The Definitive Guide to Linux Network Programming

KEIR DAVIS, JOHN W. TURNER, AND NATHAN YOCOM

Apress®

ISBN (pbk): 1-59059-322-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jim Sumser

Technical Reviewer: Kirk Bauer

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Laura Cheu

Copy Edit Manager and Copy Editor: Nicole LeClerc

Production Manager: Kari Brooks

Production Editor: Janet Vail

Compositor: Molly Sharp, ContentWorks

Proofreader: Linda Seifert

Indexer: James Minkin

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 3

Socket Programming

In Chapter 2, we covered the basic functions used in socket programming, and demonstrated a simple socket server and a simple socket client. Yet while our server and client programs were functional, they didn't do all that much. In this chapter and the following chapters, we'll go through implementing the socket interface in real-world scenarios and creating more robust networked applications step by step.

You'll remember that in Chapter 1 we discussed two types of network communications: those that required a connection, and those that did not. Our simple server and simple client require a connection and use the Transmission Control Protocol (TCP). Thus, the server and client applications in Chapter 2 use *streaming sockets*—that is, sockets that require a connection. There are also *datagram sockets*, or sockets that don't require a connection and use the User Datagram Protocol (UDP). In this chapter, we'll step through a UDP server and a UDP client. We'll also take a look at transferring files back and forth, which is more involved than just sending strings. Finally, we'll discuss error handling and error checking as a key part of any applications you develop.

User Datagram Protocol

In Chapter 2, our client and server programs were based on streaming sockets and used TCP. There is an alternative to using TCP, and that is UDP, or what is otherwise known as datagram sockets. Remember from Chapter 1 that a UDP-based network communication has some particular qualities:

- UDP makes no guarantee of packet delivery.
- UDP datagrams can be lost and arrive out of sequence.
- UDP datagrams can be copied many times and can be sent faster than the receiving node can process them.

UDP sockets are generally used when the entire communication between the server and client can exist within a distinct network packet. Some examples of

UDP-based communications include the Domain Name Service (DNS), Network Time Protocol (NTP), and the Trivial File Transfer Protocol (TFTP). For example, a simple DNS lookup request essentially consists of just two pieces of information: the name or number in the request, and the corresponding answer from the DNS server in the response (which might even be an error). There's no need to incur the overhead of opening a streaming socket and maintaining a connection for such a simple communication. An NTP request is similar. It consists of a question ("What time is it?") and the server's answer. No ongoing communications are necessary. More information on when to use UDP over TCP, or vice versa, is covered in Chapter 7.

UDP Server

Like our streaming socket example in Chapter 2, you can't have a client without having a server first, so let's create a simple UDP server. To begin, we'll set up the `#include` statements for the header files that we'll need.

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

Next, we define a constant that is 1KB in size. We'll use this constant to define the sizes of any message buffers we use.

```
#define MAXBUF 1024
```

Our server starts with standard declaration for `main()`, followed by the initialization of some variables we'll need. We define a character buffer of size `MAXBUF` to hold anything we need to send to or receive from our socket, and we set up two `sockaddr_in` structures: one for the server and one for the client. These structures will hold the metadata about the endpoints in our communication.

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

    int udpSocket;
    int returnStatus = 0;
    int addrlen = 0;
    struct sockaddr_in udpServer, udpClient;
    char buf[MAXBUF];
```

Before we continue, we check to make sure we have the right number of arguments. The only argument we need for our server is the number of the port that the server should listen on for client connections. This could easily be a constant like `MAXBUF`, but using a command-line argument means our server can listen on any unused port without recompiling it. If we don't have the right number of arguments, we exit.

```
/* check for the right number of arguments */
if (argc < 2)
{
    fprintf(stderr, "Usage: %s <port>\n", argv[0]);
    exit(1);
}
```

Assuming we are good to go, the first thing we should do is create our socket. Note that this process looks nearly identical to the socket we created in the simple server shown in Listing 2-1 in the previous chapter, except that in this case, we're using `SOCK_DGRAM` instead of `SOCK_STREAM`. By using `SOCK_DGRAM`, we're creating a UDP socket instead of a TCP socket.

