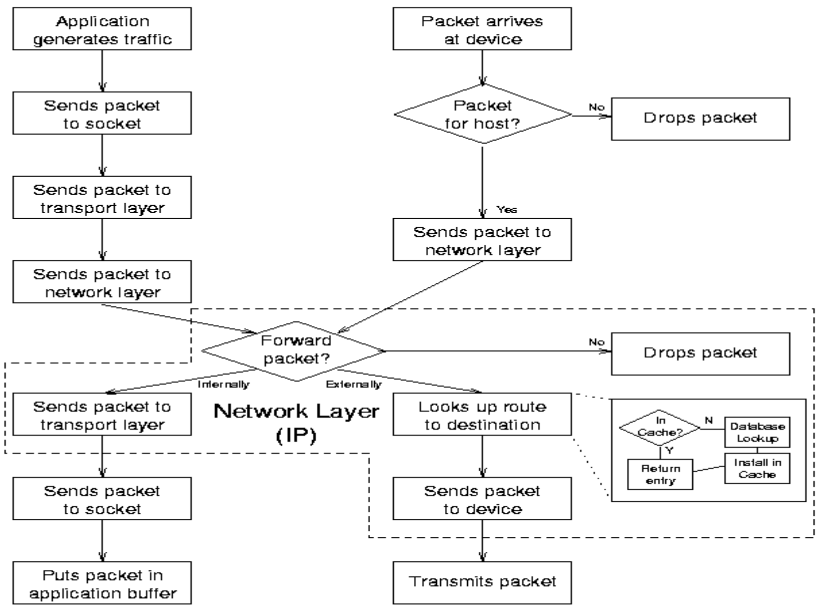
Introduction to Socket Programming Part II

In the last lab we learned what a socket is and looked at an example of a UDP server and a UDP client. In today's lab session we will look at the TCP server and TCP client.

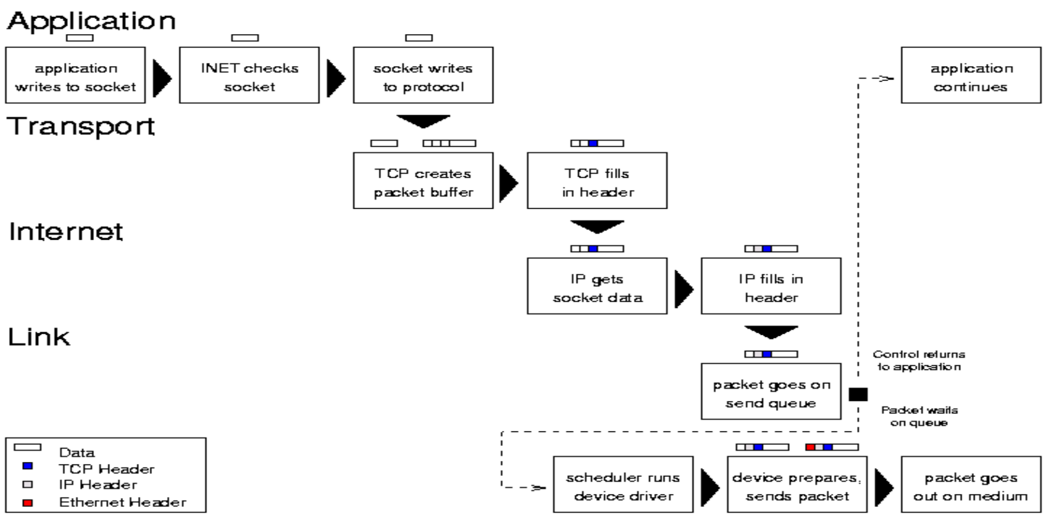
Client Server Model

In the last lab session we discussed clients and servers. 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.

