective.

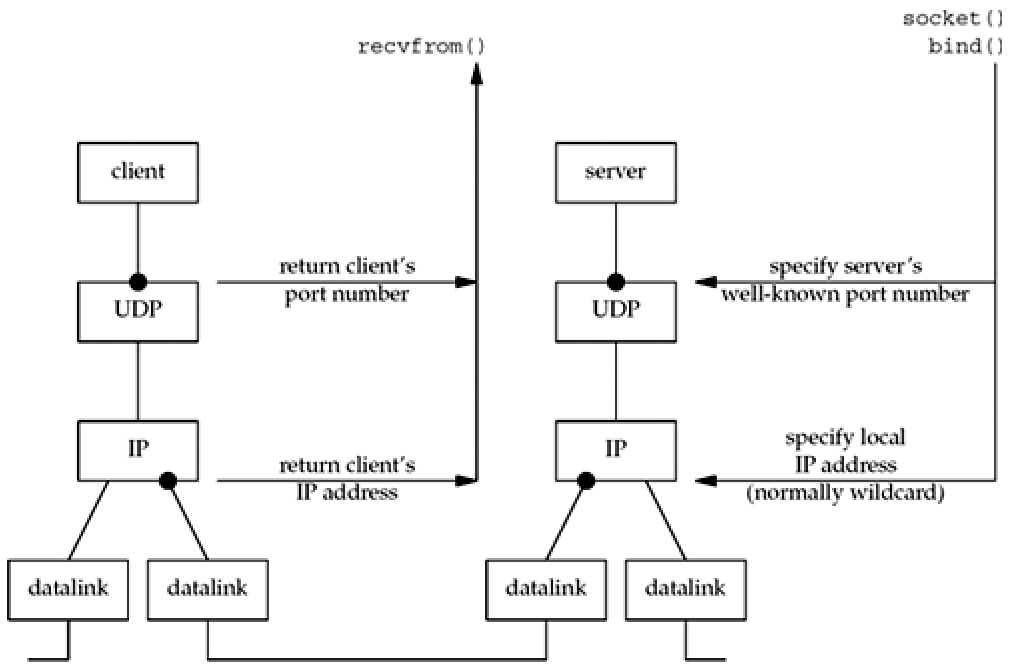

socket()
sendto()
client
server
client ephemeral port chosen by UDP
specify server's well-known port number
UDP
UDP
IP
IP
client IP address chosen by IP (based on routing)
specify server's IP address
datalink
datalink
datalink
datalink

The client must specify the server's IP address and port number for the call to `sendto`. Normally, the client's IP address and port are chosen automatically by the kernel, although we mentioned that the client can call `bind` if it so chooses. If these two values for the client are chosen by the kernel, we also mentioned that the client's ephemeral port is chosen once, on the first `sendto`, and then it never changes. The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not `bind` a specific IP address to the socket. The reason is shown in Figure 8.11: If the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right. In this worst-case scenario, the client's IP address, as chosen by the kernel based on the outgoing datalink, would change for every datagram.

What happens if the client `binds` an IP address to its socket, but the kernel decides that an outgoing datagram must be sent out some other datalink? In this case the IP datagram will contain a source IP address that is different from the IP address of the outgoing datalink (see Exercise 8.6).

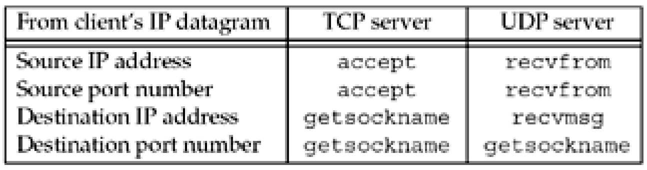
Figure 8.12 shows the same four values, but from the server's perspective.

