Programming With Sockets 2

This chapter presents the socket interface and illustrates them with sample programs. The programs demonstrate the Internet domain sockets.

What Are Sockets	*page 12*
Socket Tutorial	*page 14*
Standard Routines	*page 31*
Client-Server Programs	*page 34*
Advanced Topics	*page 41*
Moving Socket Applications to Solaris 2.x	*page 54*

Sockets are Multithread Safe

The interface described in this chapter is multithread safe. Applications that contain socket function calls can be used freely in a multithreaded application.

SunOS Binary Compatibility

There are two major changes from SunOS 4.x that hold true for Solaris 2.x releases. The binary compatibility package allows SunOS 4.x–based dynamically linked socket applications to run in Solaris 2.x.

1. You must explicitly specify the socket library (`-lsocket`) on the compilation line.

2

2. You must recompile all SunOS 4.x socket-based applications with the socket library to run under Solaris 2.x. The differences in the two socket implementations are outlined in “Moving Socket Applications to Solaris 2.x” on page 54.

What Are Sockets

Sockets are the 4.2 Berkeley software distribution (BSD) UNIX interface to network protocols. It has been an integral part of SunOS releases since 1981. They are commonly referred to as Berkeley sockets or BSD sockets. Since the days of early UNIX, applications have used the file system model of input/output to access devices and files. The file system model is sometimes called *open-close-read-write* after the basic system calls used in this model. However, the interaction between user processes and network protocols are more complex than the interaction between user processes and I/O devices.

A socket is an endpoint of communication to which a name can be bound. A socket has a *type* and one associated process. Sockets were designed to implement the client-server model for interprocess communication where:

- The interface to network protocols needs to accommodate multiple communication protocols, such as TCP/IP, XNS, and UNIX domain.
- The interface to network protocols need to accommodate server code that waits for connections and client code that initiates connections.
- They also need to operate differently, depending on whether communication is connection-oriented or connectionless.
- Application programs may wish to specify the destination address of the datagrams it delivers instead of binding the address with the `open()` call.

To address these issues and others, sockets are designed to accommodate network protocols, while still behaving like UNIX files or devices whenever it makes sense to. Applications create sockets when they need to. Sockets work with the `open()`, `close()`, `read()`, and `write()` system calls, and the operating system can differentiate between the file descriptors for files, and file descriptors for sockets.

UNIX domain sockets are named with UNIX paths. For example, a socket may be named `/tmp/foo`. UNIX domain sockets communicate only between processes on a single host. Sockets in the UNIX domain are not considered part

of the network protocols because they can only be used to communicate with processes within the same UNIX system. They are rarely used today and are only briefly covered in this manual.

Socket Libraries

The socket interface routines are in a library that must be linked with the application. The libraries `libsocket.so` and `libsocket.a` are contained in `/usr/lib` with the rest of the system service libraries. The difference is that `libsocket.so` is used for dynamic linking, whereas `libsocket.a` is used for static linking. Static linking is *strongly* discouraged.

Socket Types

Socket types define the communication properties visible to a user. The Internet domain sockets provide access to the TCP/IP transport protocols. The Internet domain is identified by the value `AF_INET`. Sockets exchange data only with sockets in the same domain.

Three types of sockets are supported:

1. Stream sockets allow processes to communicate using TCP. A stream socket provides bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. Once the connection has been established, data can be read from and written to these sockets as a byte stream. The socket type is `SOCK_STREAM`.

2. Datagram sockets allow processes to use UDP to communicate. A datagram socket supports bidirectional flow of messages. A process on a datagram socket may receive messages in a different order from the sending sequence and may receive duplicate messages. Record boundaries in the data are preserved. The socket type is `SOCK_DGRAM`.

3. Raw sockets provide access to ICMP. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. They are provided to support developing new communication protocols or for access to more esoteric facilities of an existing protocol. Only superuser processes may use raw sockets. The socket type is `SOCK_RAW`. See “Selecting Specific Protocols” on page 46 for further information.

2

Socket Tutorial

This section covers the basic methodologies of using sockets.

Socket Creation

The `socket()` call creates a socket,

```
s = socket(domain, type, protocol);
```

in the specified domain and of the specified type. If the protocol is unspecified (a value of `0`), the system selects a protocol that supports the requested socket type. The socket handle (a file descriptor) is returned.

The domain is specified by one of the constants defined in `<sys/socket.h>`. For the UNIX domain the constant is `AF_UNIX`. For the Internet domain it is `AF_INET`. Constants named AF_<*suite*> specify the address format to use in interpreting names.

Socket types are defined in `<sys/socket.h>`. `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW` is supported by `AF_INET` and `AF_UNIX`. The following creates a stream socket in the Internet domain:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call results in a stream socket with the TCP protocol providing the underlying communication. A datagram socket for intramachine use is created by:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Use the default protocol (the *protocol* argument is `0`) in most situations. You can specify a protocol other than the default, as described in "Advanced Topics" on page 41.

Binding Local Names

A socket is created with no name. A remote process has no way to refer to a socket until an address is bound to it. Communicating processes are connected through addresses. In the Internet domain, a connection is composed of local and remote addresses, and local and remote ports. In the UNIX domain, a connection is composed of (usually) one or two path names. In most domains, connections must be unique.

In the Internet domain, there may never be duplicate ordered sets, such as: <protocol, local address, local port, foreign address, foreign port>. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate ordered sets such as: <local pathname, foreign pathname>. The path names may not refer to existing files.

The bind() call allows a process to specify the local address of the socket. This forms the set <local address, local port> (or <local pathname>) while connect() and accept() complete a socket's association. The bind() system call is used as follows:

```
bind (s, name, namelen);
```

s is the socket handle. The bound name is a byte string that is interpreted by the supporting protocol(s). Internet domain names contain an Internet address and port number. UNIX domain names contain a path name and a family. Code Example 2-1 binds the name /tmp/foo to a UNIX domain socket.

Code Example 2-1 Bind Name to Socket

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
     strlen(addr.sun_path) + sizeof (addr.sun_family));
```

Note that in determining the size of an AF_UNIX socket address, null bytes are not counted, which is why strlen() use is fine.

The file name referred to in addr.sun_path is created as a socket in the system file name space. The caller must have write permission in the directory where addr.sun_path is created. The file should be deleted by the caller when it is no longer needed. AF_UNIX sockets can be deleted with unlink().

Binding an Internet address is more complicated. The call is similar:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
 ...
bind (s, (struct sockaddr *) &sin, sizeof sin);
```

The content of the address `sin` is described in "Address Binding" on page 47, where Internet address bindings are discussed.

Connection Establishment

Connection establishment is usually asymmetric, with one process acting as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the `connect()` call initiates a connection. In the UNIX domain, this might appear as:

```
struct sockaddr_un server;
server.sun.family = AF_UNIX;
 ...
connect(s, (struct sockaddr *)&server,
        strlen(server.sun_path) + sizeof (server.sun_family));
```

while in the Internet domain it might be:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. See "Signals and Process Group ID" on page 45. This is the usual way that local addresses are bound to a socket on the client side.

In the examples that follow, only `AF_INET` sockets are described.

