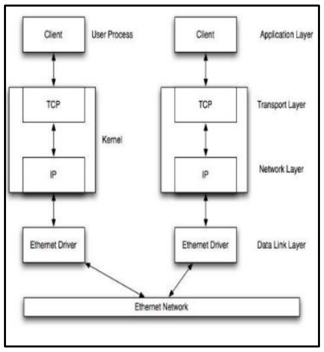
## UNIT-1

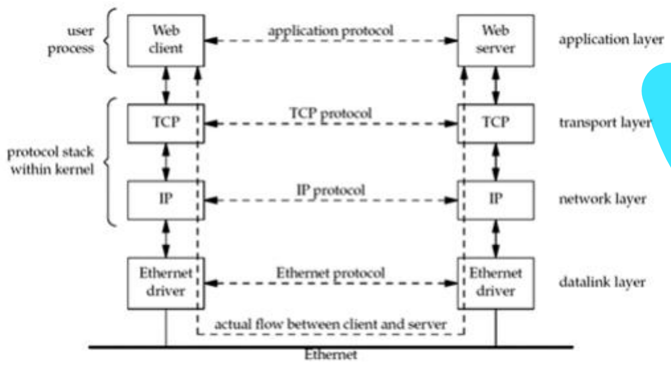
**1. What is network programming? With neat diagram, Explain the Client and Server Communication over LAN and WAN.**

**Ans: Network programming** involves writing programs to communicate with processes either on the same or on other machines on the network using standard Protocols. High-level decision must be made as to which program would initiate the communication first and when responses are expected.

### Communication over LAN:

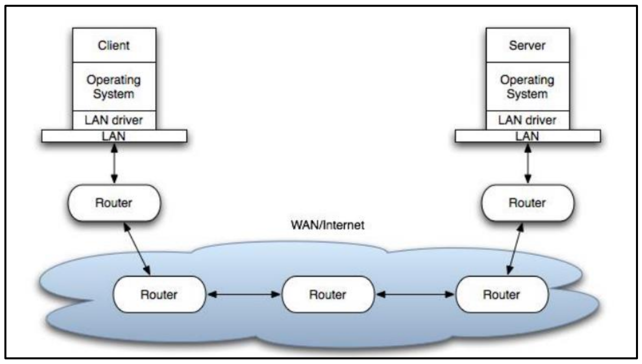
- If the client and server are on the same Ethernet, we would have the arrangement shown in Figure.
- The client and server communicate using an application protocol, however the transport layers communicate using TCP.
- The actual flow of information between client and server goes down the protocol stack on one side, across the network, and up the protocol stack on the other side.
- Client & Server are typically user processes, while TCP and IP protocols are normally part of the protocol stack within the kernel.
- The four layers labelled in the diagram are the Application layer, Transport layer, Network layer, Data-link layer.
- Some clients and servers use the User Datagram Protocol (UDP) instead of TCP.

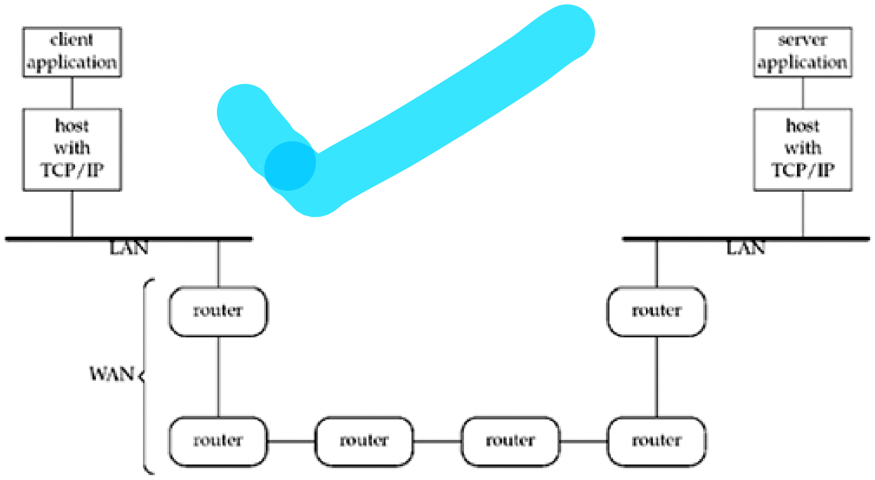




## Communication over WAN:

- The client & server on different LANs is connected to a Wide Area Network (WAN) via a router.
- Routers are the building blocks of WANs.
- The largest WAN today is the Internet.
- Many companies build their own WANs and these private WANs may or may not be connected to the Internet.





