Introduction to Socket Programming

1. Sockets

A socket is a conceptual endpoint for a logical network between two processes. A socket is a 'point' where data can be sent to or received from another process. a socket can send and receive data from a 'remote' process, i.e., a process that runs on a different physical 'host', connected to our host through a computer network.

TCP and UDP protocols

There are two major protocols used for network traffic in the transport layer in the Internet. They are namely, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Both of them reside in the Transport Layer of OSI model. Transport layer is the layer that focuses on process to process communication. i.e., breaking the message into manageable pieces (packets) and multiplexing data between processes.

The difference between the two in the view of the programmer is the reliability of the two. TCP aims to provide an error free and a reliable data transfer while the UDP expects the upper layers to treat the data transfer as inherently unreliable and handle the errors and ordering of packets.

To provide the reliability, TCP creates a 'circuit', essentially a path for the packets to travel to the destination. This preserves the ordering of the packets and prevents duplication. Therefore, these are guaranteed when using TCP.

UDP on the other hand does not create such a path. The packets are on their own to figure out a path from the source to destination. This might result in packet drops, duplication of packets and out of order sequencing of packets. Therefore UDP does not provide any guarantee for any of these.

Both of these protocols are implemented on top of Internet Protocol (IP). A basic protocol in Internet Layer of the OSI model. This is the layer that focuses on host to host communication. Internet Protocol essentially treats the underlying network as unreliable and does not provide any error checking or sequencing services.

2. Creating a UDP server

A server is the one who waits for a client to connect to. Therefore the client should know the ip address and the port of the server. But the server does not have to know the clients' address. The following steps should be taken to write a UDP server.

Algorithm

1. Create a socket with appropriate options for UDP.
2. Initialize a `sockaddr_in` struct with appropriate options for our server address.
3. Bind the socket with the created `sockaddr_in` struct.
4. Send and receive messages from clients that `sockaddr_in` struct is configured to use.

```
/* Sample UDP server */

#include <sys/socket.h>
#include <netinet/in.h>
#include <strings.h>
#include <stdio.h>

int main(int argc, char**argv)
{
      int sockfd,n;
      struct sockaddr_in servaddr, cliaddr;
      socklen_t len;
      char mesg[1000];
```

```
        char* banner = "Hello UDP client! This is UDP server";

        sockfd=socket(AF_INET,SOCK_DGRAM,0);

        servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
        servaddr.sin_port=htons(32000);
        bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));

        len = sizeof(cliaddr);
        n = recvfrom(sockfd,mesg,1000,0,(struct sockaddr *)&cliaddr,&len);
        sendto(sockfd,banner,n,0,(struct sockaddr *)&cliaddr,sizeof(cliaddr));
        mesg[n] = 0;
        printf("Received: %s\n",mesg);
        return 0;
}
```

The `socket()` call

The `socket()` function call creates a socket. The return value of this call is an integer called socket descriptor. In UNIX like systems, this is actually a file descriptor and common operations on file descriptors such as `read()` and `write()` still work with it. Additionally, specialized functions such as send, `sendto()`, `sendmsg()`, `recv()`, `recvfrom()` and `recvmsg()` can also be used with socket descriptors to send and receive data.

To create a socket three arguments are needed. First, a domain. Different domains exist. For example AF_UNIX option is used to communicate in local domain (i.e., inside the local host iteself) and AF_INET6 option is used to communicate using IP v6. The option `AF_INET` specifies that we're using IP v4 to communicate.

The next argument specifies the type of the socket. `SOCK_DGRAM` specifies that the socket is of type UDP.

The last argument specifies a protocol. This is useful if more than one protocol is supported by the specified type of socket. In this case, there is only one. Therefore this number is kept as zero.

Input output of UNIX systems
Most of the I/O devices are treated as files in UNIX systems. A common interface, `read()` and `write()` is provided to the application programmer by the kernel to deal with these.

