Wrox Programmer to Programmer™

Beginning Linux® Programming

4th Edition

Neil Matthew, Richard Stones

15

Sockets

In this chapter, you look at yet another method of process communication, but one with a crucial difference from those we've discussed in Chapters 13 and 14. Until now, all the facilities we've discussed have relied on shared resources on a single computer system. The resource varies; it can be file system space, shared physical memory, or message queues, but only processes running on a single machine can use them.

The Berkeley versions of UNIX introduced a new communication tool, the *socket interface*, which is an extension of the concept of a pipe, covered in Chapter 13. Socket interfaces are available on Linux. You can use sockets in much the same way as pipes, but they include communication across a network of computers. A process on one machine can use sockets to communicate with a process on another, which allows for client/server systems that are distributed across a network. Sockets may also be used between processes on the same machine.

Also, the sockets interface has been made available for Windows via a publicly available specification called *Windows Sockets*, or *WinSock*. Windows socket services are provided by a `Winsock.dll` system file. So, Windows programs can communicate across a network to Linux and UNIX computers and vice versa, thus implementing client/server systems. Although the programming interface for WinSock isn't quite the same as UNIX sockets, it still has sockets as its basis.

We can't cover the extensive Linux networking capabilities in a single chapter, so you'll find here a description of the principal programmatic networking interfaces. These should allow you to write your own network programs. Specifically, we look at the following:

- How a socket connection operates
- Socket attributes, addresses, and communications
- Network information and the Internet daemon (`inetd/xinetd`)
- Clients and servers

What Is a Socket?

A *socket* is a communication mechanism that allows client/server systems to be developed either locally, on a single machine, or across networks. Linux functions such as printing, connecting to databases, and serving web pages as well as network utilities such as `rlogin` for remote login and `ftp` for file transfer usually use sockets to communicate.

Sockets are created and used differently from pipes because they make a clear distinction between client and server. The socket mechanism can implement multiple clients attached to a single server.

Socket Connections

You can think of socket connections as telephone calls into a busy building. A call comes into an organization and is answered by a receptionist who puts the call through to the correct department (the server process) and from there to the right person (the server socket). Each incoming call (client) is routed to an appropriate end point and the intervening operators are free to deal with further calls. Before you look at the way socket connections are established in Linux systems, you need to understand how they operate for socket applications that maintain a connection.

First, a server application creates a socket, which like a file descriptor is a resource assigned to the server process and that process alone. The server creates it using the system call `socket`, and it can't be shared with other processes.

Next, the server process gives the socket a name. Local sockets are given a filename in the Linux file system, often to be found in `/tmp` or `/usr/tmp`. For network sockets, the filename will be a service identifier (port number/access point) relevant to the particular network to which the clients can connect. This identifier allows Linux to route incoming connections specifying a particular port number to the correct server process. For example, a web server typically creates a socket on port 80, an identifier reserved for the purpose. Web browsers know to use port 80 for their HTTP connections to web sites the user wants to read. A socket is named using the system call `bind`. The server process then waits for a client to connect to the named socket. The system call, `listen`, creates a queue for incoming connections. The server can accept them using the system call `accept`.

When the server calls `accept`, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client. The named socket remains for further connections from other clients. If the server is written appropriately, it can take advantage of multiple connections. A web server will do this so that it can serve pages to many clients at once. For a simple server, further clients wait on the listen queue until the server is ready again.

The client side of a socket-based system is more straightforward. The client creates an unnamed socket by calling `socket`. It then calls `connect` to establish a connection with the server by using the server's named socket as an address.

Once established, sockets can be used like low-level file descriptors, providing two-way data communications.

Try It Out A Simple Local Client

Here's an example of a very simple socket client program, `client1.c`. It creates an unnamed socket and connects it to a server socket called `server_socket`. We cover the details of the `socket` system call a little later, after we've discussed some addressing issues.

1. Make the necessary `include`s and set up the variables:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int sockfd;
    int len;
    struct sockaddr_un address;
    int result;
    char ch = 'A';
```

2. Create a socket for the client:

```
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

3. Name the socket as agreed with the server:

```
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "server_socket");
    len = sizeof(address);
```

4. Connect your socket to the server's socket:

```
    result = connect(sockfd, (struct sockaddr *)&address, len);

    if(result == -1) {
        perror("oops: client1");
        exit(1);
    }
```

5. You can now read and write via `sockfd`:

```
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

This program fails when you run it because you haven't yet created the server-side named socket. (The exact error message may differ from system to system.)

```
$ ./client1
oops: client1: No such file or directory
$
```

Try It Out A Simple Local Server

Here's a very simple server program, `server1.c`, that accepts connections from the client. It creates the server socket, binds it to a name, creates a listen queue, and accepts connections.

1. Make the necessary `include`s and set up the variables:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_un server_address;
    struct sockaddr_un client_address;
```

2. Remove any old sockets and create an unnamed socket for the server:

```
    unlink("server_socket");
    server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

3. Name the socket:

```
    server_address.sun_family = AF_UNIX;
    strcpy(server_address.sun_path, "server_socket");
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

4. Create a connection queue and wait for clients:

```
    listen(server_sockfd, 5);
    while(1) {
        char ch;

        printf("server waiting\n");
```

5. Accept a connection:

```
        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);
```

6. Read and write to client on `client_sockfd`:

```
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}
```

How It Works

The server program in this example can serve only one client at a time. It just reads a character from the client, increments it, and writes it back. In more sophisticated systems, where the server has to perform more work on the client's behalf, this wouldn't be acceptable, because other clients would be unable to connect until the server had finished. You'll see a couple of ways to allow multiple connections later.

When you run the server program, it creates a socket and waits for connections. If you start it in the background so that it runs independently, you can then start clients in the foreground.

```
$ ./server1 &
[1] 1094
$ server waiting
```

As it waits for connections, the server prints a message. In the preceding example, the server waits for a file system socket and you can see it with the normal `ls` command.

Remember that it's good practice to remove a socket when you've finished with it, even if the program terminates abnormally via a signal. This keeps the file system from getting cluttered with unused files.

```
$ ls -lF server_socket
srwxr-xr-x    1 neil     users           0 2007-06-23 11:41 server_socket=
```

The device type is "socket," shown by the `s` at the front of the permissions and the `=` at the end of the name. The socket has been created just as an ordinary file would be, with permissions modified by the current `umask`. If you use the `ps` command, you can see the server running in the background. It's shown sleeping (`STAT` is `S`) and is therefore not consuming CPU resources.

```
$ ps lx
F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
0  1000 23385 10689  17   0  1424  312 361800 S    pts/1      0:00 ./server1
```

Now, when you run the client program, you are successful in connecting to the server. Because the server socket exists, you can connect to it and communicate with the server.

```
$ ./client1
server waiting
char from server = B
$
```

The output from the server and the client get mixed on the terminal, but you can see that the server has received a character from the client, incremented it, and returned it. The server then continues and waits for the next client. If you run several clients together, they will be served in sequence, although the output you see may be more mixed up.

```
$ ./client1 & ./client1 & ./client1 &
[2] 23412
[3] 23413
[4] 23414
server waiting
char from server = B
server waiting
char from server = B
server waiting
char from server = B
server waiting
[2]   Done                    client1
[3]-  Done                    client1
[4]+  Done                    client1
$
```

Socket Attributes

To fully understand the system calls used in this example, you need to learn a little about UNIX networking.

Sockets are characterized by three attributes: *domain*, *type*, and *protocol*. They also have an address used as their name. The formats of the addresses vary depending on the domain, also known as the *protocol family*. Each protocol family can use one or more address families to define the address format.

Socket Domains

Domains specify the network medium that the socket communication will use. The most common socket domain is `AF_INET`, which refers to Internet networking that's used on many Linux local area networks and, of course, the Internet itself. The underlying protocol, *Internet Protocol* (IP), which only has one address family, imposes a particular way of specifying computers on a network. This is called the *IP address*.

