Introduction to Socket Programming
Part II

Code snipet:

```
struct sockaddr_in  myAddressStruct;

//Fill in the address information into myAddressStruct here,  (will be explained in detail shortly)
```

connect(socket_file_descriptor, $\underbrace{\text{(struct sockaddr *) \&myAddressStruct}}$, sizeof(myAddressStruct));

Now lets discuss how to fill in the sockaddr_in structure:

```
struct sockaddr_in{

        sa_family_t  sin_family        /*Address/Protocol Family*/
        unit16_t       sin_port          /* Port number   --Network Byte Order-- */
        struct in_addr  sin_addr        /*A struct for the 32 bit IP Address  */
        unsigned char sin_zero[8]      /*Just ignore this it is just padding*/
};

struct in_addr{
        unit32_t     s_addr   /*32 bit IP Address   --Network Byte Order-- */
};
```

For the sa_family variable sin_family always use the constant: PF_INET   or   AF_INET
***Always initialize address structures with bzero() or memset() before filling them in ***
Make sure you use the byte ordering functions when necessary for the port and IP address variables otherwise there will be strange things a happening to your packets

Converting between dotted decimal strings and Network Address values (Read pgs 70 – 74 Stevens)

To convert a string dotted decimal IP4 address to a NETWORK BYTE ORDERED 32 bit value use the functions:

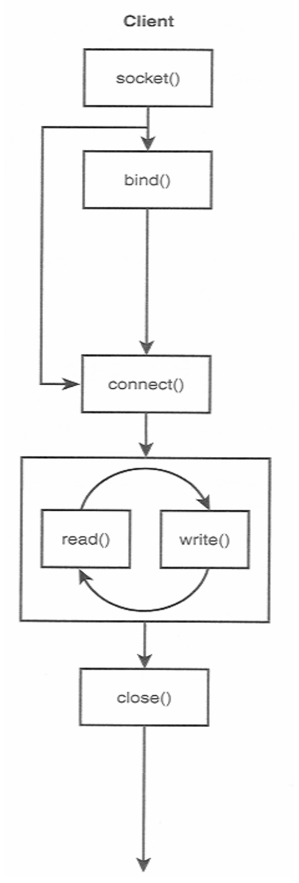
- inet_addr()
- inet_aton()

To convert a 32 bit NETWORK BYTE ORDERED to a IP4 dotted decimal string use:

- inet_ntoa()

Read Stevens also for more recent functions which work with both IP4 and IP6, however, in this class we will only be working with IP4.

6.) Outline of a TCP Client (Read Chapter 4 in Stevens)



Step 1: Create a socket :   int socket(int family, int type, int protocol)
(Read pgs 86-88 Stevens)

Creating a socket is in some ways similar to opening a file. This function creates a file descriptor and returns it from the function call. You later use this file descriptor for reading, writing and using with other socket functions

Parameters:
family: AF_INET   or   PF_INET   (These are the IP4 family)
type: SOCK_STREAM (for TCP)   or   SOCK_DGRAM (for UDP)
protocol: IPPROTO_TCP (for TCP) or IPPROTO_UDP (for UDP) or use 0

Step 2: Binding a socket: This is unnecessary for a client, what bind does is (and will be discussed in detail in the server section) is associate a port number to the application. If you skip this step with a TCP client, a temporary port number is automatically assigned, so it is just better to skip this step with the client.

Step 3: Connecting to a Server:
(Read pgs 89 – 90 Stevens)

```
int connect(int socket_file_descriptor, const struct sockaddr *ServerAddress, socklen_t AddressLength);
```

Once you have created a socket and have filled in the address structure of the server you want to connect to, the next thing to do is to connect to that server. This is done with the connect function listed above.

**This is one of the socket functions which requires an address structure so remember to type cast it to the generic socket structure when passing it to the second argument **

Connect performs the three-way handshake with the server and returns when the connection is established or an error occurs.

Once the connection is established you can begin reading and writing to the socket.

Step 4: Read and Writing to the socket will be discussed shortly
Step 5: Closing the socket will be discussed shortly

7.) Communicating with send and recv (Read pgs 48 – 49 , 77 – 81, 354 – 357 Stevens)

read/write
These are the same functions you use with files but you can use them with sockets as well. However, it is extremely important you understand how they work so please read Stevens carefully to get a full understanding.

Lets start with write()
```
int write(int file_descriptor, const void * buf, size_t message_length);
```

The return value is the number of bytes written. The number of bytes written may be less than the message_length. What this function does is transfer the data from you application to a buffer in the kernel on your machine, it does not directly transmit the data over the network. This is extremely important to understand otherwise you will end up with many headaches trying to debug your programs. TCP is in complete control of sending the data and this is implemented inside the kernel. Due to network congestion or errors, TCP may not decide to send your data right away, even when the function call returns. TCP has an elaborate sliding window mechanism which you will learn about in class to control the rate at which data is sent. Read pages 48-49, 77-78 in Stevens very carefully.

Now let us discuss reading from a socket.
```
int read(int file_descriptor, char *buffer, size_t buffer_length);
```

The value returned is the number of bytes read which may not be buffer_length!
As with write(), read() only transfers data from a buffer in the kernel to your application , you are not directly reading the byte stream from the remote host, but rather TCP is in control and buffers the data for

your application. Read Stevens very carefully, especially pages 77-78 to understand more how to properly use read.