The `sockaddr_in` struct

This struct stores IP socket address. An IP socket address contains an IP address of an interface as well as a port number. It contains the following fields:

1. `sin_family`: specifies an address family. Must be `AF_INET`
2. `sin_addr`: stores a struct of in_port_t which contains only one field: `s_addr` which stores the IP address in the network byte order

   The network and host byte orders and `htonl()` and `htons()` functions

   There are two ways that stored in the memory of a computer can be interpreted. They are: the value in the lowest address considered as the most significant byte (named Big Endian) and the value in the lowest address considered as the least significant byte (named Little Endian). This is called the 'byte order' of the architecture. The x86 and x86_64 architectures are little endian while the network communication is big endian. Whichever the byte order of the two are, `htonl()` and `htons()` functions converts the byte order of the host machine to the byte order of the network. In the struct, it is the convention to store the port number in the network byte order.

3. `sin_port` : Stores the port to be used in the network byte order.

`Bind()` call

This gives the socket a 'name' with the given IP address. This basically registers that the socket is using that IP address. Since the server is the initial receiver of the data, this is an essential step. The bind `()` function expects the first argument to be in `struct sockaddr*` type. This is essentially the same as `struct sockaddr_in` in most applications and the casting is done just to get rid of a possible warning.

In the program, you might notice that, `recv` from has an uninitialized argument `cliaddr` of type `struct sockaddr_in*` as the second argument. The use of `recvfrom` over `read` or `recv` is that, once the data is received, it fills the fields of `cliaddr` with the information of the source the data is received from. This way, the server could figure out the ip address information of the sender and reply back to it.

As with `read(), recv()` is always blocking.

3. Creating a UDP client

Algorithm

1. Create a socket
2. Initialize a `sockaddr_in` struct with the information of the server intended to connect to
3. Send and receive messages

```
/* Sample UDP client */
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv)
{
      int sockfd,n;
      struct sockaddr_in servaddr;
      char sendline[] = "Hello UDP server! This is UDP client";
      char recvline[1000];

      if (argc != 2)
      {
            printf("usage:  ./%s <IP address>\n",argv[0]);
            return -1;
      }

      sockfd=socket(AF_INET,SOCK_DGRAM,0);

      bzero(&servaddr,sizeof(servaddr));
      servaddr.sin_family = AF_INET;
      servaddr.sin_addr.s_addr=inet_addr(argv[1]);
      servaddr.sin_port=htons(32000);

      sendto(sockfd,sendline,strlen(sendline),0,(struct sockaddr
                                          *)&servaddr,sizeof(servaddr));
      n=recvfrom(sockfd,recvline,10000,0,NULL,NULL);
      recvline[n]=0;
      printf("Received: %s\n",recvline);
      return 0;
}
```

Unlike the server, the client does not have to bind the socket since the server does not need to know the clients' ip address initially before the client initiate the communication with the server.

