


```
int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, cli_len, n;
```

`sockfd` and `newsockfd` are file descriptors, i.e. array subscripts into the [file descriptor table](#). These two variables store the values returned by the socket system call and the accept system call.

`portno` stores the port number on which the server accepts connections.

`cli_len` stores the size of the address of the client. This is needed for the accept system call.

`n` is the return value for the `read()` and `write()` calls; i.e. it contains the number of characters read or written.

```
char buffer[256];
```

The server reads characters from the socket connection into this buffer.

```
struct sockaddr_in serv_addr, cli_addr;
```

A `sockaddr_in` is a structure containing an internet address. This structure is defined in `netinet/in.h`. Here is the definition:

```
struct sockaddr_in
{
    short    sin_family; /* must be AF_INET */
    u_short  sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8]; /* Not used, must be zero */
};
```

An `in_addr` structure, defined in the same header file, contains only one field, a unsigned long called `s_addr`.

The variable `serv_addr` will contain the address of the server, and `cli_addr` will contain the address of the client which connects to the server.

```
if (argc < 2)
{
    fprintf(stderr, "ERROR, no port provided\n");
    exit(1);
}
```

The user needs to pass in the port number on which the server will accept connections as an argument. This code displays an error message if the user fails to do this.

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```

The `socket()` system call creates a new socket. It takes three arguments. The first is the address domain of the socket.

Recall that there are two possible address domains, the unix domain for two processes which share a common file system, and the Internet domain for any two hosts on the Internet. The symbol constant `AF_UNIX` is used for the former, and `AF_INET` for the latter (there are actually many other options which can be used here for specialized purposes).

The second argument is the type of socket. Recall that there are two choices here, a stream socket in which characters are read in a continuous stream as if from a file or pipe, and a datagram socket, in which messages are read in chunks. The two symbolic constants are `SOCK_STREAM` and `SOCK_DGRAM`.

The third argument is the protocol. If this argument is zero (and it always should be except for unusual circumstances), the operating system will choose the most appropriate protocol. It will choose TCP for stream sockets and UDP for datagram sockets.

The socket system call returns an entry into the file descriptor table (i.e. a small integer). This value is used for all subsequent references to this socket. If the socket call fails, it returns -1.

In this case the program displays and error message and exits. However, this system call is unlikely to fail.

This is a simplified description of the socket call; there are numerous other choices for domains and types, but these are the most common. The [socket\(\) man page](#) has more information.

```
bzero((char *) &serv_addr, sizeof(serv_addr));
```

The function `bzero()` sets all values in a buffer to zero. It takes two arguments, the first is a pointer to the buffer and the second is the size of the buffer. Thus, this line initializes `serv_addr` to zeros. ----

```
portno = atoi(argv[1]);
```

The port number on which the server will listen for connections is passed in as an argument, and this statement uses the `atoi()` function to convert this from a string of digits to an integer.

```
serv_addr.sin_family = AF_INET;
```

The variable `serv_addr` is a structure of type `struct sockaddr_in`. This structure has four fields. The first field is `short sin_family`, which contains a code for the address family. It should always be set to the symbolic constant `AF_INET`.

```
serv_addr.sin_port = htons(portno);
```

The second field of `serv_addr` is unsigned short `sin_port`, which contain the port number. However, instead of simply copying the port number to this field, it is necessary to convert this to [network byte order](#) using the function `htons()` which converts a port number in host byte order to a port number in network byte order.

```
serv_addr.sin_addr.s_addr = INADDR_ANY;
```

The third field of `sockaddr_in` is a structure of type `struct in_addr` which contains only a single field unsigned long `s_addr`. This field contains the IP address of the host. For server code, this will always be the IP address of the machine on which the server is running, and there is a symbolic constant `INADDR_ANY` which gets this address.

```
if (bind(sockfd, (struct sockaddr *) &serv_addr,
    sizeof(serv_addr)) < 0)
    error("ERROR on binding");
```

