MODEL ENGINEERING COLLEGE

(Unit of I H R D)

THRIKKAKARA, ERNAKULAM

LABORATORY RECORD

. . . .

NAME	
BRANCH	
SEMESTER	ROLL No . . . .

Certified that this is the Bonafide work done by

. . . .

Staff-in-Charge ***Head of the Department***

Register No . . . . Thrikkakara

Year & Month . . . .

Date . . . .

INDEX

SI. No.	NAME OF EXPERIMENT	DATE	PAGE No.	SIGNATURE

INDEX

SI. No.	NAME OF EXPERIMENT	DATE	PAGE No.	SIGNATURE

OUTPUT:

```
$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 1500
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0xfe<compat,link,site,host>
        loop  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0


$ netstat -r
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
127.0.0.0       0.0.0.0         255.0.0.0       U         0 0          0 lo
127.0.0.1       0.0.0.0         255.255.255.255 U         0 0          0 lo
127.255.255.255 0.0.0.0         255.255.255.255 U         0 0          0 lo
224.0.0.0       0.0.0.0         240.0.0.0       U         0 0          0 lo
255.255.255.255 0.0.0.0         255.255.255.255 U         0 0          0 lo


$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=116 time=24.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=116 time=22.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=116 time=22.3 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=116 time=23.7 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=116 time=22.4 ms
```

Experiment Number: 1 **Date:**

BASIC NETWORKING COMMANDS

AIM:

To familiarise the basics of network configuration files and networking commands in Linux.

THEORY:

- **IFCONFIG** - configure a network interface
  Ifconfig is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.
  If no arguments are given, ifconfig displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the given interface only; if a single -a argument is given, it displays the status of all interfaces, even those that are down. Otherwise, it configures an interface.

- **NETSTAT** - Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.
  Netstat prints information about the Linux networking subsystem. The type of information printed is controlled by the first argument, as follows:

  (none)
  By default, netstat displays a list of open sockets. If you don't specify any address families, then the active sockets of all configured address families will be printed.

  --route, -r
  Display the kernel routing tables.

  --groups, -g
  Display multicast group membership information for IPv4 and IPv6.

  --interfaces, -i
  Display a table of all network interfaces.

  --masquerade, -M
  Display a list of masqueraded connections.

  --statistics, -s
  Display summary statistics for each protocol.

- **PING** - send ICMP ECHO_REQUEST to network hosts
  ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (“pings”) have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of “pad” bytes used to fill out the packet.
  ping works with both IPv4 and IPv6. Using only one of them explicitly can be enforced by specifying -4 or -6. ping can also send IPv6 Node Information Queries (RFC4620). Intermediate hops may not be allowed, because IPv6 source routing was deprecated (RFC5095).

```
$ arp -n
Address                  HWtype  HWaddress               Flags Mask      Iface
192.168.18.192           ether   50:eb:71:3b:39:9f       C               enp0s3
192.168.18.135           ether   08:00:27:ae:78:cc       C               enp0s3
_gateway                 ether   44:00:4d:e9:a1:17       C               enp0s3


$ telnet 192.168.18.135
Trying 192.168.18.135...
Connected to 192.168.18.135.
Escape character is '^]'.
Ubuntu 20.04.3 LTS
mec login: mec
Password:
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-37-generic x86_64)

  * Documentation:      https://help.ubuntu.com
  * Management:         https://landscape.canonical.com
  * Support:            https://ubuntu.com/advantage

83 updates can be applied immediately.
34 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

Your Hardware Enablement Stack (HWE) is supported until April 2025.


$ ftp 192.168.4.25
Connected to 192.168.4.25.
220-Welcome to the Pandemonia ftp server
220-No anonymous access
220-Authernticated access only
220 ++++++++++
Name (192.168.4.25:mec):
331 Password required for mec
Password:
230 Logged on
Remote system type is UNIX.
ftp> quit



$ finger mec
Login: mec
Name: mec
Directory: /home/mec
Shell: /bin/bash
On since Sun June 7 10:32 (IST) on :0 from :0 (messages off)
No mail.
No Plan.
```

- **ARP** - manipulate the system ARP cache
  Arp manipulates or displays the kernel's IPv4 network neighbour cache. It can add entries to the table, delete one or display the current content.