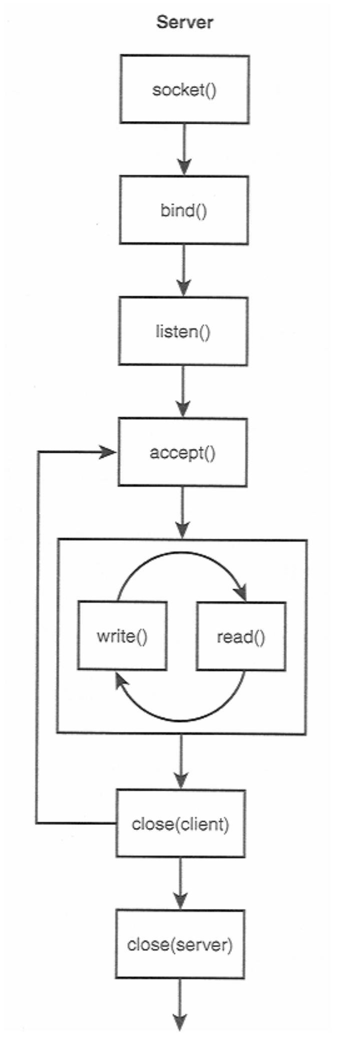
recv/send (Read pgs 354 – 357 Stevens)

8.) Shutting down sockets
Ater you are finished reading and writing to your socket you most call the close system call on the socket file descriptor just as you do on a normal file descriptor otherwise you waste system resources.

The close() function: int close(int filedescriptor);

The shutdown() function
You can also shutdown a socket in a partial way which is often used when forking off processes. You can shutdown the socket so that it won't send anymore or you could also shutdown the socket so that it won't read anymore as well. This function is not so important now but will be discussed in detail later. You can look at the man pages for a full description of this function.

9.) Outline of a TCP Server (Read Chapter 4 in Stevens).



Step 1: Creating a socket: Same as in the client
Step 2: Binding an address and port number
(Read pgs 91 – 93 Stevens)
int bind(int socket_file_descriptor, const struct sockaddr * LocalAddress, socklen_t AddressLength);

We need to associate an IP address and port number to our application. A client that wants to connect to our server needs both of these details in order to connect to our server. Notice the difference between this function and the connect() function of the client. The connect function specifies a remote address that the client wants to connect to, while here, the server is specifying to the bind function a local IP address of one of its Network Interfaces and a local port number.

**Again make sure that you cast the structure as a generic address structure in this function **

You also do not need to find information about the IP addresses associated with the host you are working on. You can specify: INNADDR_ANY to the address structure and the bind function will use on of the available (there may be more than one) IP addresses. Read Stevens page 92 for more details.

Step 3: Listen for incoming connections
(Read pgs 93 – 99 Stevens)

Binding is like waiting by a specific phone in your house, and Listening is waiting for it to ring.

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

The backlog parameter can be read in Stevens on page 94. It is important in determining how many connections the server will connect with.

Typical values for backlog are 5 – 10.

Step 4: Accepting a connection.
(Read pgs 99 –100 Stevens)

```
int accept (int socket_file_descriptor, struct sockaddr * ClientAddress, socklen_t *addrlen);
```

accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually used for listening for new incoming connections. Servers will be discussed in much more detail in a later lab.

Again, make sure you type cast to the generic socket address structure

Note that the last parameter is a pointer. You are not specifying the length, the kernel is and returning the value to your application, the same with the ClientAddress. After a connection with a client is established the address of the client must be made available to your server, otherwise how could you communicate back with the client? Therefore, the accept() function call fills in the address structure and length of the address structure for your use. Then accept() returns a new file descriptor, and it is this file descriptor with which you will read and write to the client.

10.) Handling Errors

When writing your programs you must account for and deal with errors related to any of the socket related functions! This means if a socket related function error occurs in your program, the program should exit nicely and display any useful information relating to the error it can give. For example, your program may call the bind() function trying to bind to a port which is not available. A value of –1 will be returned from this function indicating a failure, so you should catch such an error and display a proper error message. Of course, there is always the problem of cluttering up your code with error handling statements causing difficulty for someone trying to read the code and understand the flow. Therefore, some thought in terms of style should be given in dealing with how to incorporate error handling routines. Two common practices are as follows:

A.) A simple bailout function

B.) Wrapper Functions

You are not required to follow either of these practices, but if you create your own style in handling errors try to keep it simple yet effective.

11.) Gathering host information

Gathering information about your local host: int uname(struct utsname * buf)

```
struct utsname{
        char sysname[SYS_NMLN];
        char nodename[SYS_NMLN];
        char release[SYS_NMLN];
        char version[SYS_NMLN];
        char machine[SYS_NMLN];
        char domainname[SYS_NMLN];
};
```

see also :

- gethostname()
- getdomainname()

Gathering information about a remote host: struct hostent *gethostbyname(const char *name)

```
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*/
};
```

Example:

```
        struct hostent *ptr;

        ptr = gethostbyname(www.ucr.edu);

        for (int i = 0; ptr->h_aliases[i] != NULL; ++i)  printf("alias = %s\n", ptr->h_aliases[i]);
        if (ptr->h_addrtype == AF_INET)
                for (int i = 0; ptr->h_addr_list[i] != NULL; ++i)
                        printf("Address = %s\n", *(struct in_addr *) ptr->h_addr_list[i]);
```

12.) Summary of Functions

For specific and up-to-date information about each of the following functions, please use the online man pages and Steven's Unix Network Programming Vol. I.

Socket creation and destruction:

- socket()
- close()
- shutdown()

Client:
- connect()
- bind()

Server:
- accept()
- bind()
- listen()

Data Transfer:
- send()
- recv()
- write()
- read()

Miscellaneous:
- bzero()
- memset()

Host Information:
- uname()
- gethostbyname()
- gethostbyaddr()

Address Conversion:
- inet_aton()
- inet_addr()
- inet_ntoa()