Figure 8.12. Summary of UDP client/server from server's perspective.


recvfrom()
socket()
bind()
client
server
UDP
UDP
return client's port number
specify server's well-known port number
IP
IP
return client's IP address
specify local IP address (normally wildcard)
datalink
datalink
datalink
datalink

There are at least four pieces of information that a server might want to know from an arriving IP datagram: the source IP address, destination IP address, source port number, and destination port number. Figure 8.13 shows the function calls that return this information for a TCP server and a UDP server.

Figure 8.13. Information available to server from arriving IP datagram.



From client's IP datagram	TCP server	UDP server
Source IP address	`accept`	`recvfrom`
Source port number	`accept`	`recvfrom`
Destination IP address	`getsockname`	`recvmsg`
Destination port number	`getsockname`	`getsockname`

A TCP server always has easy access to all four pieces of information for a connected socket, and these four values remain constant for the lifetime of a connection. With a UDP socket, however, the destination IP address can only be obtained by setting the `IP_RECVDSTADDR` socket option for IPv4 or the `IPV6_PKTINFO` socket option for IPv6 and then calling `recvmsg` instead of `recvfrom`. Since UDP is connectionless, the destination IP address can change for each datagram that is sent to the server. A UDP server can also receive datagrams destined for one of the host's broadcast addresses or for a multicast address, as we will discuss in Chapters 20 and 21. We will show how to determine the destination address of a UDP datagram in Section 22.2, after we cover the `recvmsg` function.

8.11 'connect' Function with UDP

We mentioned at the end of Section 8.9 that an asynchronous error is not returned on a UDP socket unless the socket has been connected. Indeed, we are able to call `connect` (Section 4.3) for a UDP socket. But this does not result in anything like a TCP connection: There is no three-way handshake. Instead, the kernel just checks for any immediate errors (e.g., an obviously unreachable destination), records the IP address and port number of the peer (from the socket address structure passed to `connect`), and returns immediately to the calling process.

Overloading the `connect` function with this capability for UDP sockets is confusing. If the convention that `sockname` is the local protocol address and `peername` is the foreign protocol address is used, then a better name would have been `setpeername`. Similarly, a better name for the `bind` function would be `setsockname`.

With this capability, we must now distinguish between

- An *unconnected UDP socket*, the default when we create a UDP socket
- A *connected UDP socket*, the result of calling `connect` on a UDP socket

With a connected UDP socket, three things change, compared to the default unconnected UDP socket:

1. We can no longer specify the destination IP address and port for an output operation. That is, we do not use `sendto`, but `write` or `send` instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by `connect`.

   Similar to TCP, we can call `sendto` for a connected UDP socket, but we cannot specify a destination address. The fifth argument to `sendto` (the pointer to the socket address structure) must be a null pointer, and the sixth argument (the size of the socket address structure) should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.

2. We do not need to use `recvfrom` to learn the sender of a datagram, but `read`, `recv`, or `recvmsg` instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in `connect`. Datagrams destined to the connected UDP socket's local protocol address (e.g., IP address and port) but arriving from a protocol address other than the one to which the socket was connected are not passed to the connected socket. This limits a connected UDP socket to exchanging datagrams with one and only one peer.

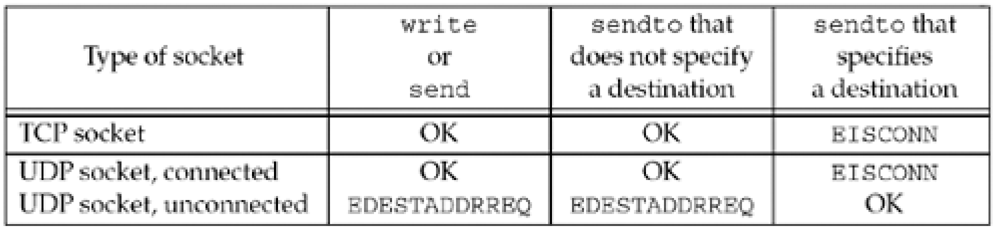
Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to `connect` to a multicast or broadcast address.