```
/* create a socket */
udpSocket = socket(AF_INET, SOCK_DGRAM, 0);
if (udpSocket == -1)
{
    fprintf(stderr, "Could not create a socket!\n");
    exit(1);
}
else {
    printf("Socket created.\n");
}
```

Next, we populate the `sockaddr_in` structure reserved for our server with the information we have. We use `INADDR_ANY` so that our socket will bind to any of the local addresses that are configured, and we use the port number that was passed as an argument. Note the call to `htons()` when dealing with the port. This ensures that the integer used for the port number is stored correctly for our architecture.

```
/* set up the server address and port */
/* use INADDR_ANY to bind to all local addresses */
udpServer.sin_family = AF_INET;
udpServer.sin_addr.s_addr = htonl(INADDR_ANY);

/* use the port passed as argument */
udpServer.sin_port = htons(atoi(argv[1]));
```

Next, we bind to the socket and prepare to receive connections. Our call to bind() uses the UDP socket descriptor we created previously, as well as a pointer to our server's sockaddr_in structure and the size of the structure itself.

```
/* bind to the socket */
returnStatus = bind(udpSocket, (struct sockaddr*)&udpServer,
                    sizeof(udpServer));
if (returnStatus == 0) {
    fprintf(stderr, "Bind completed!\n");
}
else {
    fprintf(stderr, "Could not bind to address!\n");
    close(udpSocket);
    exit(1);
}
```

Now that our setup is complete, we use a while loop that will keep our server running until it receives a signal to terminate from the operating system. The main loop in our server will listen on its socket for communications from a client. Remember that UDP is connectionless, so the client will not be making a request to set up and maintain a connection. Rather, the client will simply transmit its request or its information, and the server will wait for it. Because UDP does not guarantee delivery, the server may or may not receive the client's information.

You'll notice that we use the recvfrom() function instead of the listen(), accept(), and read() functions. recvfrom() is a blocking function, much like accept(). The function will wait on the socket until communications are received. Because UDP is connectionless, our server has no idea which client will be sending information to it, and to reply, the server will need to store the client's information locally so that it can send a response to the client if necessary. We'll use the sockaddr_in structure reserved for our client to store that information.

```
while (1)
{
    addrlen = sizeof(udpClient);
    returnStatus = recvfrom(udpSocket, buf, MAXBUF, 0,
                            (struct sockaddr*)&udpClient, &addrlen);
```

recvfrom() takes our size constant as one of its arguments. This means that the maximum amount of information that can be received from a client is limited to the size of the buffer. Any extra bytes of information that are sent are discarded. The recvfrom() function returns the total number of bytes received from the client or –1 if there's an error.

```
        if (returnStatus == -1) {
            fprintf(stderr, "Could not receive message!\n");
        }
        else {

            printf("Received: %s\n", buf);
            /* a message was received so send a confirmation */
            strcpy(buf, "OK");
            returnStatus = sendto(udpSocket, buf, strlen(buf)+1, 0,
                                  (struct sockaddr*)&udpClient,
                                   sizeof(udpClient));

            if (returnStatus == -1) {
                fprintf(stderr, "Could not send confirmation!\n");
            }
            else {
                printf("Confirmation sent.\n");
            }
        }
    }
```

After receiving information from a client and displaying the message to the console, our server resets the buffer and sends a confirmation message back. This way, our client knows that the server got the information and made a reply. Our response uses the complimentary function to `recvfrom()`, called `sendto()`. Much like `recvfrom()`, `sendto()` takes our socket descriptor, the message buffer, and a pointer to the `sockaddr_in` structure for our client as arguments. The return from `sendto()` is the number of bytes sent or –1 if there is an error.