The "next generation" Internet Protocol, IPv6, has been designed to overcome some of the problems with the standard IP, notably the limited number of addresses that are available. IPv6 uses a different socket domain, `AF_INET6`, *and a different address format. It is expected to eventually replace IP, but this process will take many years. Although there are implementations of IPv6 for Linux, it is beyond the scope of this book.*

Although names almost always refer to networked machines on the Internet, these are translated into lower-level IP addresses. An example IP address is 192.168.1.99. All IP addresses are represented by four numbers, each less than 256, a so-called *dotted quad*. When a client connects across a network via sockets, it needs the IP address of the server computer.

There may be several services available at the server computer. A client can address a particular service on a networked machine by using an IP port. A port is identified within the system by assigning a unique 16-bit integer and externally by the combination of IP address and port number. The sockets are communication end points that must be bound to ports before communication is possible.

Servers wait for connections on particular ports. Well-known services have allocated port numbers that are used by all Linux and UNIX machines. These are usually, but not always, numbers less than 1024. Examples are the printer spooler (515), `rlogin` (513), `ftp` (21), and `httpd` (80). The last of these is the standard port for web servers. Usually, port numbers less than 1024 are reserved for system services and may only be served by processes with superuser privileges. X/Open defines a constant in `netdb.h`, `IPPORT_RESERVED`, to stand for the highest reserved port number.

Because there is a standard set of port numbers for standard services, computers can easily connect to each other without having to figure out the correct port. Local services may use nonstandard port addresses.

The domain in the first example is the UNIX file system domain, `AF_UNIX`, which can be used by sockets based on a single computer that perhaps isn't networked. When this is so, the underlying protocol is file input/output and the addresses are filenames. The address that you used for the server socket was `server_socket`, which you saw appear in the current directory when you ran the server application.

Other domains that might be used include `AF_ISO` for networks based on ISO standard protocols and `AF_XNS` for the Xerox Network System. We won't cover these here.

Socket Types

A socket domain may have a number of different ways of communicating, each of which might have different characteristics. This isn't an issue with `AF_UNIX` domain sockets, which provide a reliable two-way communication path. In networked domains, however, you need to be aware of the characteristics of the underlying network and how different communication mechanisms are affected by them.

Internet protocols provide two communication mechanisms with distinct levels of service: *streams* and *datagrams*.

Stream Sockets

Stream sockets (in some ways similar to standard input/output streams) provide a connection that is a sequenced and reliable two-way byte stream. Thus, data sent is guaranteed not to be lost, duplicated, or reordered without an indication that an error has occurred. Large messages are fragmented, transmitted, and reassembled. This is similar to a file stream, which also accepts large amounts of data and splits it up for writing to the low-level disk in smaller blocks. Stream sockets have predictable behavior.

Stream sockets, specified by the type `SOCK_STREAM`, are implemented in the `AF_INET` domain by TCP/IP connections. They are also the usual type in the `AF_UNIX` domain. We concentrate on `SOCK_STREAM` sockets in this chapter because they are more commonly used in programming network applications.

TCP/IP stands for Transmission Control Protocol/Internet Protocol. IP is the low-level protocol for packets that provides routing through the network from one computer to another. TCP provides sequencing, flow control, and retransmission to ensure that large data transfers arrive with all of the data present and correct or with an appropriate error condition reported.

Datagram Sockets

In contrast, a datagram socket, specified by the type `SOCK_DGRAM`, doesn't establish and maintain a connection. There is also a limit on the size of a datagram that can be sent. It's transmitted as a single network message that may get lost, duplicated, or arrive out of sequence — ahead of datagrams sent after it.

Datagram sockets are implemented in the `AF_INET` domain by UDP/IP connections and provide an unsequenced, unreliable service. (UDP stands for User Datagram Protocol.) However, they are relatively inexpensive in terms of resources, because network connections need not be maintained. They're fast because there is no associated connection setup time.

Datagrams are useful for "single-shot" inquiries to information services, for providing regular status information, or for performing low-priority logging. They have the advantage that the death of a server doesn't unduly inconvenience a client and would not require a client restart. Because datagram-based servers usually retain no connection information, they can be stopped and restarted without disturbing their clients.

For now, we leave the topic of datagrams; see the "Datagrams" section near the end of this chapter for more information.

Socket Protocols

Where the underlying transport mechanism allows for more than one protocol to provide the requested socket type, you can select a specific protocol for a socket. In this chapter, we concentrate on UNIX network and file system sockets, which don't require you to choose a protocol other than the default.

Creating a Socket

The `socket` system call creates a socket and returns a descriptor that can be used for accessing the socket.

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

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

The socket created is one end point of a communication channel. The `domain` parameter specifies the address family, the `type` parameter specifies the type of communication to be used with this socket, and `protocol` specifies the protocol to be employed.

Domains include those in the following table:

Domain	Description
`AF_UNIX`	UNIX internal (file system sockets)
`AF_INET`	ARPA Internet protocols (UNIX network sockets)

Domain	Description
AF_ISO	ISO standard protocols
AF_NS	Xerox Network Systems protocols
AF_IPX	Novell IPX protocol
AF_APPLETALK	Appletalk DDS

The most common socket domains are AF_UNIX, which is used for local sockets implemented via the UNIX and Linux file systems, and AF_INET, which is used for UNIX network sockets. The AF_INET sockets may be used by programs communicating across a TCP/IP network including the Internet. The Windows Winsock interface also provides access to this socket domain.

The socket parameter type specifies the communication characteristics to be used for the new socket. Possible values include SOCK_STREAM and SOCK_DGRAM.

SOCK_STREAM is a sequenced, reliable, connection-based two-way byte stream. For an AF_INET domain socket, this is provided by default by a TCP connection that is established between the two end points of the stream socket when it's connected. Data may be passed in both directions along the socket connection. The TCP protocols include facilities to fragment and reassemble long messages and to retransmit any parts that may be lost in the network.

SOCK_DGRAM is a datagram service. You can use this socket to send messages of a fixed (usually small) maximum size, but there's no guarantee that the message will be delivered or that messages won't be reordered in the network. For AF_INET sockets, this type of communication is provided by UDP datagrams.

The protocol used for communication is usually determined by the socket type and domain. There is normally no choice. The protocol parameter is used where there is a choice. 0 selects the default protocol, which is used in all the examples in this chapter.

The socket system call returns a descriptor that is in many ways similar to a low-level file descriptor. When the socket has been connected to another end-point socket, you can use the read and write system calls with the descriptor to send and receive data on the socket. The close system call is used to end a socket connection.

Socket Addresses

Each socket domain requires its own address format. For an AF_UNIX socket, the address is described by a structure, sockaddr_un, defined in the sys/un.h include file.

```
struct sockaddr_un {
    sa_family_t   sun_family;    /* AF_UNIX */
    char          sun_path[];    /* pathname */
};
```

So that addresses of different types may be passed to the socket-handling system calls, each address format is described by a similar structure that begins with a field (in this case, sun_family) that specifies the address type (the socket domain). In the AF_UNIX domain, the address is specified by a filename in the sun_path field of the structure.

On current Linux systems, the type sa_family_t, defined by X/Open as being declared in sys/un.h, is taken to be a short. Also, the pathname specified in sun_path is limited in size (Linux specifies 108 characters; others may use a manifest constant such as UNIX_MAX_PATH). Because address structures may vary in size, many socket calls require or provide as an output a length to be used for copying the particular address structure.

In the AF_INET domain, the address is specified using a structure called sockaddr_in, defined in netinet/in.h, which contains at least these members:

```
struct sockaddr_in {
    short int            sin_family;  /* AF_INET */
    unsigned short int   sin_port;    /* Port number */
    struct in_addr       sin_addr;    /* Internet address */
};
```

The IP address structure, in_addr, is defined as follows:

```
struct in_addr {
    unsigned long int    s_addr;
};
```

The four bytes of an IP address constitute a single 32-bit value. An AF_INET socket is fully described by its domain, IP address, and port number. From an application's point of view, all sockets act like file descriptors and are addressed by a unique integer value.

Naming a Socket

To make a socket (as created by a call to socket) available for use by other processes, a server program needs to give the socket a name. Thus, AF_UNIX sockets are associated with a file system pathname, as you saw in the server1 example. AF_INET sockets are associated with an IP port number.

