Beej's Guide to Network Programming

Using Internet Sockets

Brian "Beej" Hall

beej@piratehaven.org

Revision History

Revision Version 1.0.0 August, 1995	Revised by: beej
Initial version.	
Revision Version 1.5.5 January 13, 1999	Revised by: beej
Latest HTML version.	
Revision Version 2.0.0 March 6, 2001	Revised by: beej
Converted to DocBook XML, corrections, additions.	
Revision Version 2.3.1 October 8, 2001	Revised by: beej
Fixed typos, syntax error in client.c, added some stuff to the Q&A section.	

Table of Contents

1. 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 `struct`s 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.

1.1. 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... `:-)`

1.2. Platform and Compiler

The code contained within this document was compiled on a Linux PC using Gnu's **gcc** compiler. It should, however, build on just about any platform that uses **gcc**. Naturally, this doesn't apply if you're programming for Windows–see the section on Windows programming, below.

1.3. Official Homepage

This official location of this document is at California State University, Chico, at `http://www.ecst.csuchico.edu/~beej/guide/net/`[1].

1.4. Note for Solaris/SunOS Programmers

When compiling for Solaris or SunOS, you need to specify some extra command-line switches for linking in the proper libraries. In order to do this, simply add "`-lnsl -lsocket -lresolv`" to the end of the compile command, like so:

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

If you still get errors, you could try further adding a "`-lxnet`" to the end of that command line. I don't know what that does, exactly, but some people seem to need it.

Another place that you might find problems is in the call to `setsockopt()`. The prototype differs from that on my Linux box, so instead of:

```
int yes=1;
```

enter this:

```
char yes='1';
```

As I don't have a Sun box, I haven't tested any of the above information–it's just what people have told me through email.

1.5. Note for Windows Programmers

I have a particular dislike for Windows, and encourage you to try Linux, BSD, or Unix instead. That being said, you can still use this stuff under Windows.

First, ignore pretty much all of the system header files I mention in here. All you need to include is:

```
#include <winsock.h>
```

Wait! You also have to make a call to `WSAStartup()` before doing anything else with the sockets library. The code to do that looks something like this:

```
#include <winsock.h>

{
    WSADATA wsaData;   // if this doesn't work
    //WSAData wsaData; // then try this instead

    if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0) {
        fprintf(stderr, "WSAStartup failed.\n");
        exit(1);
    }
```

You also have to tell your compiler to link in the Winsock library, usually called `wsock32.lib` or `winsock32.lib` or somesuch. Under VC++, this can be done through the `Project` menu, under `Settings...`. Click the `Link` tab, and look for the box titled "Object/library modules". Add "wsock32.lib" to that list.

Or so I hear.

Finally, you need to call `WSACleanup()` when you're all through with the sockets library. See your online help for details.

Once you do that, the rest of the examples in this tutorial should generally apply, with a few exceptions. For one thing, you can't use `close()` to close a socket–you need to use `closesocket()`, instead. Also, `select()` only works with socket descriptors, not file descriptors (like `0` for `stdin`).

There is also a socket class that you can use, `CSocket`. Check your compilers help pages for more information.

To get more information about Winsock, read the Winsock FAQ[2] and go from there.

Finally, I hear that Windows has no `fork()` system call which is, unfortunately, used in some of my examples. Maybe you have to link in a POSIX library or something to get it to work, or you can use `CreateProcess()` instead. `fork()` takes no arguments, and `CreateProcess()` takes about 48 billion arguments. If you're not up to that, the `CreateThread()` is a little easier to digest...unfortunately a discussion about multithreading is beyond the scope of this document. I can only talk about so much, you know!

1.6. Email Policy

I’m generally available to help out with email questions so feel free to write in, but I can’t guarantee a response. I lead a pretty busy life and there are times when I just can’t answer a question you have. When that’s the case, I usually just delete the message. It’s nothing personal; I just won’t ever have the time to give the detailed answer you require.

As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you’re getting, and anything else you think might help me troubleshoot), you’re much more likely to get a response. For more pointers, read ESR’s document, How To Ask Questions The Smart Way[3].

If you don’t get a response, hack on it some more, try to find the answer, and if it’s still elusive, then write me again with the information you’ve found and hopefully it will be enough for me to help out.

Now that I’ve badgered you about how to write and not write me, I’d just like to let you know that I *fully* appreciate all the praise the guide has received over the years. It’s a real morale boost, and it gladdens me to hear that it is being used for good! `:-)` Thank you!

1.7. Mirroring

You are more than welcome to mirror this site, whether publically or privately. If you publically mirror the site and want me to link to it from the main page, drop me a line at `<beej@piratehaven.org>`.

1.8. Note for Translators

If you want to translate the guide into another language, write me at `<beej@piratehaven.org>` and I’ll link to your translation from the main page.

Feel free to add your name and email address to the translation.

Sorry, but due to space constraints, I cannot host the translations myself.

1.9. Copyright and Distribution

Educators are especially encouraged to recommend or supply copies of this guide to their students.

2. 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**[4], **man recv**[5]) socket calls.

"But, hey!" you might be exclaiming right about now. "If it's a file descriptor, why in the name of Neptune 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.