  ARP stands for Address Resolution Protocol, which is used to find the media access control address of a network neighbour for a given IPv4 Address.

- **TELNET** - user interface to the TELNET protocol
  The telnet command is used for interactive communication with another host using the TELNET protocol. It begins in command mode, where it prints a telnet prompt ("telnet> "). If telnet is invoked with a host argument, it performs an open command implicitly.

- **FTP** - Internet file transfer program
  Ftp is the user interface to the Internet standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

  The client host and an optional port number with which ftp is to communicate may be specified on the command line. If this is done, ftp will immediately attempt to establish a connection to an FTP server on that host; otherwise, ftp will enter its command interpreter and await instructions from the user. When ftp is awaiting commands from the user the prompt 'ftp>' is provided to the user.

- **FINGER** - user information lookup program
  The finger displays information about the system users.
  Options are:

  -s Finger displays the user's login name, real name, terminal name and write status (as a ``*'' after the terminal name if write permission is denied), idle time, login time, office location and office phone number.

  -l Produces a multi-line format displaying all of the information described for the -s option as well as the user's home directory, home phone number, login shell, mail status, and the contents of the files ".plan", ".project", ".pgpkey" and ".forward" from the user's home directory.

  -p Prevents the -l option of finger from displaying the contents of the ".plan", ".project" and ".pgpkey" files.

  -m Prevent matching of user names. User is usually a login name; however, matching will also be done on the users' real names, unless the -m option is supplied. All name matching performed by finger is case insensitive.

  If no options are specified, finger defaults to the -l style output if operands are provided, otherwise to the -s style. Note that some fields may be missing, in either format, if information is not available for them.
  If no arguments are specified, finger will print an entry for each user currently logged into the system.

  Finger may be used to look up users on a remote machine. The format is to specify a user as "user@host", or "@host", where the default output format for the former is the -l style, and the default output format for the latter is the -s style. The -l option is the only option that may be passed to a remote machine.
  If standard output is a socket, finger will emit a carriage return (^M) before every linefeed (^J). This is for processing remote finger requests when invoked by fingerd(8).

```
$ traceroute google.com
traceroute to google.com (172.217.23.14), 30 hops max, 60 byte packets
 1  10.8.8.1 (10.8.8.1)  14.499 ms  15.335 ms  15.956 ms
 2  h37-220-13-49.host.redstation.co.uk (37.220.13.49)  17.811 ms  18.669 ms
19.346 ms
 3  92.zone.2.c.dc9.redstation.co.uk (185.20.96.137)  19.096 ms  19.757 ms  20.892
ms
 4  203.lc3.redstation.co.uk (185.5.3.221)  28.160 ms  28.415 ms  28.665 ms
 5  100.core1.the.as20860.net (62.128.218.33)  26.739 ms  27.840 ms  28.847 ms
 6  110.core2.thn.as20860.net (62.128.218.26)  29.112 ms  18.466 ms  19.835 ms
 7  be97.asr01.thn.as20860.net (62.128.222.205)  19.986 ms  20.488 ms  21.354 ms
 8  * * *
 9  216.239.48.143 (216.239.48.143)  24.364 ms 216.239.48.113 (216.239.48.113)
25.069 ms  25.592 ms
10  108.170.233.199 (108.170.233.199)  26.239 ms  27.369 ms  28.031 ms
11  lhr35s01-in-f14.1e100.net (172.217.23.14)  28.642 ms  29.311 ms  29.815 ms
```

```
$ whois tryhackme.com
Domain Name: TRYHACKME.COM
Registry Domain ID: 2282723194_DOMAIN_COM-VRSN
Registrar WHOIS Server: whois.namecheap.com
Registrar URL: http://www.namecheap.com
Updated Date: 2021-05-01T19:43:23Z
Creation Date: 2018-07-05T19:46:15Z
Registry Expiry Date: 2027-07-05T19:46:15Z
Registrar: NameCheap, Inc.
Registrar IANA ID: 1068
Registrar Abuse Contact Email: abuse@namecheap.com
Registrar Abuse Contact Phone: +1.6613102107
Domain Status: clientTransferProhibited
https://icann.org/epp#clientTransferProhibited Name Server:
KIP.NS.CLOUDFLARE.C>Name Server: KIP.NS.CLOUDFLARE.COM
Name Server: UMA.NS.CLOUDFLARE.COM
DNSSEC: unsigned
URL of the ICANN Whois Inaccuracy Complaint Form: https://www.icann.org/wicf/
>>> Last update of whois database: 2022-04-17T12:11:36Z <<<

For more information on Whois status codes, please visit https://icann.org/epp
```