**2. What is a netstat? List the command with at least 5 options along with the sample output with a brief description on the nature of the output produced. (07 marks)**

**Ans:** Netstat is a command line network utility tool used to display network connections, routing tables, interface statistics, masquerade connections and other network related information on Unix like OS.

This program serves multiple purposes:

- It shows the status of networking endpoints.
- It shows the multicast groups that a host belongs to on each interface.
- It shows the per-protocol statistics with the -s option.
- It displays the routing table with the -r option and the interface information with the -i option.

**1. Display All Active Network Connections:**

**netstat -a**

Sample Output:

```
Active Internet connections (servers and established)

Proto Recv-Q Send-Q Local Address          Foreign Address         State

tcp        0      0 *:ssh                  *:*                     LISTEN

tcp        0      0 localhost:ipp          *:*                     LISTEN

tcp        0      0 localhost:smtp         *:*                     LISTEN
```

**2. Display Listening Ports:**

**netstat -l**

Sample Output:

```
Active Internet connections (only servers)

Proto Recv-Q Send-Q Local Address          Foreign Address         State

tcp        0      0 *:ssh                  *:*                     LISTEN
```

**3. Display Network Statistics:**

**netstat -s**

Sample Output:

```
Ip:

    12345 total packets received
```

```
12 with invalid headers

0 forwarded
```

**4. Display Routing Table:**

**netstat -r**

Sample Output:

```
Kernel IP routing table

Destination  Gateway      Genmask       Flags  MSS Window  irtt Iface

default      router       0.0.0.0       UG     0 0         0 eth0

192.168.1.0  0.0.0.0      255.255.255.0 U      0 0         0 eth0
```

**5. Display PID and Program Names:**

**netstat -p**

Sample Output:

```
Proto Recv-Q Send-Q Local Address        Foreign Address      State      PID/Program name

tcp      0      0 *:ssh                 *:*                  LISTEN     1234/sshd
```

**3. Write a program to implement TCP daytime client.**

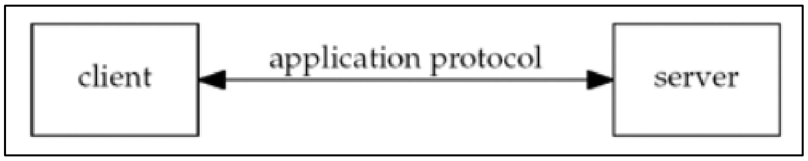
**Ans:**

```
#include "unp.h"
int main(int argc, char **argv){
        int sockfd, n;
        char recvline[MAXLINE + 1];
        struct sockaddr_in servaddr;
        if (argc != 2)
                err_quit("usage: a.out ");
        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                err_sys("socket error");
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(13);
        if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
                err_quit("inet_pton error for %s", argv[1]);
        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
                err_sys("connect error");
        while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
                recvline[n] = 0;
                if (fputs(recvline, stdout) == EOF)
                        err_sys("fputs error");
        }
        if (n < 0)
                err_sys("read error");
        exit(0);
}
```

**4. What is network protocol? With a neat block diagram explain the network application for client and server.**

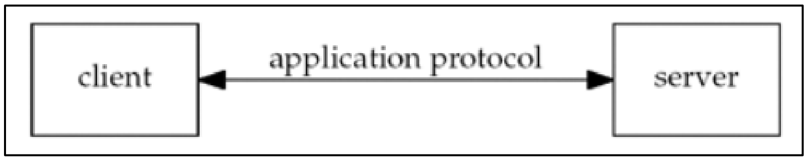
**Ans:** A network protocol is an accepted set of rules that govern data communication between different devices in the network. It determines what is being communicated, how it is being communicated, and when it is being communicated.