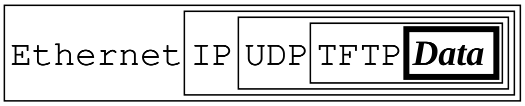
2.1. Two Types of Internet Sockets

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.

Figure 1. Data Encapsulation.



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, web browsers use the HTTP protocol which uses stream sockets to get pages. Indeed, if you telnet to a web site on port 80, and type "`GET /`", 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[6] 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[7].) IP deals primarily with Internet routing and is not generally responsible for data integrity.

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[8].)

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.

2.2. 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.

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 rarely a 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 (*Ethernet, ATM, or whatever*)

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[9] if you really really care. If you never learn about it, well, you'll live.

3. `struct`s and Data Handling

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 bear 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 Network Byte Order, you have to call a function (such as `htons()`) to change it from "Host Byte Order". If I don't say "Network Byte Order", then you must leave the value in Host Byte Order.

(For the curious, "Network Byte Order" is also know as "Big-Endian Byte Order".)

My First Struct™–`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 since you don't want to tediously pack the address in the *`sa_data`* by hand.

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 `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; // that's a 32-bit long, or 4 bytes
};
```

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 `#define`s.)

3.1. 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! (As much as Bill Gates would like to think otherwise.) Remember: put your bytes in Network Byte 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.

3.2. 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 "`10.12.110.57`" 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("10.12.110.57");
```

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.

Actually, there's a cleaner interface you can use instead of `inet_addr()`: it's called `inet_aton()` ("aton" means "ascii to network"):

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

int inet_aton(const char *cp, struct in_addr *inp);
```

And here's a sample usage, while packing a `struct sockaddr_in` (this example will make more sense to you when you get to the sections on `bind()` and `connect()`.)

```
struct sockaddr_in my_addr;

my_addr.sin_family = AF_INET;          // host byte order
my_addr.sin_port = htons(MYPORT);      // short, network byte order
inet_aton("10.12.110.57", &(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
```

`inet_aton()`, *unlike practically every other socket-related function*, returns non-zero on success, and zero on failure. And the address is passed back in *`inp`*.

Unfortunately, not all platforms implement `inet_aton()` so, although its use is preferred, the older more common `inet_addr()` is used in this guide.

All right, now you can convert string IP addresses to their binary representations. 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 192.168.4.14
a2 = inet_ntoa(ina2.sin_addr);  // this is 10.12.110.57
printf("address 1: %s\n",a1);
printf("address 2: %s\n",a2);
```

will print:

```
address 1: 10.12.110.57
address 2: 10.12.110.57
```

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.)

4. System Calls or Bust

This is the section where we get into the system calls that allow you to access the network functionality of a Unix box. When you call one of these functions, the kernel takes over and does all the work for you automagically.

The place most people get stuck around here is what order to call these things in. In that, the **man** pages are no use, as you've probably discovered. Well, to help with that dreadful situation, I've tried to lay out the system calls in the following sections in *exactly* (approximately) the same order that you'll need to call them in your programs.

That, coupled with a few pieces of sample code here and there, some milk and cookies (which I fear you will have to supply yourself), and some raw guts and courage, and you'll be beaming data around the Internet like the Son of Jon Postel!

4.1. `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" to have socket() choose the correct protocol based on the *type*. (Notes: there are many more *domain*s than I've listed. There are many more *type*s 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.)

In some documentation, you'll see mention of a mystical "PF_INET". This is a weird etherial beast that is rarely seen in nature, but I might as well clarify it a bit here. Once a long time ago, it was thought that maybe a address family (what the "AF" in "AF_INET" stands for) might support several protocols that were referenced by their protocol family (what the "PF" in "PF_INET" stands for). That didn't happen. Oh well. So the correct thing to do is to use AF_INET in your struct sockaddr_in and PF_INET in your call to socket(). But practically speaking, you can use AF_INET everywhere. And, since that's what W. Richard Stevens does in his book, that's what I'll do here.

Fine, fine, fine, but what good is this socket? The answer is that it's really no good by itself, and you need to read on and make more system calls for it to make any sense.

4.2. 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".) The port number is used by the kernel to match an incoming packet to a certain process's socket descriptor. 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>
#include <netinet/in.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("10.12.110.57");
    memset(&(my_addr.sin_zero), '\0', 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 (unless you're the superuser)! You can have any port number above that, right up to 65535 (provided they aren't already being used by another program.)

Sometimes, you might notice, you try to rerun a server and `bind()` fails, claiming "Address already in use." What does that mean? Well, a bit a of socket that was connected is still hanging around in the kernel, and it's hogging the port. You can either wait for it to clear (a minute or so), or add code to your program allowing it to reuse the port, like this:

```
    int yes=1;
//char yes='1'; // Solaris people use this
```

```
// lose the pesky "Address already in use" error message
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
}
```

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** where you only care about the remote port), you can simply call `connect()`, it'll check to see if the socket is unbound, and will `bind()` it to an unused local port if necessary.

4.3. `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 `"10.12.110.57"` on port `"23"` (the standard telnet port.) Yow! What do you do now?