- **TRACEROUTE** - print the route packets trace to network host
  traceroute tracks the route packets taken from an IP network on their way to a given host. It utilizes the IP protocol's time to live (TTL) field and attempts to elicit an ICMP TIME_EXCEEDED response from each gateway along the path to the host.

  traceroute6 is equivalent to traceroute -6

  tcptraceroute is equivalent to traceroute –T

  lft , the Layer Four Traceroute, performs a TCP traceroute, like traceroute -T , but attempts to provide compatibility with the original such implementation, also called "lft".

  The only required parameter is the name or IP address of the destination host . The optional packet_len`gth is the total size of the probing packet (default 60 bytes for IPv4 and 80 for IPv6). The specified size can be ignored in some situations or increased up to a minimal value.

  This program attempts to trace the route an IP packet would follow to some internet host by launching probe packets with a small ttl (time to live) then listening for an ICMP "time exceeded" reply from a gateway. We start our probes with a ttl of one and increase by one until we get an ICMP "port unreachable" (or TCP reset), which means we got to the "host", or hit a max (which defaults to 30 hops). Three probes (by default) are sent at each ttl setting and a line is printed showing the ttl, address of the gateway and round trip time of each probe. The address can be followed by additional information when requested. If the probe answers come from different gateways, the address of each responding system will be printed. If there is no response within a certain timeout, an "*" (asterisk) is printed for that probe.

  After the trip time, some additional annotation can be printed: !H, !N, or !P (host, network or protocol unreachable), !S (source route failed), !F (fragmentation needed), !X (communication administratively prohibited), !V (host precedence violation), !C (precedence cutoff in effect), or !<num> (ICMP unreachable code <num>). If almost all the probes result in some kind of unreachable, traceroute will give up and exit.

  We don't want the destination host to process the UDP probe packets, so the destination port is set to an unlikely value (you can change it with the -p flag). There is no such a problem for ICMP or TCP tracerouting (for TCP we use half-open technique, which prevents our probes to be seen by applications on the destination host).

  In the modern network environment the traditional traceroute methods can not be always applicable, because of widespread use of firewalls. Such firewalls filter the "unlikely" UDP ports, or even ICMP echoes.

- **WHOIS** - client for the whois directory service
  whois searches for an object in a RFC 3912 database.

  This version of the whois client tries to guess the right server to ask for the specified object. If no guess can be made it will connect to whois.networksolutions.com for NIC handles or whois.arin.net for IPv4 addresses and network names.

RESULT:

Basics of network configuration files and networking commands in Linux were understood.

Experiment Number: 2 **Date:**

SOCKET PROGRAMMING

AIM:

To familiarize and understand the use and functioning of system calls used for network programming in Linux.