- For example, a web server is typically thought of as a long-running program that sends network messages only in response to requests coming in from the network.
- The other side of the protocol is a web client, such as a browser which always initiates communication with the server.
- This organization into client and server is used by most network-aware applications. Deciding that the client always initiates requests tends to simplify the protocol as well as the programs themselves.



**5. What are the design decisions made before writing network programs? Justify the decisions made in a typical client server model (05 marks)**

- When writing programs that communicate across a computer network, one must first invent a *protocol*, an agreement on how those programs will communicate.
- Before delving into the design details of a protocol, high-level decisions must be made about which program is expected to initiate communication and when responses are expected.
- For example, a Web server is typically thought of as a long-running program (or *daemon*) that sends network messages only in response to requests coming in from the network.
- The other side of the protocol is a Web client, such as a browser, which always initiates communication with the server.
- This organization into *client* and *server* is used by most network-aware applications.
- Deciding that the client always initiates requests tends to simplify the protocol as well as the programs themselves.
- Some of the more complex network applications also require *asynchronous callback* communication, where the server initiates a message to the client.
- But it is far more common for applications to stick to the basic client/server model shown in figure.



**6. Explain with a neat diagram the following: a. TCP Connection Establishment b. TCP connection termination.**

**OR**

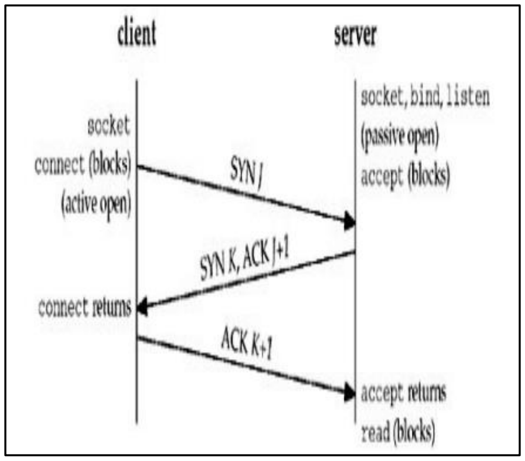
**Explain with a neat diagram the following: (12 marks)**

**a. TCP Connection Establishment b. TCP data transfer c. TCP connection termination**

**Ans:**

**TCP Connection Establishment:** The following scenario occurs when a TCP connection is established:

- The server must be prepared to accept an incoming connection. This is normally done by calling socket, bind, and listen and is called a passive open.
- The client issues an active open by calling connect. This causes the client TCP to send a "synchronize" (SYN) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options.
- The server must acknowledge (ACK) the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the client's SYN in a single segment.
- The client must acknowledge the server's SYN.
- The minimum number of packets required for this exchange is three; hence, this is called TCP's three-way handshake.

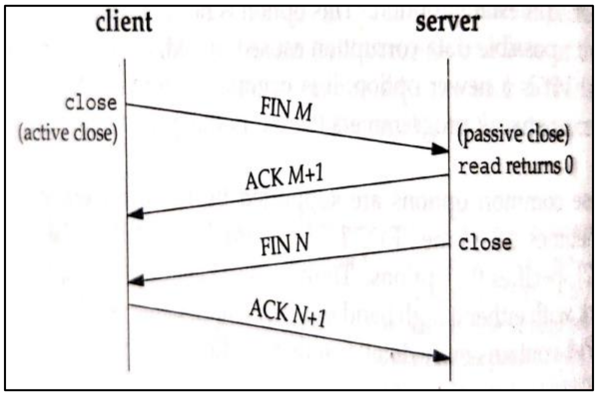


**TCP data transfer:**

- The data is transferred in packets.
- If the data is received, then the receiving device acknowledges it by sending an ACK segment.
- If the receiving device doesn't send an ACK segment, retransmission of data packets occurs.

