Beej's Guide to Network Programming

Using Internet Sockets

Version 1.5.4 (17-May-1998)
[http://www.ecst.csuchico.edu/~beej/guide/net]

Intro

Hey! Socket programming got you down? Is this stuff just a little too difficult to figure out from the `man` pages? You want to do cool Internet programming, but you don't have time to wade through a gob of `structs` trying to figure out if you have to call `bind()` before you `connect()`, etc., etc.

Well, guess what! I've already done this nasty business, and I'm dying to share the information with everyone! You've come to the right place. This document should give the average competent C programmer the edge s/he needs to get a grip on this networking noise.

Audience

This document has been written as a tutorial, not a reference. It is probably at its best when read by individuals who are just starting out with socket programming and are looking for a foothold. It is certainly not the *complete* guide to sockets programming, by any means.

Hopefully, though, it'll be just enough for those man pages to start making sense... :-)

Platform and Compiler

Most of the code contained within this document was compiled on a Linux PC using Gnu's `gcc` compiler. It was also found to compile on HPUX using `gcc`. Note that every code snippet was not individually tested.

Contents:

What is a socket?

You hear talk of "sockets" all the time, and perhaps you are wondering just what they are exactly. Well, they're this: a way to speak to other programs using standard Unix file descriptors.

What?

Ok--you may have heard some Unix hacker state, "Jeez, *everything* in Unix is a file!" What that person may have been talking about is the fact that when Unix programs do any sort of I/O, they do it by reading or writing to a file descriptor. A file descriptor is simply an integer associated with an open file. But (and here's the catch), that file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in Unix **is** a file! So when you want to communicate with another program over the Internet you're gonna do it through a file descriptor, you'd better believe it.

"Where do I get this file descriptor for network communication, Mr. Smarty-Pants?" is probably the last question on your mind right now, but I'm going to answer it anyway: You make a call to the `socket()` system routine. It returns the socket descriptor, and you communicate through it using the specialized `send()` and `recv()` ("`man send`", "`man recv`") socket calls.

"But, hey!" you might be exclaiming right about now. "If it's a file descriptor, why in the hell can't I just use the normal `read()` and `write()` calls to communicate through the socket?" The short answer is, "You can!" The longer answer is, "You can, but `send()` and `recv()` offer much greater control over your data transmission."

What next? How about this: there are all kinds of sockets. There are DARPA Internet addresses (Internet Sockets), path names on a local node (Unix Sockets), CCITT X.25 addresses (X.25 Sockets that you can safely ignore), and probably many others depending on which Unix flavor you run. This document deals only with the first: Internet Sockets.

Two Types of Internet Sockets

What's this? There are two types of Internet sockets? Yes. Well, no. I'm lying. There are more, but I didn't want to scare you. I'm only going to talk about two types here. Except for this sentence, where I'm going to tell you that "Raw Sockets" are also very powerful and you should look them up.

All right, already. What are the two types? One is "Stream Sockets"; the other is "Datagram Sockets", which may hereafter be referred to as "`SOCK_STREAM`" and "`SOCK_DGRAM`", respectively. Datagram sockets are sometimes called "connectionless sockets" (though they can be `connect()`'d if you really want. See `connect()`, below.

Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order "1, 2", they will arrive in the order "1, 2" at the opposite end. They will also be error free. Any errors you do encounter are figments of your own deranged mind, and are not to be discussed here.

What uses stream sockets? Well, you may have heard of the `telnet` application, yes? It uses stream sockets. All the characters you type need to arrive in the same order you type them, right? Also, WWW browsers use the HTTP protocol which uses stream sockets to get pages. Indeed, if you telnet to a WWW site on port 80, and type "GET pagename", it'll dump the HTML back at you!

How do stream sockets achieve this high level of data transmission quality? They use a protocol called "The Transmission Control Protocol", otherwise known as "TCP" (see RFC-793 for extremely detailed info on TCP.) TCP makes sure your data arrives sequentially and error-free. You may have heard "TCP" before as the better half of "TCP/IP" where "IP" stands for "Internet Protocol" (see RFC-791.) IP deals with Internet routing only.

Cool. What about Datagram sockets? Why are they called connectionless? What is the deal, here, anyway? Why are they unreliable? Well, here are some facts: if you send a datagram, it may arrive. It may arrive out of order. If it arrives, the data within the packet will be error-free.

Datagram sockets also use IP for routing, but they don't use TCP; they use the "User Datagram Protocol", or "UDP" (see RFC-768.)

Why are they connectionless? Well, basically, it's because you don't have to maintain an open connection as you do with stream sockets. You just build a packet, slap an IP header on it with destination information, and send it out. No connection needed. They are generally used for packet-by-packet transfers of information. Sample applications: tftp, bootp, etc.

"Enough!" you may scream. "How do these programs even work if datagrams might get lost?!" Well, my human friend, each has it's own protocol on top of UDP. For example, the tftp protocol says that for each packet that gets sent, the recipient has to send back a packet that says, "I got it!" (an "ACK" packet.) If the sender of the original packet gets no reply in, say, five seconds, he'll re-transmit the packet until he finally gets an ACK. This acknowledgment procedure is very important when implementing `SOCK_DGRAM` applications.

Low level Nonsense and Network Theory

Since I just mentioned layering of protocols, it's time to talk about how networks really work, and to show some examples of how `SOCK_DGRAM` packets are built. Practically, you can probably skip this section. It's good background, however.

[Encapsulated Protocols Image] Hey, kids, it's time to learn about **Data Encapsulation**! This is very very important. It's so important that you might just learn about it if you take the networks course here at Chico State ;-). Basically, it says this: a packet is born, the packet is wrapped ("encapsulated") in a header (and maybe footer) by the first protocol (say, the TFTP protocol), then the whole thing (TFTP header included) is encapsulated again by the next protocol (say, UDP), then again by the next (IP), then again by the final protocol on the hardware (physical) layer (say, Ethernet).