3. Asynchronous errors are returned to the process for connected UDP sockets.

   The corollary, as we previously described, is that unconnected UDP sockets do not receive asynchronous errors.

Figure 8.14 summarizes the first point in the list with respect to 4.4BSD.

Figure 8.14. TCP and UDP sockets: can a destination protocol address be specified?

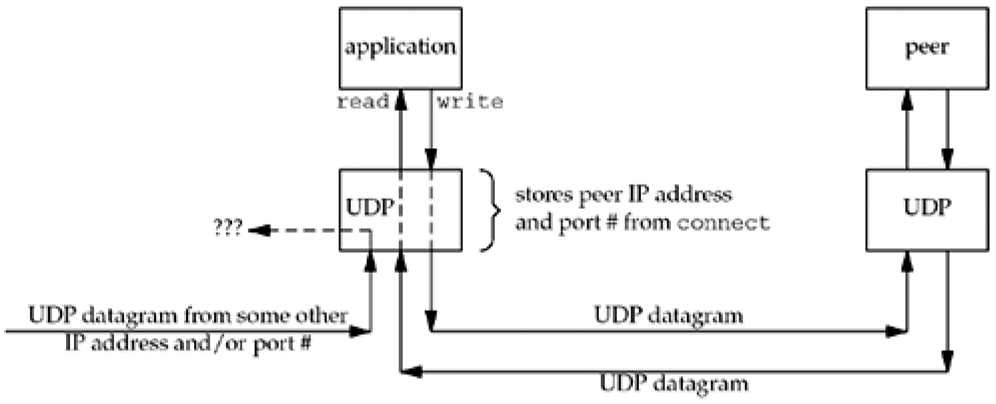


Type of socket	`write` or `send`	`sendto` that does not specify a destination	`sendto` that specifies a destination
TCP socket	OK	OK	`EISCONN`
UDP socket, connected	OK	OK	`EISCONN`
UDP socket, unconnected	`EDESTADDRREQ`	`EDESTADDRREQ`	OK

The POSIX specification states that an output operation that does not specify a destination address on an unconnected UDP socket should return `ENOTCONN`, not `EDESTADDRREQ`.

Figure 8.15 summarizes the three points that we made about a connected UDP socket.

Figure 8.15. Connected UDP socket.


application
read
write
peer
UDP
stores peer IP address and port # from connect
UDP
???
UDP datagram from some other IP address and/or port #
UDP datagram
UDP datagram

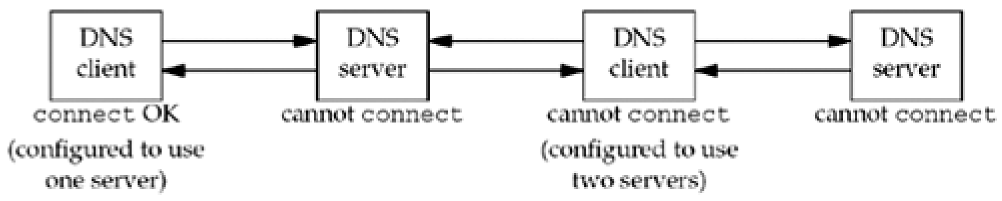
The application calls `connect`, specifying the IP address and port number of its peer. It then uses `read` and `write` to exchange data with the peer.

Datagrams arriving from any other IP address or port (which we show as "???" in Figure 8.15) are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is `connected`. These datagrams could be delivered to some other UDP socket on the host. If there is no other matching socket for the arriving datagram, UDP will discard it and generate an ICMP "port unreachable" error.

In summary, we can say that a UDP client or server can call `connect` only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls `connect`, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP); in this case, both the client and server can call `connect`.

The DNS provides another example, as shown in Figure 8.16.

Figure 8.16. Example of DNS clients and servers and the connect function.



A DNS client can be configured to use one or more servers, normally by listing the IP addresses of the servers in the file `/etc/resolv.conf`. If a single server is listed (the leftmost box in the figure), the client can call `connect`, but if multiple servers are listed (the second box from the right in the figure), the client cannot call `connect`. Also, a DNS server normally handles any client request, so the servers cannot call `connect`.