To receive a client's connection, a server must perform two steps after binding its socket. The first is to indicate how many connection requests can be queued. The second step is to accept a connection:

```
struct sockaddr_in from;
 ...
listen(s, 5);                 /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

s is the socket bound to the address to which the connection request is sent. The second parameter of `listen()` specifies the maximum number of outstanding connections that may be queued. `from` is a structure that is filled

with the address of the client. A `NULL` pointer may be passed. *fromlen* is the length of the structure. (In the UNIX domain, `from` is declared a `struct sockaddr_un`.)

`accept()` normally blocks. `accept()` returns a new socket descriptor that is connected to the requesting client. The value of *fromlen* is changed to the actual size of the address.

There is no way for a server to indicate that it will accept connections only from specific addresses. The server can check the from-address returned by `accept()` and close a connection with an unacceptable client. A server can accept connections on more than one socket, or avoid blocking on the accept call. These techniques are presented in "Advanced Topics" on page 41.

Connection Errors

An error is returned if the connection is unsuccessful (however, an address bound by the system remains). Otherwise, the socket is associated with the server and data transfer may begin.

Table 2-2 lists some of the more common errors returned when a connection attempt fails.

Table 2-2 Socket Connection Errors

Socket Errors	Error Description
`ENOBUFS`	Lack of memory available to support the call.
`EPROTONOSUPPORT`	Request for an unknown protocol.
`EPROTOTYPE`	Request for an unsupported type of socket.
`ETIMEDOUT`	No connection established in specified time. This happens when the destination host is down or when problems in the network result in lost transmissions.
`ECONNREFUSED`	The host refused service. This happens when a server process is not present at the requested address.
`ENETDOWN or EHOSTDOWN`	These errors are caused by status information delivered by the underlying communication interface.

Table 2-2 Socket Connection Errors *(Continued)*

Socket Errors	Error Description
ENETUNREACH or EHOSTUNREACH	These operational errors can occur either because there is no route to the network or host, or because of status information returned by intermediate gateways or switching nodes. The status returned is not always sufficient to distinguish between a network that is down and a host that is down.

Data Transfer

This section describes the functions to send and receive data. You can send or receive a message with the normal `read()` and `write()` system calls:

```
write(s, buf, sizeof buf);
read(s, buf, sizeof buf);
```

Or the calls `send()` and `recv()` can be used:

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

`send()` and `recv()` are very similar to `read()` and `write()`, but the `flags` argument is important. The flags, defined in `<sys/socket.h>`, can be specified as a nonzero value if one or more of the following is required:

`MSG_OOB`	send and receive out-of-band data
`MSG_PEEK`	look at data without reading
`MSG_DONTROUTE`	send data without routing packets

Out-of-band data is specific to stream sockets. When `MSG_PEEK` is specified with a `recv()` call, any data present is returned to the user but treated as still unread. The next `read()` or `recv()` call on the socket returns the same data. The option to send data without routing applied to the outgoing packets is currently used only by the routing table management process and is unlikely to be interesting to most users.

Closing Sockets

A `SOCK_STREAM` socket can be discarded by a `close()` system call. If data is queued to a socket that promises reliable delivery after a `close()`, the protocol continues to try to transfer the data. If the data is still undelivered after an arbitrary period, it is discarded.

`shutdown()` closes `SOCK_STREAM` sockets gracefully. Both processes can acknowledge that they are no longer sending. This call has the form:

```
shutdown(s, how);
```

where `how` is `0` disallows further receives, `1` disallows further sends, and `2` disallows both.

Connecting Stream Sockets

Figure 2-1 and the next two code examples illustrate initiating and accepting an Internet domain stream connection.

To initiate a connection, the client program in Code Example 2-2 creates a stream socket and calls `connect()`, specifying the address of the socket to connect to. If the target socket exists and the request is accepted, the connection is complete and the program can send data. Data are delivered in sequence with no message boundaries. The connection is destroyed when either socket is closed. For more information about data representation routines, such as `ntohl()`, `ntohs()`, `htons()`, and `htonl()`, in this program, see the `byteorder(3N)` man page.

The program in Code Example 2-3 is a server. It creates a socket and binds a name to it, then displays the port number. The program calls `listen()` to mark the socket ready to accept connection requests and initialize a queue for the requests. The rest of the program is an infinite loop. Each pass of the loop accepts a new connection and removes it from the queue, creating a new socket. The server reads and displays the messages from the socket and closes it. The use of `INADDR_ANY` is explained in “Address Binding” on page 47.

2

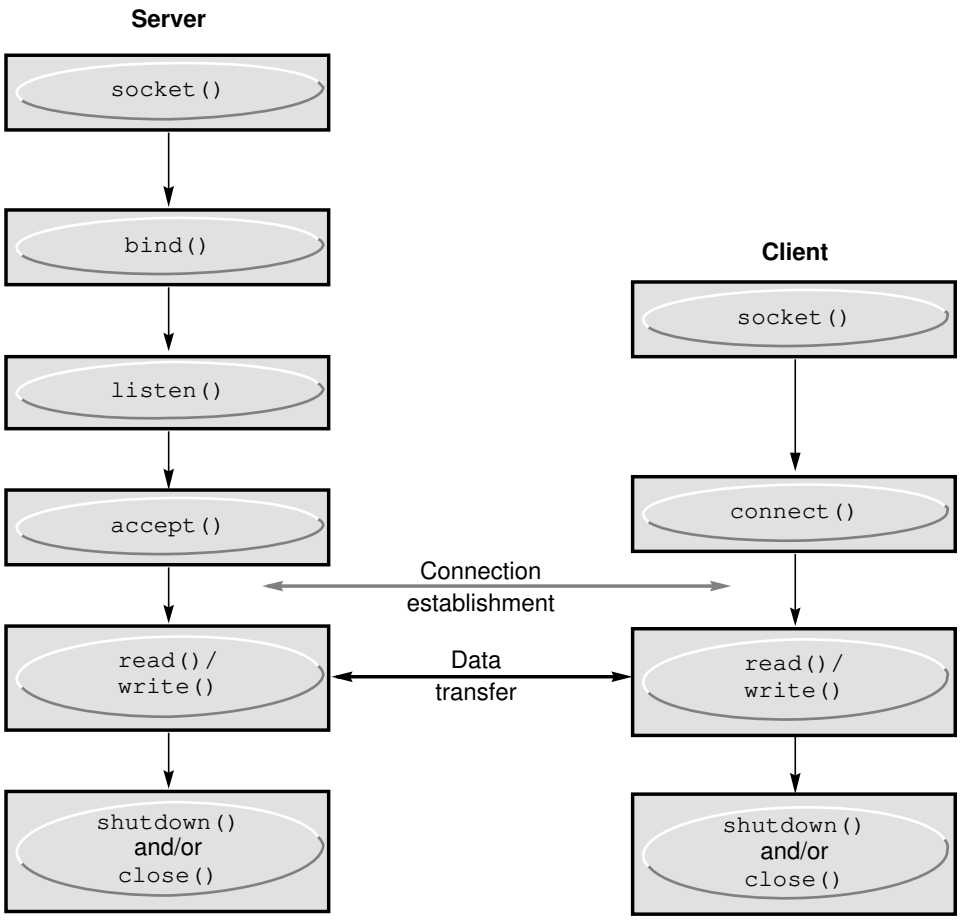

Figure 2-1 Connection-Oriented Communication Using Stream Sockets

Code Example 2-2 Internet Domain Stream Connection (Client)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."
```

```
/*
 * This program creates a socket and initiates a connection with the
 * socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 * Usage: pgm host port
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_in server;
    struct hostent *hp, *gethostbyname();
    char buf[1024];

    /* Create socket. */
    sock = socket( AF_INET, SOCK_STREAM, 0 );
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    server.sin_family = AF_INET;
    hp = gethostbyname(argv[1] );
/*
 * gethostbyname returns a structure including the network address
 * of the specified host.
 */
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    memcpy((char *) &server.sin_addr, (char *) hp->h_addr,
        hp->h_length);
    server.sin_port = htons(atoi( argv[2]));
    if (connect(sock, (struct sockaddr *) &server, sizeof server)
        == -1) {
        perror("connecting stream socket");
        exit(1);
    }
    if (write( sock, DATA, sizeof DATA ) == -1)
        perror("writing on stream socket");
```

```
    close(sock);
    exit(0);
}
```