When another computer receives the packet, the hardware strips the Ethernet header, the kernel strips the IP and UDP headers, the TFTP program strips the TFTP header, and it finally has the data.

Now I can finally talk about the infamous **Layered Network Model**. This Network Model describes a system of network functionality that has many advantages over other models. For instance, you can write sockets programs that are exactly the same without caring how the data is physically transmitted (serial, thin Ethernet, AUI, whatever) because programs on lower levels deal with it for you. The actual network hardware and topology is transparent to the socket programmer.

Without any further ado, I'll present the layers of the full-blown model. Remember this for network class exams:

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

The Physical Layer is the hardware (serial, Ethernet, etc.). The Application Layer is just about as far from the physical layer as you can imagine--it's the place where users interact with the network.

Now, this model is so general you could probably use it as an automobile repair guide if you really wanted to. A layered model more consistent with Unix might be:

- Application Layer (*telnet, ftp, etc.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP and routing*)
- Network Access Layer (*was Network, Data Link, and Physical*)

At this point in time, you can probably see how these layers correspond to the encapsulation of the original data.

See how much work there is in building a simple packet? Jeez! And you have to type in the packet headers yourself using "`cat`"! Just kidding. All you have to do for stream sockets is `send()` the data out. All you have to do for datagram sockets is encapsulate the packet in the method of your choosing and `sendto()` it out. The kernel builds the Transport Layer and Internet Layer on for you and the hardware does the Network Access Layer. Ah, modern technology.

So ends our brief foray into network theory. Oh yes, I forgot to tell you everything I wanted to say about routing: nothing! That's right, I'm not going to talk about it at all. The router strips the packet to the IP header, consults its routing table, blah blah blah. Check out the IP RFC if you really really care. If you never learn about it, well, you'll live.

structs

Well, we're finally here. It's time to talk about programming. In this section, I'll cover various data types used by the sockets interface, since some of them are a real bitch to figure out.

First the easy one: a socket descriptor. A socket descriptor is the following type:

```
    int
```

Just a regular `int`.

Things get weird from here, so just read through and bear with me. Know this: there are two byte orderings: most significant byte (sometimes called an "octet") first, or least significant byte first. The former is called "Network Byte Order". Some machines store their numbers internally in Network Byte Order, some don't. When I say something has to be in NBO, you have to call a function (such as `htons()`) to change it from "Host Byte Order". If I don't say "NBO", then you must leave the value in Host Byte Order.

My First Struct(TM)--`struct sockaddr`. This structure holds socket address information for many types of sockets:

```
    struct sockaddr {
        unsigned short    sa_family;    /* address family, AF_xxx       */
        char              sa_data[14];  /* 14 bytes of protocol address */
    };
```

`sa_family` can be a variety of things, but it'll be "`AF_INET`" for everything we do in this document. `sa_data` contains a destination address and port number for the socket. This is rather unwieldy.

To deal with `struct sockaddr`, programmers created a parallel structure: `struct sockaddr_in` ("in" for "Internet".)

```
    struct sockaddr_in {
        short int          sin_family;  /* Address family               */
        unsigned short int sin_port;    /* Port number                  */
        struct in_addr     sin_addr;    /* Internet address             */
        unsigned char      sin_zero[8]; /* Same size as struct sockaddr */
    };
```

This structure makes it easy to reference elements of the socket address. Note that `sin_zero` (which is included to pad the structure to the length of a `struct sockaddr`) should be set to all zeros with the function `bzero()` or `memset()`. Also, and this is the **important** bit, a pointer to a `struct sockaddr_in` can be cast to a pointer to a `struct sockaddr` and vice-versa. So even though `socket()` wants a `struct sockaddr *`, you can still use a `struct sockaddr_in` and cast it at the last minute! Also, notice that `sin_family` corresponds to `sa_family` in a `struct sockaddr` and should be set to "`AF_INET`". Finally, the `sin_port` and `sin_addr` must be in **Network Byte Order**!

"But," you object, "how can the entire structure, `struct in_addr sin_addr`, be in Network Byte Order?" This question requires careful examination of the structure `struct in_addr`, one of the worst unions alive:

```
    /* Internet address (a structure for historical reasons) */
    struct in_addr {
        unsigned long s_addr;
    };
```

Well, it *used* to be a union, but now those days seem to be gone. Good riddance. So if you have declared "ina" to be of type struct sockaddr_in, then "ina.sin_addr.s_addr" references the 4 byte IP address (in Network Byte Order). Note that even if your system still uses the God-awful union for struct in_addr, you can still reference the 4 byte IP address in exactly the same way as I did above (this due to #defines.)

Convert the Natives!

We've now been lead right into the next section. There's been too much talk about this Network to Host Byte Order conversion--now is the time for action!

All righty. There are two types that you can convert: short (two bytes) and long (four bytes). These functions work for the unsigned variations as well. Say you want to convert a short from Host Byte Order to Network Byte Order. Start with "h" for "host", follow it with "to", then "n" for "network", and "s" for "short": h-to-n-s, or htons() (read: "Host to Network Short").

It's almost too easy...

You can use every combination if "n", "h", "s", and "l" you want, not counting the really stupid ones. For example, there is NOT a stolh() ("Short to Long Host") function--not at this party, anyway. But there are:

- htons()--"Host to Network Short"
- htonl()--"Host to Network Long"
- ntohs()--"Network to Host Short"
- ntohl()--"Network to Host Long"

Now, you may think you're wising up to this. You might think, "What do I do if I have to change byte order on a char?" Then you might think, "Uh, never mind." You might also think that since your 68000 machine already uses network byte order, you don't have to call htonl() on your IP addresses. You would be right, BUT if you try to port to a machine that has reverse network byte order, your program will fail. Be portable! This is a Unix world! Remember: put your bytes in Network Order before you put them on the network.