The `bind()` system call binds a socket to an address, in this case the address of the current host and port number on which the server will run. It takes three arguments, the socket file descriptor, the address to which is bound, and the size of the address to which it is bound. The second argument is a pointer to a structure of type `sockaddr`, but what is passed in is a structure of type `sockaddr_in`, and so this must be cast to the correct type. This can fail for a number of reasons, the most obvious being that this socket is already in use on this machine. The [bind\(\) manual](#) has more information.

```
listen(sockfd,5);
```

The `listen` system call allows the process to listen on the socket for connections. The first argument is the socket file descriptor, and the second is the size of the backlog queue, i.e., the number of connections that can be waiting while the process is handling a particular connection. This should be set to 5, the maximum size permitted by most systems. If the first argument is a valid socket, this call cannot fail, and so the code doesn't check for errors. The [listen\(\) man page](#) has more information.

```
cli_len = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &cli_len);
if (newsockfd < 0)
    error("ERROR on accept");
```

The `accept()` system call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. The second argument is a reference pointer to the address of the client on the other end of the connection, and the third argument is the size of this structure. The [accept\(\) man page](#) has more information.

```
bzero(buffer,256);
n = read(newsockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
printf("Here is the message: %s\n",buffer);
```

Note that we would only get to this point after a client has successfully connected to our server. This code initializes the buffer using the `bzero()` function, and then reads from the socket. Note that the read call uses the new file descriptor, the one returned by `accept()`, not the original file descriptor returned by `socket()`. Note also that the `read()` will block until there is something for it to read in the socket, i.e. after the client has executed a `write()`.

It will read either the total number of characters in the socket or 255, whichever is less, and return the number of characters read. The [read\(\) man page](#) has more information.

```
n = write(newsockfd, "I got your message", 18);
if (n < 0) error("ERROR writing to socket");
```

Once a connection has been established, both ends can both read and write to the connection. Naturally, everything written by the client will be read by the server, and everything written by the server will be read by the client. This code simply writes a short message to the client. The last argument of write is the size of the message. The [write\(\) man page](#) has more information.

```
return 0;
}
```

This terminates main and thus the program. Since main was declared to be of type int as specified by the ascii standard, some compilers complain if it does not return anything.

Client code

As before, we will go through the program `client.c` line by line.

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

The header files are the same as for the server with one addition. The file `netdb.h` defines the structure `hostent`, which will be used below.

```
void error(char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
```

The `error()` function is identical to that in the server, as are the variables `sockfd`, `portno`, and `n`. The variable `serv_addr` will contain the address of the server to which we want to connect. It is of type [struct sockaddr_in](#).

The variable `server` is a pointer to a structure of type `hostent`. This structure is defined in the header file `netdb.h` as follows:

```
struct hostent
{
    char *h_name; /* official name of host */
    char **h_aliases; /* alias list */
    int h_addrtype; /* host address type */
    int h_length; /* length of address */
    char **h_addr_list; /* list of addresses from name server */
    #define h_addr h_addr_list[0] /* address, for backward compatibility */
};
```

It defines a host computer on the Internet. The members of this structure are:

<code>h_name</code>	Official name of the host.
<code>h_aliases</code>	A zero terminated array of alternate names for the host.
<code>h_addrtype</code>	The type of address being returned; currently always AF_INET.
<code>h_length</code>	The length, in bytes, of the address.
<code>h_addr_list</code>	A pointer to a list of network addresses for the named host. Host addresses are returned in network byte order.

Note that `h_addr` is an alias for the first address in the array of network addresses.

```
char buffer[256];
if (argc < 3)
{
    fprintf(stderr, "usage %s hostname port\n", argv[0]);
    exit(0);
}
portno = atoi(argv[2]);
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```

All of this code is the same as that in the server.

