Introduction to Socket Programming

Part I

Outline

1.) Introduction
2.) The Client / Server Model
3.) The Socket Interface and Features of a TCP connection
4.) Byte Ordering
5.) Address Structures, Ports, Address conversion functions
6.) Outline of a TCP Client
7.) Communicating with (send and recv)/(write and read)
8.) Shutting down sockets
9.) Outline of a TCP Server
10.) Handling Errors, Loopback
11.) Gathering host information
12.) Summary of Socket Functions

*****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.

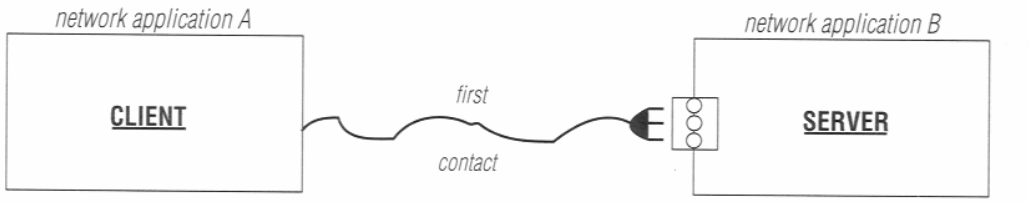
1.) Introduction

In this Lab you will be introduced to socket programming at a very elementary level. Specifically, we will focus on TCP socket connections which will 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 and then if time permitted we will build a simple TCP client and server.

For Some History of Sockets: (Read pgs 18 – 20 Stevens)

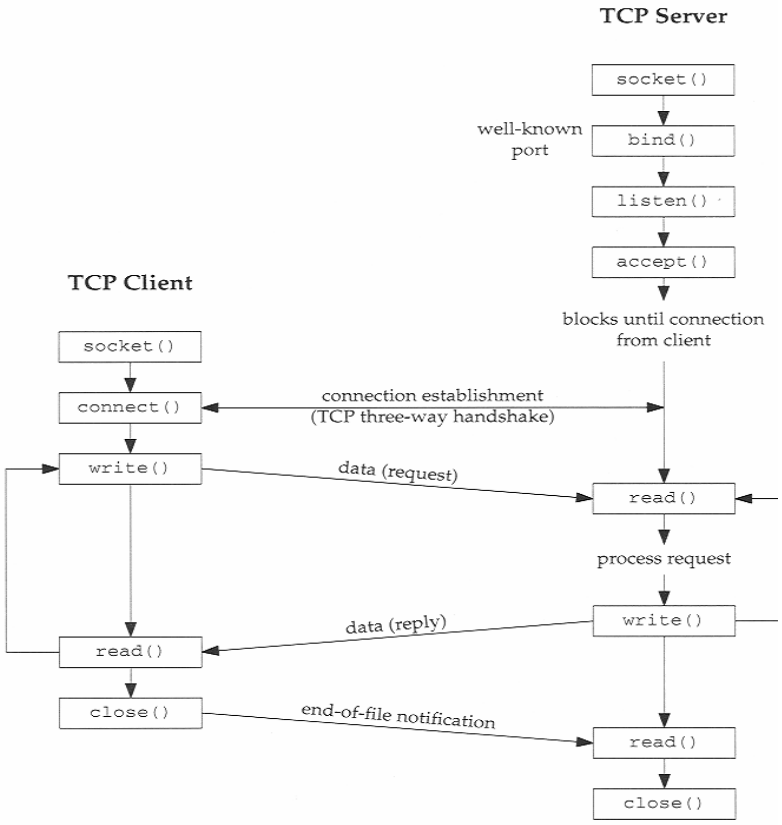
2.) The Client / Server Model

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.



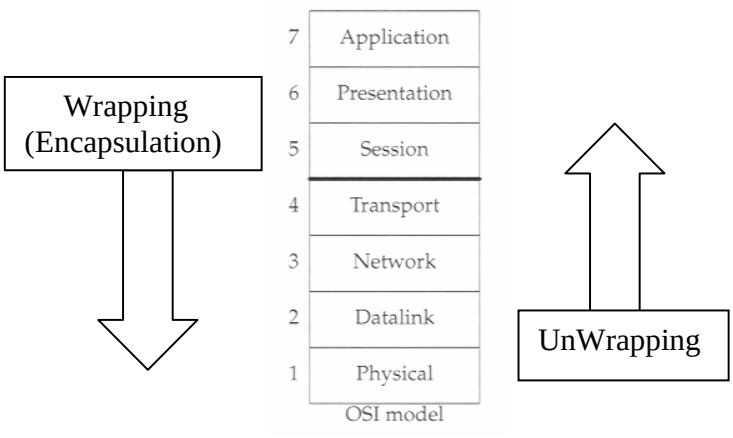
A client initiates communications to a server.

Outline of a client-server network interaction:

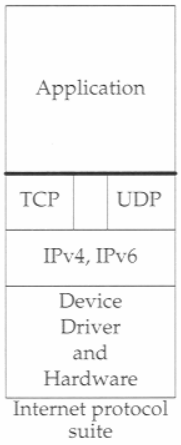


3.) The Socket Interface and Features of a TCP connection

The OSI Layers:

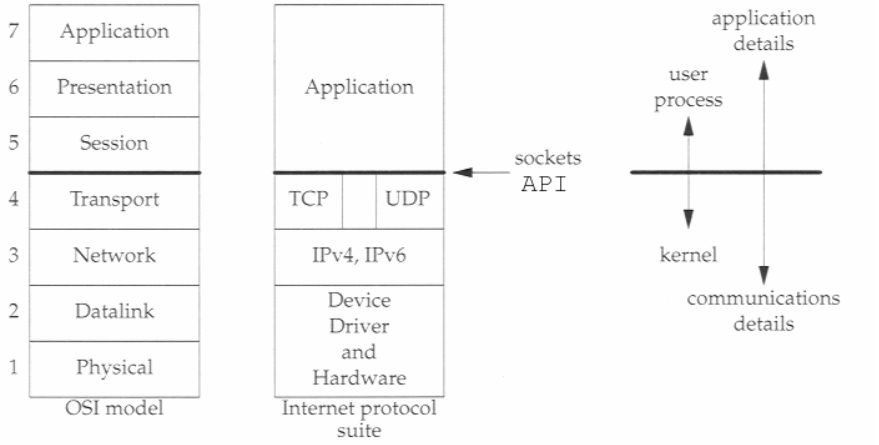


The Internet Layers:



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: (Read pgs 32 – 41 Stevens)

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

TCP three-way Handshake: (Read pg 34 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 (Read pgs 66-68 in Stevens)

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: (Read pgs 41-43 in Stevens)

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: (Read pgs 69 - 70 , 58 – 63 Stevens) Read Carefully!

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 and 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.