```
    /*cleanup */
    close(udpSocket);
    return 0;
}
```

If our server receives a termination signal from the operating system, the `while` loop terminates. Before exiting, we clean up our socket descriptor. As you can see, our UDP server is remarkably similar to our TCP server, except for a few major differences. First, the socket type is `SOCK_DGRAM` instead of `SOCK_STREAM`. Second, the server has to store information about the client locally to send a response, since a connection isn't maintained. Third, the server doesn't acknowledge receipt of the client's request unless we provide the code to do it.

To run our UDP server, compile it and then run it using a port number as the only argument. You should see output similar to this:

```
[user@host projects]$ cc -o simpleUDPServer simpleUDPServer.c
[user@host projects]$ ./simpleUDPServer 8888
Socket created.
Bind completed!
```

At this point, the server is waiting to receive communications from any clients that choose to send them.

UDP Client

Now that we have our UDP server, let's cover a simple UDP client. Like our server, we'll start out with the relevant `#include` directives and a corresponding constant declaration setting up the maximum size for any message buffers. This constant should obviously be the same for both the server and the client; otherwise, it's possible to lose data.

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

#define MAXBUF 1024
```

We use the standard declaration for main and initialize the variables we need, just like we did for our server. We also check to make sure we have the right number of arguments. In this case, we need two: the IP address of the server and the port number that the server is using.

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

    int udpSocket;
    int returnStatus;
    int addrlen;
    struct sockaddr_in udpClient, udpServer;
    char buf[MAXBUF];
    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s <ip address> <port>\n", argv[0]);
        exit(1);
    }
```

We create our datagram socket and set up our client's `sockaddr_in` structure with the information we have. In this case, we don't need to declare a specific port number for our client. The operating system will assign one randomly that our client can use to send its request. Since the UDP packet the server receives contains a header field with the source IP address as well as the source port, the server will have plenty of information to use when sending its response. Once our information is ready, we bind to the socket and prepare to transmit.

```
    /* create a socket */
    udpSocket = socket(AF_INET, SOCK_DGRAM, 0);

    if (udpSocket == -1)
    {
        fprintf(stderr, "Could not create a socket!\n");
        exit(1);
    }
    else {
        printf("Socket created.\n");
    }

    /* client address */
    /* use INADDR_ANY to use all local addresses */
    udpClient.sin_family = AF_INET;
    udpClient.sin_addr.s_addr = INADDR_ANY;
    udpClient.sin_port = 0;

    returnStatus = bind(udpSocket, (struct sockaddr*)&udpClient,
                        sizeof(udpClient));

    if (returnStatus == 0) {
        fprintf(stderr, "Bind completed!\n");
    }
    else {
        fprintf(stderr, "Could not bind to address!\n");
        close(udpSocket);
        exit(1);
    }
```

Before we transmit, we need to set up our message, which in this case is a simple string. We also need to populate the `sockaddr_in` structure that we'll use to represent the server we want to contact.

```
/* set up the message to be sent to the server */
strcpy(buf, "For Professionals, By Professionals.\n");

/* server address */
/* use the command-line arguments */
udpServer.sin_family = AF_INET;
udpServer.sin_addr.s_addr = inet_addr(argv[1]);
udpServer.sin_port = htons(atoi(argv[2]));
```

Everything is set up, so we use the `sendto()` function to send our request to the server.

```
returnStatus = sendto(udpSocket, buf, strlen(buf)+1, 0,
                      (struct sockaddr*)&udpServer, sizeof(udpServer));

if (returnStatus == -1) {
    fprintf(stderr, "Could not send message!\n");
}
else {

    printf("Message sent.\n");
```

If the value we get back from our call to `sendto()` tells us our request was sent, our client gets ready to receive the server's response using the `recvfrom()` function.

```
        /* message sent: look for confirmation */
        addrlen = sizeof(udpServer);

        returnStatus = recvfrom(udpSocket, buf, MAXBUF, 0,
                                (struct sockaddr*)&udpServer, &addrlen);
        if (returnStatus == -1) {
            fprintf(stderr, "Did not receive confirmation!\n");
        }
        else {
            buf[returnStatus] = 0;
            printf("Received: %s\n", buf);
        }

    }
```

Assuming we got some information from `recvfrom()`, we display it to the screen, clean up, and then exit.