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

int bind(int socket, const struct sockaddr *address, size_t address_len);
```

The bind system call assigns the address specified in the parameter, address, to the unnamed socket associated with the file descriptor socket. The length of the address structure is passed as address_len.

The length and format of the address depend on the address family. A particular address structure pointer will need to be cast to the generic address type (struct sockaddr *) in the call to bind.

On successful completion, bind returns 0. If it fails, it returns -1 and sets errno to one of the following values:

Errno Value	Description
`EBADF`	The file descriptor is invalid.
`ENOTSOCK`	The file descriptor doesn't refer to a socket.
`EINVAL`	The file descriptor refers to an already-named socket.
`EADDRNOTAVAIL`	The address is unavailable.
`EADDRINUSE`	The address has a socket bound to it already.

There are some more values for `AF_UNIX` sockets:

Errno value	Description
`EACCESS`	Can't create the file system name due to permissions.
`ENOTDIR, ENAMETOOLONG`	Indicates a poor choice of filename.

Creating a Socket Queue

To accept incoming connections on a socket, a server program must create a queue to store pending requests. It does this using the `listen` system call.

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

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

A Linux system may limit the maximum number of pending connections that may be held in a queue. Subject to this maximum, `listen` sets the queue length to `backlog`. Incoming connections up to this queue length are held pending on the socket; further connections will be refused and the client's connection will fail. This mechanism is provided by `listen` to allow incoming connections to be held pending while a server program is busy dealing with a previous client. A value of `5` for `backlog` is very common.

The `listen` function will return `0` on success or `-1` on error. Errors include `EBADF`, `EINVAL`, and `ENOTSOCK`, as for the `bind` system call.

Accepting Connections

Once a server program has created and named a socket, it can wait for connections to be made to the socket by using the `accept` system call.

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

int accept(int socket, struct sockaddr *address, size_t *address_len);
```

The accept system call returns when a client program attempts to connect to the socket specified by the parameter socket. The client is the first pending connection from that socket's queue. The accept function creates a new socket to communicate with the client and returns its descriptor. The new socket will have the same type as the server listen socket.

The socket must have previously been named by a call to bind and had a connection queue allocated by listen. The address of the calling client will be placed in the sockaddr structure pointed to by address. A null pointer may be used here if the client address isn't of interest.

The address_len parameter specifies the length of the client structure. If the client address is longer than this value, it will be truncated. Before calling accept, address_len must be set to the expected address length. On return, address_len will be set to the actual length of the calling client's address structure.

If there are no connections pending on the socket's queue, accept will block (so that the program won't continue) until a client does make connection. You can change this behavior by using the O_NONBLOCK flag on the socket file descriptor, using the fcntl function in your code like this:

```
int flags = fcntl(socket, F_GETFL, 0);

fcntl(socket, F_SETFL, O_NONBLOCK|flags);
```

The accept function returns a new socket file descriptor when there is a client connection pending or -1 on error. Possible errors are similar to those for bind and listen, with the addition of EWOULDBLOCK, where O_NONBLOCK has been specified and there are no pending connections. The error EINTR will occur if the process is interrupted while blocked in accept.

Requesting Connections

Client programs connect to servers by establishing a connection between an unnamed socket and the server listen socket. They do this by calling connect.

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

int connect(int socket, const struct sockaddr *address, size_t address_len);
```

The socket specified by the parameter socket is connected to the server socket specified by the parameter address, which is of length address_len. The socket must be a valid file descriptor obtained by a call to socket.

If it succeeds, connect returns 0, and -1 is returned on error. Possible errors this time include the following:

Errno Value	Description
EBADF	An invalid file descriptor was passed in socket.
EALREADY	A connection is already in progress for this socket.
ETIMEDOUT	A connection timeout has occurred.
ECONNREFUSED	The requested connection was refused by the server.

If the connection can't be set up immediately, connect will block for an unspecified timeout period. Once this timeout has expired, the connection will be aborted and connect will fail. However, if the call to connect is interrupted by a signal that is handled, the connect call will fail (with errno set to EINTR), but the connection attempt won't be aborted — it will be set up asynchronously, and the program will have to check later to see if the connection was successful.

As with accept, the blocking nature of connect can be altered by setting the O_NONBLOCK flag on the file descriptor. In this case, if the connection can't be made immediately, connect will fail with errno set to EINPROGRESS and the connection will be made asynchronously.

Though asynchronous connections can be tricky to handle, you can use a call to select on the socket file descriptor to check that the socket is ready for writing. We cover select later in this chapter.

Closing a Socket

You can terminate a socket connection at the server and client by calling close, just as you would for low-level file descriptors. You should always close the socket at both ends. For the server, you should do this when read returns zero. Note that the close call may block if the socket has untransmitted data, is of a connection-oriented type, and has the SOCK_LINGER option set. You learn about setting socket options later in this chapter.

Socket Communications

Now that we have covered the basic system calls associated with sockets, let's take a closer look at the example programs. You'll try to convert them to use a network socket rather than a file system socket. The file system socket has the disadvantage that, unless the author uses an absolute pathname, it's created in the server program's current directory. To make it more generally useful, you need to create it in a globally accessible directory (such as /tmp) that is agreed between the server and its clients. For network sockets, you need only choose an unused port number.

For the example, select port number 9734. This is an arbitrary choice that avoids the standard services (you can't use port numbers below 1024 because they are reserved for system use). Other port numbers are often listed, with the services provided on them, in the system file /etc/services. When you're writing socket-based applications, always choose a port number not listed in this configuration file.

Be aware that there is a deliberate error in the programs client2.c and server2.c that you will fix in client3.c and server3.c. Please do not use the code from client2.c and server2.c in your own programs.

You'll run your client and server across a local network, but network sockets are not only useful on a local area network; any machine with an Internet connection (even a modem dial-up) can use network sockets to communicate with others. You can even use a network-based program on a stand-alone UNIX computer because a UNIX computer is usually configured to use a loopback network that contains only itself. For illustration purposes, this example uses this loopback network, which can also be useful for debugging network applications because it eliminates any external network problems.

The loopback network consists of a single computer, conventionally called `localhost`, with a standard IP address of 127.0.0.1. This is the local machine. You'll find its address listed in the network hosts file, `/etc/hosts`, with the names and addresses of other hosts on shared networks.

Each network with which a computer communicates has a hardware interface associated with it. A computer may have different network names on each network and certainly will have different IP addresses. For example, Neil's machine `tilde` has three network interfaces and therefore three addresses. These are recorded in `/etc/hosts` as follows:

```
127.0.0.1      localhost             # Loopback
192.168.1.1    tilde.localnet        # Local, private Ethernet
158.152.X.X    tilde.demon.co.uk     # Modem dial-up
```

The first is the simple loopback network, the second is a local area network accessed via an Ethernet adapter, and the third is the modem link to an Internet service provider. You can write a network socket-based program to communicate with servers accessed via any of these interfaces without alteration.

Try It Out Network Client

Here's a modified client program, `client2.c`, to connect to a network socket via the loopback network. It contains a subtle bug concerned with hardware dependency, but we'll discuss that later in this chapter.

1. Make the necessary `include`s and set up the variables:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char ch = 'A';
```

2. Create a socket for the client:

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

3. Name the socket, as agreed with the server:

```
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = 9734;
    len = sizeof(address);
```

The rest of this program is the same as client1.c from earlier in this chapter. When you run this version, it fails to connect because there isn't a server running on port 9734 on this machine.

```
$ ./client2
oops: client2: Connection refused
$
```

How It Works

The client program used the sockaddr_in structure from the include file netinet/in.h to specify an AF_INET address. It tries to connect to a server on the host with IP address 127.0.0.1. It uses a function, inet_addr, to convert the text representation of an IP address into a form suitable for socket addressing. The manual page for inet has more information on other address translation functions.

Try It Out Network Server

You also need to modify the server program to wait for connections on your chosen port number. Here's a modified server: server2.c.