Calling connect Multiple Times for a UDP Socket

A process with a connected UDP socket can call `connect` again for that socket for one of two reasons:

- To specify a new IP address and port
- To unconnect the socket

The first case, specifying a new peer for a connected UDP socket, differs from the use of `connect` with a TCP socket: `connect` can be called only one time for a TCP socket.

To unconnect a UDP socket, we call `connect` but set the family member of the socket address structure (`sin_family` for IPv4 or `sin6_family` for IPv6) to `AF_UNSPEC`. This might return an error of `EAFNOSUPPORT` (p. 736 of TCPv2), but that is acceptable. It is the process of calling `connect` on an already connected UDP socket that causes the socket to become unconnected (pp. 787□788 of TCPv2).

The Unix variants seem to differ on exactly how to unconnect a socket, and you may encounter approaches that work on some systems and not others. For example, calling `connect` with `NULL` for the address works only on some systems (and on some, it only works if the third argument, the length, is nonzero). The POSIX specification and BSD man pages are not much help here, only mentioning that a *null address* should be used and not mentioning the error return (even on success) at all. The most portable solution is to zero out an address structure, set the family to `AF_UNSPEC` as mentioned above, and pass it to `connect`.

Another area of disagreement is around the local binding of a socket during the unconnect process. AIX keeps both the chosen local IP address and the port, even from an implicit bind. FreeBSD and Linux set the local IP address back to all zeros, even if you previously called `bind`, but leave the port number intact. Solaris sets the local IP address back to all zeros if it was assigned by an implicit bind; but if the program called `bind` explicitly, then the IP address remains unchanged.

Performance

When an application calls `sendto` on an unconnected UDP socket, Berkeley-derived kernels temporarily connect the socket, send the datagram, and then unconnect the socket (pp. 762□763 of TCPv2). Calling `sendto` for two datagrams on an unconnected UDP socket then involves the following six steps by the kernel:

- Connect the socket
- Output the first datagram
- Unconnect the socket
- Connect the socket
- Output the second datagram
- Unconnect the socket

Another consideration is the number of searches of the routing table. The first temporary connect searches the routing table for the destination IP address and saves (caches) that information. The second temporary connect notices that the destination address equals the destination of the cached routing table information (we are assuming two `sendto`s to the same destination) and we do not need to search the routing table again (pp. 737□738 of TCPv2).

When the application knows it will be sending multiple datagrams to the same peer, it is more efficient to connect the socket explicitly. Calling `connect` and then calling `write` two times involves the following steps by the kernel:

- Connect the socket
- Output first datagram
- Output second datagram

In this case, the kernel copies only the socket address structure containing the destination IP address and port one time, versus two times when `sendto` is called twice. [Partridge and Pink 1993] note that the temporary connecting of an unconnected UDP socket accounts for nearly one-third of the cost of each UDP transmission.

8.12 'dg_cli' Function (Revisited)

We now return to the `dg_cli` function from Figure 8.8 and recode it to call `connect`. Figure 8.17 shows the new function.

Figure 8.17 dg_cli function that calls connect.

udpcliserv/dgcliconnect.c

```
#include    "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int     n;
    char    sendline[MAXLINE], recvline[MAXLINE + 1];

    Connect(sockfd, (SA *) pservaddr, servlen);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Write(sockfd, sendline, strlen(sendline));

        n = Read(sockfd, recvline, MAXLINE);

        recvline[n] = 0;        /* null terminate */
        Fputs(recvline, stdout);
    }
}
```

The changes are the new call to `connect` and replacing the calls to `sendto` and `recvfrom` with calls to `write` and `read`. This function is still protocol-independent since it doesn't look inside the socket address structure that is passed to `connect`. Our client `main` function, Figure 8.7, remains the same.