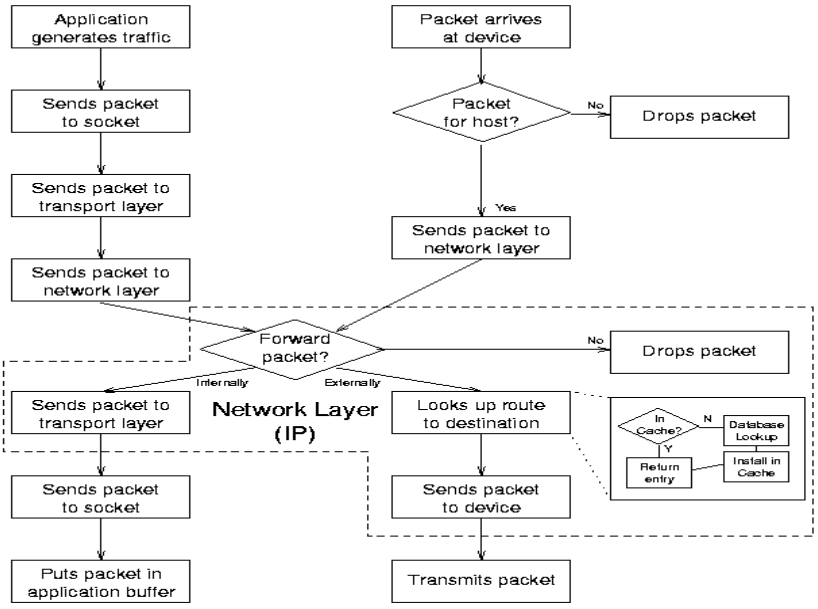
Exercises – 1

1. A server does not reply only to one client and exit. It waits indefinitely, replying for all clients until it is forced to terminate by the host computer. Modify the server to do this
2. An echo server is a simple server that receives a message from a client and sends the same message back to the client. Write a simple echo server which achieves this task.
3. Write a server that uses the hypothetical Capitalizer Service Protocol (CSP) to provide a capitalizer service. CSP works as follows:
   - First the client initializes the connection by sending an integer which represents the number of sentences it is going to send. (Let's say the number is n) the CSP server sends the characters 'ack' back to the client, acknowledging the message.
   - CSP server then sits in a while loop that iterates 'n' times. At each iteration the server does the following:
     - receive a sentence from the client.
     - Capitalize it
     - send it back to the client.
4. Write a time server that, when connected, sits in a loop and sends the current time each second to the receiver

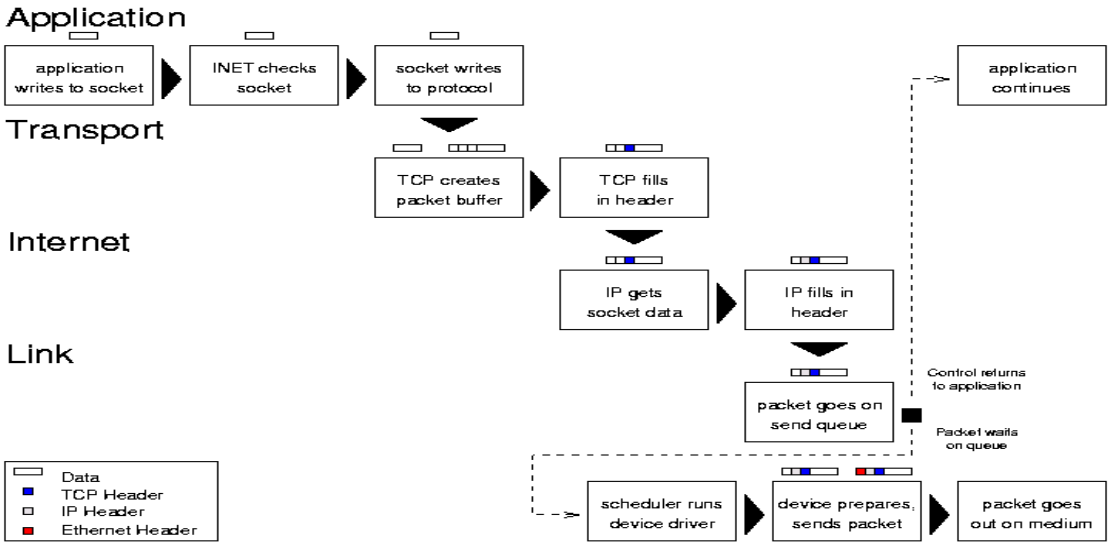
4. Client Server Model

The server is the one who initially wait for the client. The need for such a process as a server is because there is no method to invoke a program by a message through a network. Therefore, if two programs are to communicate through the network, both of them should be running at the same time for the message to get passed. Since the two programs reside on two physically distant two hosts, there is no way of invoking one program when the other is invoked. Therefore, one of them should start first and wait indefinitely until the other one decides to communicate. The one who waits is called server while the other one is called the client.