```
    /* cleanup */
    close(udpSocket);
    return 0;

}
```

The key difference between a UDP communication and a TCP communication is that neither the server nor the client has any guarantee that they will receive anything at all. In our example, we had the server send back a confirmation message, but in the real world, a UDP server wouldn't do anything but send back the result of the client's request. For example, in the case of a network time-server, a client would request the current time, and the server's reply would be the current time, provided no errors occurred. If the client didn't receive a reply within a certain time frame, it would typically issue the request again. No confirmation messages or other status messages would be sent, and the server would take no action if the client didn't receive the response. In a TCP communication, the TCP layer on the server would resend the response until the client acknowledged receipt. This is different from a UDP communication, where the client and server applications are wholly responsible for any necessary acknowledgments or retries.

File Transfer

So far, our socket examples have demonstrated the programming concepts involved, but haven't done anything but pass a few strings back and forth. That's fine for example purposes, but what about other kinds of data, such as binary data? As you'll see in this section, sending binary information is a little different than sending simple strings.

Now that you've seen both a TCP connection and a UDP connection in action, let's take a look at what it takes to transfer a file. For this example, we'll use TCP (or streaming) sockets. Our server will bind to a particular port and listen for connections. When a connection is created, the client will send the name of a file to the server, and then the server will read the name of the file, retrieve the file from disk, and send that file back to the client.

One key difference in this example from our earlier TCP server is the use of two sockets instead of one. Why two sockets? With two sockets, our server can handle the request from a client while still accepting more connections from other clients. The connections will be put into a queue on a first-come, first-served basis. If we didn't use two sockets and were busy handling a request, then a second client would be unable to connect and would get a "connection refused" error message. By using two sockets, we can let our server handle its response while lining up other connections to be handled as soon as it's done, without returning an error. Note that this is different than servers that can handle more than one connection at a time.

The Server

Our file transfer server starts out with the familiar #include directives, with one new one. Because we're going to be working with files, we need to include fcntl.h, which contains some constants and other definitions we may need.

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

In this example, we're going to hard-code the port number using a constant called SERVERPORT. We'll do the same on the client. This could easily be changed to use an argument from the command line. We'll also use the same constant called MAXBUF to define the maximum size of our transfer buffers. When we initialize our variables, we'll add a second socket descriptor, giving us two to work with: socket1 and socket2. We'll also define two sockaddr_in structures: one for the server and one for the client.

```
    #define SERVERPORT    8888
    #define MAXBUF        1024

int main()
{

    int socket1,socket2;
    int addrlen;
    struct sockaddr_in xferServer, xferClient;
    int returnStatus;
```

First, we create our socket using SOCK_STREAM. We'll use the first socket descriptor, socket1, for this. If for some reason we can't create our socket, we'll print an error message and exit.

```
/* create a socket */
socket1 = socket(AF_INET, SOCK_STREAM, 0);

if (socket1 == -1)
{
    fprintf(stderr, "Could not create socket!\n");
    exit(1);
}
```

Next, we set up our `sockaddr` structures, using `INADDR_ANY` to bind to all of the local IP addresses and setting the port number to our `SERVERPORT` constant. After we're set up, we make a call to `bind()` to bind our first socket descriptor to the IP address and port.

```
/* bind to a socket, use INADDR_ANY for all local addresses */
xferServer.sin_family = AF_INET;
xferServer.sin_addr.s_addr = INADDR_ANY;
xferServer.sin_port = htons(SERVERPORT);

returnStatus = bind(socket1,
                    (struct sockaddr*)&xferServer,
                    sizeof(xferServer));

if (returnStatus == -1)
{
    fprintf(stderr, "Could not bind to socket!\n");
    exit(1);
}
```

If our call to `bind()` is successful, our next step is to tell our program to listen for requests on the socket using the `listen()` function. For arguments, we'll pass our socket descriptor and the number 5. The second argument defines the *backlog* of pending connections that are allowed. In our case, we are telling our socket that we will allow five connections to be waiting in the queue.

NOTE *The behavior of the backlog argument changed with version 2.2 of the Linux kernel. Previously, the backlog argument represented the maximum number of connection* requests *allowed in the queue. With 2.2 and later, the behavior changed to represent the maximum number of* completed, *established sockets allowed in the queue, not just simple requests.*