Code Example 2-3 Accepting an Internet Stream Connection (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop.
 * Each time through the loop it accepts a connection and prints
 * data from it. When the connection breaks, or the client closes
 * the connection, the program accepts a new connection.
 */

main()
{
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;

    /* Create socket. */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Bind socket using wildcards.*/
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = 0;
    if (bind(sock, (struct sockaddr *) &server, sizeof server)
        == -1)
        perror("binding stream socket");
        exit(1);
    }
    /* Find out assigned port number and print it out. */
    length = sizeof server;
    if (getsockname(sock,(struct sockaddr *) &server,&length)
```

```
        == -1) {
            perror("getting socket name");
            exit(1);
        }
        printf("Socket port #%d\n", ntohs(server.sin_port));
        /* Start accepting connections. */
        listen(sock, 5);
        do {
            msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
            if (msgsock == -1
                perror("accept");
            else do {
                memset(buf, 0, sizeof buf);
                if ((rval = read(msgsock,buf, 1024)) == -1)
                    perror("reading stream message");
                if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval != 0);
            close(msgsock);
        } while(TRUE);
        /*
         * Since this program has an infinite loop, the socket "sock" is
         * never explicitly closed. However, all sockets will be closed
         * automatically when a process is killed or terminates normally.
         */
         exit(0);
}
```

Datagram Sockets

A datagram socket provides a symmetric data exchange interface. There is no requirement for connection establishment. Each message carries the destination address. Figure 2-2 shows the flow of communication between server and client.

Datagram sockets are created as described in “Socket Creation” on page 14. If a particular local address is needed, the `bind()` operation must precede the first data transmission. Otherwise, the system sets the local address and/or port when data is first sent. To send data, the `sendto()` call is used:

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are the same as in connection-oriented sockets. The *to* and *tolen* values indicate the address of the intended recipient of the message. A locally detected error condition (such as an unreachable network) causes a return of `-1` and *errno* to be set to the error number.

To receive messages on a datagram socket, the `recvfrom()` call is used:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from,
          &fromlen);
```

Before the call, *fromlen* is set to the size of the *from* buffer. On return it is set to the size of the address from which the datagram was received.

Datagram sockets can also use the `connect()` call to associate a socket with a specific destination address. It can then use the `send()` call. Any data sent on the socket without explicitly specifying a destination address is addressed to the connected peer, and only data received from that peer is delivered. Only one connected address is permitted for one socket at a time. A second `connect()` call changes the destination address. Connect requests on datagram sockets return immediately. The system simply records the peer's address. `accept()`, and `listen()` are not used with datagram sockets.

While a datagram socket is connected, errors from previous `send()` calls may be returned asynchronously. These errors can be reported on subsequent operations on the socket, or an option of `getsockopt`, `SO_ERROR`, can be used to interrogate the error status.

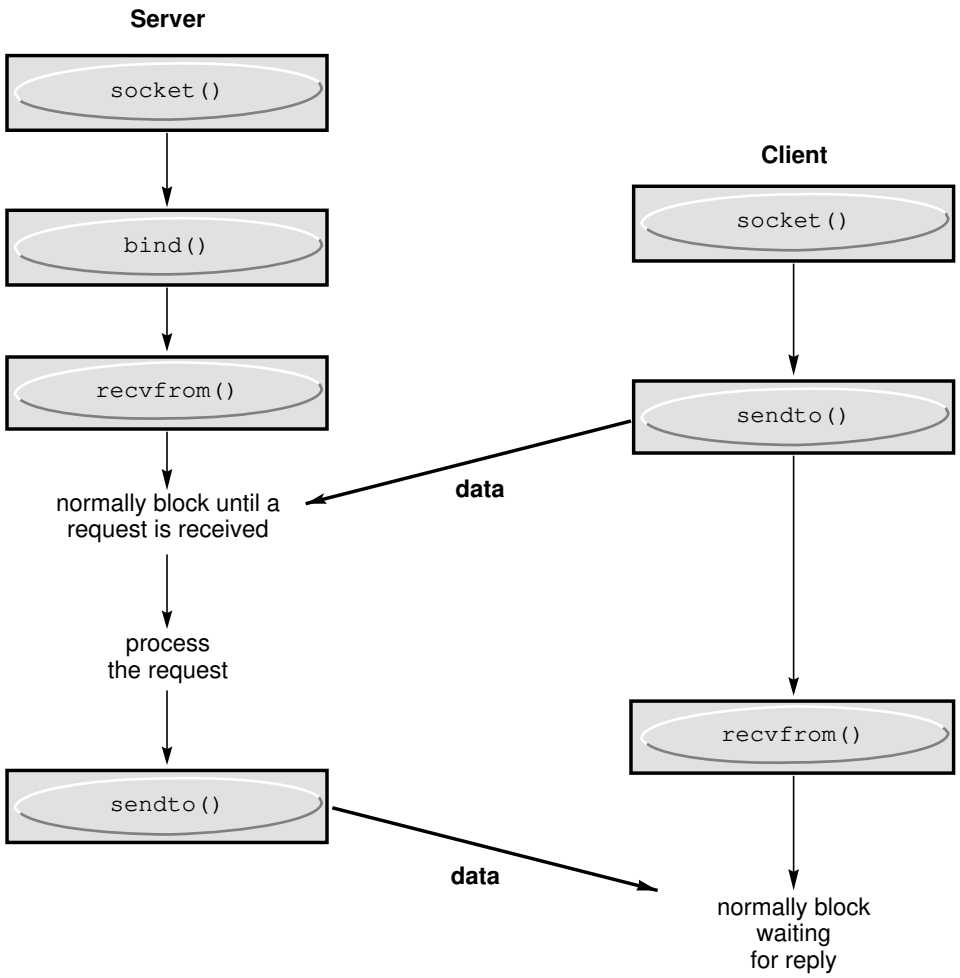

Figure 2-2 Connectionless Communication Using Datagram Sockets

Code Example 2-4 shows how to read an Internet call, and Code Example 2-5 shows how to send an Internet call.