1. Make the necessary includes and set up the variables:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
```

2. Create an unnamed socket for the server:

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

3. Name the socket:

```
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_address.sin_port = 9734;
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

From here on, the listing follows server1.c exactly. Running client2 and server2 will show the same behavior you saw earlier with client1 and server1.

How It Works

The server program creates an AF_INET domain socket and arranges to accept connections on it. The socket is bound to your chosen port. The address specified determines which computers are allowed to connect. By specifying the loopback address, as in the client program, you are restricting communications to the local machine.

If you want to allow the server to communicate with remote clients, you must specify a set of IP addresses that you are willing to allow. You can use the special value, INADDR_ANY, to specify that you'll accept connections from all of the interfaces your computer may have. If you chose to, you could distinguish between different network interfaces to separate, for example, internal Local Area Network and external Wide Area Network connections. INADDR_ANY is a 32-bit integer value that you can use in the sin_addr.s_addr field of the address structure. However, you have a problem to resolve first.

Host and Network Byte Ordering

When we run these versions of the server and client programs on an Intel processor–based Linux machine, we can see the network connections by using the netstat command. This command will also be available on most UNIX systems configured for networking. It shows the client/server connection waiting to close down. The connection closes down after a small timeout. (Again, the exact output may vary among different versions of Linux.)

```
$ ./server2 & ./client2
[3] 23770
server waiting
server waiting
char from server = B
$ netstat –A inet
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address   Foreign Address  (State)       User
tcp        1      0 localhost:1574  localhost:1174   TIME_WAIT     root
```

Before you try out further examples in this book, be sure to terminate running example server programs, because they will compete to accept connections from clients and you'll see confusing results. You can kill them all (including ones covered later in the chapter) with the following command:

```
killall server1 server2 server3 server4 server5
```

You can see the port numbers that have been assigned to the connection between the server and the client. The local address shows the server, and the foreign address is the remote client. (Even though it's on the same machine, it's still connected over a network.) To ensure that all sockets are distinct, these client ports are typically different from the server listen socket and unique to the computer.

However, the local address (the server socket) is given as 1574 (or you may see `mvel-lm` as a service name), but the port chosen in the example is 9734. Why are they different? The answer is that port numbers and addresses are communicated over socket interfaces as binary numbers. Different computers use different byte ordering for integers. For example, an Intel processor stores the 32-bit integer as four consecutive bytes in memory in the order 1-2-3-4, where 1 is the most significant byte. IBM PowerPC processors would store the integer in the byte order 4-3-2-1. If the memory used for integers were simply copied byte-by-byte, the two different computers would not be able to agree on integer values.

To enable computers of different types to agree on values for multibyte integers transmitted over a network, you need to define a network ordering. Client and server programs must convert their internal integer representation to the network ordering before transmission. They do this by using functions defined in `netinet/in.h`. These are

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

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

These functions convert 16-bit and 32-bit integers between native host format and the standard network ordering. Their names are abbreviations for conversions — for example, "host to network, long" (`htonl`) and "host to network, short" (`htons`). For computers where the native ordering is the same as network ordering, these represent null operations.

To ensure correct byte ordering of the 16-bit port number, your server and client need to apply these functions to the port address. The change to `server3.c` is

```
        server_address.sin_addr.s_addr = htonl(INADDR_ANY);
        server_address.sin_port = htons(9734);
```

You don't need to convert the function call, `inet_addr("127.0.0.1")`, because `inet_addr` is defined to produce a result in network order. The change to `client3.c` is

```
        address.sin_port = htons(9734);
```

The server has also been changed to allow connections from any IP address by using `INADDR_ANY`.

Now, when you run `server3` and `client3`, you see the correct port being used for the local connection.

```
$ netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address   (State)       User
tcp        1      0 localhost:9734  localhost:1175    TIME_WAIT     root
```

Remember that if you're using a computer that has the same native and network byte ordering, you won't see any difference. It's still important always to use the conversion functions to allow correct operation with clients and servers on computers with a different architecture.

Network Information

So far, your client and server programs have had addresses and port numbers compiled into them. For a more general server and client program, you can use network information functions to determine addresses and ports to use.

If you have permission to do so, you can add your server to the list of known services in `/etc/services`, which assigns a name to port numbers so that clients can use symbolic services rather than numbers.

Similarly, given a computer's name, you can determine the IP address by calling host database functions that resolve addresses for you. They do this by consulting network configuration files, such as `/etc/hosts`, or network information services, such as NIS (Network Information Services, formerly known as Yellow Pages) and DNS (Domain Name Service).

Host database functions are declared in the interface header file `netdb.h`.

```
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
struct hostent *gethostbyname(const char *name);
```

The structure returned by these functions must contain at least these members:

```
struct hostent {
    char *h_name;          /* name of the host */
    char **h_aliases;      /* list of aliases (nicknames) */
    int h_addrtype;        /* address type */
    int h_length;          /* length in bytes of the address */
    char **h_addr_list     /* list of address (network order) */
};
```

If there is no database entry for the specified host or address, the information functions return a null pointer.

Similarly, information concerning services and associated port numbers is available through some service information functions.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

The `proto` parameter specifies the protocol to be used to connect to the service, either `"tcp"` for `SOCK_STREAM` TCP connections or `"udp"` for `SOCK_DGRAM` UDP datagrams.

The structure `servent` contains at least these members:

```
struct servent {
    char *s_name;         /* name of the service */
    char **s_aliases;     /* list of aliases (alternative names) */
    int s_port;           /* The IP port number */
    char *s_proto;        /* The service type, usually "tcp" or "udp" */
};
```

You can gather host database information about a computer by calling `gethostbyname` and printing the results. Note that the address list needs to be cast to the appropriate address type and converted from network ordering to a printable string using the `inet_ntoa` conversion, which has the following definition:

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

char *inet_ntoa(struct in_addr in)
```

The function converts an Internet host address to a string in dotted quad format. It returns `-1` on error, but POSIX doesn't define any specific errors. The other new function you use is `gethostname`.

```
#include <unistd.h>

int gethostname(char *name, int namelength);
```

This function writes the name of the current host into the string given by `name`. The hostname will be null-terminated. The argument `namelength` indicates the length of the string `name`, and the returned hostname will be truncated if it's too long to fit. `gethostname` returns `0` on success and `-1` on error, but again no errors are defined in POSIX.

Try It Out Network Information

This program, `getname.c`, gets information about a host computer.

1. As usual, make the appropriate `include`s and declare the variables:

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *host, **names, **addrs;
    struct hostent *hostinfo;
```

2. Set the host to the argument supplied with the `getname` call, or by default to the user's machine:

```
    if(argc == 1) {
        char myname[256];
        gethostname(myname, 255);
        host = myname;
    }
    else
        host = argv[1];
```

3. Call `gethostbyname` and report an error if no information is found:

```
    hostinfo = gethostbyname(host);
    if(!hostinfo) {
            fprintf(stderr, "cannot get info for host: %s\n", host);
            exit(1);
        }
```

4. Display the hostname and any aliases that it may have:

```
        printf("results for host %s:\n", host);
        printf("Name: %s\n", hostinfo -> h_name);
        printf("Aliases:");
        names = hostinfo -> h_aliases;
        while(*names) {
            printf(" %s", *names);
            names++;
        }
        printf("\n");
```

5. Warn and exit if the host in question isn't an IP host:

```
        if(hostinfo -> h_addrtype != AF_INET) {
            fprintf(stderr, "not an IP host!\n");
            exit(1);
        }
```

6. Otherwise, display the IP address(es):

```
        addrs = hostinfo -> h_addr_list;
        while(*addrs) {
            printf(" %s", inet_ntoa(*(struct in_addr *)*addrs));
            addrs++;
        }
        printf("\n");
        exit(0);
    }
```

Alternatively, you could use the function `gethostbyaddr` to determine which host has a given IP address. You might use this in a server to find out where the client is calling from.