**TCP connection termination:** While it takes three segments to establish a connection, it takes four to terminate a connection.

- One application calls *close* first, and we say that this end performs the active close. This end's TCP sends an FIN segment, which means it is finished sending data.
- The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file (after any data that may have already been queued for the application to receive), since the receipt of the FIN means the application will not receive any additional data on the connection.
- Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.
- The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.



## 7. Write a program to implement TCP daytime client for IPV6

```
int main()
{
int s;
struct sockaddr_in6 addr;
s = socket(AF_INET6, SOCK_STREAM, 0);
addr.sin6_family = AF_INET6;
addr.sin6_port = htons(5000);
inet_pton(AF_INET6, "::1", &addr.sin6_addr);
connect(s, (struct sockaddr*)&addr, sizeof(addr));
while ((n = read(sockfd, recvline, 1000)) > 0) {
recvline[n] = 0;
fputs(recvline, stdout);
}
close(sockfd);
return 0;
}
```

**8. What are wrapper functions? Develop the wrapper functions for the following:**

**a) Socket function.**

**b) Pthread_mutex_lock**

**OR**

**Explain the Error Handling using Wrapper functions**

**Wrapper functions:** In any real-world program, it is essential to check every function call for an error return. We check for errors from socket, inet_pton, connect, read, and fputs, and when one occurs, we call our own functions, err_quit and err_sys, to print an error message and terminate the program.

Since terminating on an error is the common case, we can shorten our programs by defining a *wrapper function* that performs the actual function call, tests the return value, and terminates on an error.
Whenever you encounter a function name in the text that begins with an uppercase letter, that is one of our wrapper functions. It calls a function whose name is the same but begins with the lowercase letter.

**a) Socket function**

```
int
Socket(int family, int type, int protocol)
{
        int n;
        if ( (n = socket(family, type, protocol)) < 0)
        err_sys("socket error");
        return (n);
}
```

**b) Pthread_mutex_lock**

```
void
Pthread_mutex_lock(pthread_mutex_t *mptr)
{
        int n;
        if ( (n = pthread_mutex_lock(mptr)) == 0)
        return;
        errno = n;
        err_sys("pthread_mutex_lock error");
}
```

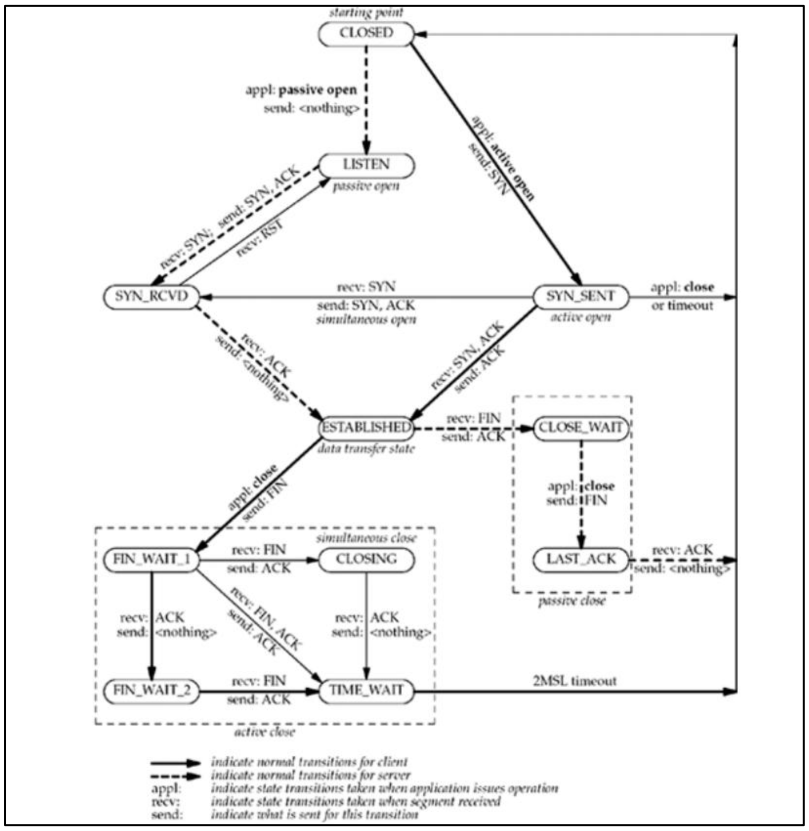
**9. Write a program to implement TCP daytime Server.**