A final point: why do sin_addr and sin_port need to be in Network Byte Order in a struct sockaddr_in, but sin_family does not? The answer: sin_addr and sin_port get encapsulated in the packet at the IP and UDP layers, respectively. Thus, they must be in Network Byte Order. However, the sin_family field is only used by the kernel to determine what type of address the structure contains, so it must be in Host Byte Order. Also, since sin_family does **not** get sent out on the network, it can be in Host Byte Order.

IP Addresses and How to Deal With Them

Fortunately for you, there are a bunch of functions that allow you to manipulate IP addresses. No need to figure them out by hand and stuff them in a long with the << operator.

First, let's say you have a `struct sockaddr_in ina`, and you have an IP address "132.241.5.10" that you want to store into it. The function you want to use, `inet_addr()`, converts an IP address in numbers-and-dots notation into an unsigned long. The assignment can be made as follows:

```
    ina.sin_addr.s_addr = inet_addr("132.241.5.10");
```

Notice that `inet_addr()` returns the address in Network Byte Order already--you don't have to call `htonl()`. Swell!

Now, the above code snippet isn't very robust because there is no error checking. See, `inet_addr()` returns `-1` on error. Remember binary numbers? `(unsigned)-1` just happens to correspond to the IP address 255.255.255.255! That's the broadcast address! Wrongo. Remember to do your error checking properly.

All right, now you can convert string IP addresses to `long`s. What about the other way around? What if you have a `struct in_addr` and you want to print it in numbers-and-dots notation? In this case, you'll want to use the function `inet_ntoa()` ("ntoa" means "network to ascii") like this:

```
    printf("%s",inet_ntoa(ina.sin_addr));
```

That will print the IP address. Note that `inet_ntoa()` takes a `struct in_addr` as an argument, not a `long`. Also notice that it returns a pointer to a char. This points to a statically stored char array within `inet_ntoa()` so that each time you call `inet_ntoa()` it will overwrite the last IP address you asked for. For example:

```
    char *a1, *a2;
    .
    .
    a1 = inet_ntoa(ina1.sin_addr);  /* this is 198.92.129.1 */
    a2 = inet_ntoa(ina2.sin_addr);  /* this is 132.241.5.10 */
    printf("address 1: %s\n",a1);
    printf("address 2: %s\n",a2);
```

will print:

```
    address 1: 132.241.5.10
    address 2: 132.241.5.10
```

If you need to save the address, `strcpy()` it to your own character array.

That's all on this topic for now. Later, you'll learn to convert a string like "whitehouse.gov" into its corresponding IP address (see DNS, below.)

`socket()`--Get the File Descriptor!

I guess I can put it off no longer--I have to talk about the `socket()` system call. Here's the breakdown:

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

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

But what are these arguments? First, `domain` should be set to "`AF_INET`", just like in the `struct sockaddr_in` (above.) Next, the `type` argument tells the kernel what kind of socket this is: `SOCK_STREAM` or `SOCK_DGRAM`. Finally, just set `protocol` to "`0`". (Notes: there are many more `domains` than I've listed. There are many more `types` than I've listed. See the `socket()` man page. Also, there's a "better" way to get the `protocol`. See the `getprotobyname()` man page.)

`socket()` simply returns to you a socket descriptor that you can use in later system calls, or `-1` on error. The global variable `errno` is set to the error's value (see the `perror()` man page.)

`bind()`--What port am I on?

Once you have a socket, you might have to associate that socket with a port on your local machine. (This is commonly done if you're going to `listen()` for incoming connections on a specific port--MUDs do this when they tell you to "telnet to x.y.z port 6969".) If you're going to only be doing a `connect()`, this may be unnecessary. Read it anyway, just for kicks.

Here is the synopsis for the `bind()` system call:

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

`sockfd` is the socket file descriptor returned by `socket()`. `my_addr` is a pointer to a `struct sockaddr` that contains information about your address, namely, port and IP address. `addrlen` can be set to `sizeof(struct sockaddr)`.

Whew. That's a bit to absorb in one chunk. Let's have an example:

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

#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

    my_addr.sin_family = AF_INET;     /* host byte order */
    my_addr.sin_port = htons(MYPORT); /* short, network byte order */
    my_addr.sin_addr.s_addr = inet_addr("132.241.5.10");
    bzero(&(my_addr.sin_zero), 8);    /* zero the rest of the struct */

    /* don't forget your error checking for bind(): */
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    .
    .
    .
```

There are a few things to notice here. `my_addr.sin_port` is in Network Byte Order. So is `my_addr.sin_addr.s_addr`. Another thing to watch out for is that the header files might differ from system to system. To be sure, you should check your local man pages.

Lastly, on the topic of `bind()`, I should mention that some of the process of getting your own IP address and/or port can can be automated:

```
        my_addr.sin_port = 0; /* choose an unused port at random */
        my_addr.sin_addr.s_addr = INADDR_ANY;  /* use my IP address */
```

See, by setting `my_addr.sin_port` to zero, you are telling `bind()` to choose the port for you. Likewise, by setting `my_addr.sin_addr.s_addr` to `INADDR_ANY`, you are telling it to automatically fill in the IP address of the machine the process is running on.

If you are into noticing little things, you might have seen that I didn't put `INADDR_ANY` into Network Byte Order! Naughty me. However, I have inside info: `INADDR_ANY` is really zero! Zero still has zero on bits even if you rearrange the bytes. However, purists will point out that there could be a parallel dimension where `INADDR_ANY` is, say, 12 and that my code won't work there. That's ok with me:

```
        my_addr.sin_port = htons(0); /* choose an unused port at random */
        my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  /* use my IP address */
```

Now we're so portable you probably wouldn't believe it. I just wanted to point that out, since most of the code you come across won't bother running `INADDR_ANY` through `htonl()`.

`bind()` also returns `-1` on error and sets `errno` to the error's value.

Another thing to watch out for when calling `bind()`: don't go underboard with your port numbers. All ports below 1024 are RESERVED! You can have any port number above that, right up to 65535 (provided they aren't already being used by another program.)

One small extra final note about `bind()`: there are times when you won't absolutely have to call it. If you are `connect()`'ing to a remote machine and you don't care what your local port is (as is the case with `telnet`), you can simply call `connect()`, it'll check to see if the socket is unbound, and will `bind()` it to an unused local port.

connect()--Hey, you!

Let's just pretend for a few minutes that you're a telnet application. Your user commands you (just like in the movie *TRON*) to get a socket file descriptor. You comply and call `socket()`. Next, the user tells you to connect to "132.241.5.10" on port "23" (the standard telnet port.) Oh my God! What do you do now?

Lucky for you, program, you're now perusing the section on `connect()`--how to connect to a remote host. You read furiously onward, not wanting to disappoint your user...

The `connect()` call is as follows:

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

    int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

`sockfd` is our friendly neighborhood socket file descriptor, as returned by the `socket()` call, `serv_addr` is a `struct sockaddr` containing the destination port and IP address, and `addrlen` can be set to `sizeof(struct sockaddr)`.

Isn't this starting to make more sense? Let's have an example:

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

    #define DEST_IP   "132.241.5.10"
    #define DEST_PORT 23

    main()
    {
        int sockfd;
        struct sockaddr_in dest_addr;   /* will hold the destination addr */

        sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

        dest_addr.sin_family = AF_INET;        /* host byte order */
        dest_addr.sin_port = htons(DEST_PORT); /* short, network byte order */
        dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
        bzero(&(dest_addr.sin_zero), 8);       /* zero the rest of the struct */

        /* don't forget to error check the connect()! */
        connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
        .
        .
        .
```

Again, be sure to check the return value from `connect()`--it'll return `-1` on error and set the variable `errno`.

Also, notice that we didn't call `bind()`. Basically, we don't care about our local port number; we only care where we're going. The kernel will choose a local port for us, and the site we connect to will automatically get this information from us. No worries.

`listen()`--Will somebody please call me?

Ok, time for a change of pace. What if you don't want to connect to a remote host. Say, just for kicks, that you want to wait for incoming connections and handle them in some way. The process is two step: first you `listen()`, then you `accept()` (see below.)

The listen call is fairly simple, but requires a bit of explanation:

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

`sockfd` is the usual socket file descriptor from the `socket()` system call. `backlog` is the number of connections allowed on the incoming queue. What does that mean? Well, incoming connections are going to wait in this queue until you `accept()` them (see below) and this is the limit on how many can queue up. Most systems silently limit this number to about 20; you can probably get away with setting it to `5` or `10`.

Again, as per usual, `listen()` returns `-1` and sets `errno` on error.

Well, as you can probably imagine, we need to call `bind()` before we call `listen()` or the kernel will have us listening on a random port. Bleah! So if you're going to be listening for incoming connections, the sequence of system calls you'll make is:

```
    socket();
    bind();
    listen();
    /* accept() goes here */
```

I'll just leave that in the place of sample code, since it's fairly self-explanatory. (The code in the `accept()` section, below, is more complete.) The really tricky part of this whole sha-bang is the call to `accept()`.

`accept()`--"Thank you for calling port 3490."

Get ready--the `accept()` call is kinda weird! What's going to happen is this: someone far far away will try to `connect()` to your machine on a port that you are `listen()`'ing on. Their connection will be queued up waiting to be `accept()`'ed. You call `accept()` and you tell it to get the pending connection. It'll return to you a *brand new socket file descriptor* to use for this single connection! That's right, suddenly you have *two socket file descriptors* for the price of one! The original one is still listening on your port and the newly created one is finally ready to `send()` and `recv()`. We're there!

The call is as follows:

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

     int accept(int sockfd, void *addr, int *addrlen);
```

`sockfd` is the `listen()`'ing socket descriptor. Easy enough. `addr` will usually be a pointer to a local `struct sockaddr_in`. This is where the information about the incoming connection will go (and you can determine which host is calling you from which port). `addrlen` is a local integer variable that should be set to `sizeof(struct sockaddr_in)` before its address is passed to `accept()`. Accept will not put more than that many bytes into `addr`. If it puts fewer in, it'll change the value of `addrlen` to reflect that.

Guess what? `accept()` returns `-1` and sets `errno` if an error occurs. Betcha didn't figure that.

Like before, this is a bunch to absorb in one chunk, so here's a sample code fragment for your perusal:

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

    #define MYPORT 3490    /* the port users will be connecting to */

    #define BACKLOG 10     /* how many pending connections queue will hold */

    main()
    {
        int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
        struct sockaddr_in my_addr;    /* my address information */
        struct sockaddr_in their_addr; /* connector's address information */
        int sin_size;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

        my_addr.sin_family = AF_INET;         /* host byte order */
        my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
        my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
        bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

        /* don't forget your error checking for these calls: */
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
        listen(sockfd, BACKLOG);

        sin_size = sizeof(struct sockaddr_in);
        new_fd = accept(sockfd, &their_addr, &sin_size);
        .
        .
        .
```

Again, note that we will use the socket descriptor `new_fd` for all `send()` and `recv()` calls. If you're only getting one single connection ever, you can `close()` the original `sockfd` in order to prevent more incoming connections on the same port, if you so desire.

`send()` and `recv()`--Talk to me, baby!

These two functions are for communicating over stream sockets or connected datagram sockets. If you want to use regular unconnected datagram sockets, you'll need to see the section on `sendto()` and `recvfrom()`, below.