Lucky for you, program, you're now perusing the section on `connect()`–how to connect to a remote host. So read furiously onward! No time to lose!

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>
#include <netinet/in.h>

#define DEST_IP   "10.12.110.57"
#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);
memset(&(dest_addr.sin_zero), '\0', 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 remote port). The kernel will choose a local port for us, and the site we connect to will automatically get this information from us. No worries.

4.4. `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()`.

4.5. `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 with it 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>
#include <netinet/in.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
    memset(&(my_addr.sin_zero), '\0', 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, (struct sockaddr *)&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 listening *sockfd* in order to prevent more incoming connections on the same port, if you so desire.

4.6. 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 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.)

Wait! recv() can return 0. This can mean only one thing: the remote side has closed the connection on you! A return value of 0 is recv()'s way of letting you know this has occurred.

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!

4.7. `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.

4.8. `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.)

It's important to note that `shutdown()` doesn't actually close the file descriptor–it just changes its usability. To free a socket descriptor, you need to use `close()`.

Nothing to it.

4.9. `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[10] for more info.)

4.10. `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.

4.11. DNS–You say "whitehouse.gov", I say "198.137.240.92"

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 "198.137.240.92".

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[11]:

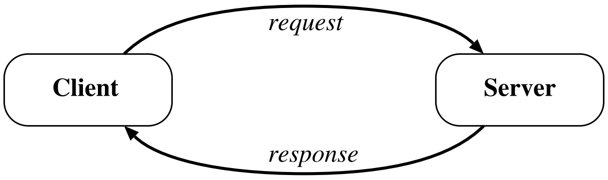
```
/*
** getip.c - a hostname lookup demo
*/

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

Figure 2. Client-Server Interaction.



5. 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.

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.

5.1. 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[12]: (Note: a trailing backslash on a line means that the line is continued on the next.)

```
/*
** server.c - a stream socket server demo
*/

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

```
#include <sys/wait.h>
#include <signal.h>

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

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

void sigchld_handler(int s)
{
    while(wait(NULL) > 0);
}

int main(void)
{
    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;
    struct sigaction sa;
    int yes=1;

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

    if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
        perror("setsockopt");
        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; // automatically fill with my IP
    memset(&(my_addr.sin_zero), '\0', 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);
    }

    sa.sa_handler = sigchld_handler; // reap all dead processes
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        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
            close(sockfd); // child doesn't need the listener
            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
    }

    return 0;
}
```

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.

(Also, this whole `sigaction()` thing might be new to you–that's ok. The code that's there is responsible for reaping zombie processes that appear as the `fork()`ed child processes exit. If you make lots of zombies and don't reap them, your system administrator will become agitated.)

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

5.2. 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[13]:

```
/*
** client.c - a stream socket client demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.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
        perror("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);
    memset(&(their_addr.sin_zero), '\0', 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-1, 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.

5.3. 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`[14]:

```
/*
** listener.c - a datagram sockets "server" demo
*/

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

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

#define MAXBUFLEN 100

int main(void)
{
    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; // automatically fill with my IP
    memset(&(my_addr.sin_zero), '\0', 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-1, 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);

    return 0;
}
```

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`[15]:

```
/*
** talker.c - a datagram "client" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.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
        perror("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);
    memset(&(their_addr.sin_zero), '\0', 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()`.

6. Slightly Advanced Techniques

These aren't *really* advanced, but they're getting out of the more basic levels we've already covered. In fact, if you've gotten this far, you should consider yourself fairly accomplished in the basics of Unix network programming! Congratulations!

So here we go into the brave new world of some of the more esoteric things you might want to learn about sockets. Have at it!

6.1. Blocking

Blocking. You've heard about it–now what the heck 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()`.

6.2. `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 which 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 μ (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 around 100 milliseconds, so you might have to wait 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[16] waits 2.5 seconds for something to appear on standard input:

```
/*
** select.c - a select() demo
*/

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

#define STDIN 0  // file descriptor for standard input

int main(void)
{
    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");

    return 0;
}
```

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.

Some Unices update the time in your `struct timeval` to reflect the amount of time still remaining before a timeout. But others do not. Don't rely on that occurring if you want to be portable. (Use `gettimeofday()` if you need to track time elapsed. It's a bummer, I know, but that's the way it is.)

What happens if a socket in the read set closes the connection? Well, in that case, `select()` returns with that socket descriptor set as "ready to read". When you actually do `recv()` from it, `recv()` will return `0`. That's how you know the client has closed the connection.

One more 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.

But, by popular demand, here is an in-depth example. Unfortunately, the difference between the dirt-simple example, above, and this one here is significant. But have a look, then read the description that follows it.