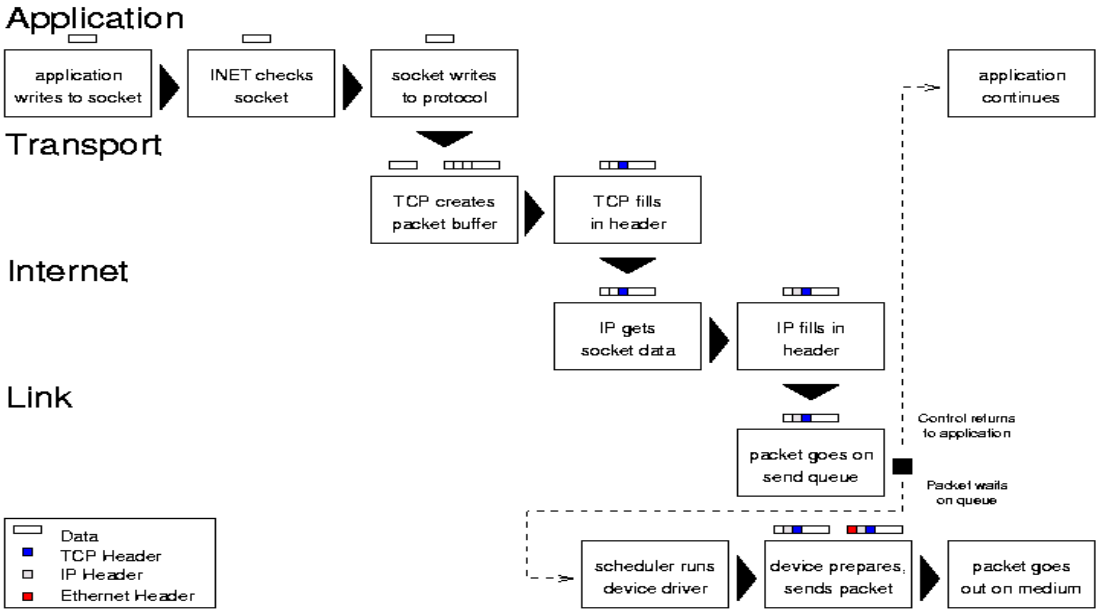
A message received through the network is first received by the operating system kernel. Most kernels have different restrictions on packets and ports. The following diagram summarizes the process.



The following diagram summarizes how the host kernel handles packets from the application to the point where the packet is delivered to the outside media.



The following diagram summarizes how the host kernel handles packets from outside media to the point where the packet is delivered to the application.



TCP

As we discussed earlier, TCP and UDP are two protocols in the transport layer of the Internet Protocol Suite. We discussed UDP last time and in this lab, we will look at TCP.

TCP is a protocol that ensures packet ordering and reliability. To maintain these qualities, a 'virtual circuit' or a path is created between the server and host. This makes the implementation of TCP significantly complex than UDP.

5. Creating a TCP server

Algorithm:

1. Create a socket with appropriate options for TCP.
2. Initialize a `sockaddr_in` struct with appropriate options for our server address.
3. Bind the socket with the created `sockaddr_in struct.`

4. Listen for any incoming connections.
5. Accept connections.
6. Send and receive messages.

```
 /* Sample TCP server */

#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv)
{
      int listenfd,connfd,n;
      struct sockaddr_in servaddr,cliaddr;
      socklen_t clilen;
      char* banner = "Hello TCP client! This is TCP server";
      char buffer[1000];

      /* one socket is dedicated to listening */
      listenfd=socket(AF_INET,SOCK_STREAM,0);

      /* initialize a sockaddr_in struct with our own address information for
      binding the socket */
      servaddr.sin_family = AF_INET;
      servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
      servaddr.sin_port=htons(32000);

      /* binding */
      bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
      listen(listenfd,0);
      clilen=sizeof(cliaddr);

      /* accept the client with a different socket.  */
      connfd = accept(listenfd,(struct sockaddr *) &cliaddr, &clilen); // the
      uninitialized cliaddr variable is filled in with the
      n = recvfrom(connfd,buffer,1000,0,(struct sockaddr *)&cliaddr,&clilen);//
information of the client by recvfrom function
      buffer[n] = 0;
      sendto(connfd,banner,strlen(banner),0,(struct sockaddr
*)&cliaddr,sizeof(cliaddr));
      printf("Received:%s\n",buffer);
      return 0;
}
```

The `bind()` call

This gives the socket a 'name' with the given IP address. This basically registers that the socket is using that IP address. Since the server is the initial receiver of the data, this is an essential step. The bind `()` function expects the first argument to be in `struct sockaddr*` type. This is essentially the same as `struct sockaddr_in` in most applications and the casting is done just to get rid of a possible warning. Bind is not a blocking call.

The `listen()` call

The `listen()` call makes the given socket a passive socket. i.e., this socket is used only for wait for a connection from a client and accepts it. It does not pass any other messages to the client. The remainder of the communication should be done by some other socket. This cannot be done with a UDP server.