Code Example 2-4 Reading Internet Domain Datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <stdio.h>

/*
 * The include file <netinet/in.h> defines sockaddr_in as:
 *  struct sockaddr_in {
 *      short       sin_family;
 *      u_short     sin_port;
 *      struct      in_addr sin_addr;
 *      char        sin_zero[8];
 * };
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */

main()
{
    int sock, length;
    struct sockaddr_in name;
    char buf[1024];

    /* Create socket from which to read. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name with wildcards. */
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = 0;
    if (bind(sock,(struct sockaddr *)&name, sizeof name) == -1) {
        perror("binding datagram socket");
        exit(1);
    }
    /* Find assigned port value and print it out. */
    length = sizeof(name);
    if (getsockname(sock,(struct sockaddr *) &name, &length)
        == -1) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs( name.sin_port));
    /* Read from the socket. */
    if ( read(sock, buf, 1024) == -1 )
        perror("receiving datagram packet");
    printf("-->%s\n", buf);
```

```
    close(sock);
    exit(0);
}
```

Code Example 2-5 Sending an Internet Domain Datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the
 * command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_in name;
    struct hostent *hp, *gethostbyname();

    /* Create socket on which to send. */
    sock = socket(AF_INET,SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
     * Construct name, with no wildcards, of the socket to ``send''
     * to. gethostbyname returns a structure including the network
     * address of the specified host. The port number is taken from
     * the command line.
     */
    hp = gethostbyname(argv[1]);
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    memcpy((char *) &name.sin_addr, (char *) hp->h_addr,
            hp->h_length);
    name.sin_family = AF_INET;
```

```
    name.sin_port = htons( atoi( argv[2] ));
    /* Send message. */
    if (sendto(sock,DATA, sizeof DATA ,0,
        (struct sockaddr *) &name,sizeof name) == -1)
        perror("sending datagram message");
    close(sock);
    exit(0);
}
```

Input/Output Multiplexing

Requests can be multiplexed among multiple sockets or files. The `select()` call is used to do this:

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The first argument of `select()` is the number of file descriptors in the lists pointed to by the next three arguments.

The second, third, and fourth arguments of `select()` are pointers to three sets of file descriptors: a set of descriptors to read on, a set to write on, and a set on which exception conditions are accepted. Out-of-band data is the only exceptional condition. Any of these pointers can be a properly cast null. Each set is a structure containing an array of long integer bit masks. The size of the array is set by `FD_SETSIZE` (defined in `select.h`). The array is long enough to hold one bit for each `FD_SETSIZE` file descriptor.

The macros `FD_SET` (*fd*, &*mask*) and `FD_CLR` (*fd*, &*mask*) add and delete, respectively, the file descriptor *fd* in the set `mask`. The set should be zeroed before use, and the macro `FD_ZERO`(&*mask*) clears the set `mask`.

A time-out value may be specified. If the `timeout` pointer is `NULL`, `select()` blocks until a descriptor is selectable, or until a signal is received. If the fields in `timeout` are set to `0`, `select()` polls and returns immediately.

`select()` normally returns the number of file descriptors selected. `select()` returns a `0` if the time-out has expired. `select()` returns `-1` for an error or interrupt with the error number in *errno* and the file descriptor masks unchanged.

For a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

Test the status of a file descriptor in a select mask with the `FD_ISSET`(*fd*, &*mask*) macro. It returns a nonzero value if *fd* is in the set `mask`, and `0` if it is not. Use `select()` followed by a `FD_ISSET` (*fd*, &*mask*) macro on the read set to check for queued connect requests on a socket.

Code Example 2-6 shows how to select on a "listening" socket for readability to determine when a new connection can be picked up with a call to `accept()`. The program accepts connection requests, reads data, and disconnects on a single socket.

Code Example 2-6 Check for Pending Connections With `select()`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
 */

main()
{
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;

    /* Open a socket and bind it as in previous examples. */
```

```
    /* Start accepting connections. */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        to.tv_usec = 0;
        if (select(1, &ready, (fd_set *)0, (fd_set *)0, &to) == -1) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0,
              (int *)0);
            if (msgsock == -1)
                perror("accept");
            else do {
                memset(buf, 0, sizeof buf);
                if ((rval = read(msgsock, buf, 1024)) == -1)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
    } while (TRUE);
    exit(0);
}
```

In previous versions of the `select()` routine, its arguments were pointers to integers instead of pointers to `fd_sets`. This style of call still works if the number of file descriptors is smaller than the number of bits in an integer.

`select()` provides a synchronous multiplexing scheme. The `SIGIO` and `SIGURG` signals described in "Advanced Topics" on page 41 provide asynchronous notification of output completion, input availability, and exceptional conditions.

Standard Routines

You may need to locate and construct network addresses. This section describes the new routines that manipulate network addresses. Unless otherwise stated, functions presented in this section apply only to the Internet domain.

Locating a service on a remote host requires many levels of mapping before client and server communicate. A service has a name for human use. The service and host names must be translated to network addresses. Finally, the address is used to locate and route to the host. The specifics of the mappings may vary between network architectures. Preferably, a network will not require that hosts be named, thus protecting the identity of their physical locations. It is more flexible to discover the location of the host when it is addressed.

Standard routines map host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers, and the appropriate protocol to use in communicating with the server process. The file `<netdb.h>` must be included when using any of these routines.

Host Names

An Internet host name to address mapping is represented by the `hostent` structure:

```
struct hostent {
    char    *h_name;          /* official name of host */
    char    **h_aliases;      /* alias list */
    int     h_addrtype;       /* hostaddrtype(e.g.,AF_INET) */
    int     h_length;         /* length of address */
    char    **h_addr_list;    /* list of addrs, null terminated */
};
/*1st addr, net byte order*/
#define h_addr h_addr_list[0]
```

`gethostbyname()` maps an Internet host name to a `hostent` structure.
`gethostbyaddr()` maps an Internet host address to a `hostent` structure.
`inet_ntoa()` maps an Internet host address to a displayable string.

The routines return a `hostent` structure containing the name of the host, its aliases, the address type (address family), and a `NULL`-terminated list of variable length addresses. The list of addresses is required because a host can have many addresses. The `h_addr` definition is for backward compatibility, and is the first address in the list of addresses in the `hostent` structure.

Network Names

There are routines to map network names to numbers, and back. These routines return a `netent` structure:

```
/*
 * Assumes that a network number fits in 32 bits.
 */
struct netent {
    char    *n_name;      /* official name of net */
    char    **n_aliases;  /* alias list */
    int     n_addrtype;   /* net address type */
    int     n_net;        /* net number, host byte order */
};
```

`getnetbyname()`, `getnetbyaddr()`, and `getnetent()` are the network counterparts to the host routines described above.

Protocol Names

The `protoent` structure defines the protocol-name mapping used with `getprotobyname()`, `getprotobynumber()`, and `getprotoent()`:

```
struct protoent {
    char    *p_name;      /* official protocol name */
    char    **p_aliases;  /* alias list */
    int     p_proto;      /* protocol number */
};
```

In the UNIX domain, no protocol database exists.

Service Names

An Internet domain service resides at a specific, well-known port and uses a particular protocol. A service name to port number mapping is described by the `servent` structure:

```
struct servent {
    char    *s_name;     /* official service name */
    char    **s_aliases; /* alias list */
    int     s_port;      /* port number, network byte order */
    char    *s_proto;    /* protocol to use */
};
```

`getservbyname()` maps service names and, optionally, a qualifying protocol to a `servent` structure. The call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification of a telnet server using any protocol. The call

