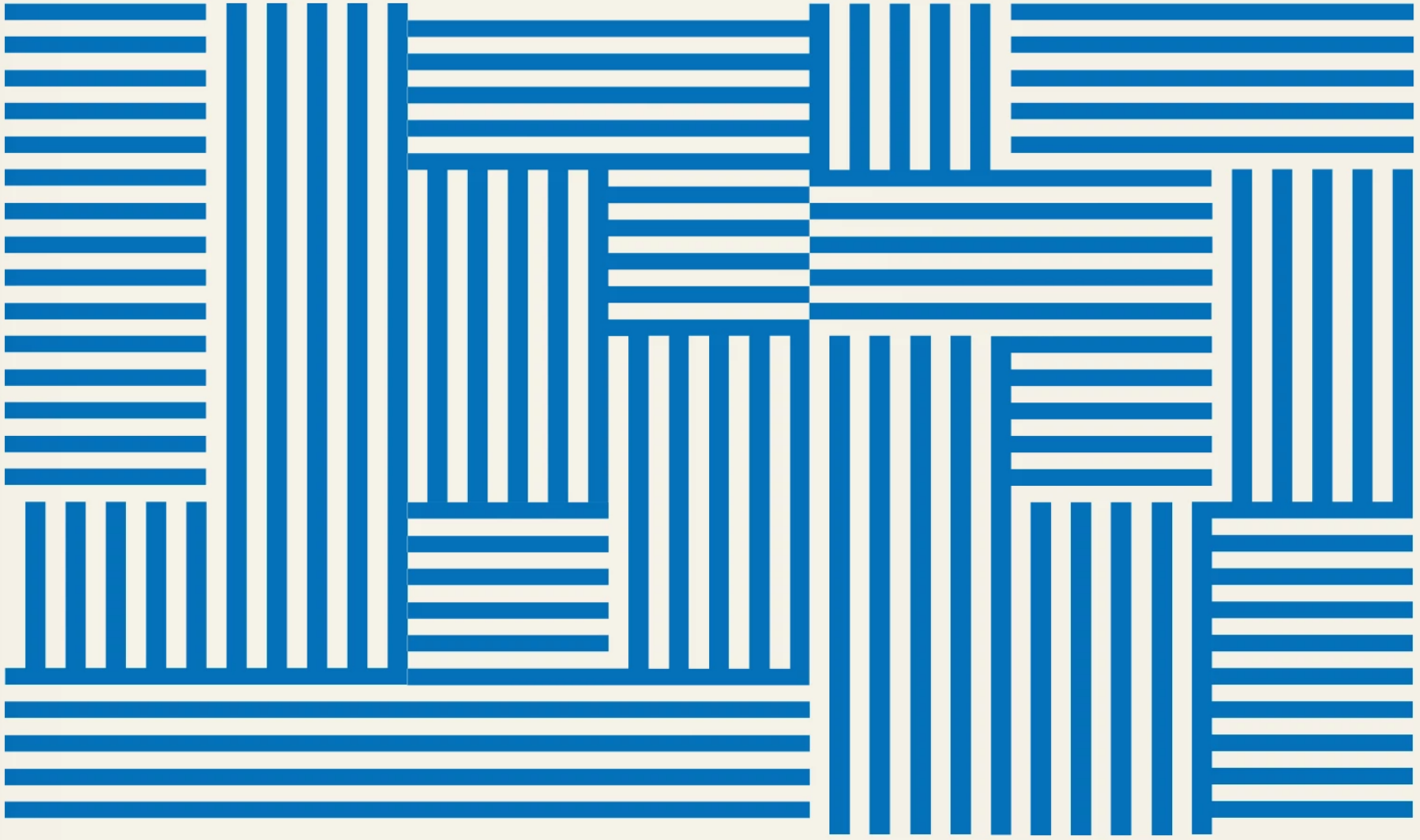


N/W programming in C/C++ (+Linux)

1

Beej's Guide to Network Programming

Using Internet Sockets

Brian "Beej Jorgensen" Hall

What is a socket? → from CN: a unique identifier for Transport L. to do "process to process" delivery.

socket = IP + port struct{}

in Unix based systems: sockets are

a way to speak to other programs using standard Unix file descriptors.

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.

Two Types of Internet Sockets

(there are more)

"Stream Sockets" → "SOCK_STREAM"

- Reliable
- Two-way connected

"Datagram Sockets" → "SOCK_DGRAM"

- Unreliable
- Connectionless

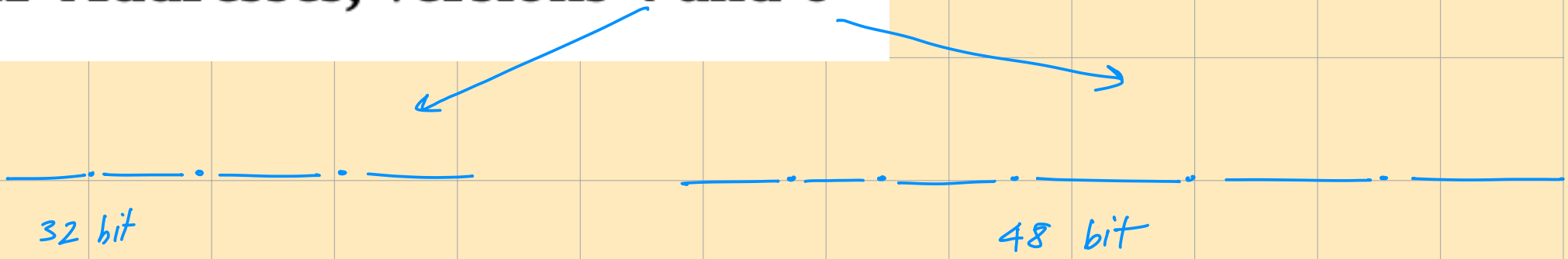
"Stream Oriented" / "Connection Oriented"

↳ TCP

↳ UDP

Things to keep in mind:

IP Addresses, versions 4 and 6



Port Numbers

(16 bits) used by processes to communicate

Think of the IP address as the street address of a hotel, and the port number as the room number. That's a decent analogy;

Byte Order

Endian-ness on diff. architectures.

how to "order" bytes correctly?