The first argument is the socket and the second argument specifies the number of connections to be queued. More than one client could try to connect to the server at a time, but one TCP socket could only serve one client. Therefore, the `listen()` call queues connection requests until a socket accepts the connection.
Listen is not blocking.

The `accept()` call

The `accept()` call gets the first client out of the listen() queue and creates a connection with it. The uninitialized cliaddr struct is initialized by the accept() function with the information of the incoming connection which can be used to communicate with the client. Accept is blocking function, i.e., the function does not return until the connection is not established.

6. Creating a TCP client

Algorithm:

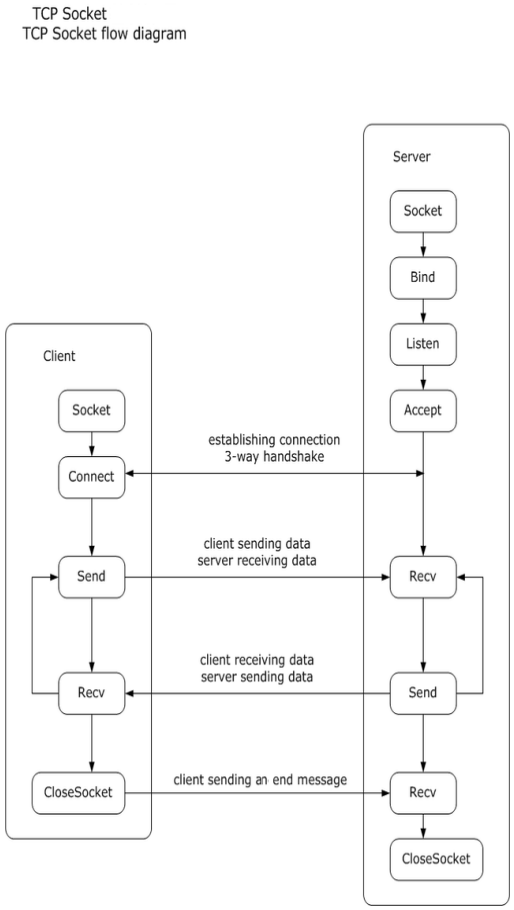
1. Create a socket with appropriate options for TCP.
2. Initialize a sockaddr_in struct with appropriate options for the server address.
3. Connect to the server.
4. Send and receive messages.

```
/* Sample TCP client */

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

int main(int argc, char**argv)
{
      int sockfd,n;
      struct sockaddr_in servaddr;
      char banner[] = "Hello TCP server! This is TCP client";
      char buffer[1000];

      if (argc != 2)
      {
            printf("usage:  ./%s <IP address>\n",argv[0]);
            return -1;
      }
      /* socket to connect */
      sockfd=socket(AF_INET,SOCK_STREAM,0);

      /* IP address information of the server to connect to */
      servaddr.sin_family = AF_INET;
      servaddr.sin_addr.s_addr=inet_addr(argv[1]);
      servaddr.sin_port=htons(32000);

      connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

      sendto(sockfd,banner,strlen(banner),0, (struct sockaddr
                                          *)&servaddr,sizeof(servaddr));
      n=recvfrom(sockfd,buffer,10000,0,NULL,NULL);
      buffer[n]=0;
      printf("Received:%s\n",buffer);
      return 0;
}
```

The `Connect()` call

Creates a connection between the socket and the address. When a UPD client calls connect(), the communication happens only with the given server alone until another connect() call is made. A UDP client may call connect() any number of times while a TCP client could call it successfully only once on one socket. Connect call is blocking, which means, until server accepts the connection, the client won't proceed with the rest of the program.

The following diagram shows a summary of the steps taken for the communication between the server and the client



Exercises – 2

Write a server similar to an FTP server which sends a file over a network using a TCP connection along with its client. The client and the server follow the following protocol:

- For now, let's just assume that the server has only one specific file called 'serverfile.txt'.
- The client connects and sends 'request' string, requesting the file 'serverfile.txt' from the server. Server responds with the size of the file (you can hard code the size of the file for now)
- Server then sits in a loop, sending the file, 1000 bytes at a time.
- Upon receiving each 1000 bytes, the client appends the received part to a file called 'serverfile.txt'.
- The client disconnects upon receiving the full file