How It Works

The `getname` program calls `gethostbyname` to extract the host information from the host database. It prints out the hostname, its aliases (other names the computer is known by), and the IP addresses that the host uses on its network interfaces. On one of the authors' systems, running the example and specifying `tilde` gave the two interfaces: Ethernet and modem.

```
$ ./getname tilde
results for host tilde:
Name: tilde.localnet
Aliases: tilde
 192.168.1.1 158.152.x.x
```

When you use the hostname, `localhost`, the loopback network is given.

```
$ ./getname localhost
results for host localhost:
Name: localhost
Aliases:
 127.0.0.1
```

You can now modify your client to connect to any named host. Instead of connecting to your example server, you'll connect to a standard service so that you can extract the port number.

Most UNIX and some Linux systems make their system time and date available as a standard service called `daytime`. Clients may connect to this service to discover the server's idea of the current time and date. Here's a client program, `getdate.c`, that does just that.

Try It Out Connecting to a Standard Service

1. Start with the usual `include`s and declarations:

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

int main(int argc, char *argv[])
{
    char *host;
    int sockfd;
    int len, result;
    struct sockaddr_in address;
    struct hostent *hostinfo;
    struct servent *servinfo;
    char buffer[128];

    if(argc == 1)
        host = "localhost";
    else
        host = argv[1];
```

2. Find the host address and report an error if none is found:

```
    hostinfo = gethostbyname(host);
    if(!hostinfo) {
        fprintf(stderr, "no host: %s\n", host);
        exit(1);
    }
```

3. Check that the `daytime` service exists on the host:

```
    servinfo = getservbyname("daytime", "tcp");
    if(!servinfo) {
        fprintf(stderr,"no daytime service\n");
        exit(1);
    }
    printf("daytime port is %d\n", ntohs(servinfo -> s_port));
```

4. Create a socket:

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

5. Construct the address for use with `connect`:

```
    address.sin_family = AF_INET;
    address.sin_port = servinfo -> s_port;
    address.sin_addr = *(struct in_addr *)*hostinfo -> h_addr_list;
    len = sizeof(address);
```

6. Then connect and get the information:

```
    result = connect(sockfd, (struct sockaddr *)&address, len);
    if(result == -1) {
        perror("oops: getdate");
        exit(1);
    }

    result = read(sockfd, buffer, sizeof(buffer));
    buffer[result] = '\0';
    printf("read %d bytes: %s", result, buffer);

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

You can use `getdate` to get the time of day from any known host.

```
$ ./getdate localhost
daytime port is 13
read 26 bytes: 24 JUN 2007 06:03:03 BST
$
```

If you receive an error message such as

```
oops: getdate: Connection refused
```

or

```
oops: getdate: No such file or directory
```

it may be because the computer you are connecting to has not enabled the daytime service. This has become the default behavior in more recent Linux systems. In the next section, you see how to enable this and other services.

How It Works

When you run this program, you can specify a host to connect to. The `daytime` service port number is determined from the network database function `getservbyname`, which returns information about network services in a similar way to host information. The program `getdate` tries to connect to the address given first in the list of alternate addresses for the specified host. If successful, it reads the information returned by the `daytime` service, a character string representing the UNIX time and date.

The Internet Daemon (xinetd/inetd)

UNIX systems providing a number of network services often do so by way of a super-server. This program (the Internet daemon, `xinetd` or `inetd`) listens for connections on many port addresses at once. When a client connects to a service, the daemon program runs the appropriate server. This cuts down on the need for servers to be running all the time; they can be started as required.

The Internet daemon is implemented in modern Linux systems by `xinetd`*. This implementation has replaced the original UNIX program,* `inetd`*, although you will still see* `inetd` *in older Linux systems and on other UNIX-like systems.*

`xinetd` is usually configured through a graphical user interface for managing network services, but it is also possible to modify its configuration files directly. These are typically `/etc/xinetd.conf` and files in the `/etc/xinetd.d` directory.

Each service that is to be provided via `xinetd` has a configuration file in `/etc/xinetd.d`. `xinetd` will read all of these configuration files when it starts up, and again if instructed to do so.

Here are a couple of example `xinetd` configuration files, first for the daytime service:

```
#default: off
# description: A daytime server. This is the tcp version.
service daytime
{
        socket_type     = stream
        protocol        = tcp
        wait            = no
        user            = root
        type            = INTERNAL
        id              = daytime-stream
        FLAGS           = IPv6 IPv4
}
```

The following configuration is for the file transfer service:

```
# default: off
# description:
#   The vsftpd FTP server serves FTP connections. It uses
#   normal, unencrypted usernames and passwords for authentication.
# vsftpd is designed to be secure.
#
```

```
# NOTE: This file contains the configuration for xinetd to start vsftpd.
#       the configuration file for vsftp itself is in /etc/vsftpd.conf
service ftp
{
#        server_args             =
#        log_on_success          += DURATION USERID
#        log_on_failure          += USERID
#        nice                    = 10
 socket_type     = stream
 protocol        = tcp
 wait            = no
 user            = root
 server          = /usr/sbin/vsftpd
}
```

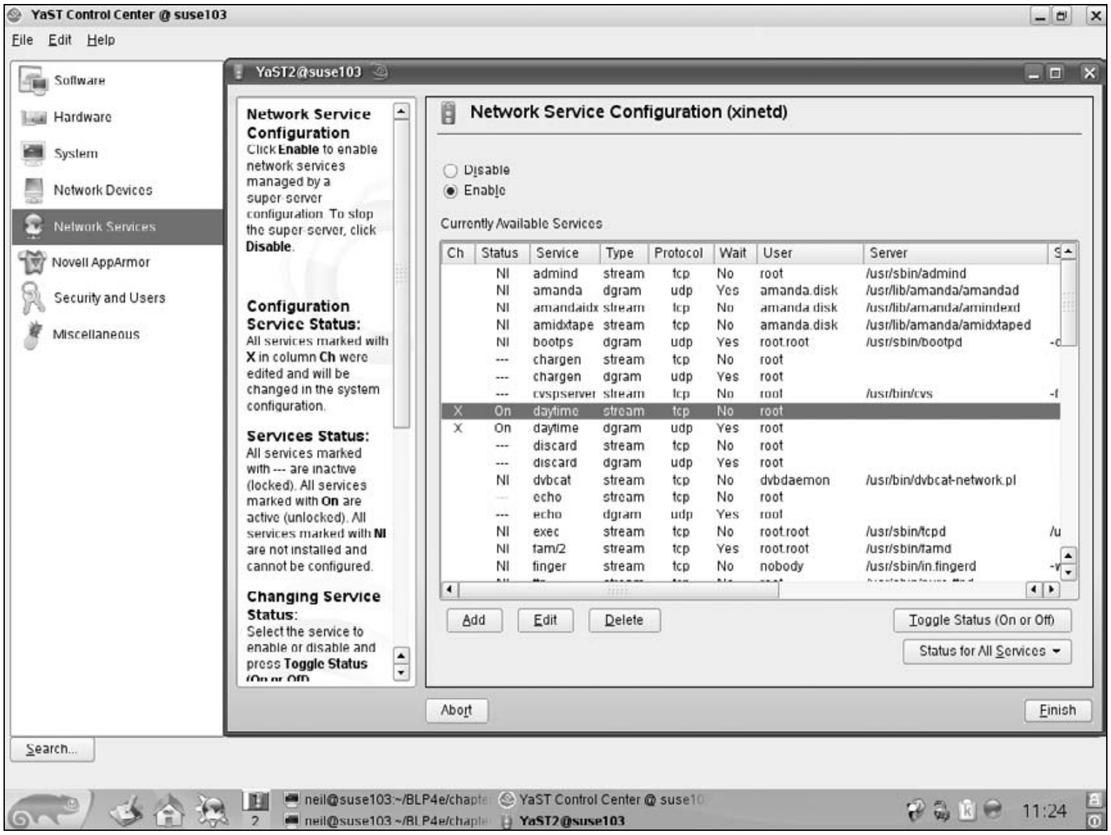
The `daytime` service that the `getdate` program connects to is actually handled by `xinetd` itself (it is marked as internal) and can be made available using both `SOCK_STREAM` (`tcp`) and `SOCK_DGRAM` (`udp`) sockets.