```
server = gethostbyname(argv[1]);
if (server == NULL)
{
    fprintf(stderr, "ERROR, no such host\n");
    exit(0);
}
```

The variable `argv[1]` contains the name of a host on the Internet, e.g. `cs.rpi.edu`. The function:

```
struct hostent *gethostbyname(char *name)
```

Takes such a name as an argument and returns a pointer to a `hostent` containing information about that host.

The field `char *h_addr` contains the IP address.

If this structure is NULL, the system could not locate a host with this name.

In the old days, this function worked by searching a system file called `/etc/hosts` but with the explosive growth of the Internet, it became impossible for system administrators to keep this file current. Thus, the mechanism by which this function works is complex, often involves querying large databases all around the country. The [gethostbyname\(\) man page](#) has more information.

```
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&serv_addr.sin_addr.s_addr,
      server->h_length);
serv_addr.sin_port = htons(portno);
```

This code sets the fields in `serv_addr`. Much of it is the same as in the server. However, because the field `server->h_addr` is a character string, we use the function:

```
void bcopy(char *s1, char *s2, int length)
```

which copies `length` bytes from `s1` to `s2`. ----

```
if (connect(sockfd, &serv_addr, sizeof(serv_addr)) < 0)
    error("ERROR connecting");
```

The `connect` function is called by the client to establish a connection to the server. It takes three arguments, the socket file descriptor, the address of the host to which it wants to connect (including the port number), and the size of this address. This function returns 0 on success and -1 if it fails. The [connect\(\) man page](#) has more information.

Notice that the client needs to know the port number of the server, but it does not need to know its own port number. This is typically assigned by the system when `connect` is called.

```
printf("Please enter the message: ");
bzero(buffer, 256);
fgets(buffer, 256, stdin);
n = write(sockfd, buffer, strlen(buffer));
if (n < 0)
    error("ERROR writing to socket");
bzero(buffer, 256);
n = read(sockfd, buffer, 255);
if (n < 0)
    error("ERROR reading from socket");
printf("%s\n", buffer);
return 0;
}
```

The remaining code should be fairly clear. It prompts the user to enter a message, uses `fgets` to read the message from `stdin`, writes the message to the socket, reads the reply from the socket, and displays this reply on the screen.

Enhancements to the server code

The sample server code above has the limitation that it only handles one connection, and then dies. A "real world" server should run indefinitely and should have the capability of handling a number of simultaneous connections, each in its own process. This is typically done by forking off a new process to handle each new connection.

The following code has a dummy function called `dostuff(int sockfd)`.

This function will handle the connection after it has been established and provide whatever services the client requests. As we saw above, once a connection is established, both ends can use `read` and `write` to send information to the other end, and the details of the information passed back and forth do not concern us here.

To write a "real world" server, you would make essentially no changes to the `main()` function, and all of the code which provided the service would be in `dostuff()`.

To allow the server to handle multiple simultaneous connections, we make the following changes to the code:

1. Put the accept statement and the following code in an infinite loop.
2. After a connection is established, call `fork()` to create a new process.
3. The child process will close `sockfd` and call `#dostuff#`, passing the new socket file descriptor as an argument. When the two processes have completed their conversation, as indicated by `dostuff()` returning, this process simply exits.
4. The parent process closes `newsockfd`. Because all of this code is in an infinite loop, it will return to the accept statement to wait for the next connection.

Here is the code.

```
while (1)
{
    newsockfd = accept(sockfd,
        (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    pid = fork();
    if (pid < 0)
        error("ERROR on fork");
    if (pid == 0)
    {
        close(sockfd);
        dostuff(newsockfd);
        exit(0);
    }
    else
        close(newsockfd);
} /* end of while */
```

[Click here](#) for a complete server program which includes this change. This will run with the program `client.c`.

The zombie problem

The above code has a problem; if the parent runs for a long time and accepts many connections, each of these connections will create a zombie when the connection is terminated. A zombie is a process which has terminated but but cannot be permitted to fully die because at some point in the future, the parent of the process might execute a `wait` and would want information about the death of the child. Zombies clog up the process table in the kernel, and so they should be prevented. Unfortunately, the code which prevents zombies is not consistent across different architectures. When a child dies, it sends a `SIGCHLD` signal to its parent. On systems such as AIX, the following code in `main()` is all that is needed.