```
    returnStatus = listen(socket1, 5);

    if (returnStatus == -1)
    {
        fprintf(stderr, "Could not listen on socket!\n");
        exit(1);
    }
```

At this point, our server is listening on the socket for connections. Our next step is to build the logic to handle requests. In this control loop, we'll initialize some new variables that we'll use to keep track of our file reads and writes, as well as a standard file descriptor and a buffer to hold the filename that we'll be retrieving for clients. The key part of the loop is the call to `accept()`. Notice that we call `accept()` and pass it our first socket. The value returned by `accept()`, though, is the descriptor to yet another socket, `socket2`. This lets our server queue up other connections on the first socket while it waits to complete the one it is handling, up to the maximum limit of the backlog argument (in this case, five).

```
    for(;;)
    {

        int fd;
        int i, readCounter, writeCounter;
        char* bufptr;
        char buf[MAXBUF];
        char filename[MAXBUF];

        /* wait for an incoming connection */
        addrlen = sizeof(xferClient);

        /* use accept() to handle incoming connection requests        */
        /* and free up the original socket for other requests         */
        socket2 = accept(socket1, (struct sockaddr*)&xferClient, &addrlen);

        if (socket2 == -1)
        {
            fprintf(stderr, "Could not accept connection!\n");
            exit(1);
        }
```

`accept()` is a function that will *block* on a socket if there is no current connection. This is handy for servers, because otherwise the server stops as soon as it started, since there wouldn't be any connections on the socket. By blocking, the server waits for connections to come in on the socket.

If we have a connection, the first thing our client will do is send the name of the file it wants the server to retrieve. So, our first order of business is to read the filename sent by the client from the socket and store it in the buffer we set aside for it.

```
/* get the filename from the client over the socket */
i = 0;

if ((readCounter = read(socket2, filename + i, MAXBUF)) > 0)
{
    i += readCounter;
}

if (readCounter == -1)
{
    fprintf(stderr, "Could not read filename from socket!\n");
    close(socket2);
    continue;
}
```

We set `readCounter` to the number of bytes read from the socket. This is the length of the filename. We initially set up the filename variable to be quite large, because there's no way for us to know ahead of time which file the client will request. We want to make sure the filename we receive is only as large as it needs to be, so we will null-terminate the filename, making the buffer holding the name the right size so that we can use it later. If we don't get a filename, we'll close the socket and continue listening for other connections. If we do get a filename, we'll print a status message to the console so that we can see which files the clients are requesting. Then we'll open the file for reading, making sure to set the `O_RDONLY` flag to prevent any mishaps, such as overwriting a file or creating an empty file by mistake. The return from the `open()` call is a file descriptor.

```
filename[i+1] = '\0';

printf("Reading file %s\n", filename);

/* open the file for reading */
fd = open(filename, O_RDONLY);

if (fd == -1)
{
    fprintf(stderr, "Could not open file for reading!\n");
```

```
            close(socket2);
            continue;
        }

        /* reset the read counter */
        readCounter = 0;
```

By now we've hopefully gotten a handle to the file that our client wants. If we can't find or read the file for some reason, our server will close the socket and go back to waiting for another connection. Next, we reset the readCounter, because we're going to use it to count how many bytes we read from the disk while retrieving the file. We'll send the file to the client in chunks, using our MAXBUF constant as the size of each chunk. Because our files can be of varying sizes, we'll use nested loops to keep track of where we are in the transfer. As long as the number of bytes sent is smaller than the total file size, we'll know to keep sending.

```
        /* read the file, and send it to the client in chunks of size MAXBUF */
        while((readCounter = read(fd, buf, MAXBUF)) > 0)
        {
            writeCounter = 0;
            bufptr = buf;

            while (writeCounter < readCounter)
            {

                readCounter -= writeCounter;
                bufptr += writeCounter;
                writeCounter = write(socket2, bufptr, readCounter);

                if (writeCounter == -1)
                {
                    fprintf(stderr, "Could not write file to client!\n");
                    close(socket2);
                    continue;
                }
            }
        }
```

As before, if we run into any problems writing to the client, we close the socket and continue listening for other connections. Once our file has been sent, we'll clean up the file descriptor and close the second socket. Our server will loop back and check for more connections on socket1 until it receives a termination signal from the operating system.