This program[17] acts like a simple multi-user chat server. Start it running in one window, then **telnet** to it ("**telnet hostname 9034**") from multiple other windows. When you type something in one **telnet** session, it should appear in all the others.

```
/*
** selectserver.c - a cheezy multiperson chat server
*/

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

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

#define PORT 9034   // port we're listening on

int main(void)
{
    fd_set master;   // master file descriptor list
    fd_set read_fds; // temp file descriptor list for select()
    struct sockaddr_in myaddr;     // server address
    struct sockaddr_in remoteaddr; // client address
    int fdmax;        // maximum file descriptor number
    int listener;     // listening socket descriptor
    int newfd;        // newly accept()ed socket descriptor
    char buf[256];    // buffer for client data
    int nbytes;
    int yes=1;        // for setsockopt() SO_REUSEADDR, below
    int addrlen;
    int i, j;

    FD_ZERO(&master);    // clear the master and temp sets
    FD_ZERO(&read_fds);

    // get the listener
    if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    // lose the pesky "address already in use" error message
    if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
                                                        sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

    // bind
    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = INADDR_ANY;
    myaddr.sin_port = htons(PORT);
    memset(&(myaddr.sin_zero), '\0', 8);
    if (bind(listener, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1) {
        perror("bind");
        exit(1);
    }

    // listen
    if (listen(listener, 10) == -1) {
        perror("listen");
        exit(1);
    }

    // add the listener to the master set
    FD_SET(listener, &master);
```

```
// keep track of the biggest file descriptor
fdmax = listener; // so far, it's this one

// main loop
for(;;) {
    read_fds = master; // copy it
    if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
        perror("select");
        exit(1);
    }

    // run through the existing connections looking for data to read
    for(i = 0; i <= fdmax; i++) {
        if (FD_ISSET(i, &read_fds)) { // we got one!!
            if (i == listener) {
                // handle new connections
                addrlen = sizeof(remoteaddr);
                if ((newfd = accept(listener, (struct sockaddr *)&remoteaddr,
                                                         &addrlen)) == -1) {
                    perror("accept");
                } else {
                    FD_SET(newfd, &master); // add to master set
                    if (newfd > fdmax) {    // keep track of the maximum
                        fdmax = newfd;
                    }
                    printf("selectserver: new connection from %s on "
                        "socket %d\n", inet_ntoa(remoteaddr.sin_addr), newfd);
                }
            } else {
                // handle data from a client
                if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
                    // got error or connection closed by client
                    if (nbytes == 0) {
                        // connection closed
                        printf("selectserver: socket %d hung up\n", i);
                    } else {
                        perror("recv");
                    }
                    close(i); // bye!
                    FD_CLR(i, &master); // remove from master set
                } else {
                    // we got some data from a client
                    for(j = 0; j <= fdmax; j++) {
                        // send to everyone!
                        if (FD_ISSET(j, &master)) {
                            // except the listener and ourselves
                            if (j != listener && j != i) {
                                if (send(j, buf, nbytes, 0) == -1) {
                                    perror("send");
                                }
                            }
                        }
                    }
                }
            } // it's SO UGLY!
```

```
                }
            }
        }

        return 0;
    }
```

Notice I have two file descriptor sets in the code: *master* and *read_fds*. The first, *master*, holds all the socket descriptors that are currently connected, as well as the socket descriptor that is listening for new connections.

The reason I have the *master* set is that `select()` actually *changes* the set you pass into it to reflect which sockets are ready to read. Since I have to keep track of the connections from one call of `select()` to the next, I must store these safely away somewhere. At the last minute, I copy the *master* into the *read_fds*, and then call `select()`.

But doesn't this mean that every time I get a new connection, I have to add it to the *master* set? Yup! And every time a connection closes, I have to remove it from the *master* set? Yes, it does.

Notice I check to see when the *listener* socket is ready to read. When it is, it means I have a new connection pending, and I `accept()` it and add it to the *master* set. Similarly, when a client connection is ready to read, and `recv()` returns `0`, I know the client has closed the connection, and I must remove it from the *master* set.

If the client `recv()` returns non-zero, though, I know some data has been received. So I get it, and then go through the *master* list and send that data to all the rest of the connected clients.

And that, my friends, is a less-than-simple overview of the almighty `select()` function.

6.3. Handling Partial `send()`s

Remember back in the section about `send()`, above, when I said that `send()` might not send all the bytes you asked it to? That is, you want it to send 512 bytes, but it returns 412. What happened to the remaining 100 bytes?

Well, they're still in your little buffer waiting to be sent out. Due to circumstances beyond your control, the kernel decided not to send all the data out in one chunk, and now, my friend, it's up to you to get the data out there.

You could write a function like this to do it, too:

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

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // how many bytes we've sent
    int bytesleft = *len; // how many we have left to send
    int n;

    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }
```

```
    *len = total; // return number actually sent here

    return n==-1?-1:0; // return -1 on failure, 0 on success
}
```

In this example, *s* is the socket you want to send the data to, *buf* is the buffer containing the data, and *len* is a pointer to an `int` containing the number of bytes in the buffer.

The function returns `-1` on error (and *errno* is still set from the call to `send()`.) Also, the number of bytes actually sent is returned in *len*. This will be the same number of bytes you asked it to send, unless there was an error. `sendall()` will do it's best, huffing and puffing, to send the data out, but if there's an error, it gets back to you right away.

For completeness, here's a sample call to the function:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("We only sent %d bytes because of the error!\n", len);
}
```

What happens on the receiver's end when part of a packet arrives? If the packets are variable length, how does the receiver know when one packet ends and another begins? Yes, real-world scenarios are a royal pain in the donkeys. You probably have to *encapsulate* (remember that from the data encapsulation section way back there at the beginning?) Read on for details!