If we run this program on the host `macosx`, specifying the IP address of the host `freebsd4` (which is not running our server on port 9877), we have the following output:

```
macosx % udpcli04 172.24.37.94
hello, world
read error: Connection refused
```

The first point we notice is that we do *not* receive the error when we start the client process. The error occurs only after we send the first datagram to the server. It is sending this datagram that elicits the ICMP error from the server host. But when a TCP client calls `connect`, specifying a server host that is not running the server process, `connect` returns the error because the call to `connect` causes the TCP three-way handshake to happen, and the first packet of that handshake elicits an RST from the server TCP (Section 4.3).

Figure 8.18 shows the `tcpdump` output.

Figure 8.18 tcpdump output when running Figure 8.17.

```
macosx % tcpdump
1  0.0                    macosx.51139 > freebsd4.9877: udp 13
2  0.006180 ( 0.0062)     freebsd4 > macosx: icmp: freebsd4 udp port 9877
unreachable
```

We also see in Figure A.15 that this ICMP error is mapped by the kernel into the error `ECONNREFUSED`, which corresponds to the message string output by our `err_sys` function: "Connection refused."

Unfortunately, not all kernels return ICMP messages to a connected UDP socket, as we have shown in this section. Normally, Berkeley-derived kernels return the error, while System V kernels do not. For example, if we run the same client on a Solaris 2.4 host and `connect` to a host that is not running our server, we can watch with `tcpdump` and verify that the ICMP "port unreachable" error is returned by the server host, but the client's call to `read` never returns. This bug was fixed in Solaris 2.5. UnixWare does not return the error, while AIX, Digital Unix, HP-UX, and Linux all return the error.

8.13 Lack of Flow Control with UDP

We now examine the effect of UDP not having any flow control. First, we modify our `dg_cli` function to send a fixed number of datagrams. It no longer reads from standard input. Figure 8.19 shows the new version. This function writes 2,000 1,400-byte UDP datagrams to the server.

We next modify the server to receive datagrams and count the number received. This server no longer echoes datagrams back to the client. Figure 8.20 shows the new `dg_echo` function. When we terminate the server with our terminal interrupt key (`SIGINT`), it prints the number of received datagrams and terminates.

Figure 8.19 dg_cli function that writes a fixed number of datagrams to the server.

udpcliserv/dgcliloop1.c

```
#include    "unp.h"

#define NDG     2000        /*  datagrams to send */
#define DGLEN    1400        /*  length of each datagram */

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int     i;
    char    sendline[DGLEN];

    for (i = 0; i < NDG; i++) {
        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
    }
}
```

Figure 8.20 dg_echo function that counts received datagrams.

udpcliserv/dgecholoop1.c

```
#include    "unp.h"

static void recvfrom_int(int);
static int count;

void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
    socklen_t len;
    char    mesg[MAXLINE];

    Signal(SIGINT, recvfrom_int);

    for ( ; ; ) {
        len = clilen;
        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

        count++;
    }
}

static void
recvfrom_int(int signo)
{
    printf("\nreceived %d datagrams\n", count);
    exit(0);
}
```

We now run the server on the host `freebsd`, a slow SPARCStation. We run the client on the RS/6000 system `aix`, connected directly with 100Mbps Ethernet. Additionally, we run `netstat -s` on the server, both before and after, as the statistics that are output tell us how many datagrams were lost. Figure 8.21 shows the output on the server.

Figure 8.21 Output on server host.

```
freebsd % netstat -s -p udp
udp:
       71208 datagrams received
       0 with incomplete header
       0 with bad data length field
       0 with bad checksum
       0 with no checksum
       832 dropped due to no socket
       16 broadcast/multicast datagrams dropped due to no socket
       1971 dropped due to full socket buffers
       0 not for hashed pcb
       68389 delivered
       137685 datagrams output
freebsd % udpserv06              start our server
                                                               we run the
client here
    ^C                              we type our interrupt key after the client
is finished
received 30 datagrams
freebsd % netstat -s -p udp
udp:
       73208 datagrams received
       0 with incomplete header
       0 with bad data length field
```

```
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        3941 dropped due to full socket buffers
        0 not for hashed pcb
        68419 delivered
        137685 datagrams output
```

The client sent 2,000 datagrams, but the server application received only 30 of these, for a 98% loss rate. There is *no* indication whatsoever to the server application or to the client application that these datagrams were lost. As we have said, UDP has no flow control and it is unreliable. It is trivial, as we have shown, for a UDP sender to overrun the receiver.