All righty. There are two types of numbers 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").

Function	Description
`htons()`	host `to` network `short`
`htonl()`	host `to` network `long`
`ntohs()`	network `to` host `short`
`ntohl()`	network `to` host `long`

Basically, you'll want to convert the numbers to Network Byte Order before they go out on the wire, and convert them to Host Byte Order as they come in off the wire.

Program elements :

"TCP":

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

```
int
```

Just a regular `int`.

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

— used to "pref" the socked address structures
— " in hostname lookups,
service name " etc.

You'll load this struct up a bit, and then call `getaddrinfo()`. It'll return a pointer to a new linked list of these structures filled out with all the goodies you need.

You can force it to use IPv4 or IPv6 in the `ai_family` field, or leave it as `AF_UNSPEC` to use whatever. This is cool because your code can be IP version-agnostic.

Note that this is a linked list: `ai_next` points at the next element—there could be several results for you to choose from. I'd use the first result that worked, but you might have different business needs; I don't know everything, man!

You'll see that the `ai_addr` field in the `struct addrinfo` is a pointer to a `struct sockaddr`. This is where we start getting into the nitty-gritty details of what's inside an IP address structure.

Anyway, the `struct sockaddr` 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` (IPv4) or `AF_INET6` (IPv6) 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") to be used with IPv4.

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)

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

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 `connect()` wants a `struct sockaddr*`, you can still use a `struct sockaddr_in` and cast it at the last minute!

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, notice that `sin_family` corresponds to `sa_family` in a `struct sockaddr` and should be set to "`AF_INET`". Finally, the `sin_port` must be in *Network Byte Order* (by using `htons()`!)

First, let's say you have a `struct sockaddr_in ina`, and you have an IP address "`10.12.110.57`" or "`2001:db8:63b3:1::3490`" that you want to store into it. The function you want to use, `inet_pton()`, converts an IP address in numbers-and-dots notation into either a `struct in_addr` or a `struct in6_addr` depending on whether you specify `AF_INET` or `AF_INET6`. ("`pton`" stands for "presentation to network"—you can call it "printable to network" if that's easier to remember.) The conversion can be made as follows:

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

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? (Or a `struct in6_addr` that you want in, uh, "hex-and-colons" notation.) In this case, you'll want to use the function `inet_ntop()` ("ntop" means "network to presentation"—you can call it "network to printable" if that's easier to remember), like this:

```
// IPv4:

char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

printf("The IPv4 address is: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6;    // pretend this is loaded with something

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

from IPv4 to IPv6

But I just want to know what to change in my code to get it going with IPv6! Tell me now!

Ok! Ok!

Almost everything in here is something I've gone over, above, but it's the short version for the impatient. (Of course, there is more than this, but this is what applies to the guide.)

1. First of all, try to use `getaddrinfo()` to get all the `struct sockaddr` info, instead of packing the structures by hand. This will keep you IP version-agnostic, and will eliminate many of the subsequent steps.
2. Any place that you find you're hard-coding anything related to the IP version, try to wrap up in a helper function.
3. Change `AF_INET` to `AF_INET6`.
4. Change `PF_INET` to `PF_INET6`.
5. Change `INADDR_ANY` assignments to `in6addr_any` assignments, which are slightly different:

```
struct sockaddr_in sa;
```

```
struct sockaddr_in6 sa6;

sa.sin_addr.s_addr = INADDR_ANY;  // use my IPv4 address
sa6.sin6_addr = in6addr_any; // use my IPv6 address
```

Also, the value `IN6ADDR_ANY_INIT` can be used as an initializer when the `struct in6_addr` is declared, like so:

```
struct in6_addr ia6 = IN6ADDR_ANY_INIT;
```

6. Instead of `struct sockaddr_in` use `struct sockaddr_in6`, being sure to add "6" to the fields as appropriate (see `structs`, above). There is no `sin6_zero` field.
7. Instead of `struct in_addr` use `struct in6_addr`, being sure to add "6" to the fields as appropriate (see `structs`, above).
8. Instead of `inet_aton()` or `inet_addr()`, use `inet_pton()`.
9. Instead of `inet_ntoa()`, use `inet_ntop()`.
10. Instead of `gethostbyname()`, use the superior `getaddrinfo()`.
11. Instead of `gethostbyaddr()`, use the superior `getnameinfo()` (although `gethostbyaddr()` can still work with IPv6).
12. `INADDR_BROADCAST` no longer works. Use IPv6 multicast instead.

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.

System Calls

`getaddrinfo()`—Prepare to launch!

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.

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[1] 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.

Eg:

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 addrinfos`, 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()`.

← Server

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.
```

`socket()`—Get the File Descriptor!

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

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

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.

`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 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.

`connect()`—Hey, you!

The `connect()` call is as follows:

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

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

`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:

```
1 getaddrinfo();
2 socket();
3 bind();
4 listen();
5 /* accept() goes here */
```

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

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

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

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

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, 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!

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

↳ Connection breakdown phase → One side's "Fin" msg.

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:

`how`	Effect
`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).