```
            close(fd);
            close(socket2);

        }

    close (socket1);
    return 0;

}
```

Once the termination signal is received, we'll clean up by closing the original socket (socket1), and exiting.

The Client

Our file transfer client is slightly less complicated than our server. We begin with the standard #include directives and the same constant declarations that we used in the server, namely MAXBUF and SERVERPORT.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

  #define SERVERPORT    8888
  #define MAXBUF        1024
```

Next we do the standard setup and initialization, including the creation of a streaming socket and the population of a sockaddr_in structure for our server.

```
int main(int argc, char* argv[])
{
    int sockd;
    int counter;
    int fd;
    struct sockaddr_in xferServer;
    char buf[MAXBUF];
    int returnStatus;
    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s <ip address> <filename> [dest filename]\n",
                argv[0]);
        exit(1);
    }

    /* create a socket */
    sockd = socket(AF_INET, SOCK_STREAM, 0);

    if (sockd == -1)
    {
        fprintf(stderr, "Could not create socket!\n");
        exit(1);
    }

    /* set up the server information */
    xferServer.sin_family = AF_INET;
    xferServer.sin_addr.s_addr = inet_addr(argv[1]);
    xferServer.sin_port = htons(SERVERPORT);

    /* connect to the server */
    returnStatus = connect(sockd,
                           (struct sockaddr*)&xferServer,
                           sizeof(xferServer));

    if (returnStatus == -1)
    {
        fprintf(stderr, "Could not connect to server!\n");
        exit(1);
    }
```

Once we have a successful connection to our server, our first task is to send the name of the file we want the server to retrieve for us. This was passed to our client as a command-line argument.

```
    /* send the name of the file we want to the server */
    returnStatus = write(sockd, argv[2], strlen(argv[2])+1);

    if (returnStatus == -1)
    {
        fprintf(stderr, "Could not send filename to server!\n");
        exit(1);
    }
```

Once our filename is sent to the server, we don't need to send anything else, so we call the `shutdown()` function to set our socket to read-only.

```
    /* call shutdown to set our socket to read-only */
    shutdown(sockd, SHUT_WR);
```

Next, we set up our destination file. If a destination filename was an argument on our command line, we use that and open it for writing. If no destination filename was given, we use `stdout`, denoted by a file descriptor of 1.

```
    /* open up a handle to our destination file to receive the contents */
    /* from the server   */
    fd = open(argv[3], O_WRONLY | O_CREAT | O_APPEND);

    if (fd == -1)
    {
        fprintf(stderr, "Could not open destination file, using stdout.\n");
        fd = 1;
    }
```

As long as the server is sending us chunks of the file, read them from the socket and write them out to the system. In this example, we are writing the chunks to standard out (`stdout`). This could be changed easily to use the filename we sent to the server.

```
    /* read the file from the socket as long as there is data */
    while ((counter = read(sockd, buf, MAXBUF)) > 0)
    {
        /* send the contents to stdout */
        write(fd, buf, counter);
    }

    if (counter == -1)
    {
        fprintf(stderr, "Could not read file from socket!\n");
        exit(1);
    }

  close(sockd);
  return 0;

}
```

Once all of the chunks have been read from the server, clean up by closing the socket, and exit.