If we look at the `netstat` output, the total number of datagrams received by the server host (not the server application) is 2,000 (73,208 - 71,208). The counter "dropped due to full socket buffers" indicates how many datagrams were received by UDP but were discarded because the receiving socket's receive queue was full (p. 775 of TCPv2). This value is 1,970 (3,491 - 1,971), which when added to the counter output by the application (30), equals the 2,000 datagrams received by the host. Unfortunately, the `netstat` counter of the number dropped due to a full socket buffer is systemwide. There is no way to determine which applications (e.g., which UDP ports) are affected.

The number of datagrams received by the server in this example is not predictable. It depends on many factors, such as the network load, the processing load on the client host, and the processing load on the server host.

If we run the same client and server, but this time with the client on the slow Sun and the server on the faster RS/6000, no datagrams are lost.

```
aix % udpserv06
^?                           we type our interrupt key after the client is finished
received 2000 datagrams
```

UDP Socket Receive Buffer

The number of UDP datagrams that are queued by UDP for a given socket is limited by the size of that socket's receive buffer. We can change this with the `SO_RCVBUF` socket option, as we described in Section 7.5. The default size of the UDP socket receive buffer under FreeBSD is 42,080 bytes, which allows room for only 30 of our 1,400-byte datagrams. If we increase the size of the socket receive buffer, we expect the server to receive additional datagrams. Figure 8.22 shows a modification to the `dg_echo` function from Figure 8.20 that sets the socket receive buffer to 240 KB.

Figure 8.22 dg_echo function that increases the size of the socket receive queue.

udpcliserv/dgecholoop2.c

```
#include    "unp.h"

static void recvfrom_int(int);
static int count;

void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
    int     n;
    socklen_t len;
    char    mesg[MAXLINE];

    Signal(SIGINT, recvfrom_int);

    n = 220 * 1024;
    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

    for ( ; ; ) {
        len = clilen;
        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

        count++;
    }
}

static void
recvfrom_int(int signo)
{
    printf("\nreceived %d datagrams\n", count);
    exit(0);
}
```

If we run this server on the Sun and the client on the RS/6000, the count of received datagrams is now 103. While this is slightly better than the earlier example with the default socket receive buffer, it is no panacea.

Why do we set the receive socket buffer size to 220 x 1,024 in Figure 8.22? The maximum size of a socket receive buffer in FreeBSD 5.1 defaults to 262,144 bytes (256 x 1,024), but due to the buffer allocation policy (described in Chapter 2 of

TCPv2), the actual limit is 233,016 bytes. Many earlier systems based on 4.3BSD restricted the size of a socket buffer to around 52,000 bytes.

8.14 Determining Outgoing Interface with UDP

A connected UDP socket can also be used to determine the outgoing interface that will be used to a particular destination. This is because of a side effect of the `connect` function when applied to a UDP socket: The kernel chooses the local IP address (assuming the process has not already called `bind` to explicitly assign this). This local IP address is chosen by searching the routing table for the destination IP address, and then using the primary IP address for the resulting interface.

Figure 8.23 shows a simple UDP program that `connects` to a specified IP address and then calls `getsockname`, printing the local IP address and port.