THEORY:

SOCKET OPERATIONS

- **SOCKET(2)**

```
NAME
```
socket - create an endpoint for communication

```
SYNOPSIS
       #include <sys/types.h>
        #include <sys/socket.h>
        int socket(int domain, int type, int protocol);
```

```
DESCRIPTION
```
socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The domain argument specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>.

```
RETURN VALUE
```
On success, a file descriptor for the new socket is returned. On error, -1 is returned, and errno is set appropriately.

- **SOCKETPAIR(2)**

```
NAME
```
socketpair - create a pair of connected sockets

```
SYNOPSIS
        #include <sys/types.h>
        #include <sys/socket.h>
        int socketpair(int domain, int type, int protocol, int sv[2]);
```

DESCRIPTION

The socketpair() call creates an unnamed pair of connected sockets in the specified domain, of the specified type, and using the optionally specified protocol. For further details of these arguments, see socket(2).

The file descriptors used in referencing the new sockets are returned in sv[0] and sv[1]. The two sockets are indistinguishable.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, errno is set appropriately, and sv is left unchanged.

On Linux (and other systems), socketpair() does not modify sv on failure. A requirement standardizing this behavior was added in POSIX.1-2016.

- **GETSOCKOPT(2) and SETSOCKOPT(2)**

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

```
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int getsockopt(int sockfd, int level, int optname,
                      void *optval, socklen_t *optlen);

       int setsockopt(int sockfd, int level, int optname,
                      const void *optval, socklen_t optlen);
```

DESCRIPTION

getsockopt() and setsockopt() manipulate options for the socket referred to by the file descriptor sockfd. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified.

The arguments optval and optlen are used to access option values for setsockopt(). For getsockopt() they identify a buffer in which the value for the requested option(s) are to be returned. For getsockopt(), optlen is a value-result argument, initially containing the size of the buffer pointed to by optval, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, optval may be NULL.

RETURN VALUE

On success, zero is returned for the standard options. On error, -1 is returned, and errno is set appropriately.

Netfilter allows the programmer to define custom socket options with associated handlers; for such options, the return value on success is the value returned by the handler.

- **GETSOCKNAME(2)**

NAME

getsockname - get socket name

SYNOPSIS

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

DESCRIPTION

getsockname() returns the current address to which the socket sockfd is bound, in the buffer pointed to by addr. The addrlen argument should be initialized to indicate the amount of space (in bytes) pointed to by addr. On return it contains the actual size of the socket address.

The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

- **GETPEERNAME(2)**

NAME

getpeername - get name of connected peer socket

SYNOPSIS

```
#include <sys/socket.h>
 int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

DESCRIPTION

getpeername() returns the address of the peer connected to the socket sockfd, in the buffer pointed to by addr. The addrlen argument should be initialized to indicate the amount of space pointed to by addr. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

- **BIND(2)**

NAME

bind - bind a name to a socket

SYNOPSIS
```
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>
       int bind(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);
```

DESCRIPTION

When a socket is created with socket(2), it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called “assigning a name to a socket”.

It is normally necessary to assign a local address using bind() before a SOCK_STREAM socket may receive connections.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

- **ACCEPT(2)**

NAME

accept, accept4 - accept a connection on a socket

SYNOPSIS
```
      #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>
       int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <sys/socket.h>
       int accept4(int sockfd, struct sockaddr *addr,
                   socklen_t *addrlen, int flags);
```

DESCRIPTION

The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET).

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.

The argument sockfd is a socket that has been created with socket(2), bound to a local address with bind(2), and is listening for connections after a listen(2).

The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer.

The addrlen argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by addr; on return it will contain the actual size of the peer address.

The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

If no pending connections are present on the queue, and the socket is not marked as nonblocking, accept() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, accept() fails with the error EAGAIN or EWOULDBLOCK.

In order to be notified of incoming connections on a socket, you can use select(2), poll(2), or epoll(7). A readable event will be delivered when a new connection is attempted and you may then call accept() to get a socket for that connection.

If flags is 0, then accept4() is the same as accept().

RETURN VALUE

On success, these system calls return a nonnegative integer that is a file descriptor for the accepted socket. On error, -1 is returned, errno is set appropriately, and addrlen is left unchanged.

- **CONNECT(2)**

NAME

connect - initiate a connection on a socket

SYNOPSIS