6.4. Son of Data Encapsulation

What does it really mean to encapsulate data, anyway? In the simplest case, it means you'll stick a header on there with either some identifying information or a packet length, or both.

What should your header look like? Well, it's just some binary data that represents whatever you feel is necessary to complete your project.

Wow. That's vague.

Okay. For instance, let's say you have a multi-user chat program that uses `SOCK_STREAM`s. When a user types ("says") something, two pieces of information need to be transmitted to the server: what was said and who said it.

So far so good? "What's the problem?" you're asking.

The problem is that the messages can be of varying lengths. One person named "tom" might say, "Hi", and another person named "Benjamin" might say, "Hey guys what is up?"

So you `send()` all this stuff to the clients as it comes in. Your outgoing data stream looks like this:

```
t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

And so on. How does the client know when one message starts and another stops? You could, if you wanted, make all messages the same length and just call the `sendall()` we implemented, above. But that wastes bandwidth! We don't want to `send()` 1024 bytes just so "tom" can say "Hi".

So we *encapsulate* the data in a tiny header and packet structure. Both the client and server know how to pack and unpack (sometimes referred to as "marshal" and "unmarshal") this data. Don't look now, but we're starting to define a *protocol* that describes how a client and server communicate!

In this case, let's assume the user name is a fixed length of 8 characters, padded with `'\0'`. And then let's assume the data is variable length, up to a maximum of 128 characters. Let's have a look a sample packet structure that we might use in this situation:

1. `len` (1 byte, unsigned) – The total length of the packet, counting the 8-byte user name and chat data.
2. `name` (8 bytes) – The user's name, NUL-padded if necessary.
3. `chatdata` (*n*-bytes) – The data itself, no more than 128 bytes. The length of the packet should be calculated as the length of this data plus 8 (the length of the name field, above).

Why did I choose the 8-byte and 128-byte limits for the fields? I pulled them out of the air, assuming they'd be long enough. Maybe, though, 8 bytes is too restrictive for your needs, and you can have a 30-byte name field, or whatever. The choice is up to you.

Using the above packet definition, the first packet would consist of the following information (in hex and ASCII):

```
   0A     74 6F 6D 00 00 00 00 00      48 69
(length)  T  o  m    (padding)         H  i
```

And the second is similar:

```
   14     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(length)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w  ...
```

(The length is stored in Network Byte Order, of course. In this case, it's only one byte so it doesn't matter, but generally speaking you'll want all your binary integers to be stored in Network Byte Order in your packets.)

When you're sending this data, you should be safe and use a command similar to `sendall()`, above, so you know all the data is sent, even if it takes multiple calls to `send()` to get it all out.

Likewise, when you're receiving this data, you need to do a bit of extra work. To be safe, you should assume that you might receive a partial packet (like maybe we receive "`00 14 42 65 6E`" from Benjamin, above, but that's all we get in this call to `recv()`). We need to call `recv()` over and over again until the packet is completely received.

But how? Well, we know the number of bytes we need to receive in total for the packet to be complete, since that number is tacked on the front of the packet. We also know the maximum packet size is 1+8+128, or 137 bytes (because that's how we defined the packet.)

What you can do is declare an array big enough for two packets. This is your work array where you will reconstruct packets as they arrive.

Every time you `recv()` data, you'll feed it into the work buffer and check to see if the packet is complete. That is, the number of bytes in the buffer is greater than or equal to the length specified in the header (+1, because the length in the header doesn't include the byte for the length itself.) If the number of bytes in the buffer is less than 1, the packet is not complete, obviously. You have to make a special case for this, though, since the first byte is garbage and you can't rely on it for the correct packet length.

Once the packet is complete, you can do with it what you will. Use it, and remove it from your work buffer.

Whew! Are you juggling that in your head yet? Well, here's the second of the one-two punch: you might have read past the end of one packet and onto the next in a single `recv()` call. That is, you have a work buffer with one complete packet, and an incomplete part of the next packet! Bloody heck. (But this is why you made your work buffer large enough to hold *two* packets–in case this happened!)

Since you know the length of the first packet from the header, and you've been keeping track of the number of bytes in the work buffer, you can subtract and calculate how many of the bytes in the work buffer belong to the second (incomplete) packet. When you've handled the first one, you can clear it out of the work buffer and move the partial second packed down the to front of the buffer so it's all ready to go for the next `recv()`.

(Some of you readers will note that actually moving the partial second packet to the beginning of the work buffer takes time, and the program can be coded to not require this by using a circular buffer. Unfortunately for the rest of you, a discussion on circular buffers is beyond the scope of this article. If you're still curious, grab a data structures book and go from there.)

I never said it was easy. Ok, I did say it was easy. And it is; you just need practice and pretty soon it'll come to you naturally. By Excalibur I swear it!

7. 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?

7.1. man Pages

Try the following man pages, for starters:

- `htonl()`[18]
- `htons()`[19]
- `ntohl()`[20]
- `ntohs()`[21]
- `inet_aton()`[22]
- `inet_addr()`[23]
- `inet_ntoa()`[24]
- `socket()`[25]
- `socket options`[26]
- `bind()`[27]

- `connect()`[28]
- `listen()`[29]
- `accept()`[30]
- `send()`[31]
- `recv()`[32]
- `sendto()`[33]
- `recvfrom()`[34]
- `close()`[35]
- `shutdown()`[36]
- `getpeername()`[37]
- `getsockname()`[38]
- `gethostbyname()`[39]
- `gethostbyaddr()`[40]
- `getprotobyname()`[41]
- `fcntl()`[42]
- `select()`[43]
- `perror()`[44]
- `gettimeofday()`[45]

7.2. Books

For old-school actual hold-it-in-your-hand pulp paper books, try some of the following excellent guides. Note the prominent Amazon.com logo. What all this shameless commercialism means is that I basically get a kickback (Amazon.com store credit, actually) for selling these books through this guide. So if you're going to order one of these books anyway, why not send me a special thank you by starting your spree from one of the links, below.

Besides, more books for me might ultimately lead to more guides for you. `;-)`

[46]

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 013490012X[47], 0130810819[48].

Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0130183806[49], 0139738436[50], 0138487146[51].

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3: 0201633469[52], 020163354X[53], 0201634953[54].

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 1565923227[55].

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201563177[56].

Using C on the UNIX System by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0937175234. *Out of print.*

7.3. Web References

On the web:

BSD Sockets: A Quick And Dirty Primer[57] (has other great Unix system programming info, too!)

The Unix Socket FAQ[58]

Client-Server Computing[59]

Intro to TCP/IP[60] (gopher)

Internet Protocol Frequently Asked Questions[61]

The Winsock FAQ[62]

7.4. RFCs

RFCs[63]–the real dirt:

RFC-768[64]–The User Datagram Protocol (UDP)

RFC-791[65]–The Internet Protocol (IP)

RFC-793[66]–The Transmission Control Protocol (TCP)

RFC-854[67]–The Telnet Protocol

RFC-951[68]–The Bootstrap Protocol (BOOTP)

RFC-1350[69]–The Trivial File Transfer Protocol (TFTP)

8. Common Questions

Q: Where can I get those header files?

A: If you don't have them on your system already, you probably don't need them. Check the manual for your particular platform. If you're building for Windows, you only need to `#include <winsock.h>`.

Q: What do I do when `bind()` reports "Address already in use"?

A: You have to use `setsockopt()` with the `SO_REUSEADDR` option on the listening socket. Check out the section on `bind()` and the section on `select()` for an example.

Q: How do I get a list of open sockets on the system?

A: Use the netstat. Check the man page for full details, but you should get some good output just typing:

```
$ netstat
```

The only trick is determining which socket is associated with which program. `:-)`

Q: How can I view the routing table?

A: Run the route command (in `/sbin` on most Linuxes) or the command netstat -r.

Q: How can I run the client and server programs if I only have one computer? Don't I need a network to write network program?

A: Fortunately for you, virtually all machines implement a loopback network "device" that sits in the kernel and pretends to be a network card. (This is the interface listed as "`lo`" in the routing table.)

Pretend you're logged into a machine named "`goat`". Run the client in one window and the server in another. Or start the server in the background ("**server &**") and run the client in the same window. The upshot of the loopback device is that you can either **client goat** or **client localhost** (since "`localhost`" is likely defined in your `/etc/hosts` file) and you'll have the client talking to the server without a network!

In short, no changes are necessary to any of the code to make it run on a single non-networked machine! Huzzah!

Q: How can I tell if the remote side has closed connection?

A: You can tell because `recv()` will return `0`.

Q: How do I implement a "ping" utility? What is ICMP? Where can I find out more about raw sockets and `SOCK_RAW`?

A: All your raw sockets questions will be answered in W. Richard Stevens' UNIX Network Programming books. See the books section of this guide.

Q: How do I build for Windows?

A: First, delete Windows and install Linux or BSD. `};-)`. No, actually, just see the section on building for Windows in the introduction.

Q: How do I build for Solaris/SunOS? I keep getting linker errors when I try to compile!

A: The linker errors happen because Sun boxes don't automatically compile in the socket libraries. See the section on building for Solaris/SunOS in the introduction for an example of how to do this.

Q: Why does `select()` keep falling out on a signal?

A: Signals tend to cause blocked system calls to return `-1` with *`errno`* set to `EINTR`. When you set up a signal handler with `sigaction()`, you can set the flag `SA_RESTART`, which is supposed to restart the system call after it was interrupted.

Naturally, this doesn't always work.

My favorite solution to this involves a `goto` statement. You know this irritates your professors to no end, so go for it!