Figure 8.23 UDP program that uses connect to determine outgoing interface.

udpcliserv/udpcli09.c

```
#include     "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd;
    socklen_t len;
    struct sockaddr_in cliaddr, servaddr;

    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>");

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

    len = sizeof(cliaddr);
    Getsockname(sockfd, (SA *) &cliaddr, &len);
    printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

    exit(0);
}
```

If we run the program on the multihomed host `freebsd`, we have the following output:

```
freebsd % udpcli09 206.168.112.96
local address 12.106.32.254:52329

freebsd % udpcli09 192.168.42.2
local address 192.168.42.1:52330

freebsd % udpcli09 127.0.0.1
local address 127.0.0.1:52331
```

The first time we run the program, the command-line argument is an IP address that follows the default route. The kernel assigns the local IP address to the primary address of the interface to which the default route points. The second time, the argument is the IP address of a system connected to a second Ethernet interface, so the kernel assigns the local IP address to the primary address of this second interface. Calling `connect` on a UDP socket does not send anything to that host; it is entirely a local operation that saves the peer's IP address and port. We also see that calling `connect` on an unbound UDP socket also assigns an ephemeral port to the socket.

Unfortunately, this technique does not work on all implementations, mostly SVR4-derived kernels. For example, this does not work on Solaris 2.5, but it works on AIX, HP-UX 11, MacOS X, FreeBSD, Linux, and Solaris 2.6 and later.

8.15 TCP and UDP Echo Server Using 'select'

We now combine our concurrent TCP echo server from Chapter 5 with our iterative UDP echo server from this chapter into a single server that uses `select` to multiplex a TCP and UDP socket. Figure 8.24 is the first half of this server.

Create listening TCP socket

14–22 A listening TCP socket is created that is bound to the server's well-known port. We set the `SO_REUSEADDR` socket option in case connections exist on this port.

Create UDP socket

23–29 A UDP socket is also created and bound to the same port. Even though the same port is used for TCP and UDP sockets, there is no need to set the `SO_REUSEADDR` socket option before this call to `bind`, because TCP ports are independent of UDP ports.

Figure 8.25 shows the second half of our server.

Establish signal handler for SIGCHLD

30 A signal handler is established for `SIGCHLD` because TCP connections will be handled by a child process. We showed this signal handler in Figure 5.11.

Prepare for select

31–32 We initialize a descriptor set for `select` and calculate the maximum of the two descriptors for which we will wait.

Figure 8.24 First half of echo server that handles TCP and UDP using select.

udpcliserv/udpservselect01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd, udpfd, nready, maxfdp1;
 6     char    mesg[MAXLINE];
 7     pid_t   childpid;
 8     fd_set  rset;
 9     ssize_t n;
10     socklen_t len;
11     const int on = 1;
12     struct sockaddr_in cliaddr, servaddr;
13     void    sig_chld(int);

14         /* create listening TCP socket */
15     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

16     bzero(&servaddr, sizeof(servaddr));
```

```
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19     servaddr.sin_port = htons(SERV_PORT);

20     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

22     Listen(listenfd, LISTENQ);

23         /* create UDP socket */
24     udpfd = Socket(AF_INET, SOCK_DGRAM, 0);

25     bzero(&servaddr, sizeof(servaddr));
26     servaddr.sin_family = AF_INET;
27     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28     servaddr.sin_port = htons(SERV_PORT);

29     Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
```

Call select

34□41 We call `select`, waiting only for readability on the listening TCP socket or readability on the UDP socket. Since our `sig_chld` handler can interrupt our call to `select`, we handle an error of `EINTR`.

Handle new client connection

42□51 We `accept` a new client connection when the listening TCP socket is readable, `fork` a child, and call our `str_echo` function in the child. This is the same sequence of steps we used in Chapter 5.

Figure 8.25 Second half of echo server that handles TCP and UDP using select.

udpcliserv/udpservselect01.c

```
30     Signal(SIGCHLD, sig_chld);    /* must call waitpid() */

31     FD_ZERO(&rset);
32     maxfdp1 = max(listenfd, udpfd) + 1;
33     for ( ; ; ) {
34         FD_SET(listenfd, &rset);
35         FD_SET(udpfd, &rset);
36         if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
37             if (errno == EINTR)
38                 continue;    /* back to for() */
39             else
40                 err_sys("select error");
41         }