The `send()` call:

```
    int send(int sockfd, const void *msg, int len, int flags);
```

`sockfd` is the socket descriptor you want to send data to (whether it's the one returned by `socket()` or the one you got with `accept()`.) `msg` is a pointer to the data you want to send, and `len` is the length of that data in bytes. Just set `flags` to `0`. (See the `send()` man page for more information concerning flags.)

Some sample code might be:

```
    char *msg = "Beej was here!";
    int len, bytes_sent;
    .
    .
    len = strlen(msg);
    bytes_sent = send(sockfd, msg, len, 0);
    .
    .
    .
```

`send()` returns the number of bytes actually sent out--**this might be less than the number you told it to send!** See, sometimes you tell it to send a whole gob of data and it just can't handle it. It'll fire off as much of the data as it can, and trust you to send the rest later. Remember, if the value returned by `send()` doesn't match doesn't match the value in `len`, it's up to you to send the rest of the string. The good news is this: if the packet is small (less than 1K or so) it will *probably* manage to send the whole thing all in one go. Again, `-1` is returned on error, and `errno` is set to the error number.

The `recv()` call is similar in many respects:

```
    int recv(int sockfd, void *buf, int len, unsigned int flags);
```

`sockfd` is the socket descriptor to read from, `buf` is the buffer to read the information into, `len` is the maximum length of the buffer, and `flags` can again be set to `0`. (See the `recv()` man page for flag information.)

`recv()` returns the number of bytes actually read into the buffer, or `-1` on error (with `errno` set, accordingly.)

There, that was easy, wasn't it? You can now pass data back and forth on stream sockets! Whee! You're a Unix Network Programmer!

`sendto()` and `recvfrom()`--Talk to me, DGRAM-style

"This is all fine and dandy," I hear you saying, "but where does this leave me with unconnected datagram sockets?" No problemo, amigo. We have just the thing.

Since datagram sockets aren't connected to a remote host, guess which piece of information we need to give before we send a packet? That's right! The destination address! Here's the scoop:

```
    int sendto(int sockfd, const void *msg, int len, unsigned int flags,
               const struct sockaddr *to, int tolen);
```

As you can see, this call is basically the same as the call to `send()` with the addition of two other pieces of information. `to` is a pointer to a `struct sockaddr` (which you'll probably have as a `struct sockaddr_in` and cast it at the last minute) which contains the destination IP address and port. `tolen` can simply be set to `sizeof(struct sockaddr)`.

Just like with `send()`, `sendto()` returns the number of bytes actually sent (which, again, might be less than the number of bytes you told it to send!), or `-1` on error.

Equally similar are `recv()` and `recvfrom()`. The synopsis of `recvfrom()` is:

```
    int recvfrom(int sockfd, void *buf, int len, unsigned int flags
                 struct sockaddr *from, int *fromlen);
```

Again, this is just like `recv()` with the addition of a couple fields. `from` is a pointer to a local `struct sockaddr` that will be filled with the IP address and port of the originating machine. `fromlen` is a pointer to a local `int` that should be initialized to `sizeof(struct sockaddr)`. When the function returns, `fromlen` will contain the length of the address actually stored in `from`.

`recvfrom()` returns the number of bytes received, or `-1` on error (with `errno` set accordingly.)

Remember, if you `connect()` a datagram socket, you can then simply use `send()` and `recv()` for all your transactions. The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information for you.

`close()` and `shutdown()`--Get outta my face!

Whew! You've been `send()`'ing and `recv()`'ing data all day long, and you've had it. You're ready to close the connection on your socket descriptor. This is easy. You can just use the regular Unix file descriptor `close()` function:

```
    close(sockfd);
```

This will prevent any more reads and writes to the socket. Anyone attempting to read or write the socket on the remote end will receive an error.

Just in case you want a little more control over how the socket closes, you can use the `shutdown()` function. It allows you to cut off communication in a certain direction, or both ways (just like `close()` does.) Synopsis:

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

`sockfd` is the socket file descriptor you want to shutdown, and `how` is one of the following:

- `0` - Further receives are disallowed
- `1` - Further sends are disallowed
- `2` - Further sends and receives are disallowed (like `close()`)

`shutdown()` returns `0` on success, and `-1` on error (with `errno` set accordingly.)

If you deign to use `shutdown()` on unconnected datagram sockets, it will simply make the socket unavailable for further `send()` and `recv()` calls (remember that you can use these if you `connect()` your datagram socket.)

Nothing to it.

`getpeername()`--Who are you?

This function is so easy.

It's so easy, I almost didn't give it it's own section. But here it is anyway.

The function `getpeername()` will tell you who is at the other end of a connected stream socket. The synopsis:

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

    int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

`sockfd` is the descriptor of the connected stream socket, `addr` is a pointer to a `struct sockaddr` (or a `struct sockaddr_in`) that will hold the information about the other side of the connection, and `addrlen` is a pointer to an `int`, that should be initialized to `sizeof(struct sockaddr)`.

The function returns `-1` on error and sets `errno` accordingly.

Once you have their address, you can use `inet_ntoa()` or `gethostbyaddr()` to print or get more information. No, you can't get their login name. (Ok, ok. If the other computer is running an ident daemon, this is possible. This, however, is beyond the scope of this document. Check out RFC-1413 for more info.)

`gethostname()`--Who am I?

Even easier than `getpeername()` is the function `gethostname()`. It returns the name of the computer that your program is running on. The name can then be used by `gethostbyname()`, below, to determine the IP address of your local machine.

What could be more fun? I could think of a few things, but they don't pertain to socket programming. Anyway, here's the breakdown:

```
    #include <unistd.h>

    int gethostname(char *hostname, size_t size);
```

The arguments are simple: `hostname` is a pointer to an array of chars that will contain the hostname upon the function's return, and `size` is the length in bytes of the `hostname` array.