```
signal(SIGCHLD, SIG_IGN);
```

This says to ignore the `SIGCHLD` signal. However, on systems running SunOS, you have to use the following code:

```
void *SigCatcher(int n)
{
    wait3(NULL, WNOHANG, NULL);
}
...
int main()
{
    ...
    signal(SIGCHLD, SigCatcher);
    ...
}
```

The function `SigCatcher()` will be called whenever the parent receives a `SIGCHLD` signal (i.e. whenever a child dies). This will in turn call `wait3` which will receive the signal. The `WNOHANG` flag is set, which causes this to be a non-blocking wait (one of my favorite [http://rampages.onramp.net/~greid/oxyamoron.htm oxyamorons]).

Alternative types of sockets

This example showed a stream socket in the Internet domain. This is the most common type of connection. A second type of connection is a datagram socket. You might want to use a datagram socket in cases where there is only one message being sent from the client to the server, and only one message being sent back. There are several differences between a datagram socket and a stream socket.

1. Datagrams are unreliable, which means that if a packet of information gets lost somewhere in the Internet, the sender is not told (and of course the receiver does not know about the existence of the message). In contrast, with a stream socket, the underlying TCP protocol will detect that a message was lost because it was not acknowledged, and it will be retransmitted without the process at either end knowing about this.
2. Message boundaries are preserved in datagram sockets. If the sender sends a datagram of 100 bytes, the receiver must read all 100 bytes at once. This can be contrasted with a stream socket, where if the sender wrote a 100 byte message, the receiver could read it in two chunks of 50 bytes or 100 chunks of one byte.
3. The communication is done using special system calls `sendto()` and `recvfrom()` rather than the more generic `read()` and `write()`.
4. There is a lot less overhead associated with a datagram socket because connections do not need to be established and broken down, and packets do not need to be acknowledged. This is why datagram sockets are often used when the service to be provided is short, such as a time-of-day service.

[Click here](#) for the server code using a datagram socket.

[Click here](#) for the client code using a datagram socket.

These two programs can be compiled and run in exactly the same way as the server and client using a stream socket.

Most of the server code is similar to the stream socket code. Here are the differences.

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

Note that when the socket is created, the second argument is the symbolic constant `SOCK_DGRAM` instead of `SOCK_STREAM`. The protocol will be UDP, not TCP. ---

```
fromLen = sizeof(struct sockaddr_in);
while (1)
{
    n = recvfrom(sock, buf, 1024, 0, (struct sockaddr *) &from, &fromlen);
    if (n < 0) error("recvfrom");
}
```

Servers using datagram sockets do not use the `listen()` or the `accept()` system calls. After a socket has been bound to an address, the program calls `recvfrom()` to read a message. This call will block until a message is received. The `recvfrom()` system call takes six arguments. The first three are the same as those for the `read()` call, the socket file descriptor, the buffer into which the message will be read, and the maximum number of bytes. The fourth argument is an integer argument for flags. This is ordinarily set to zero. The fifth argument is a pointer to a [sockaddr_in structure](#). When the call returns, the values of this structure will have been filled in for the other end of the connection (the client). The size of this structure will be in the last argument, a pointer to an integer. This call returns the number of bytes in the message. (or -1 on an error condition). The [recvfrom\(\) man page](#) has more information.