```
sp = getservbyname("telnet", "tcp");
```

returns the telnet server that uses the TCP protocol. `getservbyport()` and `getservent()` are also provided. `getservbyport()` has an interface similar to that of `getservbyname()`; an optional protocol name may be specified to qualify lookups.

Other Routines

In addition to address-related database routines, there are several other routines that simplify manipulating names and addresses. Table 2-3 summarizes the routines for manipulating variable-length byte strings and byte-swapping network addresses and values.

Table 2-3 Run-Time Library Routines

Call	Synopsis
`memcmp` (*s1*, *s2*, *n*)	Compares byte-strings; `0` if same, not `0` otherwise
`memcpy` (*s1*, *s2*, *n*)	Copies n bytes from *s2* to *s1*
`memset` (*base*, *value*, *n*)	Sets *n* bytes to `value` starting at `base`
`htonl` (*val*)	32-bit quantity from host into network byte order
`htons` (*val*)	16-bit quantity from host into network byte order
`ntohl` (*val*)	32-bit quantity from network into host byte order
`ntohs` (*val*)	16-bit quantity from network into host byte order

The byte-swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures the host byte ordering is different from network byte order, so, programs must sometimes byte-swap values. Routines that return network addresses do so in network order. There are byte-swapping problems only when interpreting network addresses. For example, the following code formats a TCP or UDP port:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On certain machines, where these routines are not needed, they are defined as null macros.

Client-Server Programs

The most common form of distributed application is the client/server model. In this scheme, client processes request services from a server process.

An alternate scheme is a service server that can eliminate dormant server processes. An example is `inetd`, the Internet service daemon. `inetd` listens at a variety of ports, determined at start up by reading a configuration file. When a connection is requested on an `inetd` serviced port, `inetd` spawns the appropriate server to serve the client. Clients are unaware that an intermediary has played any part in the connection. `inetd` is described in more detail in “inetd Daemon” on page 53.

Servers

Most servers are accessed at well-known Internet port numbers or UNIX domain names. Code Example 2-7 illustrates the main loop of a remote-login server.

Code Example 2-7 Remote Login Server

```
main(argc, argv)
    int argc;
    char *argv[];
{
    int f;
    struct sockaddr_in from;
    struct sockaddr_in sin;
    struct servent *sp;
```

```
    sp = getservbyname("login", "tcp");
    if (sp == (struct servent *) NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service");
        exit(1);
    }
    ...
#ifndef DEBUG
    /* Disassociate server from controlling terminal. */
    ...
#endif
    sin.sin_port = sp->s_port;/* Restricted port */
    sin.sin_addr.s_addr = INADDR_ANY;
    ...
    f = socket(AF_INET, SOCK_STREAM, 0);
    ...
    if (bind( f, (struct sockaddr *) &sin, sizeof sin ) == -1) {
        ...
    }
    ...
    listen(f, 5);
    while (TRUE) {
        int g, len = sizeof from;
        g = accept(f, (struct sockaddr *) &from, &len);
        if (g == -1) {
            if (errno != EINTR)
                syslog(LOG_ERR, "rlogind: accept: %m");
            continue;
        }
        if (fork() == 0) {
            close(f);
            doit(g, &from);
        }
        close(g);
    }
    exit(0);
}
```

First, the server gets its service definition, as Code Example 2-8 shows.

Code Example 2-8 Remote Login Server: Step 1

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
    fprintf(stderr, "rlogind: tcp/login: unknown service\n");
    exit(1);
}
```

The result from `getservbyname()` is used later to define the Internet port at which the program listens for service requests. Some standard port numbers are in `/usr/include/netinet/in.h`.

In the non-`DEBUG` mode of operation, the server dissociates from the controlling terminal of its invoker, shown in Code Example 2-9.

Code Example 2-9 Dissociating from the Controlling Terminal

```
(void) close(0);
(void) close(1);
(void) close(2);
(void) open("/", O_RDONLY);
(void) dup2(0, 1);
(void) dup2(0, 2);
setsid();
```

This prevents the server from receiving signals to the process group of the controlling terminal. Once a server has dissociated itself, it cannot send reports of errors to a terminal and must log errors with `syslog()`.

A server next creates a socket and listens for service requests. `bind()` insures that the server listens at the expected location. (The remote login server listens at a restricted port number, so it runs as super-user.)

Code Example 2-10 illustrates the main body of the loop.

Code Example 2-10 Remote Login Server: Main Body

```
while(TRUE) {
    int g, len = sizeof(from);
    if (g = accept(f, (struct sockaddr *) &from, &len) == -1) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    if (fork() == 0) {                              /* Child */
        close(f);
        doit(g, &from);
    }
    close(g);                                       /* Parent */
}
```

`accept()` blocks until a client requests service. `accept()` returns a failure indication if it is interrupted by a signal such as `SIGCHLD`. The return value from `accept()` is checked and an error is logged with `syslog()` if an error has occurred.

The server then forks a child process and invokes the main body of the remote login protocol processing. The socket used by the parent to queue connection requests is closed in the child. The socket created by `accept()` is closed in the parent. The address of the client is passed to `doit()` for authenticating clients.

Clients

This section describes the steps taken by the client remote login process. As in the server, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
    fprintf(stderr,"rlogin: tcp/login: unknown service");
    exit(1);
}
```

Next, the destination host is looked up with a `gethostbyname()` call:

```
hp = gethostbyname(argv[1]);
if (hp == (struct hostent *) NULL) {
    fprintf(stderr, "rlogin: %s: unknown host", argv[1]);
    exit(2);
}
```

Then, connect to the server at the requested host and start the remote login protocol. The address buffer is cleared and filled with the Internet address of the foreign host and the port number at which the login server listens:

```
memset((char *) &server, 0, sizeof server);
memcpy((char*) &server.sin_addr,hp->h_addr,hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. `connect()` implicitly does a `bind()`, since `s` is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
    perror("rlogin: socket");
    exit(3);
}
```

```
 ...
if (connect(s, (struct sockaddr *) &server, sizeof server) < 0) {
    perror("rlogin: connect");
    exit(4);
}
```

Connectionless Servers

Some services use datagram sockets. The `rwho` service provides status information on hosts connected to a local area network. (Avoid running `in.rwho`; it causes heavy network traffic.) This service requires the ability to broadcast information to all hosts connected to a particular network. It is an example of datagram socket use.

A user on a host running the `rwho` server may get the current status of another host with `ruptime`. Typical output is illustrated in Code Example 2-11.

Code Example 2-11 Output of `ruptime` Program

```
itchy up 9:45, 5 users, load 1.15, 1.39, 1.31
scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59
click up 10:10, 0 users, load 0.27, 0.15, 0.14
clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65
ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41
dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56
peninsula down 0:24
wood down 17:04
carpediem down 16:09
chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

Status information is periodically broadcast by the `rwho` server processes on each host. The server process also receives the status information and updates a database. This database is interpreted for the status of each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Use of broadcast is fairly inefficient, since a lot of net traffic is generated. Unless the service is used widely and frequently, the expense of periodic broadcasts outweighs the simplicity.

A simplified version of the `rwho` server is shown in Code Example 2-12. It does two tasks: receives status information broadcast by other hosts on the network and supplies the status of its host. The first task is done in the main loop of the program: Packets received at the `rwho` port are checked to be sure they were

sent by another `rwho` server process, and are stamped with the arrival time. They then update a file with the status of the host. When a host has not been heard from for an extended time, the database routines assume the host is down and logs it. This application is prone to error, as a server may be down while a host is up.