The function returns `0` on successful completion, and `-1` on error, setting `errno` as usual.

DNS--You say "whitehouse.gov", I say "198.137.240.100"

In case you don't know what DNS is, it stands for "Domain Name Service". In a nutshell, you tell it what the human-readable address is for a site, and it'll give you the IP address (so you can use it with `bind()`, `connect()`, `sendto()`, or whatever you need it for.) This way, when someone enters:

```
    $ telnet whitehouse.gov
```

`telnet` can find out that it needs to `connect()` to "198.137.240.100".

But how does it work? You'll be using the function `gethostbyname()`:

```
    #include <netdb.h>

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

As you see, it returns a pointer to a `struct hostent`, the layout of which is as follows:

```
    struct hostent {
        char    *h_name;
        char    **h_aliases;
        int     h_addrtype;
        int     h_length;
        char    **h_addr_list;
    };
    #define h_addr h_addr_list[0]
```

And here are the descriptions of the fields in the `struct hostent`:

- `h_name -` Official name of the host.
- `h_aliases -` A NULL-terminated array of alternate names for the host.
- `h_addrtype -` The type of address being returned; usually `AF_INET`.
- `h_length -` The length of the address in bytes.
- `h_addr_list -` A zero-terminated array of network addresses for the host. Host addresses are in Network Byte Order.
- `h_addr -` The first address in `h_addr_list`.

`gethostbyname()` returns a pointer to the filled `struct hostent`, or NULL on error. (But `errno` is **not** set--**`h_errno`** is set instead. See `herror()`, below.)

But how is it used? Sometimes (as we find from reading computer manuals), just spewing the information at the reader is not enough. This function is certainly easier to use than it looks.

Here's an example program:

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

    int main(int argc, char *argv[])
    {
        struct hostent *h;

        if (argc != 2) {  /* error check the command line */
            fprintf(stderr,"usage: getip address\n");
            exit(1);
        }

        if ((h=gethostbyname(argv[1])) == NULL) {  /* get the host info */
            herror("gethostbyname");
            exit(1);
        }

        printf("Host name  : %s\n", h->h_name);
        printf("IP Address : %s\n",inet_ntoa(*((struct in_addr *)h->h_addr)));

        return 0;
    }
```

With `gethostbyname()`, you can't use `perror()` to print error message (since `errno` is not used). Instead, call `herror()`.

It's pretty straightforward. You simply pass the string that contains the machine name ("whitehouse.gov") to `gethostbyname()`, and then grab the information out of the returned `struct hostent`.

The only possible weirdness might be in the printing of the IP address, above. `h->h_addr` is a `char *`, but `inet_ntoa()` wants a `struct in_addr` passed to it. So I cast `h->h_addr` to a `struct in_addr *`, then dereference it to get at the data.

Client-Server Background

It's a client-server world, baby. Just about everything on the network deals with client processes talking to server processes and vice-versa. Take `telnet`, for instance. When you connect to a remote host on port 23 with telnet (the client), a program on that host (called `telnetd`, the server) springs to life. It handles the incoming telnet connection, sets you up with a login prompt, etc.

Figure 2. The Client-Server Relationship.

The exchange of information between client and server is summarized in Figure 2.

Note that the client-server pair can speak `SOCK_STREAM`, `SOCK_DGRAM`, or anything else (as long as they're speaking the same thing.) Some good examples of client-server pairs are `telnet`/`telnetd`, `ftp`/`ftpd`, or `bootp`/`bootpd`. Every time you use `ftp`, there's a remote program, `ftpd`, that serves you.

Often, there will only be one server on a machine, and that server will handle multiple clients using `fork()`. The basic routine is: server will wait for a connection, `accept()` it, and `fork()` a child process to handle it. This is what our sample server does in the next section.

A Simple Stream Server

All this server does is send the string "`Hello, World!\n`" out over a stream connection. All you need to do to test this server is run it in one window, and telnet to it from another with:

```
$ telnet remotehostname 3490
```

where `remotehostname` is the name of the machine you're running it on.

The server code: (Note: a trailing backslash on a line means that the line is continued on the next.)

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define MYPORT 3490    /* the port users will be connecting to */

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         /* host byte order */
    my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
    bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) \
                                                              == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    while(1) {  /* main accept() loop */
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, \
                                                      &sin_size)) == -1) {
            perror("accept");
            continue;
        }
        printf("server: got connection from %s\n", \
                                           inet_ntoa(their_addr.sin_addr));
        if (!fork()) { /* this is the child process */
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  /* parent doesn't need this */

        while(waitpid(-1,NULL,WNOHANG) > 0); /* clean up child processes */
    }
}
```

In case you're curious, I have the code in one big `main()` function for (I feel) syntactic clarity. Feel free to split it into smaller functions if it makes you feel better.

You can also get the string from this server by using the client listed in the next section.

A Simple Stream Client

This guy's even easier than the server. All this client does is connect to the host you specify on the command line, port 3490. It gets the string that the server sends.

The client source:

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

    #define PORT 3490    /* the port client will be connecting to */

    #define MAXDATASIZE 100 /* max number of bytes we can get at once */

    int main(int argc, char *argv[])
    {
        int sockfd, numbytes;
        char buf[MAXDATASIZE];
        struct hostent *he;
        struct sockaddr_in their_addr; /* connector's address information */

        if (argc != 2) {
            fprintf(stderr,"usage: client hostname\n");
            exit(1);
        }

        if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
            herror("gethostbyname");
            exit(1);
        }

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }

        their_addr.sin_family = AF_INET;      /* host byte order */
        their_addr.sin_port = htons(PORT);    /* short, network byte order */
        their_addr.sin_addr = *((struct in_addr *)he->h_addr);
        bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

        if (connect(sockfd, (struct sockaddr *)&their_addr, \
                                              sizeof(struct sockaddr)) == -1) {
            perror("connect");
            exit(1);
        }

        if ((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {
            perror("recv");
            exit(1);
        }

        buf[numbytes] = '\0';

        printf("Received: %s",buf);

        close(sockfd);

        return 0;
    }
```

Notice that if you don't run the server before you run the client, `connect()` returns "Connection refused". Very useful.

