Introduction to Socket Programming

Part I : TCP Clients, Servers; Host information

Keywords: sockets, client-server, network programming-socket functions, OSI layering, byte-ordering

Outline:

*****NOTE******

This introduction is not intended to be a thorough and in depth coverage of the sockets API but only to give a general outline of elementary TCP socket usage. Please refer to Richard Stevens book : "Unix Network Programming" Volume 1 for details about any of the functions covered here, and also use the online man pages for more specific details about each function.

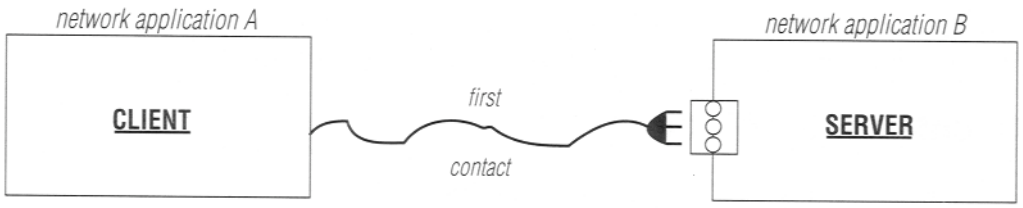
1.) Introduction

In this Lab you will be introduced to socket programming at a very elementary level. Specifically, we will focus on TCP socket connections which are a fundamental part of socket programming since they provide a connection oriented service with both flow and congestion control. What this means to the programmer is that a TCP connection provides a reliable connection over which data can be transferred with little effort required on the programmers part; TCP takes care of the reliability, flow control, congestion control for you. First the basic concepts will be discussed, then we will learn how to implement a simple TCP client and server.

2.) The Client / Server Model

It is possible for two network applications to begin simultaneously, but it is impractical to require it. Therefore, it makes sense to design communicating network applications to perform complementary network operations in sequence, rather than simultaneously. The server executes first and waits to

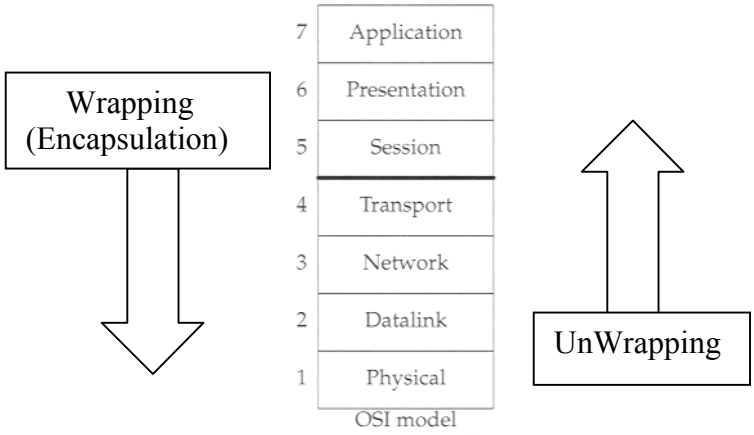
receive; the client executes second and sends the first network packet to the server. After initial contact, either the client or the server is capable of sending and receiving data.



A client initiates communications to a server.

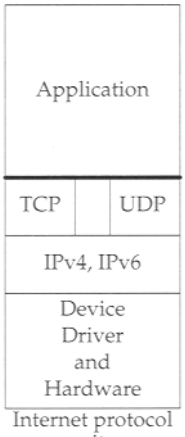
3.) The Socket Interface and Features of a TCP connection

The OSI Layers:



OSI model

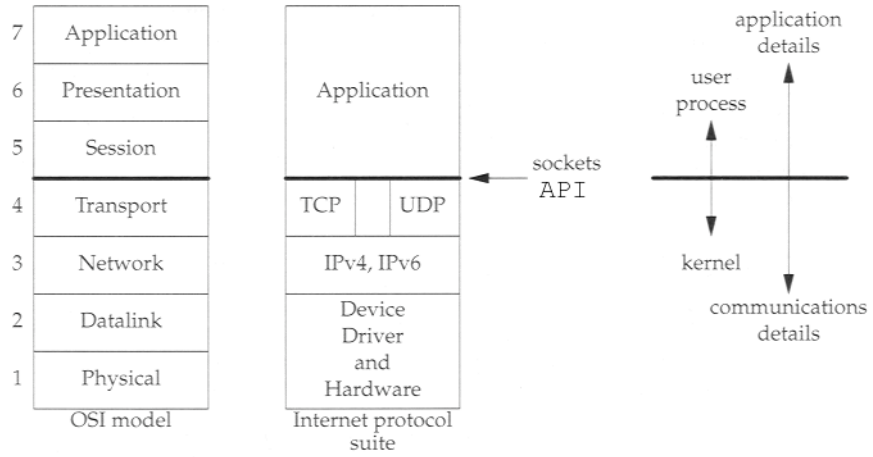
The Internet Layers:



Internet protocol suite

The Internet does not strictly obey the OSI model but rather merges several of the protocols layers together.