Code Example 2-12 `rwho` Server

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    net = getnetbyname("localnet");
    sin.sin_addr = inet_makeaddr(net->n_net, INADDR_ANY);
    sin.sin_port = sp->s_port;
    ...
    s = socket(AF_INET, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
              == -1) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof sin);
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    while(1) {
        struct whod wd;
        int cc, whod, len = sizeof from;
        cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
             (struct sockaddr *) &from, &len);
        if (cc <= 0) {
            if (cc == -1 && errno != EINTR)
                syslog(LOG_ERR, "rwhod: recv: %m");
            continue;
        }
        if (from.sin_port != sp->s_port) {
            syslog(LOG_ERR, "rwhod: %d: bad from port",
                   ntohs(from.sin_port));
            continue;
        }
        ...
        if (!verify( wd.wd_hostname)) {
```

```
                syslog(LOG_ERR, "rwhod: bad host name from %x",
                       ntohl(from.sin_addr.s_addr));
                continue;
            }
            (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
            whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
            ...
            (void) time(&wd.wd_recvtime);
            (void) write(whod, (char *) &wd, cc);
            (void) close(whod);
        }
        exit(0);
}
```

The second server task is to supply the status of its host. This requires periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other `rwho` servers to hear. This task is run by a timer and triggered with a signal. Locating the system status information is involved but uninteresting.

Status information is broadcast on the local network. For networks that do not support broadcast, another scheme must be used.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe problem. The system isolates host-specific data from applications by providing system calls that return the required data. (For example, `uname()` returns the host's official name.) The `ioctl()` call lets you find the networks to which a host is directly connected. A local network broadcasting mechanism has been implemented at the socket level. Combining these two features lets a process broadcast on any directly connected local network that supports broadcasting in a site-independent manner. This solves the problem of deciding how to propagate status with `rwho`, or more generally in broadcasting. Such status is broadcast to connected networks at the socket level, where the connected networks have been obtained through the appropriate `ioctl()` calls. "Broadcasting and Determining Network Configuration" on page 49 details the specifics of broadcasting.

Advanced Topics

For most programmers, the mechanisms already described are enough to build distributed applications. Others will need some of the features in this section.

Out-of-Band Data

The stream socket abstraction includes out-of-band data. Out-of-band data is a logically independent transmission channel between a pair of connected stream sockets. Out-of-band data is delivered independently of normal data. The out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data, and at least one message may be pending delivery at any time.

For communications protocols that support only in-band signaling (that is, urgent data is delivered in sequence with normal data), the message is extracted from the normal data stream and stored separately. This lets users choose between receiving the urgent data in order and receiving it out of sequence, without having to buffer the intervening data.

You can peek (with `MSG_PEEK`) at out-of-band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by `SIGURG` with the appropriate `fcntl()` call, as described in "Interrupt Driven Socket I/O" on page 44 for `SIGIO`. If multiple sockets have out-of-band data waiting delivery, a `select()` call for exceptional conditions can be used to determine the sockets with such data pending.

A logical mark is placed in the data stream at the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal is received, all data up to the mark in the data stream is discarded.

To send an out-of-band message, the `MSG_OOB` flag is applied to `send()` or `sendto()`. To receive out-of-band data, specify `MSG_OOB` to `recvfrom()` or `recv()` (unless out-of-band data is taken in line, in which case the `MSG_OOB` flag is not needed). The `SIOCATMARK ioctl` tells whether the read pointer currently points at the mark in the data stream:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is `1` on return, the next read returns data after the mark. Otherwise, assuming out-of-band data has arrived, the next read provides data sent by the client before sending the out-of-band signal. The routine in the remote login process that flushes output on receipt of an interrupt or quit signal is shown in Code Example 2-13. This code reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

Code Example 2-13 Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <sys/ioctl.h>
#include <sys/file.h>

...

oob()
{
    int out = FWRITE;
    char waste[BUFSIZ];
    int mark;

    /* flush local terminal output */
    ioctl(1, TIOCFLUSH, (char *) &out);
    while(1) {
        if (ioctl(rem, SIOCATMARK, &mark) == -1) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof waste);
    }
    if (recv(rem, &mark, 1, MSG_OOB) == -1) {
        perror("recv");
        ...
    }
    ...
}
```

A process can also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (for example, TCP, the protocol used to provide socket streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a `recv()` is done with the `MSG_OOB` flag. In that case, the call returns the error of `EWOULDBLOCK`. Also, there may be enough in-

band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data before the urgent data can be delivered.

There is also a facility to retain the position of urgent in-line data in the socket stream. This is available as a socket-level option, `SO_OOBINLINE`. See the `getsockopt(3N)` manpage for usage. With this option, the position of urgent data (the mark) is retained, but the urgent data immediately follows the mark in the normal data stream returned without the `MSG_OOB` flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Nonblocking Sockets

Some applications require sockets that do not block. For example, requests that cannot complete immediately and would cause the process to be suspended (awaiting completion) are not executed. An error code would be returned. Once a socket is created and any connection to another socket is made, it may be made nonblocking by `fcntl()` as Code Example 2-14 shows.

Code Example 2-14 Set Nonblocking Socket

```
#include <fcntl.h>
#include <sys/file.h>
 ...
int fileflags;
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fileflags = fcntl(s, F_GETFL, 0) == -1)
    perror("fcntl F_GETFL");
    exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1)
    perror("fcntl F_SETFL, FNDELAY");
    exit(1);
}
 ...
```

When doing I/O on a nonblocking socket, check for the error `EWOULDBLOCK` (in `<errno.h>`), which occurs when an operation would normally block. `accept()`, `connect()`, `send()`, `recv()`, `read()`, and `write()` can all

return EWOULDBLOCK. If an operation such as a send() cannot be done in its entirety, but partial writes work (such as when using a stream socket), the data that can be sent immediately are processed, and the return value is the amount actually sent.

Asynchronous Sockets

Asynchronous communication between processes is required in real-time applications. Asynchronous sockets must be SOCK_STREAM type. Make a socket asynchronous as shown in Code Example 2-15.

Code Example 2-15 Making a Socket Asynchronous

```
#include <fcntl.h>
#include <sys/file.h>
 ...
int fileflags;
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fileflags = fcntl(s, F_GETFL ) == -1)
    perror("fcntl F_GETFL");
    exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) < 0)
    perror("fcntl F_SETFL, FNDELAY | FASYNC");
    exit(1);
}
 ...
```

After sockets are initialized, connected, and made asynchronous, communication is similar to reading and writing a file asynchronously. A send(), write(), recv(), or read() initiates a data transfer. A data transfer is completed by a signal-driven I/O routine, described in the next section.

Interrupt Driven Socket I/O

The SIGIO signal notifies a process when a socket (actually any file descriptor) has finished a data transfer. There are three steps in using SIGIO:

- Set up a SIGIO signal handler with the signal() or sigvec() calls.

- Use `fcntl()` to set the process ID or process group ID to receive the signal to its own process ID or process group ID (the default process group of a socket is group `0`).
- Convert the socket to asynchronous as shown in "Asynchronous Sockets" on page 44.

Sample code to allow a given process to receive information on pending requests as they occur for a socket is shown in Code Example 2-16. With the addition of a handler for `SIGURG`, this code can also be used to prepare for receipt of `SIGURG` signals.

