5. System Calls or Bust

This is the section where we get into the system calls (and other library calls) that allow you to access the network functionality of a Unix box, or any box that supports the sockets API for that matter (BSD, Windows, Linux, Mac, what-have-you.) 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!

*(Please note that for brevity, many code snippets below do not include necessary error checking. And they very commonly assume that the result from calls to **`getaddrinfo()`** succeed and return a valid entry in the linked list. Both of these situations are properly addressed in the stand-alone programs, though, so use those as a model.)*

5.1. `getaddrinfo()`—Prepare to launch!

This is a real workhorse of a function with a lot of options, but usage is actually pretty simple. It helps set up the `struct`s you need later on.

A tiny bit of history: it used to be that you would use a function called **`gethostbyname()`** to do DNS lookups. Then you'd load that information by hand into a `struct sockaddr_in`, and use that in your calls.

This is no longer necessary, thankfully. (Nor is it desirable, if you want to write code that works for both IPv4 and IPv6!) In these modern times, you now have the function **`getaddrinfo()`** that does all kinds of good stuff for you, including DNS and service name lookups, and fills out the `struct`s you need, besides!

Let's take a look!

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

int getaddrinfo(const char *node,     // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

You give this function three input parameters, and it gives you a pointer to a linked-list, *res*, of results.

~~Google Adsense~~

The *node* parameter is the host name to connect to, or an IP address.

Next is the parameter *service*, which can be a port number, like "80", or the name of a particular service (found in [The IANA Port List]() or the */etc/services* file on your Unix machine) like "http" or "ftp" or "telnet" or "smtp" or whatever.

Finally, the *hints* parameter points to a `struct addrinfo` that you've already filled out with relevant information.

Here's a sample call if you're a server who wants to listen on your host's IP address, port 3490. Note that this doesn't actually do any listening or network setup; it merely sets up structures we'll use later:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}

// servinfo now points to a linked list of 1 or more struct addrinfos

// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo); // free the linked-list
```

Notice that I set the *ai_family* to `AF_UNSPEC`, thereby saying that I don't care if we use IPv4 or IPv6. You can set it to `AF_INET` or `AF_INET6` if you want one or the other specifically.

Also, you'll see the `AI_PASSIVE` flag in there; this tells **`getaddrinfo()`** to assign the address of my local host to the socket structures. This is nice because then you don't have to hardcode it. (Or you can put a specific address in as the first parameter to **`getaddrinfo()`** where I currently have `NULL`, up there.)

Then we make the call. If there's an error (**`getaddrinfo()`** returns non-zero), we can print it out using the function **`gai_strerror()`**, as you see. If everything works properly, though, *servinfo* will point to a linked list of `struct addrinfo`s, each of which contains a `struct sockaddr` of some kind that we can use later! Nifty!

Finally, when we're eventually all done with the linked list that **`getaddrinfo()`** so graciously allocated for us, we can (and should) free it all up with a call to **`freeaddrinfo()`**.

Here's a sample call if you're a client who wants to connect to a particular server, say "www.example.net" port 3490. Again, this doesn't actually connect, but it sets up the structures we'll use later:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo now points to a linked list of 1 or more struct addrinfos

// etc.
```

~~Google Adsense~~

I keep saying that `servinfo` is a linked list with all kinds of address information. Let's write a quick demo program to show off this information. This short program will print the IP addresses for whatever host you specify on the command line:

```
/*
** showip.c -- show IP addresses for a host given on the command line
*/

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

int main(int argc, char *argv[])
{
    struct addrinfo hints, *res, *p;
    int status;
    char ipstr[INET6_ADDRSTRLEN];

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

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // AF_INET or AF_INET6 to force version
    hints.ai_socktype = SOCK_STREAM;

    if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
        return 2;
    }

    printf("IP addresses for %s:\n\n", argv[1]);

    for(p = res;p != NULL; p = p->ai_next) {
        void *addr;
        char *ipver;

        // get the pointer to the address itself,
        // different fields in IPv4 and IPv6:
        if (p->ai_family == AF_INET) { // IPv4
            struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
            addr = &(ipv4->sin_addr);
            ipver = "IPv4";
        } else { // IPv6
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
            addr = &(ipv6->sin6_addr);
            ipver = "IPv6";
        }

        // convert the IP to a string and print it:
        inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
        printf("  %s: %s\n", ipver, ipstr);
    }

    freeaddrinfo(res); // free the linked list

    return 0;
}
```

~~Google Adsense~~

As you see, the code calls **`getaddrinfo()`** on whatever you pass on the command line, that fills out the linked list pointed to by *`res`*, and then we can iterate over the list and print stuff out or do whatever.

(There's a little bit of ugliness there where we have to dig into the different types of `struct sockaddr`s depending on the IP version. Sorry about that! I'm not sure of a better way around it.)

Sample run! Everyone loves screenshots:

```
$ showip www.example.net
IP addresses for www.example.net:

  IPv4: 192.0.2.88

$ showip ipv6.example.com
IP addresses for ipv6.example.com:

  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

Now that we have that under control, we'll use the results we get from **`getaddrinfo()`** to pass to other socket functions and, at long last, get our network connection established! Keep reading!

5.2. `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? They allow you to say what kind of socket you want (IPv4 or IPv6, stream or datagram, and TCP or UDP).

It used to be people would hardcode these values, and you can absolutely still do that. (*`domain`* is `PF_INET` or `PF_INET6`, *`type`* is `SOCK_STREAM` or `SOCK_DGRAM`, and *`protocol`* can be set to `0` to choose the proper protocol for the given `type`. Or you can call **`getprotobyname()`** to look up the protocol you want, "tcp" or "udp".)

~~Google Adsense~~

(This `PF_INET` thing is a close relative of the `AF_INET` that you can use when initializing the *`sin_family`* field in your `struct sockaddr_in`. In fact, they're so closely related that they actually have the same value, and many programmers will call **`socket()`** and pass `AF_INET` as the first argument instead of **`PF_INET`**. Now, get some milk and cookies, because it's times for a story. Once upon a time, 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 referred to by their protocol family (what the "PF" in "`PF_INET`" stands for). That didn't happen. And they all lived happily ever after, The End. So the most correct thing to do is to use `AF_INET` in your `struct sockaddr_in` and `PF_INET` in your call to **`socket()`**.)

Anyway, enough of that. What you really want to do is use the values from the results of the call to **`getaddrinfo()`**, and feed them into **`socket()`** directly like this:

```
int s;
struct addrinfo hints, *res;

// do the lookup
// [pretend we already filled out the "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// [again, you should do error-checking on getaddrinfo(), and walk
// the "res" linked list looking for valid entries instead of just
// assuming the first one is good (like many of these examples do.)
// See the section on client/server for real examples.]

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

`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 *`errno`* man page for more details, and a quick note on using *`errno`* in multithreaded programs.)

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.

5.3. `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—multiplayer network games do this when they tell you to "connect to 192.168.5.10 port 3490".) 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()`** (because you're the client, not the server), this is probably 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`* is the length in bytes of that address. ~~Google Adsense~~

Whew. That's a bit to absorb in one chunk. Let's have an example that binds the socket to the host the program is running on, port 3490:

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

By using the `AI_PASSIVE` flag, I'm telling the program to bind to the IP of the host it's running on. If you want to bind to a specific local IP address, drop the `AI_PASSIVE` and put an IP address in for the first argument to **`getaddrinfo()`**.

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

Lots of old code manually packs the `struct sockaddr_in` before calling **`bind()`**. Obviously this is IPv4-specific, but there's really nothing stopping you from doing the same thing with IPv6, except that using **`getaddrinfo()`** is going to be easier, generally. Anyway, the old code looks something like this:

```
// !!! THIS IS THE OLD WAY !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
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', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

In the above code, you could also assign `INADDR_ANY` to the *`s_addr`* field if you wanted to bind to your local IP address (like the `AI_PASSIVE` flag, above.) The IPv6 version of `INADDR_ANY` is a global variable *`in6addr_any`* that is assigned into the *`sin6_addr`* field of your `struct sockaddr_in6`. (There is also a macro `IN6ADDR_ANY_INIT` that you can use in a variable initializer.)

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 ~~Google Adsense~~ 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 little bit of a 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.

5.4. `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`* is the length in bytes of the server address structure.

All of this information can be gleaned from the results of the **`getaddrinfo()`** call, which rocks.

Is this starting to make more sense? I can't hear you from here, so I'll just have to hope that it is. Let's have an example where we make a socket connection to "www.example.com", port 3490:

```
struct addrinfo hints, *res;
int sockfd;

```

~~Google Adsense~~

```
// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

Again, old-school programs filled out their own `struct sockaddr_in`s to pass to **`connect()`**. You can do that if you want to. See the similar note in the **`bind()`** section, above.

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.

5.5. `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()`** so that the server is running on a specific port. (You have to be able to tell your buddies which port to connect to!) So if you're going to be listening for incoming connections, the sequence of system calls you'll make is:

```
getaddrinfo();
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 sh ~~Google Adsense~~ call to **accept()**.

5.6. 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 for more new connections, and the newly created one is finally ready to **send()** and **recv()**. We're there!

The call is as follows:

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

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

sockfd is the **listen()**ing socket descriptor. Easy enough. *addr* will usually be a pointer to a local `struct sockaddr_storage`. 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_storage)` 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

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! don't forget your error checking for these calls !!

    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // make a socket, bind it, and listen on it:

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // now accept an incoming connection:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

    // ready to communicate on socket descriptor new_fd!
    .
    .
    .
```

~~Google Adsense~~

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.

5.7. 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 *proba* ~~Google Adsense~~ 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, 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!

5.8. `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, socklen_t 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 will probably be another `struct sockaddr_in` or `struct sockaddr_in6` or `struct sockaddr_storage` that you cast at the last minute) which contains the destination IP address and port. *`tolen`*, an `int` deep-down, can simply be set to `sizeof *to` or `sizeof(struct sockaddr_storage)`.

To get your hands on the destination address structure, you'll probably either get it from **`getaddrinfo()`**, or from **`recvfrom()`**, below, or you'll fill it out by hand.

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_storage` 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 *from` or `sizeof(struct sockaddr_storage)`. When the function returns, *`fromlen`* will contain the length of the address actually stored in *`from`*.

~~Google Adsense~~

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

So, here's a question: why do we use `struct sockaddr_storage` as the socket type? Why not `struct sockaddr_in`? Because, you see, we want to not tie ourselves down to IPv4 or IPv6. So we use the generic `struct sockaddr_storage` which we know will be big enough for either.

(So... here's another question: why isn't `struct sockaddr` itself big enough for any address? We even cast the general-purpose `struct sockaddr_storage` to the general-purpose `struct sockaddr`! Seems extraneous and redundant, huh. The answer is, it just isn't big enough, and I'd guess that changing it at this point would be Problematic. So they made a new one.)

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.

5.9. `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 ~~Google Adsense~~ **`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.

(Except to remember that if you're using Windows and Winsock that you should call **`closesocket()`** instead of **`close()`**.)

5.10. `getpeername()`—Who are you?

This function is so easy.

It's so easy, I almost didn't give it its 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 *addr` or `sizeof(struct sockaddr)`.

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

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

5.11. `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 *`hostna`* ~~Google Adsense~~

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

<< Prev Beej's Guide to Network Programming Next >>