Where is the socket programming interface in relation to the protocol stack?



Features of a TCP connection:

- Connection Oriented
- Reliability
  1. Handles lost packets
  2. Handles packet sequencing
  3. Handles duplicated packets
- Full Duplex
- Flow Control
- Congestion Control

TCP versus UDP as a Transport Layer Protocol:

TCP	UDP
Reliable, guaranteed	Unreliable. Instead, prompt delivery of packets.
Connection-oriented	Connectionless
Used in applications that require safety gurantee. (eg. file applications.)	Used in media applications. (eg. video or voice transmissions.)
Flow control, sequencing of packets, error-control.	No flow or sequence control, user must handle these manually.
Uses byte stream as unit of transfer. (stream sockets)	Uses datagrams as unit of transfer. (datagram sockets)
Allows to send multiple packets with a single ACK.	
Allows two-way data exchange, once the connection is established. (full-duplex)	Allows data to be transferred in one direction at once. (half-duplex)
e.g. Telnet uses stream sockets. (everything you write on one side appears exaclt in same order on the other side)	e.g. TFTP (trivial file transfer protocol) uses datagram sockets.

TCP three-way Handshake: (Read pg 37 Stevens)

Sockets versus File I/O

Working with sockets is very similar to working with files. The socket() and accept() functions both return handles (file descriptor) and reads and writes to the sockets requires the use of these handles (file descriptors). In Linux, sockets and file descriptors also share the same file descriptor table. That is, if you open a file and it returns a file descriptor with value say 8, and then immediately open a socket, you will be given a file descriptor with value 9 to reference that socket. Even though sockets and files share the same file descriptor table, they are still very different. Sockets have addresses associated with them whereas files do not, notice that this distinguishes sockets form pipes, since pipes do not have addresses with which they associate. You cannot randomly access a socket like you can a file with lseek(). Sockets must be in the correct state to perform input or output.

File I/O	*Network I/O*
open a file	open a socket
	name the socket
	associate with another socket
read and write	send and receive between sockets
close the file	close the socket

4.) Byte Ordering

Port numbers and IP Addresses (both discussed next) are represented by multi-byte data types which are placed in packets for the purpose of routing and multiplexing. Port numbers are two bytes (16 bits) and IP4 addresses are 4 bytes (32 bits), and a problem arises when transferring multi-byte data types between different architectures. Say Host A uses a "*big-endian*" architecture and sends a packet across the network to Host B which uses a "*little-endian*" architecture. If Host B looks at the address to see if the packet is for him/her (choose a gender!), it will interpret the bytes in the opposite order and will wrongly conclude that it is not his/her packet. ***The Internet uses big-endian*** and we call it the *network-byte-order*, and it is really not important to know which method it uses since we have the following functions to convert host-byte-ordered values into network-byte-ordered values and vice versa:

To convert port numbers (16 bits):

Host -> Network

```
unit16_t   htons( uint16_t  hostportnumber  )
```

Network -> Host

```
unit16_t   ntohs( uint16_t  netportnumber  )
```

To convert IP4 Addresses (32 bits):

Host -> Network

unit32_t htonl(uint32_t hostportnumber)

Network -> Host

Unit32_t ntohl(uint32_t netportnumber)

5.) Address Structures, Ports, Address conversion functions

Overview of IP4 addresses:

IP4 addresses are 32 bits long. They are expressed commonly in what is known as dotted decimal notation. Each of the four bytes which makes up the 32 address are expressed as an integer value (0 – 255) and separated by a dot. For example, 138.23.44.2 is an example of an IP4 address in dotted decimal notation. There are conversion functions which convert a 32 bit address into a dotted decimal string and vice versa which will be discussed later.

Often times though the IP address is represented by a domain name, for example, hill.ucr.edu. Several functions described later will allow you to convert from one form to another (Magic provided by DNS!).

The importance of IP addresses follows from the fact that each host on the Internet has a unique IP address. Thus, although the Internet is made up of many networks of networks with many different types of architectures and transport mediums, it is the IP address which provides a cohesive structure so that at least theoretically, (there are routing issues involved as well), any two hosts on the Internet can communicate with each other.

