1. `structs` and Data Handling

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

Equivalent & will use:

```
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
};
```

- `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
- `len` is the length of that data in bytes.
- A pointer to a `struct sockaddr_in` can be cast to a pointer to a `struct sockaddr` and vice-versa.
- The `sin_port` and `sin_addr` 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.

```
// Internet address (a structure for historical reasons)
struct in_addr {
    unsigned long s_addr; // that's a 32-bit long, or 4 bytes
};
```

1.1 Convert the Natives!

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

1.2 IP Addresses and How to Deal With Them

```
struct sockaddr_in ina
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

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);
```

Usage:

```
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
```

1.3 Ip to string

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

`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
```

1.4 . Private (Or Disconnected) Networks

`10.x.x.x` and `192.168.x.x`, where x is 0-255, generally. Less common is `172.y.x.x`, where y goes between 16 and 31.

2. System Calls

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

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

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

- `domain` should be set to `PF_INET`.
- `type` argument tells the kernel what kind of socket this is: `SOCK_STREAM` or `SOCK_DGRAM`.
- set `protocol` to `0` to have `socket()` choose the correct `protocol` based on the `type`.
- `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).
- This `PF_INET` thing is a close relative of the `AF_INET` that you used 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.
- The most correct thing to do is to use `AF_INET` in your `struct sockaddr_in` and `PF_INET` in your call to `socket()`.

2.2 `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".)

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

Example:

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

#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(PF_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` & `my_addr.sin_addr.s_addr` are in Network Byte Order.
- The header files might differ from system to system. To be sure, you should check your local man pages.

Some of the process of getting your own IP address and/or port 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
```

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. (`INADDR_ANY` is really zero)

Purist way of usage:

```
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
```

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 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);
}
```

There are times when you won't absolutely have to call `bind()`. 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.

2.3 `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?

```
#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
- `addrlen` can be set to `sizeof(struct sockaddr)`.

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(PF_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));
    .
    .
    .
```

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.

2.4 `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()`

```
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 (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.)

- returns -1 and sets `errno` on error
- we need to call `bind()` before we call `listen()` or the kernel will have us listening on a random port. 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 */
```

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

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()`.

```
#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
- `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.
- returns -1 and sets `errno` if an error occurs.

```
#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(PF_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);
    .
    .
    .
```

- 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.

2.6 `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
- `len` is the length of that data in bytes.
- Just set `flags` to `0`. (See the `send()` man page for more information concerning flags.)
- returns the number of bytes actually sent out - this might be less than the number you told it to send. (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).
- `-1` is returned on error, and `errno` is set to the error number.

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

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.
- `flags` can again be set to `0`.
- returns the number of bytes actually read into the buffer, or `-1` on error (with errno set, accordingly.)

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

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);
```

- `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
- `len` is the length of that data in bytes.
- Just set `flags` to `0`. (See the `send()` man page for more information concerning flags.)
- `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.

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

- `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`.
- 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.

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

```
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.
- `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())
- returns `0` on success, and `-1` on error

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()`.

2.9 `getpeername()`--Who are you?

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.
- `addrlen` is a pointer to an int, that should be initialized to `sizeof(struct sockaddr)`.
- 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.

2.10 gethostname()--Who am I?

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.

```
#include <unistd.h>

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

- `hostname` is a pointer to an array of chars that will contain the hostname upon the function's return.
- `size` is the length in bytes of the hostname array.
- returns `0` on successful completion, and `-1` on error, setting `errno` as usual.

2.11 DNS--You say "whitehouse.gov", I say "63.161.169.137"

```
#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]
```

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

```
/*
** 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;
}
```

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.