Datagram Sockets

I really don't have that much to talk about here, so I'll just present a couple of sample programs: `talker.c` and `listener.c`.

`listener` sits on a machine waiting for an incoming packet on port 4950. `talker` sends a packet to that port, on the specified machine, that contains whatever the user enters on the command line.

Here is the source for `listener.c`:

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>
    #include <string.h>
    #include <sys/types.h>
    #include <netinet/in.h>
    #include <sys/socket.h>
    #include <sys/wait.h>

    #define MYPORT 4950    /* the port users will be connecting to */

    #define MAXBUFLEN 100

    main()
    {
        int sockfd;
        struct sockaddr_in my_addr;    /* my address information */
        struct sockaddr_in their_addr; /* connector's address information */
        int addr_len, numbytes;
        char buf[MAXBUFLEN];
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }

        my_addr.sin_family = AF_INET;         /* host byte order */
        my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
        my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
        bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) \
                                                                       == -1) {
            perror("bind");
            exit(1);
        }

        addr_len = sizeof(struct sockaddr);
        if ((numbytes=recvfrom(sockfd, buf, MAXBUFLEN, 0, \
                           (struct sockaddr *)&their_addr, &addr_len)) == -1) {
            perror("recvfrom");
            exit(1);
        }

        printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
        printf("packet is %d bytes long\n",numbytes);
        buf[numbytes] = '\0';
        printf("packet contains \"%s\"\n",buf);

        close(sockfd);
    }
```

Notice that in our call to `socket()` we're finally using `SOCK_DGRAM`. Also, note that there's no need to `listen()` or `accept()`. This is one of the perks of using unconnected datagram sockets!

Next comes the source for `talker.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define MYPORT 4950    /* the port users will be connecting to */

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; /* connector's address information */
    struct hostent *he;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
        herror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;      /* host byte order */
    their_addr.sin_port = htons(MYPORT);  /* short, network byte order */
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0, \
         (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n",numbytes,inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

And that's all there is to it! Run `listener` on some machine, then run `talker` on another. Watch them communicate! Fun G-rated excitement for the entire nuclear family!

Except for one more tiny detail that I've mentioned many times in the past: connected datagram sockets. I need to talk about this here, since we're in the datagram section of the document. Let's say that `talker` calls `connect()` and specifies the `listener`'s address. From that point on, `talker` may only sent to and receive from the address specified by `connect()`. For this reason, you don't have to use `sendto()` and `recvfrom()`; you can simply use `send()` and `recv()`.

Blocking

Blocking. You've heard about it--now what the hell is it? In a nutshell, "block" is techie jargon for "sleep". You probably noticed that when you run `listener`, above, it just sits there until a packet arrives. What happened is that it called `recvfrom()`, there was no data, and so `recvfrom()` is said to "block" (that is, sleep there) until some data arrives.

Lots of functions block. `accept()` blocks. All the `recv*()` functions block. The reason they can do this is because they're allowed to. When you first create the socket descriptor with `socket()`, the kernel sets it to blocking. If you don't want a socket to be blocking, you have to make a call to `fcntl()`:

```
    #include <unistd.h>
    #include <fcntl.h>
    .
    .
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    fcntl(sockfd, F_SETFL, O_NONBLOCK);
    .
    .
```

By setting a socket to non-blocking, you can effectively "poll" the socket for information. If you try to read from a non-blocking socket and there's no data there, it's not allowed to block--it will return `-1` and `errno` will be set to `EWOULDBLOCK`.

Generally speaking, however, this type of polling is a bad idea. If you put your program in a busy-wait looking for data on the socket, you'll suck up CPU time like it was going out of style. A more elegant solution for checking to see if there's data waiting to be read comes in the following section on select().

select()--Synchronous I/O Multiplexing

This function is somewhat strange, but it's very useful. Take the following situation: you are a server and you want to listen for incoming connections as well as keep reading from the connections you already have.

No problem, you say, just an accept() and a couple of recv()s. Not so fast, buster! What if you're blocking on an accept() call? How are you going to recv() data at the same time? "Use non-blocking sockets!" No way! You don't want to be a CPU hog. What, then?

select() gives you the power to monitor several sockets at the same time. It'll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions, if you really want to know that.

Without any further ado, I'll offer the synopsis of select():

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

        int select(int numfds, fd_set *readfds, fd_set *writefds,
                   fd_set *exceptfds, struct timeval *timeout);
```

The function monitors "sets" of file descriptors; in particular readfds, writefds, and exceptfds. If you want to see if you can read from standard input and some socket descriptor, sockfd, just add the file descriptors 0 and sockfd to the set readfds. The parameter numfds should be set to the values of the highest file descriptor plus one. In this example, it should be set to sockfd+1, since it is assuredly higher than standard input (0).

When select() returns, readfds will be modified to reflect which of the file descriptors you selected is ready for reading. You can test them with the macro FD_ISSET(), below.

Before progressing much further, I'll talk about how to manipulate these sets. Each set is of the type fd_set. The following macros operate on this type:

- FD_ZERO(fd_set *set) - clears a file descriptor set
- FD_SET(int fd, fd_set *set) - adds fd to the set
- FD_CLR(int fd, fd_set *set) - removes fd from the set
- FD_ISSET(int fd, fd_set *set) - tests to see if fd is in the set

Finally, what is this weirded out struct timeval? Well, sometimes you don't want to wait forever for someone to send you some data. Maybe every 96 seconds you want to print "Still Going..." to the terminal even though nothing has happened. This time structure allows you to specify a timeout period. If the time is exceeded and select() still hasn't found any ready file descriptors, it'll return so you can continue processing.