Example File Transfer Session

Let's run our file transfer client and server, and look at what the output would be. First, let's start the server.

```
[user@host projects]$ cc -o xferServer xferServer.c
[user@host projects]$ ./xferServer
```

At this point, the server is listening on all local IP addresses, on port 8888, which is the value of `SERVERPORT`. Feel free to change this to anything you like. Next, we compile and launch the client.

```
[user@host projects]$ cc -o xferClient xferClient.c
[user@host projects]$ ./xferClient 127.0.0.1 filename > new-filename
```

The client takes two arguments: the IP address of the server that will be sending the file and the filename to retrieve. Remember that our client uses standard out (`stdout`) to write the file it receives from the server. Because we're using `stdout`, we need to redirect `stdout` to a new location using our login shell's redirection operator, the ">" character. If we don't redirect `stdout` to a new location, one of two things will happen:

- The file contents that are retrieved will be displayed on the screen.
- The file we retrieve will actually overwrite the file we asked the server to read if we're operating in the same directory and our client and server are running on the same machine.

In our previous example, both programs are running on the same machine and in the same directory. We don't want our client's write process to compete with the server's read process, so we use the redirection operator to send the contents of the retrieved file to another filename altogether. Because our file might be a binary file, such as an image or executable instead of ASCII text, we also don't want the contents sent to the screen. Sending the contents of the file to the screen is fine if it's text, but not if it's a binary file.

Error Handling

In all of our examples so far, we've been fairly diligent about checking the results of our function calls. In general, systems return negative numbers when something goes wrong and positive numbers when something is all right. However, it is possible, especially in the case of sockets, for a function call to return an error that isn't a negative number. These error codes are defined in the `errno.h` file on your system. On most Linux systems, for example, this file can be found at `/usr/include/asm/errno.h`.

It's a good idea to check the return value after every single system call. You'll save yourself time and effort by checking the values and printing or logging a relevant message to a file or to the console screen. There are a number of built-in support features that you can use to do this. For example, in the code you've seen so far, we've been using the `fprintf()` function to print our messages to the standard error device, known as `stderr`. This device can be easily redirected using your shell's redirection operators so that the messages received by `stderr` go where you want them to go, such as to a log file or even to mail or an alphanumeric pager.

Another function that can be used for error logging is the `perror()` function. This function behaves more simply than the `fprintf()` function, because `perror()` always uses `stderr` for its output and doesn't let you do any formatting of the error message. With `fprintf()` and using `stderr` as the file descriptor, you can easily include formatted information, such as the values of any relevant variables that may have caused the problem. These values can be any of the data types supported by `fprintf()`.

Not only will you save time and effort by checking your error codes, but you'll also give your applications a solid foundation for security. It's well known that some of the primary exploits used by crackers are basic things like buffer overflows and unexpected payloads. If you take care in your programs to check the sizes of buffers and verify the content of variables, and take extra checks to ensure that the data you send or receive is the data you expect to send or receive, your applications will be less susceptible to those types of attacks. If you're expecting a date, then check to make sure the date is valid before doing anything with it. If you're setting up a 5KB buffer to store something, make sure your applications don't try to store 6KB of information there, or even 5KB + 1 byte.

One important thing to remember is that not all errors are failure errors, especially in the case of socket programming. Just because your function call returned an error doesn't mean your program can't continue, whether it's a server or a client. Remember that when making network connections over distributed networks, just about anything can happen. Because just about anything can happen, it's normal for some errors to occur, and your programs should determine which errors can be handled without stopping and which are serious failures.

For example, if you develop a server application, and the server can't do required operations like creating, binding to, or listening on a socket, then there's no reason to continue. If, however, the error returned is something expected such as a transmission error, your program should continue operations. In some cases, like UDP, your program might not even know a transmission error occurred, whereas with TCP, your program will know.

Probably the most important thing to know about error handling is that there is no "right way." The right way is the way that works for you and your application, since only you can set the design parameters. However, even though there is no "right way," it doesn't mean that you can ignore error handling. Ignoring errors will leave your applications unstable and insecure, not to mention difficult to maintain.

Summary

In this chapter, we covered building a couple of different types of socket applications. We built both a UDP server and a UDP client, and demonstrated the difference between using datagram sockets and streaming sockets. Then, using streaming sockets, we built a server and client capable of transferring any file that the server could access. This demonstrated how to send binary data instead of simple strings and stepped through a sample session. Finally, we discussed error checking and error handling, and explained that good error checking will not only save you time and effort when developing your applications, but also help make your applications more stable and more secure.