The `ftp` file transfer service is available only via `SOCK_STREAM` sockets and is provided by an external program, in this case `vsftpd`. The daemon will start these external programs when a client connects to the `ftp` port.

To activate service configuration changes, you can edit the `xinetd` configuration and send a hang-up signal to the daemon process, but we recommend that you use a more friendly way of configuring services. To allow your time-of-day client to connect, enable the `daytime` service using the tools provided on your Linux system. On SUSE and openSUSE the services may be configured from the SUSE Control Center as shown in Figure 15-1. Red Hat versions (both Enterprise Linux and Fedora) have a similar configuration interface. Here, the `daytime` service is being enabled for both TCP and UDP queries.

For systems that use `inetd` rather than `xinetd` , here's the equivalent extract from the `inetd` configuration file, `/etc/inetd.conf`, which is used by `inetd` to decide which servers to run:

```
#
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
#
# Echo, discard, daytime, and chargen are used primarily for testing.
#
daytime    stream    tcp    nowait    root    internal
daytime    dgram    udp    wait    root    internal
#
# These are standard services.
#
ftp     stream    tcp    nowait    root    /usr/sbin/tcpd    /usr/sbin/wu.ftpd
telnet  stream    tcp    nowait    root    /usr/sbin/tcpd    /usr/sbin/in.telnetd
#
# End of inetd.conf.
```



Network Service Configuration
Click **Enable** to enable network services managed by a super-server configuration. To stop the super-server, click **Disable**.

Configuration Service Status:
All services marked with **X** in column **Ch** were edited and will be changed in the system configuration.

Services Status:
All services marked with --- are inactive (locked). All services marked with **On** are active (unlocked). All services marked with **NI** are not installed and cannot be configured.

Changing Service Status:
Select the service to enable or disable and press **Toggle Status (On or Off)**

Network Service Configuration (xinetd)

Disable [✓ Enable]

Currently Available Services

Ch	Status	Service	Type	Protocol	Wait	User	Server	S
	NI	admind	stream	tcp	No	root	/usr/sbin/admind	
	NI	amanda	dgram	udp	Yes	amanda.disk	/usr/lib/amanda/amandad	
	NI	amandaidx	stream	tcp	No	amanda.disk	/usr/lib/amanda/amindexd	
	NI	amidxtape	stream	tcp	No	amanda.disk	/usr/lib/amanda/amidxtaped	
	NI	bootps	dgram	udp	Yes	root.root	/usr/sbin/bootpd	-c
	---	chargen	stream	tcp	No	root		
	---	chargen	dgram	udp	Yes	root		
	---	cvspserver	stream	tcp	No	root	/usr/bin/cvs	-f
X	On	daytime	stream	tcp	No	root		
X	On	daytime	dgram	udp	Yes	root		
	---	discard	stream	tcp	No	root		
	---	discard	dgram	udp	Yes	root		
	NI	dvbcat	stream	tcp	No	dvbdaemon	/usr/bin/dvbcat-network.pl	
	---	echo	stream	tcp	No	root		
	---	echo	dgram	udp	Yes	root		
	NI	exec	stream	tcp	No	root.root	/usr/sbin/tcpd	/u
	NI	fam/2	stream	tcp	Yes	root.root	/usr/sbin/famd	
	NI	finger	stream	tcp	No	nobody	/usr/sbin/in.fingerd	-v

neil@suse103:~/BLP4e/chapte
neil@suse103:~/BLP4e/chapte
YaST Control Center @ suse10
YaST2@suse103
11:24

Figure 15-1

Note that in this example the ftp service is provided by the external program `wu.ftpd`. If your system is running `inetd`, you can change the services provided by editing `/etc/inetd.conf` (a `#` at the start of a line indicates that the line is a comment) and restarting the `inetd` process. This can be done by sending it a hang-up signal using `kill`. To make this easier, some systems are configured so that `inetd` writes its process ID to a file. Alternatively, `killall` can be used:

```
# killall -HUP inetd
```

Socket Options

There are many options that you can use to control the behavior of socket connections — too many to detail here. The `setsockopt` function is used to manipulate options.

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

int setsockopt(int socket, int level, int option_name,
        const void *option_value, size_t option_len);
```

You can set options at various levels in the protocol hierarchy. To set options at the socket level, you must set `level` to `SOL_SOCKET`. To set options at the underlying protocol level (TCP, UDP, and so on), set `level` to the number of the protocol (from either the header file `netinet/in.h` or as obtained by the function `getprotobyname`).

The `option_name` parameter selects an option to set; the `option_value` parameter is an arbitrary value of length `option_len` bytes passed unchanged to the underlying protocol handler.

Socket level options defined in `sys/socket.h` include the following.

OPTION	DESCRIPTION
`SO_DEBUG`	Turn on debugging information.
`SO_KEEPALIVE`	Keep connections active with periodic transmissions.
`SO_LINGER`	Complete transmission before close.

`SO_DEBUG` and `SO_KEEPALIVE` take an integer `option_value` used to turn the option on (`1`) or off (`0`). `SO_LINGER` requires a `linger` structure defined in `sys/socket.h` to define the state of the option and the linger interval.

`setsockopt` returns `0` if successful, `-1` otherwise. The manual pages describe further options and errors.

Multiple Clients

So far in this chapter, you've seen how you can use sockets to implement client/server systems both locally and across networks. Once established, socket connections behave like low-level open file descriptors and in many ways like bi-directional pipes.

You might need to consider the case of multiple, simultaneous clients connecting to a server. You've seen that when a server program accepts a new connection from a client, a new socket is created and the original listen socket remains available for further connections. If the server doesn't immediately accept further connections, they will be held pending in a queue.

The fact that the original socket is still available and that sockets behave as file descriptors gives you a method of serving multiple clients at the same time. If the server calls `fork` to create a second copy of itself, the open socket will be inherited by the new child process. It can then communicate with the connecting client while the main server continues to accept further client connections. This is, in fact, a fairly easy change to make to your server program, which is shown in the following Try It Out section.

Because you're creating child processes but not waiting for them to complete, you must arrange for the server to ignore `SIGCHLD` signals to prevent zombie processes.

Try It Out A Server for Multiple Clients

1. This program, `server4.c`, begins in a similar vein to the last server, with the notable addition of an `include` for the `signal.h` header file. The variables and the procedure of creating and naming a socket are the same:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;

    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

2. Create a connection queue, ignore child exit details, and wait for clients:

```
    listen(server_sockfd, 5);

    signal(SIGCHLD, SIG_IGN);

    while(1) {
        char ch;

        printf("server waiting\n");
```

3. Accept the connection:

```
        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);
```

4. Fork to create a process for this client and perform a test to see whether you're the parent or the child:

```
        if(fork() == 0) {
```

5. If you're the child, you can now read/write to the client on `client_sockfd`. The five-second delay is just for this demonstration:

```
            read(client_sockfd, &ch, 1);
            sleep(5);
            ch++;
            write(client_sockfd, &ch, 1);
            close(client_sockfd);
            exit(0);
        }
```

6. Otherwise, you must be the parent and your work for this client is finished:

```
        else {
            close(client_sockfd);
        }
    }
}
```

The code inserts a five-second delay in the processing of the client's request to simulate server calculation or database access. If you had done this with the previous server, each run of `client3` would have taken five seconds. With the new server, you can handle multiple `client3` programs concurrently, with an overall elapsed time of just over five seconds.