The struct timeval has the follow fields:

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

Just set tv_sec to the number of seconds to wait, and set tv_usec to the number of microseconds to wait. Yes, that's *micro*seconds, not milliseconds. There are 1,000 microseconds in a millisecond, and 1,000 milliseconds in a second. Thus, there are 1,000,000 microseconds in a second. Why is it "usec"? The "u" is supposed to look like the Greek letter Mu that we use for "micro". Also, when the function returns, timeout *might* be updated to show the time still remaining. This depends on what flavor of Unix you're running.

Yay! We have a microsecond resolution timer! Well, don't count on it. Standard Unix timeslice is 100 milliseconds, so you'll probably have to wait at least that long, no matter how small you set your struct timeval.

Other things of interest: If you set the fields in your struct timeval to 0, select() will timeout immediately, effectively polling all the file descriptors in your sets. If you set the parameter timeout to NULL, it will never timeout, and will wait until the first file descriptor is ready. Finally, if you don't care about waiting for a certain set, you can just set it to NULL in the call to select().

The following code snippet waits 2.5 seconds for something to appear on standard input:

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

#define STDIN 0  /* file descriptor for standard input */

main()
{
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    /* don't care about writefds and exceptfds: */
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");
}
```

If you're on a line buffered terminal, the key you hit should be RETURN or it will time out anyway.

Now, some of you might think this is a great way to wait for data on a datagram socket--and you are right: it *might* be. Some Unices can use select in this manner, and some can't. You should see what your local man page says on the matter if you want to attempt it.

One final note of interest about select(): if you have a socket that is listen()'ing, you can check to see if there is a new connection by putting that socket's file descriptor in the readfds set.

And that, my friends, is a quick overview of the almighty select() function.

More References

You've come this far, and now you're screaming for more! Where else can you go to learn more about all this stuff?

Try the following man pages, for starters:

- `socket()`
- `bind()`
- `connect()`
- `listen()`
- `accept()`
- `send()`
- `recv()`
- `sendto()`
- `recvfrom()`
- `close()`
- `shutdown()`
- `getpeername()`
- `getsockname()`
- `gethostbyname()`
- `gethostbyaddr()`
- `getprotobyname()`
- `fcntl()`
- `select()`
- `perror()`

Also, look up the following books:

> **Internetworking with TCP/IP, volumes I-III** by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. Second edition ISBNs: 0-13-468505-9, 0-13-472242-6, 0-13-474222-2. There is a third edition of this set which covers IPv6 and IP over ATM.
>
> **Using C on the UNIX System** by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0-937175-23-4.
>
> **TCP/IP Network Administration** by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 0-937175-82-X.
>
> **TCP/IP Illustrated, volumes 1-3** by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs: 0-201-63346-9, 0-201-63354-X, 0-201-63495-3.
>
> **Unix Network Programming** by W. Richard Stevens. Published by Prentice Hall. ISBN 0-13-949876-1.

On the web:

> **BSD Sockets: A Quick And Dirty Primer**
> (http://www.cs.umn.edu/~bentlema/unix/--has other great Unix system programming info, too!)
>
> **Client-Server Computing**
> (http://pandonia.canberra.edu.au/ClientServer/socket.html)
>
> **Intro to TCP/IP** (gopher)

(gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/)

Internet Protocol Frequently Asked Questions (France)
(http://web.cnam.fr/Network/TCP-IP/)

The Unix Socket FAQ
(http://www.ibrado.com/sock-faq/)

RFCs--the real dirt:

RFC-768 -- The User Datagram Protocol (UDP)
(ftp://nic.ddn.mil/rfc/rfc768.txt)

RFC-791 -- The Internet Protocol (IP)
(ftp://nic.ddn.mil/rfc/rfc791.txt)

RFC-793 -- The Transmission Control Protocol (TCP)
(ftp://nic.ddn.mil/rfc/rfc793.txt)

RFC-854 -- The Telnet Protocol
(ftp://nic.ddn.mil/rfc/rfc854.txt)

RFC-951 -- The Bootstrap Protocol (BOOTP)
(ftp://nic.ddn.mil/rfc/rfc951.txt)

RFC-1350 -- The Trivial File Transfer Protocol (TFTP)
(ftp://nic.ddn.mil/rfc/rfc1350.txt)

Disclaimer and Call for Help

Well, that's the lot of it. Hopefully at least some of the information contained within this document has been remotely accurate and I sincerely hope there aren't any glaring errors. Well, sure, there always are.

So, if there are, that's tough for you. I'm sorry if any inaccuracies contained herein have caused you any grief, but you just can't hold me accountable. See, I don't stand behind a single word of this document, legally speaking. This is my warning to you: the whole thing could be a load of crap.

But it's probably not. After all, I've spent many many hours messing with this stuff, and implemented several TCP/IP network utilities for Windows (including Telnet) as summer work. I'm not the sockets god; I'm just some guy.

By the way, if anyone has any constructive (or destructive) criticism about this document, please send mail to beej@ecst.csuchico.edu and I'll try to make an effort to set the record straight.

In case you're wondering why I did this, well, I did it for the money. Hah! No, really, I did it because a lot of people have asked me socket-related questions and when I tell them I've been thinking about putting together a socket page, they say, "cool!" Besides, I feel that all this hard-earned knowledge is going to waste if I can't share it with others. WWW just happens to

be the perfect vehicle. I encourage others to provide similar information whenever possible.

Enough of this--back to coding! ;-)