Code Example 2-16 Asynchronous Notification of I/O Requests

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* Set the process receiving SIGIO/SIGURG signals to us. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
    perror("fcntl F_SETOWN");
    exit(1);
}
```

Signals and Process Group ID

For `SIGURG` and `SIGIO`, each socket has a process number and a process group ID. These values are initialized to zero, but may be redefined at a later time with the `F_SETOWN fcntl()`, as in the previous example. A positive third argument to `fcntl()` sets the socket's process ID. A negative third argument to `fcntl()` sets the socket's process group ID. The only allowed recipient of `SIGURG` and `SIGIO` signals is the calling process.

A similar `fcntl()`, `F_GETOWN`, returns the process number of a socket.

Reception of `SIGURG` and `SIGIO` can also be enabled by using `ioctl()` to assign the socket to the user's process group:

```
/* oobdata is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
    perror("ioctl: SIOCSPGRP");
}
```

Another signal that is useful in server processes is SIGCHLD. This signal is delivered to a process when any child process changes state. Normally, servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodically polling for exit status. For example, the remote login server loop shown previously can be augmented as shown in Code Example 2-17.

Code Example 2-17 SIGCHLD Signal

```
int reaper();
 ...
sigset(SIGCHLD, reaper);
listen(f, 5);
while (1) {
    int g, len = sizeof from;
    g = accept(f, (struct sockaddr *) &from, &len);
    if (g < 0) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    ...
}

#include <wait.h>

reaper()
{
    int options;
    int error;
    siginfo_t info;

    options = WNOHANG | WEXITED;
    bzero((char *) &info, sizeof(info));
    error = waitid(P_ALL, 0, &info, options);
}
```

If the parent server process fails to reap its children, zombie processes result.

Selecting Specific Protocols

If the third argument of the socket() call is 0, socket() selects a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available.

When using “raw” sockets to communicate directly with lower-level protocols or hardware interfaces, it may be important for the protocol argument to set up de-multiplexing. For example, raw sockets in the Internet domain can be used to implement a new protocol on IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol, determine the protocol number as defined in the protocol domain. For the Internet domain, use one of the library routines discussed in “Standard Routines” on page 31, such as `getprotobyname()`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto) ;
```

This results in a socket `s` using a stream-based connection, but with protocol type of `newtcp` instead of the default `tcp`.

Address Binding

In the Internet Protocol family, bindings are composed of local and foreign IP addresses, and of local and foreign port numbers. Port numbers are allocated in separate spaces, one for each system and one for each transport protocol (TCP or UDP). Through `bind()`, a process specifies the <*local IP address*, *local port number*> half of an association, while `connect()` and `accept()` complete a socket’s association by specifying the <*foreign IP address*, *foreign port number*> part. Since the association is created in two steps, the association-uniqueness requirement might be violated, unless care is taken. It is unrealistic to expect user programs to always know proper values to use for the local address and local port, since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

The wildcard address simplifies local address binding in the Internet domain. When an address is specified as `INADDR_ANY` (a constant defined in `<netinet/in.h>`), the system interprets the address as any valid address. Code Example 2-18 binds a specific port number to a socket, and leaves the local address unspecified.

Code Example 2-18 Bind Port Number to Socket

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
    s = socket(AF_INET, SOCK_STREAM, 0);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(MYPORT);
    bind(s, (struct sockaddr *) &sin, sizeof sin);
```

Each network interface on a host typically has a unique IP address. Sockets with wildcard local addresses may receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. For example, if a host has two interfaces with addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as in Code Example 2-18, the process can accept connection requests addressed to 128.32.0.4 or 10.0.0.78. To allow only hosts on a specific network to connect to it, a server binds the address of the interface on the appropriate network.

Similarly, a local port number can be left unspecified (specified as `0`), in which case the system selects a port number. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
sin.sin_addr.s_addr = inet_addr("127.0.0.1");
sin.sin_family = AF_INET;
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The system uses two criteria to select the local port number:

- The first is that Internet port numbers less than 1024 (`IPPORT_RESERVED`) are reserved for privileged users (that is, the superuser). Nonprivileged users may use any Internet port number greater than 1024. The largest Internet port number is 65535.
- The second criterion is that the port number is not currently bound to some other socket.

The port number and IP address of the client is found through either `accept()` (the *from* result) or `getpeername()`.

In certain cases, the algorithm used by the system to select port numbers is unsuitable for an application. This is because associations are created in a two-step process. For example, the Internet file transfer protocol specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed before address binding:

```
 ...
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

With this call, local addresses can be bound that are already in use. This does not violate the uniqueness requirement, because the system still verifies at connect time that any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error `EADDRINUSE` is returned.

Broadcasting and Determining Network Configuration

Messages sent by datagram sockets can be broadcast to reach all of the hosts on an attached network. The network must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Broadcasting is usually used for either of two reasons: to find a resource on a local network without having its address, or functions like routing require that information be sent to all accessible neighbors.

To send a broadcast message, create an Internet datagram socket:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and bind a port number to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The datagram can be broadcast on only one network by sending to the network's broadcast address. A datagram can also be broadcast on all attached networks by sending to the special address `INADDR_BROADCAST`, defined in `<netinet/in.h>`.

The system provides a mechanism to determine a number of pieces of information (including the IP address and broadcast address) about the network interfaces on the system. The `SIOCGIFCONF ioctl` call returns the interface configuration of a host in a single `ifconf` structure. This structure contains an array of `ifreq` structures, one for each address domain supported by each network interface to which the host is connected. Code Example 2-19 shows these structures defined in `<net/if.h>`.

Code Example 2-19 `net/if.h` Header File

```
struct ifreq {
#define IFNAMSIZ 16
char ifr_name[IFNAMSIZ]; /* if name, e.g., "en0" */
union {
    struct sockaddr ifru_addr;
    struct sockaddr ifru_dstaddr;
    char ifru_oname[IFNAMSIZ]; /* other if name */
    struct sockaddr ifru_broadaddr;
    short ifru_flags;
    int ifru_metric;
    char ifru_data[1]; /* interface dependent data */
    char ifru_enaddr[6];
} ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
#define ifr_oname ifr_ifru.ifru_oname
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
#define ifr_flags ifr_ifru.ifru_flags
#define ifr_metric ifr_ifru.ifru_metric
#define ifr_data ifr_ifru.ifru_data
#define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

The call that obtains the interface configuration is:

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof buf;
ifc.ifc_buf = buf;
```

```
if (ioctl(s, SIOCGIFCONF, (char *) &ifc ) < 0) {
    ...
}
```

After this call, *buf* contains an array of `ifreq` structures, one for each network to which the host is connected. These structures are ordered first by interface name and then by supported address families. `ifc.ifc_len` is set to the number of bytes used by the `ifreq` structures.

Each structure has a set of interface flags that tell whether the corresponding network is up or down, point to point or broadcast, and so on. The `SIOCGIFFLAGS ioctl` returns these flags for an interface specified by an `ifreq` structure shown in Code Example 2-20.