Ports:

Sockets are UNIQUELY identified by Internet address, end-to-end protocol, and port number.
That is why when a socket is first created it is vital to match it with a valid IP address and a port number.
In our labs we will basically be working with TCP sockets.

Ports are software objects to multiplex data between different applications. When a host receives a packet, it travels up the protocol stack and finally reaches the application layer. Now consider a user running an ftp client, a telnet client, and a web browser concurrently. To which application should the packet be delivered? Well part of the packet contains a value holding a port number, and it is this number which determines to which application the packet should be delivered.

So when a client first tries to contact a server, which port number should the client specify? For many common services, standard port numbers are defined.

Port	*Service Name, Alias*	*Description*
1	`tcpmux`	TCP port service multiplexer
7	`echo`	Echo server
9	`discard`	Like `/dev/null`
13	`daytime`	System's date/time
20	`ftp-data`	FTP data port
21	`ftp`	Main FTP connection
23	`telnet`	Telnet connection
25	`smtp, mail`	UNIX mail
37	`time, timeserver`	Time server
42	`nameserver`	Name resolution (DNS)
70	`gopher`	Text/menu information
79	`finger`	Current users
80	`www, http`	Web server

Ports 0 – 1023, are reserved and servers or clients that you create will not be able to **bind** to these ports unless you have root privilege.

Ports 1024 – 65535 are available for use by your programs, but beware other network applications maybe running and using these port numbers as well so do not make assumptions about the availability of specific port numbers. Make sure you read Stevens for more details about the available range of port numbers!

Address Structures:

Socket functions like connect(), accept(), and bind() require the use of specifically defined address structures to hold IP address information, port number, and protocol type. This can be one of the more confusing aspects of socket programming so it is necessary to clearly understand how to use the socket address structures. The difficulty is that you can use sockets to program network applications using different protocols. For example, we can use IP4, IP6, Unix local, etc. Here is the problem: Each different protocol uses a different address structure to hold its addressing information, yet they all use the same functions connect(), accept(), bind() etc. So how do we pass these different structures to a given socket function that requires an address structure? Well it may not be the way you would think it should be done and this is because sockets where developed a long time ago before things like a void pointer where features in C. So this is how it is done:

There is a generic address structure: struct sockaddr

This is the address structure which must be passed to all of the socket functions requiring an address structure. ***This means that you must type cast*** your specific protocol dependent address structure to the generic address structure when passing it to these socket functions.

Protocol specific address structures usually start with *sockaddr_* and end with a ***suffix*** depending on that protocol. For example:

struct sockaddr_in (IP4, think of in as internet)
struct sockaddr_in6 (IP6)
struct sockaddr_un (Unix local)
struct sockaddr_dl (Data link)

We will be only using the IP4 address structure: struct sockaddr_in.
So once we fill in this structure with the IP address, port number, etc we will pass this to one of our socket functions and we will need to type cast it to the generic address structure. For example:

```
struct sockaddr_in  myAddressStruct;

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

connect(socket_file_descriptor, (struct sockaddr *) &myAddressStruct,
sizeof(myAddressStruct));
```

Here is how to fill in the sockaddr_in structure:

```
struct sockaddr_in{

      sa_family_t  sin_family       /*Address/Protocol Family*/ (we'll use PF_INET)
      unit16_t       sin_port       /* 16-bit Port number   --Network Byte Ordered--
*/
      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 Ordered-- */
};
```

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

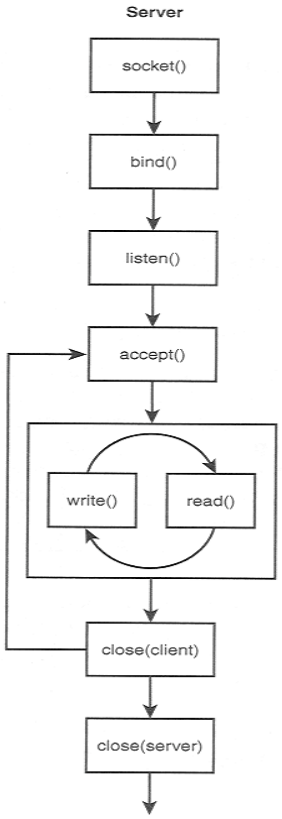
To convert a string dotted decimal IP4 address to a NETWORK BYTE ORDERED 32 bit value use the functions:

- inet_addr()
- inet_aton()

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

- inet_ntoa()

6.) Outline of a TCP Server:



Step 1: Creating a socket:

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

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 an address and port number

```
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.

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *bind()* is 0 for success and –1 for failure.

**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. This ensures that connections to a specified port will be directed to this socket, regardless of which Internet address they are sent to. This is useful if host has multiple IP addresses, then it enables the user to specify which IP address will be b_nded to which port number.

Step 3: Listen for incoming connections

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' book. It is important in determining how many connections the server will connect with. Typical values for backlog are 5 – 10.

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *listen()* is 0 for success and –1 for failure.

Step 4: Accepting a connection.

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

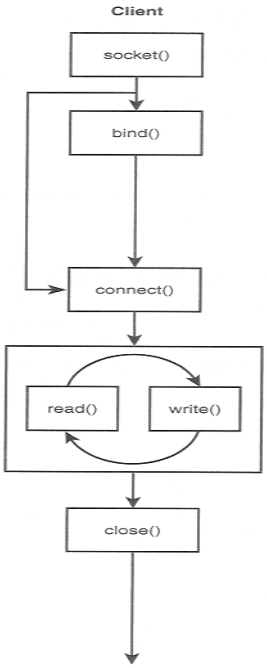
accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually used for listening for new incoming connections. Servers will be discussed in much more detail in a later lab.

It dequeues the next connection request on the queue for this socket of the server. If queue is empty, this function blocks until a connection request arrives. (read the reference book *TCP/IP Implementation in C* for more details.)

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.

7.) Outline of a TCP Client



Step 1: Create a socket : Same as in the server.

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:

```
 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.

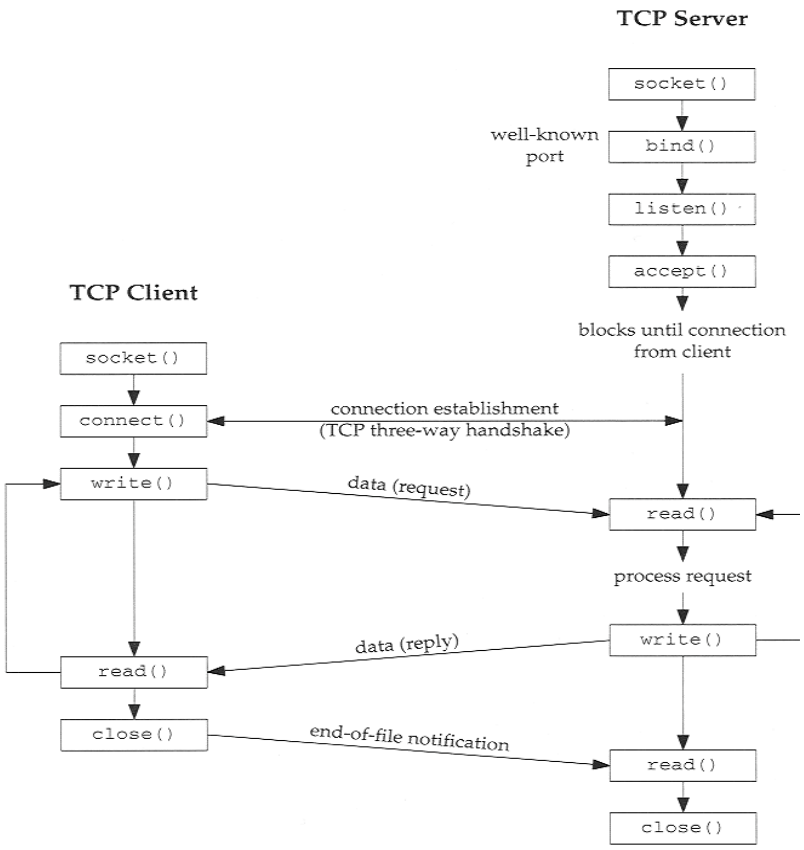
**This is one of the socket functions which requires an address structure so remember to type cast it to the generic socket structure when passing it to the second argument **

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

8.) Outline of a client-server network interaction:



Communication of 2 pairs via sockets necessitates existence of this 4-tuple:

- Local IP address
- Local Port#
- Foreign IP address
- Foreign Port#

!!!! When a server receives (accepts) the client's connection request => it *fork*s a copy of itself and lets the child handle the client. (make sure you remember these Operating Systems concepts) Therefore on the server machine, listening socket is distinct from the connected socket.

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.

Writing to a socket:

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

The return value is the number of bytes written, and –1 for failure. 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.

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! It returns –1 for failure. 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.

Shutting down sockets:

After 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.

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()