```
$ ./server4 &
[1] 26566
server waiting
$ ./client3 & ./client3 & ./client3 & ps x
[2] 26581
[3] 26582
[4] 26583
server waiting
server waiting
server waiting
  PID TTY      STAT   TIME COMMAND
26566 pts/1    S      0:00 ./server4
26581 pts/1    S      0:00 ./client3
26582 pts/1    S      0:00 ./client3
26583 pts/1    S      0:00 ./client3
26584 pts/1    R+     0:00 ps x
26585 pts/1    S      0:00 ./server4
26586 pts/1    S      0:00 ./server4
26587 pts/1    S      0:00 ./server4
$ char from server = B
char from server = B
char from server = B
ps x
  PID TTY      STAT   TIME COMMAND
26566 pts/1    S      0:00 ./server4
26590 pts/1    R+     0:00 ps x
[2]   Done                   ./client3
[3]-  Done                   ./client3
[4]+  Done                   ./client3
$
```

How It Works

The server program now creates a new child process to handle each client, so you see several server waiting messages as the main program continues to wait for new connections. The `ps` output (edited here) shows the main `server4` process, PID 26566, waiting for new clients while the three `client3` processes are being served by three children of the server. After a five-second pause, all of the clients get their results and finish. The child server processes exit to leave just the main server alone again.

The server program uses fork to handle multiple clients. In a database application, this may not be the best solution, because the server program may be quite large and there is still the problem of coordinating database accesses from multiple server copies. In fact, what you really need is a way for a single server to handle multiple clients without blocking and waiting on client requests to arrive. The solution to this problem involves handling multiple open file descriptors at once and isn't limited to socket applications. Enter select.

select

Quite often when you're writing Linux applications, you may need to examine the state of a number of inputs to determine the next action to take. For example, a communication program such as a terminal emulator needs to read the keyboard and the serial port effectively at the same time. In a single-user system, it might be acceptable to run in a "busy wait" loop, repeatedly scanning the input for data and reading it if it arrives. This behavior is expensive in terms of CPU time.

The select system call allows a program to wait for input to arrive (or output to complete) on a number of low-level file descriptors at once. This means that the terminal emulator program can block until there is something to do. Similarly, a server can deal with multiple clients by waiting for a request on many open sockets at the same time.

The select function operates on data structures, fd_set, that are sets of open file descriptors. A number of macros are defined for manipulating these sets:

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

void FD_ZERO(fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```

As suggested by their names, FD_ZERO initializes an fd_set to the empty set, FD_SET and FD_CLR set and clear elements of the set corresponding to the file descriptor passed as fd, and FD_ISSET returns nonzero if the file descriptor referred to by fd is an element of the fd_set pointed to by fdset. The maximum number of file descriptors in an fd_set structure is given by the constant FD_SETSIZE.

The select function can also use a timeout value to prevent indefinite blocking. The timeout value is given using a struct timeval. This structure, defined in sys/time.h, has the following members:

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

The time_t type is defined in sys/types.h as an integral type.

The select system call has the following prototype:

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

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);
```

A call to select is used to test whether any one of a set of file descriptors is ready for reading or writing or has an error condition pending and will optionally block until one is ready.

The nfds argument specifies the number of file descriptors to be tested, and descriptors from 0 to nfds-1 are considered. Each of the three descriptor sets may be a null pointer, in which case the associated test isn't carried out.

The select function will return if any of the descriptors in the readfds set are ready for reading, if any in the writefds set are ready for writing, or if any in errorfds have an error condition. If none of these conditions apply, select will return after an interval specified by timeout. If the timeout parameter is a null pointer and there is no activity on the sockets, the call will block forever.

When select returns, the descriptor sets will have been modified to indicate which descriptors are ready for reading or writing or have errors. You should use FD_ISSET to test them, to determine the descriptor(s) needing attention. You can modify the timeout value to indicate the time remaining until the next timeout, but this behavior isn't specified by X/Open. In the case of a timeout occurring, all descriptor sets will be empty.

The select call returns the total number of descriptors in the modified sets. It returns –1 on failure, setting errno to describe the error. Possible errors are EBADF for invalid descriptors, EINTR for return due to interrupt, and EINVAL for bad values for nfds or timeout.

Although Linux modifies the structure pointed to by timeout *to indicate the time remaining, most versions of UNIX do not. Much existing code that uses the* select *function initializes a* timeval *structure and then continues to use it without ever reinitializing the contents. On Linux, this code may operate incorrectly because Linux is modifying the* timeval *structure every time a timeout occurs. If you're writing or porting code that uses the* select *function, you should watch out for this difference and always reinitialize the timeout. Note that both behaviors are correct; they're just different!*

Try It Out select

Here is a program, select.c, to illustrate the use of select. You'll see a more complete example a little later. This program reads the keyboard (standard input — file descriptor 0) with a timeout of 2.5 seconds. It reads the keyboard only when input is ready. It's quite straightforward to extend it to include other descriptors, such as serial lines or pipes and sockets, depending on the application.

1. Begin as usual with the includes and declarations and then initialize inputs to handle input from the keyboard:

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
#include <fcntl.h>
```

```
#include <sys/ioctl.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    char buffer[128];
    int result, nread;

    fd_set inputs, testfds;
    struct timeval timeout;

    FD_ZERO(&inputs);
    FD_SET(0,&inputs);
```

2. Wait for input on `stdin` for a maximum of 2.5 seconds:

```
    while(1) {
        testfds = inputs;
        timeout.tv_sec = 2;
        timeout.tv_usec = 500000;

        result = select(FD_SETSIZE, &testfds, (fd_set *)NULL, (fd_set *)NULL,
                        &timeout);
```

3. After this time, test `result`. If there has been no input, the program loops again. If there has been an error, the program exits:

```
        switch(result) {
        case 0:
            printf("timeout\n");
            break;
        case -1:
            perror("select");
            exit(1);
```

4. If, during the wait, you have some action on the file descriptor, read the input on `stdin` and echo it whenever an <end of line> character is received, until that input is Ctrl+D:

```
        default:
            if(FD_ISSET(0,&testfds)) {
                ioctl(0,FIONREAD,&nread);
                if(nread == 0) {
                    printf("keyboard done\n");
                    exit(0);
                }
                nread = read(0,buffer,nread);
                buffer[nread] = 0;
                printf("read %d from keyboard: %s", nread, buffer);
            }
            break;
        }
    }
}
```

When you run this program, it prints `timeout` every two and a half seconds. If you type at the keyboard, it reads the standard input and reports what was typed. With most shells, the input will be sent to the program when the user presses the Enter (or Return) key or keys in a control sequence, so your program will print the input whenever you press Enter. Note that the Enter key itself is read and processed like any other character (try this by not pressing Enter, but a number of characters followed by Ctrl+D).

```
$ ./select
timeout
hello
read 6 from keyboard: hello
fred
read 5 from keyboard: fred
timeout
^D
keyboard done
$
```

How It Works

The program uses the `select` call to examine the state of the standard input. By arranging a timeout value, the program resumes every 2.5 seconds to print a timeout message. This is indicated by `select` returning zero. On end of file, the standard input descriptor is flagged as ready for input, but there are no characters to be read.

Multiple Clients

Your simple server program can benefit by using `select` to handle multiple clients simultaneously, without resorting to child processes. For real applications using this technique, you must take care that you do not make other clients wait too long while you deal with the first to connect.

The server can use `select` on both the listen socket and the clients' connection sockets at the same time. Once activity has been indicated, you can use `FD_ISSET` to cycle through all the possible file descriptors to discover which one the activity is on.

If the listen socket is ready for input, this will mean that a client is attempting to connect and you can call `accept` without risk of blocking. If a client descriptor is indicated ready, this means that there's a client request pending that you can read and deal with. A read of zero bytes will indicate that a client process has ended and you can close the socket and remove it from your descriptor set.

Try It Out An Improved Multiple Client/Server

1. For the final example, `server5.c`, you'll include the `sys/time.h` and `sys/ioctl.h` headers instead of `signal.h` as in the last program, and declare some extra variables to deal with `select`:

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

```
#include <netinet/in.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
    int result;
    fd_set readfds, testfds;
```

2. Create and name a socket for the server:

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

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);

    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

3. Create a connection queue and initialize `readfds` to handle input from `server_sockfd`:

```
    listen(server_sockfd, 5);

    FD_ZERO(&readfds);
    FD_SET(server_sockfd, &readfds);
```

4. Now wait for clients and requests. Because you have passed a null pointer as the `timeout` parameter, no timeout will occur. The program will exit and report an error if `select` returns a value less than 1:

```
while(1) {
        char ch;
        int fd;
        int nread;

        testfds = readfds;

        printf("server waiting\n");
        result = select(FD_SETSIZE, &testfds, (fd_set *)0,
            (fd_set *)0, (struct timeval *) 0);

        if(result < 1) {
            perror("server5");
            exit(1);
        }
```

5. Once you know you've got activity, you can find which descriptor it's on by checking each in turn using `FD_ISSET`:

```
        for(fd = 0; fd < FD_SETSIZE; fd++) {
            if(FD_ISSET(fd,&testfds)) {
```

6. If the activity is on `server_sockfd`, it must be a request for a new connection, and you add the associated `client_sockfd` to the descriptor set:

```
                if(fd == server_sockfd) {
                    client_len = sizeof(client_address);
                    client_sockfd = accept(server_sockfd,
                        (struct sockaddr *)&client_address, &client_len);
                    FD_SET(client_sockfd, &readfds);
                    printf("adding client on fd %d\n", client_sockfd);
                }
```

7. If it isn't the server, it must be client activity. If `close` is received, the client has gone away, and you remove it from the descriptor set. Otherwise, you "serve" the client as in the previous examples.