```
select_restart:
    if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
        if (errno == EINTR) {
            // some signal just interrupted us, so restart
            goto select_restart;
        }
        // handle the real error here:
        perror("select");
    }
```

Sure, you don't *need* to use `goto` in this case; you can use other structures to control it. But I think the `goto` statement is actually cleaner.

Q: How can I implement a timeout on a call to `recv()`?

A: Use `select()`! It allows you to specify a timeout parameter for socket descriptors that you're looking to read from. Or, you could wrap the entire functionality in a single function, like this:

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

int recvtimeout(int s, char *buf, int len, int timeout)
{
    fd_set fds;
    int n;
    struct timeval tv;

    // set up the file descriptor set
    FD_ZERO(&fds);
    FD_SET(s, &fds);
```

```
    // set up the struct timeval for the timeout
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // wait until timeout or data received
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -1; // error

    // data must be here, so do a normal recv()
    return recv(s, buf, len, 0);
}

// Sample call to recvtimeout():
    .
    .
    n = recvtimeout(s, buf, sizeof(buf), 10); // 10 second timeout

    if (n == -1) {
        // error occurred
        perror("recvtimeout");
    }
    else if (n == -2) {
        // timeout occurred
    } else {
        // got some data in buf
    }
    .
    .
```

Notice that `recvtimeout()` returns `-2` in case of a timeout. Why not return `0`? Well, if you recall, a return value of `0` on a call to `recv()` means that the remote side closed the connection. So that return value is already spoken for, and `-1` means "error", so I chose `-2` as my timeout indicator.

Q: How do I encrypt or compress the data before sending it through the socket?

A: One easy way to do encryption is to use SSL (secure sockets layer), but that's beyond the scope of this guide.

But assuming you want to plug in or implement your own compressor or encryption system, it's just a matter of thinking of your data as running through a sequence of steps between both ends. Each step changes the data in some way.

1. server reads data from file (or whereever)
2. server encrypts data (you add this part)
3. server `send()`s encrypted data

Now the other way around:

4. client `recv()`s encrypted data
5. client decrypts data (you add this part)
6. client writes data to file (or whereever)

You can also do compression at the same point that you do the encryption/decryption, above. Or you could do both! Just remember to compress before you encrypt. `:)`

Just as long as the client properly undoes what the server does, the data will be fine in the end no matter how many intermediate steps you add.

So all you need to do to use my code is to find the place between where the data is read and the data is sent (using `send()`) over the network, and stick some code in there that does the encryption.

Q: What is this "`PF_INET`" I keep seeing? Is it related to `AF_INET`?

A: Yes, yes it is. See the section on `socket()` for details.

Q: How can I write a server that accepts shell commands from a client and executes them?

A: For simplicity, lets say the client `connect()`s, `send()`s, and `close()`s the connection (that is, there are no subsequent system calls without the client connecting again.)

The process the client follows is this:

1. `connect()` to server
2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` the connection

Meanwhile, the server is handling the data and executing it:

1. `accept()` the connection from the client
2. `recv(str)` the command string
3. `close()` the connection
4. `system(str)` to run the command

Beware! Having the server execute what the client says is like giving remote shell access and people can do things to your account when they connect to the server. For instance, in the above example, what if the client sends "**rm -rf ~**"? It deletes everything in your account, that's what!

So you get wise, and you prevent the client from using any except for a couple utilities that you know are safe, like the **foobar** utility:

```
if (!strcmp(str, "foobar")) {
    sprintf(sysstr, "%s > /tmp/server.out", str);
    system(sysstr);
}
```

But you're still unsafe, unfortunately: what if the client enters "**foobar; rm -rf ~**"? The safest thing to do is to write a little routine that puts an escape ("`\`") character in front of all non-alphanumeric characters (including spaces, if appropriate) in the arguments for the command.

As you can see, security is a pretty big issue when the server starts executing things the client sends.

Q: I'm sending a slew of data, but when I `recv()`, it only receives 536 bytes or 1460 bytes at a time. But if I run it on my local machine, it receives all the data at the same time. What's going on?

A: You're hitting the MTU–the maximum size the physical medium can handle. On the local machine, you're using the loopback device which can handle 8K or more no problem. But on ethernet, which can only handle 1500 bytes with a header, you hit that limit. Over a modem, with 576 MTU (again, with header), you hit the even lower limit.

You have to make sure all the data is being sent, first of all. (See the `sendall()` function implementation for details.) Once you're sure of that, then you need to call `recv()` in a loop until all your data is read.

Read the section Son of Data Encapsulation for details on receiving complete packets of data using multiple calls to `recv()`.

Q: I'm on a Windows box and I don't have the `fork()` system call or any kind of `struct sigaction`. What to do?

A: If they're anywhere, they'll be in POSIX libraries that may have shipped with your compiler. Since I don't have a Windows box, I really can't tell you the answer, but I seem to remember that Microsoft has a POSIX compatibility layer and that's where `fork()` would be. (And maybe even `sigaction`.)

Search the help that came with VC++ for "fork" or "POSIX" and see if it gives you any clues.

If that doesn't work at all, ditch the `fork()`/`sigaction` stuff and replace it with the Win32 equivalent: `CreateProcess()`. I don't know how to use `CreateProcess()`–it takes a bazillion arguments, but it should be covered in the docs that came with VC++.

Q: How do I send data securely with TCP/IP using encryption?

A: Check out the OpenSSL project[70].

Q: I'm behind a firewall–how do I let people outside the firewall know my IP address so they can connect to my machine?