```
n = sendto(sock, "Got your message", 17, 0, (struct sockaddr *) &from, fromlen);
if (n < 0)
    error("sendto");
}
```

To send a datagram, the function `sendto()` is used. This also takes six arguments. The first three are the same as for a `write()` call, the socket file descriptor, the buffer from which the message will be written, and the number of bytes to write. The fourth argument is an int argument called flags, which is normally zero. The fifth argument is a pointer to a `sockaddr_in` structure. This will contain the address to which the message will be sent. Notice that in this case, since the server is replying to a message, the values of this structure were provided by the `recvfrom` call. The last argument is the size of this structure. Note that this is not a pointer to an int, but an int value itself. The [sendto\(\) man page](#) has more information.

The client code for a datagram socket client is the same as that for a stream socket with the following differences.

- the socket system call has `SOCK_DGRAM` instead of `SOCK_STREAM` as its second argument.

- there is no connect() system call
- instead of read and write, the client uses recvfrom and sendto which are described in detail above.

Sockets in the Unix Domain

Here is the code for a client and server which communicate using a stream socket in the Unix domain.

[U_server.c](#)
[U_client](#)

The only difference between a socket in the Unix domain and a socket in the Internet domain is the form of the address. Here is the address structure for a Unix Domain address, defined in [the header file](#).

```
struct sockaddr_un
{
    short  sun_family; /* AF_UNIX */
    char   sun_path[108]; /* path name (gag) */
};
```

The field sun_path has the form of a path name in the Unix file system. This means that both client and server have to be running the same file system. Once a socket has been created, it remain until it is explicitly deleted, and its name will appear with the ls command, always with a size of zero. Sockets in the Unix domain are virtually identical to named pipes (FIFOs).

Designing servers

There are a number of different ways to design servers. These models are discussed in detail in a book by Douglas E. Comer and David L. Stevens entitled *Internetworking with TCP/IP Volume III: Client Server Programming and Applications* published by [http://www.prenhall.com Prentice Hall] in 1996. These are summarized here.

Concurrent, connection oriented servers

The typical server in the Internet domain creates a stream socket and forks off a process to handle each new connection that it receives. This model is appropriate for services which will do a good deal of reading and writing over an extended period of time, such as a telnet server or an ftp server. This model has relatively high overhead, because forking off a new process is a time consuming operation, and because a stream socket which uses the TCP protocol has high kernel overhead, not only in establishing the connection but also in transmitting information. However, once the connection has been established, data transmission is reliable in both directions.

Iterative, connectionless servers

Servers which provide only a single message to the client often do not involve forking, and often use a datagram socket rather than a stream socket. Examples include a finger daemon or a timeofday server or an echo server (a server which merely echoes a message sent by the client). These servers handle each message as it receives them in the same process. There is much less overhead with this type of server, but the communication is unreliable. A request or a reply may get lost in the Internet, and there is no built-in mechanism to detect and handle this.

Single Process concurrent servers

A server which needs the capability of handling several clients simultaneous, but where each connection is I/O dominated (i.e. the server spends most of its time blocked waiting for a message from the client) is a candidate for a single process, concurrent server. In this model, one process maintains a number of open connections, and listens at each for a message. Whenever it gets a message from a client, it replies quickly and then listens for the next one. This type of service can be done with the select system call.

rate this article:

current rating: average rating: 2.2 (21 votes) (1=very good 6=terrible)

Your rating:

☐ Very good (1) ☐ Good (2) ☐ ok (3) ☐ average (4) ☐ bad (5) ☐ terrible (6)

[back](#)

[comment this article](#)

[RSS-Feeds](#)
[RSS-Feeds](#)
[ATOM-Feeds](#)
 - Powered by [S&P LegendCMS](#) - Running on [XGENTOO LINUX](#) -
 Copyright 2004 S&P Software Design
[W3C XHTML](#) ; [W3C CSS](#) ; [S&P ButtonMaker](#)
 - Level Triple-A Conformance to Web Content Accessibility Guidelines 1.0 -
 - Copyright and legal notices -
 Time to create this page: 13.7 ms