```
                else {
                    ioctl(fd, FIONREAD, &nread);

                    if(nread == 0) {
                        close(fd);
                        FD_CLR(fd, &readfds);
                        printf("removing client on fd %d\n", fd);
                    }

                    else {
                        read(fd, &ch, 1);
                        sleep(5);
                        printf("serving client on fd %d\n", fd);
                        ch++;
                        write(fd, &ch, 1);
                    }
                }
            }
        }
    }
}
```

In a real-world program, it would be advisable to include a variable holding the largest `fd` *number connected (not necessarily the most recent* `fd` *number connected). This would prevent looping through potentially thousands of* `fd`*s that aren't even connected and couldn't possibly be ready for reading. We've omitted it here simply for brevity's sake and to make the code simpler.*

When you run this version of the server, it deals with multiple clients sequentially in a single process.

```
$ ./server5 &
[1] 26686
server waiting
$ ./client3 & ./client3 & ./client3 & ps x
[2] 26689
[3] 26690
adding client on fd 4
server waiting
[4] 26691
  PID TTY      STAT   TIME COMMAND
26686 pts/1    S      0:00 ./server5
26689 pts/1    S      0:00 ./client3
26690 pts/1    S      0:00 ./client3
26691 pts/1    S      0:00 ./client3
26692 pts/1    R+     0:00 ps x
$ serving client on fd 4
server waiting
adding client on fd 5
server waiting
adding client on fd 6
char from server = B
serving client on fd 5
server waiting
removing client on fd 4
char from server = B
serving client on fd 6
server waiting
removing client on fd 5
server waiting
char from server = B
removing client on fd 6
server waiting

[2]   Done                    ./client3
[3]-  Done                    ./client3
[4]+  Done                    ./client3
$
```

To complete the analogy at the start of the chapter, the following table shows the parallels between socket connections and a telephone exchange.

Telephone	Network Sockets
Call company on 555-0828	Connect to IP address 127.0.0.1.
Call answered by reception	Connection established to `remote host`.
Ask for finance department	Route using specified port (9734).

Continued on next page

Telephone	Network Sockets
Call answered by finance administration	Server returns from `select`.
Call put through to free account manager	Server calls `accept`, creating new socket on extension 456.

Datagrams

In this chapter, we have concentrated on programming applications that maintain connections to their clients, using connection-oriented TCP socket connections. There are cases where the overhead of establishing and maintaining a socket connection is unnecessary.

The `daytime` service used in `getdate.c` earlier provides a good example. You create a socket, make a connection, read a single response, and close the connection. That's a lot of operations just to get the date.

The `daytime` service is also available by UDP using datagrams. To use it, you send a single datagram to the service and get a single datagram containing the date and time in response. It's simple.

Services provided by UDP are typically used where a client needs to make a short query of a server and expects a single short response. If the cost in terms of processing time is low enough, the server is able to provide this service by dealing with requests from clients one at a time, allowing the operating system to hold incoming requests in a queue. This simplifies the coding of the server.

Because UDP is not a guaranteed service, however, you may find that your datagram or your response goes missing. So if the data is important to you, you would need to code your UDP clients carefully to check for errors and retry if necessary. In practice, on a local area network, UDP datagrams are very reliable.

To access a service provided by UDP, you need to use the `socket` and `close` system calls as before, but rather than using `read` and `write` on the socket, you use two datagram-specific system calls, `sendto` and `recvfrom`.

Here's a modified version of `getdate.c` that gets the date via a UDP datagram service. Changes from the earlier version are highlighted.

```
/*  Start with the usual includes and declarations.  */

#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *host;
    int sockfd;
```

```
        int len, result;
        struct sockaddr_in address;
        struct hostent *hostinfo;
        struct servent *servinfo;
        char buffer[128];

        if(argc == 1)
            host = "localhost";
        else
            host = argv[1];

    /*  Find the host address and report an error if none is found.  */

        hostinfo = gethostbyname(host);
        if(!hostinfo) {
            fprintf(stderr, "no host: %s\n", host);
            exit(1);
        }

    /*  Check that the daytime service exists on the host.  */

        servinfo = getservbyname("daytime", "udp");
        if(!servinfo) {
            fprintf(stderr,"no daytime service\n");
            exit(1);
        }
        printf("daytime port is %d\n", ntohs(servinfo -> s_port));

    /*  Create a UDP socket.  */

        sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    /*  Construct the address for use with sendto/recvfrom...  */

        address.sin_family = AF_INET;
        address.sin_port = servinfo -> s_port;
        address.sin_addr = *(struct in_addr *)*hostinfo -> h_addr_list;
        len = sizeof(address);

        result = sendto(sockfd, buffer, 1, 0, (struct sockaddr *)&address, len);
        result = recvfrom(sockfd, buffer, sizeof(buffer), 0,
                          (struct sockaddr *)&address, &len);
        buffer[result] = '\0';
        printf("read %d bytes: %s", result, buffer);

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

As you can see, the changes required are very small. You find the daytime service with `getservbyname` as before, but you specify the datagram service by requesting the UDP protocol. You create a datagram socket using socket with a `SOCK_DGRAM` parameter. You set up the destination address as before, but now you have to send a datagram rather than just read from the socket.

Because you are not making an explicit connection to services provided by UDP, you have to have some way of letting the server know that you want to receive a response. In this case, you send a datagram (here you send a single byte from the buffer you are going to receive the response into) to the service, and it responds with the date and time.

The `sendto` system call sends a datagram from a buffer on a socket using a socket address and address length. Its prototype is essentially

```
int sendto(int sockfd, void *buffer, size_t len, int flags,
           struct sockaddr *to, socklen_t tolen);
```

In normal use, the `flags` parameter can be kept zero.

The `recvfrom` system call waits on a socket for a datagram from a specified address and receives it into a buffer. Its prototype is essentially

```
int recvfrom(int sockfd, void *buffer, size_t len, int flags,
           struct sockaddr *from, socklen_t *fromlen);
```

Again, in normal use, the `flags` parameter can be kept zero.

To keep the example short, we have omitted error handling. Both sendto and recvfrom will return |1 if an error occurs and will set errno appropriately. Possible errors include the following:

To keep the example short we have omitted error

Errno Value	Description
`EBADF`	An invalid file descriptor was passed.
`EINTR`	A signal occurred.

Unless the socket is set nonblocking using `fcntl` (as you saw for accepting TCP connections earlier), the `recvfrom` call will block indefinitely. The socket can, however, be used with `select` and a timeout to determine whether any data has arrived in the same way that you have seen with the connection-based servers earlier. Alternatively, an alarm clock signal can be used to interrupt a receive operation (see Chapter 11).

Summary

In this chapter, we've covered another method of inter-process communication: sockets. These allow you to develop true distributed client/server applications to run across networks. We briefly covered some of the host database information functions and how Linux handles standard system services with the Internet daemon. You worked through a number of client/server example programs that demonstrate networking and handling multiple clients.

Finally, you learned about the `select` system call that allows a program to be advised of input and output activity on several open file descriptors and sockets at once.