A: Unfortunately, the purpose of a firewall is to prevent people outside the firewall from connecting to machines inside the firewall, so allowing them to do so is basically considered a breach of security.

This isn't to say that all is lost. For one thing, you can still often `connect()` through the firewall if it's doing some kind of masquerading or NAT or something like that. Just design your programs so that you're always the one initiating the connection, and you'll be fine.

If that's not satisfactory, you can ask your sysadmins to poke a hole in the firewall so that people can connect to you. The firewall can forward to you either through it's NAT software, or through a proxy or something like that.

Be aware that a hole in the firewall is nothing to be taken lightly. You have to make sure you don't give bad people access to the internal network; if you're a beginner, it's a lot harder to make software secure than you might imagine.

Don't make your sysadmin mad at me. `;-)`

9. 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, let this be a warning to 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. The whole thing could be completely and utterly wrong!

But it's probably not. After all, I've spent many many hours messing with this stuff, and implemented several TCP/IP network utilities at work, have written multiplayer game engines, and so on. But 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@piratehaven.org>` 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. Ha! 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. The web just happens to be the perfect vehicle. I encourage others to provide similar information whenever possible.

Enough of this–back to coding! `;-)`

Notes

1. http://www.ecst.csuchico.edu/~beej/guide/net/
2. http://tangentsoft.net/wskfaq/
3. http://www.tuxedo.org/~esr/faqs/smart-questions.html

4. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
5. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
6. http://www.rfc-editor.org/rfc/rfc793.txt
7. http://www.rfc-editor.org/rfc/rfc791.txt
8. http://www.rfc-editor.org/rfc/rfc768.txt
9. http://www.rfc-editor.org/rfc/rfc791.txt
10. http://www.rfc-editor.org/rfc/rfc1413.txt
11. http://www.ecst.csuchico.edu/~beej/guide/net/examples/getip.c
12. http://www.ecst.csuchico.edu/~beej/guide/net/examples/server.c
13. http://www.ecst.csuchico.edu/~beej/guide/net/examples/client.c
14. http://www.ecst.csuchico.edu/~beej/guide/net/examples/listener.c
15. http://www.ecst.csuchico.edu/~beej/guide/net/examples/talker.c
16. http://www.ecst.csuchico.edu/~beej/guide/net/examples/select.c
17. http://www.ecst.csuchico.edu/~beej/guide/net/examples/selectserver.c
18. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htonl.3.inc
19. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htons.3.inc
20. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohl.3.inc
21. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohs.3.inc
22. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_aton.3.inc
23. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_addr.3.inc
24. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_ntoa.3.inc
25. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/socket.2.inc
26. http://linux.com.hk/man/showman.cgi?manpath=/man/man7/socket.7.inc
27. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/bind.2.inc
28. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/connect.2.inc
29. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/listen.2.inc
30. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/accept.2.inc
31. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
32. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
33. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/sendto.2.inc
34. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recvfrom.2.inc

35. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/close.2.inc
36. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/shutdown.2.inc
37. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getpeername.2.inc
38. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getsockname.2.inc
39. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyname.3.inc
40. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyaddr.3.inc
41. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/getprotobyname.3.inc
42. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/fcntl.2.inc
43. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/select.2.inc
44. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/perror.3.inc
45. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/gettimeofday.2.inc
46. http://www.amazon.com/exec/obidos/redirect-home/beejsguides-20
47. http://www.amazon.com/exec/obidos/ASIN/013490012X/beejsguides-20
48. http://www.amazon.com/exec/obidos/ASIN/0130810819/beejsguides-20
49. http://www.amazon.com/exec/obidos/ASIN/0130183806/beejsguides-20
50. http://www.amazon.com/exec/obidos/ASIN/0139738436/beejsguides-20
51. http://www.amazon.com/exec/obidos/ASIN/0138487146/beejsguides-20
52. http://www.amazon.com/exec/obidos/ASIN/0201633469/beejsguides-20
53. http://www.amazon.com/exec/obidos/ASIN/020163354X/beejsguides-20
54. http://www.amazon.com/exec/obidos/ASIN/0201634953/beejsguides-20
55. http://www.amazon.com/exec/obidos/ASIN/1565923227/beejsguides-20
56. http://www.amazon.com/exec/obidos/ASIN/0201563177/beejsguides-20
57. http://www.cs.umn.edu/~bentlema/unix/
58. http://www.ibrado.com/sock-faq/
59. http://pandonia.canberra.edu.au/ClientServer/
60. gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/
61. http://www-iso8859-5.stack.net/pages/faqs/tcpip/tcpipfaq.html
62. http://tangentsoft.net/wskfaq/
63. http://www.rfc-editor.org/
64. http://www.rfc-editor.org/rfc/rfc768.txt
65. http://www.rfc-editor.org/rfc/rfc791.txt

66. http://www.rfc-editor.org/rfc/rfc793.txt
67. http://www.rfc-editor.org/rfc/rfc854.txt
68. http://www.rfc-editor.org/rfc/rfc951.txt
69. http://www.rfc-editor.org/rfc/rfc1350.txt
70. http://www.openssl.org/