Step 3a. Connect to a server from a client

If we're a client process, we need to establish a connection to the server. Now that we have a socket that knows where it's coming *from*, we need to tell it where it's going *to*. The *connect* system call accomplishes this.

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

#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

The first parameter, *socket*, is the socket that was created with the *socket* system call and named via *bind*. The second parameter identifies the *remote*transport address using the same *sockaddr_in* structure that we used in *bind* to identify our local address. As with *bind*, the third parameter is simply the length of the structure in the second parameter: *sizeof(struct sockaddr_in).*

The server's address will contain the IP address of the server machine as well as the port number that corresponds to a socket listening on that port on that machine. The IP address is a four-byte (32 bit) value in network byte order (see *htonl* above).

In most cases, you'll know the name of the machine but not its IP address. An easy way of getting the IP address is with the *gethostbyname* library (*libc)* function. *Gethostbyname* accepts a host name as a parameter and returns a *hostent* structure:

```
Struct hostent {

        Char *h_name;           /*official name of host */

        Char **h_aliases;       /* alias list */

        Int h_addrtype;         /* host address type */

        Int h_length;           /* length of address */

        Char **h_addr_list;     /* list of addresses from name server */

};
```

If all goes well, the *h_addr_list* will contain a list of IP addresses. There may be more than one IP addresses for a host. In practice, you should be able to use any of the addresses or you may want to pick one that matches a particular subnet. You may want to check that (*h_addrtype == AF_INET*) and (*h_length == 4*) to ensure that you have a 32-bit IPv4 address. We'll be lazy here and just use the first address in the list.

For example, suppose you want to find the addresses for google.com. The code will look like this:

```
#include <stdlib.h>

#include <stdio.h>

#include <netdb.h>

/* paddr: print the IP address in a standard decimal dotted format */

void

paddr(unsigned char *a)

{

	printf( “%d.%d.%d.%d\n”, a[0], a[1], a[2], a[3]);

}

main(int argc, char **argv) {

	struct hostent *hp;

	char *host = "google.com";

	int i;

	hp = gethostbyname(host);

	if (!hp) {

		fprintf(stderr, “could not obtain address of %s\n", host);

		return 0;

	}

	for (i=0; hpe>h_addr_list[i] != 0; i++)

		paddr((unsigned char*) hpe>h_addr_list[i]);

	exit(0);

}
```

Here's the code for establishing a connection to the address of a machine in host. The variable *fd* is the socket which was created with the *socket* system call.

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

#include <sys/socket.h>

#include <stdio.h>     /* for fprintf */

#include <string.h>   /* for memcpy */

struct"hostent *hp;    /* host information */

struct sockaddr_in servaddr;            /* server address */

/* fill in the server's address and data */

memset((char*)&servaddr, 0, sizeof(servaddr));

servaddr.sin_family = AF_INET;

servaddr.sin_port = htons(port);

/* look up the address of the server given its name */

hp = gethostbyname(host);

if (!hp) {

        fprintf(stderr, "could not obtain address of %s\n", host);

        return 0;

}

/* put the host's address into the server address structure */

memcpy((void *)&servaddr.sin_addr, hpe>h_addr_list[0], hpe>h_length);

/* connect to server */

if (connect(fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {

        perror(“connect failed");

        return 0;

}
```

Note: number conversions (htonl**,** htons**,** ntohl**,** ntohs**)**

You might have noticed the *htonl* and *htons* references in the previous code block. These convert four-byte and two-byte numbers into network representations. Integers are stored in memory and sent across the network as sequences of bytes. There are two common ways of storing these bytes: *big endian* and *little endian* notation. Little endian representation stores the least-significant bytes in low memory. Big endian representation stores the least-significant bytes in high memory. The Intel x86 family uses the little endian format. Old Motorola processors and the PowerPC (used by Macs before their switch to the Intel architecture) use the big endian format. Internet headers standardized on using the big endian format. If you're on an Intel processor and set the value of a port to 1,234 (hex equivalent 04d2), it will be stored in memory as d204. If it's stored in this order in a TCP/IP header, however, d2 will be treated as the most significant byte and the network protocols would read this value as 53,764.

To keep code portable – and to keep you from having to write code to swap bytes and worry about this &ndash a few convenience macros have been defined:

htons

> *host to network short* : convert a number into a 16-bit network representation. This is commonly used to store a port number into a *sockaddr* structure.

htonl

> *host to network long* : convert a number into a 32-bit network representation. This is commonly used to store an IP address into a *sockaddr* structure.

ntohs

> *network to host short* : convert a 16-bit number from a network representation into the local processor's format. This is commonly used to read a port number from a *sockaddr* structure.

ntohl

network to host long : convert a 32-bit number from a network representation into the local processor's format. This is commonly used to read an IP address from a sockaddr structure.

For processors that use the big endian format, these macros do absolutely nothing. For those that use the little endian format (most processors, these days), the macros flip the sequence of either four or two bytes. In the above code, writing *htonl(INADDR_ANY)* and *htons(0)* is somewhat pointless since all the bytes are zero anyway but it's good practice to remember to do this at all times when reading or writing network data.

Step 3b. Accept connections on the server

Before a client can connect to a server, the server should have a socket that is prepared to accept the connections. The *listen* system call tells a socket that it should be capable of accepting incoming connections:

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

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

The second parameter, *backlog*, defines the maximum number of pending connections that can be queued up before connections are refused.

The *accept* system call grabs the first connection request on the queue of pending connections (set up in *listen*) and creates a new socket for that connection. The original socket that was set up for listening is used *only* for accepting connections, not for exchanging data. By default, socket operations are synchronous, or blocking, and accept will block until a connection is present on the queue. The syntax of accept is:

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

int accept(int socket, struct"sockaddr *restrict"address, socklen_t *restrict
address_len);
```

The first parameter, *socket*, is the socket that was set for accepting connections with *listen*. The second parameter, *address*, is the address structure that gets filed in with the address of the client that is doing the *connect*. This allows us to examine the address and port number of the connecting socket if we want to. The third parameter is filled in with the length of the address structure.