Code Example 2-20 Obtaining Interface Flags

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n >= 0; ifr++) {
    /*
     * Be careful not to use an interface devoted to an address
     * domain other than those intended.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    /* Skip boring cases */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
        continue;
}
```

Code Example 2-21 shows the broadcast of an interface can be obtained with the `SIOGGIFBRDADDR ioctl()`.

Code Example 2-21 Broadcast Address of an Interface

```
    if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
    }
    memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
        sizeof ifr->ifr_broadaddr);
```

The SIOGGIFBRDADDR ioctl can also be used to get the destination address of a point-to-point interface.

After the interface broadcast address is obtained, transmit the broadcast datagram with sendto():

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

Use one sendto() for each interface to which the host is connected that supports the broadcast or point-to-point addressing.

Socket Options

You can set and get several options on sockets through setsockopt() and getsockopt(); for example changing the send or receive buffer space. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

Table 2-4 shows the arguments of the calls.

Table 2-4 setsockopt() and getsockopt() Arguments

Arguments	Description
s	Socket on which the option is to be applied
level	Specifies the protocol level, i.e. socket level, indicated by the symbolic constant SOL_SOCKET in <sys/socket.h>
optname	Symbolic constant defined in <sys/socket.h> that specifies the option
optval	Points to the value of the option
optlen	Points to the length of the value of the option

For getsockopt(), *optlen* is a value-result argument, initially set to the size of the storage area pointed to by *optval* and set on return to the length of storage used.

It is sometimes useful to determine the type (for example, stream or datagram) of an existing socket. Programs invoked by inetd may need to do this by using the SO_TYPE socket option and the getsockopt() call:

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

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
    ...
}
```

After `getsockopt()`, `type` is set to the value of the socket type, as defined in `<sys/socket.h>`. For a datagram socket, `type` would be `SOCK_DGRAM`.

`inetd` *Daemon*

One of the daemons provided with the system is `inetd`. It is invoked at start-up time, and gets the services for which it listens from the `/etc/inetd.conf` file. The daemon creates one socket for each service listed in `/etc/inetd.conf`, binding the appropriate port number to each socket. See the `inetd(1M)` man page for details.

`inetd` does a `select()` on each socket, waiting for a connection request to the service corresponding to that socket. For `SOCK_STREAM` type sockets, `inetd` does an `accept()` on the listening socket, `fork()`s, `dup()`s the new socket to file descriptors `0` and `1` (`stdin` and `stdout`), closes other open file descriptors, and `exec()`s the appropriate server.

The primary benefit of `inetd` is that services that are not in use are not taking up machine resources. A secondary benefit is that `inetd` does most of the work to establish a connection. The server started by `inetd` has the socket connected to its client on file descriptors `0` and `1`, and can immediately `read()`, `write()`, `send()`, or `recv()`. Servers can use buffered I/O as provided by the `stdio` conventions, as long as they use `fflush()` when appropriate.

`getpeername()` returns the address of the peer (process) connected to a socket; it is useful in servers started by `inetd`. For example, to log the Internet address in decimal dot notation (such as 128.32.0.4, which is conventional for representing an IP address of a client), an `inetd` server could use the following:

```
struct sockaddr_in name;
int namelen = sizeof name;
 ...
if (getpeername(0, (struct sockaddr *) &name, &namelen) < 0) {
    syslog(LOG_ERR, "getpeername: %m");
    exit(1);
} else
    syslog(LOG_INFO, "Connection from %s",
            inet_ntoa(name.sin_addr));
 ...
```

Moving Socket Applications to Solaris 2.x

Sockets and the socket implementation are mostly compatible with previous releases of SunOS. But, an application programmer must be aware of some differences that are listed in the tables provided in this section.

Table 2-5 Connection-Mode Primitives (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	Solaris 2.x
`connect()`	
When `connect()` is called on an unbound socket, the protocol determines whether the endpoint is bound before the connection takes place.	When `connect()` is called on an unbound socket, that socket is always bound to an address selected by the protocol.

Table 2-6 Data Transfer Primitives (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	Solaris 2.x
`write()`	
`write()` fails with *errno* set to `ENOTCONN` if it is used on an unconnected socket.	A call to `write()` appears to succeed, but the data are discarded. The socket error option `SO_ERROR` returns `ENOTCONN` if this happens.

Table 2-6 Data Transfer Primitives (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	Solaris 2.x
`write()` can be used on type `SOCK_DGRAM` sockets (either `AF_UNIX` or `AF_INET` domains) to send zero-length data.	A call to `write()` returns `-1`, with *errno* set to `ERANGE`. `send()`, `sendto()`, or `sendmsg()` should be used to send zero-length data.
`read()`	
`read()` fails with *errno* set to `ENOTCONN` if `read()` is used on an unconnected socket.	`read()` returns zero bytes read if the socket is in blocking mode. If the socket is in non-blocking mode, it returns `-1` with *errno* set to `EAGAIN`.

Table 2-7 Information Primitives (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	Solaris 2.x
`getsockname()`	
`getsockname()` works even when a previously existing connection has been closed.	`getsockname()` returns `-1` and *errno* is set to `EPIPE` if a previously existing connection has been closed.
`ioctl()` and `fcntl()`	
The argument of the `SIOCSPGRP/FIOSETOWN/F_SETOWN` `ioctl()`s and the `F_SETOWN` `fcntl()` are a positive process ID or negative process group ID of the intended recipient list of subsequent `SIGURG` and `SIGIO` signals.	This is not the case in Solaris 2.x. The only acceptable argument of these system calls is the caller's process ID or a negative of the caller's process group ID. So, the only recipient of `SIGURG` and `SIGIO` is the calling process.

Table 2-8 Local Management (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	Solaris 2.x
`bind()`	
`bind()` uses the credentials of the user at the time of the `bind()` call to determine if the requested address is allocated or not.	`socket()` causes the user's credentials to be found and used to validate addresses used later in `bind()`.
`setsockopt()`	

Table 2-8 Local Management (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	Solaris 2.x
`setsockopt()` can be used at any time during the life of a socket.	If a socket is unbound and `setsockopt()` is used, the operation succeeds in the `AF_INET` domain but fails in the `AF_UNIX` domain.
`shutdown()`	
If `shutdown()` is called with `how` set to zero, further tries to receive data returns zero bytes (`EOF`).	If a `shutdown()` call with `how` set to zero is followed by a `read(2)` call and the socket is in nonblocking mode, `read()` returns `-1` with *errno* set to `EAGAIN`. If one of the socket receive primitives is used, the correct result (`EOF`) is returned.
If `shutdown()` is called with the value of `2` for how, further tries to receive data return `EOF`. Tries to send data return `-1` with *errno* set to `EPIPE` and a `SIGPIPE` is issued.	The same result happens, but tries to send data using `write(2)` cause *errno* to be set to `EIO`. If a socket primitive is used, the correct *errno* is returned.

Table 2-9 Signals (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	Solaris 2.x
`SIGIO`	
`SIGIO` is delivered every time new data are appended to the socket input queue.	`SIGIO` is delivered only when data are appended to a socket queue that was previously empty.
`SIGURG`	
A `SIGURG` is delivered every time new data are anticipated or actually arrive.	A `SUGURG` is delivered only when data are already pending.
`S_ISSOCK()`	
The `S_ISSOCK` macro takes the mode of a file as an argument. It returns `1` if the file is a socket and `0` otherwise.	The `S_ISSOCK` macro does not exist. Here, a socket is a file descriptor associated with a STREAMS character device that has the socket module pushed onto it.

Table 2-10 Miscellaneous Socket Issues (SunOS 4.x/Solaris 2.x)

SunOS 4.x (BSD)	**Solaris 2.x**
Invalid buffers	
If an invalid buffer is specified in a function, the function returns `-1` with *errno* set to `EFAULT`.	If an invalid buffer is specified in a function, the user's program probably dumps core.
Sockets in Directories	
If `ls -l` is executed in a directory containing a UNIX domain socket, an `s` is displayed on the left side of the mode field.	If `ls -l` is executed in a directory that contains a UNIX domain socket, a `p` is displayed on the left side of the mode field.
An `ls -F` causes an equal sign (=) to be displayed after any file name of a UNIX domain socket.	Nothing is displayed after the file name.

2