42         if (FD_ISSET(listenfd, &rset)) {
43             len = sizeof(cliaddr);
44             connfd = Accept(listenfd, (SA *) &cliaddr, &len);

45             if ( (childpid = Fork()) == 0) { /* child process */
46                 Close(listenfd);     /* close listening socket */
47                 str_echo(connfd);    /* process the request */
48                 exit(0);
49              }
50              Close(connfd);     /* parent closes connected socket */
51         }

52         if (FD_ISSET(udpfd, &rset)) {
53             len = sizeof(cliaddr);
54             n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);

55             Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
56         }
57     }
58 }
```

Handle arrival of datagram

52□57 If the UDP socket is readable, a datagram has arrived. We read it with `recvfrom` and send it back to the client with `sendto`.

8.16 Summary

Converting our echo client/server to use UDP instead of TCP was simple. But lots of features provided by TCP are missing: detecting lost packets and retransmitting, verifying responses as being from the correct peer, and the like. We will return to this topic in Section 22.5 and see what it takes to add some reliability to a UDP application.

UDP sockets can generate asynchronous errors, that is, errors that are reported some time after a packet is sent. TCP sockets always report these errors to the application, but with UDP, the socket must be connected to receive these errors.

UDP has no flow control, and this is easy to demonstrate. Normally, this is not a problem, because many UDP applications are built using a request-reply model, and not for transferring bulk data.

There are still more points to consider when writing UDP applications, but we will save these until Chapter 22, after covering the interface functions, broadcasting, and multicasting.

Exercises

8.1 We have two applications, one using TCP and the other using UDP. 4,096 bytes are in the receive buffer for the TCP socket and two 2,048-byte datagrams are in the receive buffer for the UDP socket. The TCP application calls `read` with a third argument of 4,096 and the UDP application calls `recvfrom` with a third argument of 4,096. Is there any difference?

8.2 What happens in Figure 8.4 if we replace the final argument to `sendto` (which we show as `len`) with `clilen`?

8.3 Compile and run the UDP server in Figures 8.3 and 8.4 and then the UDP client in Figures 8.7 and 8.8. Verify that the client and server work together.

8.4 Run the `ping` program in one window, specifying the `-i 60` option (send one packet every 60 seconds; some systems use `-I` instead of `-i`), the `-v` option (print all received ICMP errors), and the loopback address (normally 127.0.0.1). We will use this program to see the ICMP port unreachable returned by the server host. Next, run our client from the previous exercise in another window, specifying the IP address of some host that is not running the server. What happens?

8.5 We said with Figure 8.5 that each connected TCP socket has its own socket receive buffer. What about the listening socket; do you think it has its own socket receive buffer?

8.6 Use the `sock` program (Section C.3) and a tool such as `tcpdump` (Section C.5) to test what we claimed in Section 8.10: If the client `binds` an IP address to its socket but sends a datagram that goes out some other interface, the resulting IP datagram still contains the IP address that was bound to the socket, even though this does not correspond to the outgoing interface.

8.7 Compile the programs from Section 8.13 and run the client and server on different hosts. Put a `printf` in the client each time a datagram is written to the socket. Does this change the percentage of received packets? Why? Put a `printf` in the server each time a datagram is read from the socket. Does this change the percentage of received packets? Why?

8.8 What is the largest length that we can pass to `sendto` for a UDP/IPv4 socket, that is, what is the largest amount of data that can fit into a UDP/IPv4 datagram? What changes with UDP/IPv6?

Modify Figure 8.8 to send one maximum-size UDP datagram, read it back, and print the number of bytes returned by `recvfrom`.

8.9 Modify Figure 8.25 to conform to RFC 1122 by using `IP_RECVDSTADDR` for the UDP socket.

Chapter 9. Elementary SCTP Sockets