**Ans:**

```
#include "unp.h".

#include <time.h>

Int main(int argc, char **argv){

        int listenfd, connfd;

        struct sockaddr_in servaddr;

        char buff[MAXLINE];

        time_t ticks;

        listenfd = Socket(AF_INET, SOCK_STREAM, 0);

        bzeros(&servaddr, sizeof(servaddr));

        servaddr.sin_family = AF_INET;

        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

        servaddr.sin_port = htons(13);

        Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

        Listen(listenfd, LISTENQ);

        for ( ; ; ) {

                connfd = Accept(listenfd, (SA *) NULL, NULL);

                ticks = time(NULL);

                snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));

                Write(connfd, buff, strlen(buff));

                Close(connfd);

        }

}
```

**10. With a neat sketch explain the TCP state Transition Diagram.**

- The operation of TCP with regard to connection establishment and connection termination can be specified with a state transition diagram.
- There are 11 different states defined for a connection and the rules of TCP dictate the transitions from one state to another, based on the current state and the segment received in that state.
- If an application performs an active open in the CLOSED state, TCP sends a SYN and the new state is SYN_SENT.
- If TCP next receives a SYN with an ACK, it sends an ACK and the new state is ESTABLISHED. This final state is where most data transfer occurs.
- The two arrows leading from the ESTABLISHED state deal with the termination of a connection.
- If an application calls close before receiving a FIN (an active close), the transition is to the FIN_WAIT_1 state.
- But if an application receives a FIN while in the ESTABLISHED state (a passive close), the transition is to the CLOSE_WAIT state.
- We denote the normal client transitions with a darker solid line and the normal server transitions with a darker dashed line.
- We also note that there are two transitions that we have not talked about: a simultaneous open (when both ends send SYNs at about the same time and the SYNs cross in the network) and a simultaneous close (when both ends send FINs at the same time).



## 11. Write a short note on i) TCP ii) UDP iii) SCTP protocols

i) TCP:

- TCP provides *connections* between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.
- TCP also provides *reliability*. When TCP sends data to the other end, it requires an acknowledgment in return. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time. After some number of retransmissions, TCP will give up, with the total amount of time spent trying to send data typically between 4 and 10 minutes.
- TCP does not guarantee that the data will be received by the other endpoint, as this is impossible.
- Therefore, TCP cannot be described as a 100% reliable protocol; it provides reliable delivery of data *or* reliable notification of failure.
- TCP also *sequences* the data by associating a sequence number with every byte that it sends.
- TCP provides *flow control*.

ii) UDP:

- UDP is a simple transport-layer protocol.
- The application writes a message to a UDP socket, which is then *encapsulated* in a UDP *datagram*, which is then further encapsulated as an IP datagram, which is then sent to its destination.
- There is no guarantee that a UDP datagram will ever reach its final destination, that order will be preserved across the network, or that datagrams arrive only once.
- UDP provides a *connectionless* service, as there need not be any long-term relationship between a UDP client and server.
- The problem that we encounter with network programming using UDP is its lack of reliability. If a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not delivered to the UDP socket and is not automatically retransmitted.

iii) SCTP:

- SCTP provides services similar to those offered by UDP and TCP.
- SCTP provides *associations* between clients and servers.
- SCTP also provides applications with reliability, sequencing, flow control, and full-duplex data transfer, like TCP.
- SCTP is *message-oriented*. It provides sequenced delivery of individual records.
- SCTP can provide multiple streams between connection endpoints, each with its own reliable sequenced delivery of messages.
- SCTP also provides a multi homing feature, which allows a single SCTP endpoint to support multiple IP addresses.