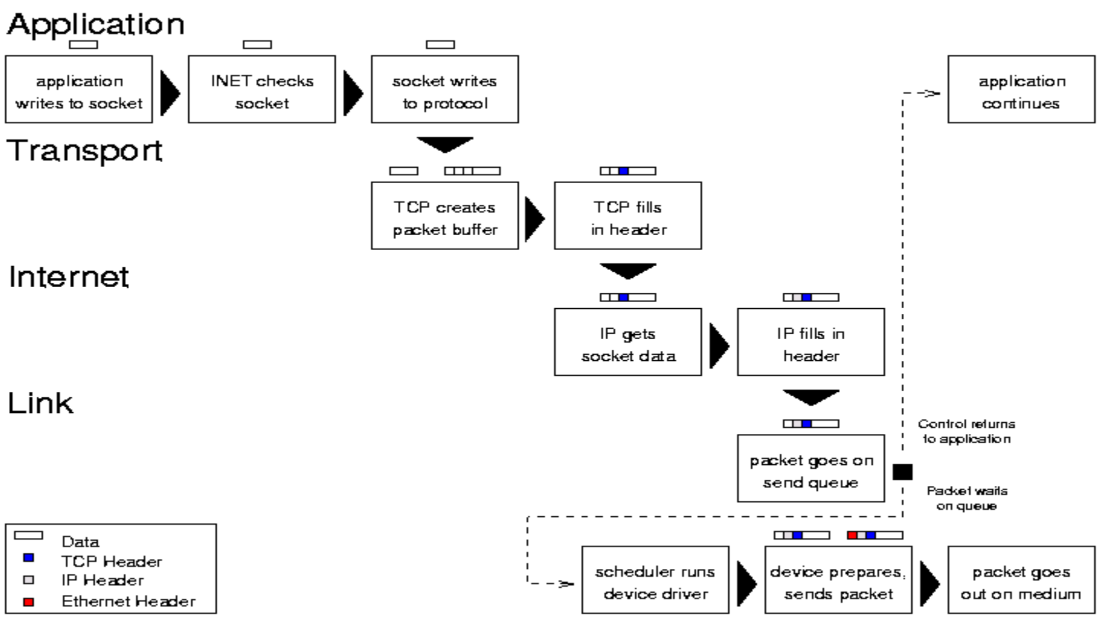


The following diagram summerizes how the host kernel handles packets from the application to the point where the packet is delivered to the outside media.



Application
application writes to socket
INET checks socket
socket writes to protocol
application continues
Transport
TCP creates packet buffer
TCP fills in header
Internet
IP gets socket data
IP fills in header
Link
packet goes on send queue
Control returns to application
Packet waits on queue
scheduler runs device driver
device prepares, sends packet
packet goes out on medium
Data
TCP Header
IP Header
Ethernet Header

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


Application
application writes to socket
INET checks socket
socket writes to protocol
application continues
Transport
TCP creates packet buffer
TCP fills in header
Internet
IP gets socket data
IP fills in header
Link
packet goes on send queue
Control returns to application
Packet waits on queue
scheduler runs device driver
device prepares, sends packet
packet goes out on medium
Data
TCP Header
IP Header
Ethernet Header

TCP
As we discussed in the last lab, 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.

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 accept 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.

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.

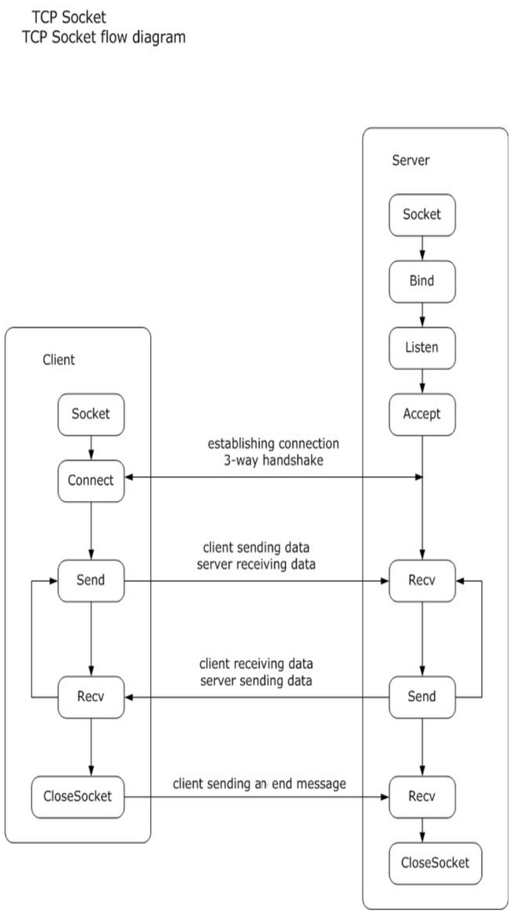
```
/* Sample TCP client */

#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

1. 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 receving each 1000 bytes, the client appends the received part to a file called 'serverfile.txt'.

The client disconnects upon receiving the full file.