```
      #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>
       int connect(int sockfd, const struct sockaddr *addr,
                      socklen_t addrlen);
```

DESCRIPTION

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The addrlen argument specifies the size of addr. The format of the address in addr is determined by the address space of the socket sockfd.

If the socket sockfd is of type SOCK_DGRAM, then addr is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type SOCK_STREAM or SOCK_SEQPACKET, this call attempts to make a connection to the socket that is bound to the address specified by addr.

Generally, connection-based protocol sockets may successfully connect() only once; connectionless protocol sockets may use connect() multiple times to change their association.

RETURN VALUE

If the connection or binding succeeds, zero is returned. On error, -1 is returned, and errno is set appropriately.

- **SHUTDOWN(2)**

NAME
shutdown - shut down part of a full-duplex connection

SYNOPSIS
```
#include <sys/socket.h>
 int shutdown(int sockfd, int how);
```

DESCRIPTION
The shutdown() call causes all or part of a full-duplex connection on the socket associated with sockfd to be shutdown. If how is SHUT_RD, further receptions will be disallowed. If how is SHUT_WR, further transmissions will be disallowed. If how is SHUT_RDWR, further receptions and transmissions will be disallowed.

RETURN VALUE
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

SEND OR RECEIVE OPERATIONS

- **SEND(2)**

NAME
send, sendto, sendmsg - send a message on a socket

SYNOPSIS
```
#include <sys/types.h>
 #include <sys/socket.h>

 ssize_t send(int sockfd, const void *buf, size_t len, int flags);

 ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                const struct sockaddr *dest_addr, socklen_t addrlen);

 ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

DESCRIPTION
The system calls send(), sendto(), and sendmsg() are used to transmit a message to another socket.
The send() call may be used only when the socket is in a connected state (so that the intended recipient is known). The only difference between send() and write(2) is the presence of flags. With a zero flags argument, send() is equivalent to write(2). Also, the following call
```
send(sockfd, buf, len, flags);
```
is equivalent to
```
sendto(sockfd, buf, len, flags, NULL, 0);
```
The argument sockfd is the file descriptor of the sending socket.

If sendto() is used on a connection-mode (SOCK_STREAM, SOCK_SEQPACKET) socket, the arguments dest_addr and addrlen are ignored (and the error EISCONN may be returned when they are not NULL and 0), and the error ENOTCONN is returned when the socket was not actually connected. Otherwise, the address of the target is given by dest_addr with addrlen specifying its size. For sendmsg(), the address of the target is given by msg.msg_name, with msg.msg_namelen specifying its size.

For send() and sendto(), the message is found in buf and has length len. For sendmsg(), the message is pointed to by the elements of the array msg.msg_iov. The sendmsg() call also allows sending ancillary data (also known as control information).

If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send(). Locally detected errors are indicated by a return value of -1.

When the message does not fit into the send buffer of the socket, send() normally blocks, unless the socket has been placed in nonblocking I/O mode. In nonblocking mode it would fail with the error EAGAIN or EWOULDBLOCK in this case. The select(2) call may be used to determine when it is possible to send more data.

RETURN VALUE

On success, these calls return the number of bytes sent. On error, -1 is returned, and errno is set appropriately.

- **SENDMMSG(2)**

NAME

sendmmsg - send multiple messages on a socket

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
 #include <sys/socket.h>

 int sendmmsg(int sockfd, struct mmsghdr *msgvec, unsigned int vlen,
                 int flags);
```

DESCRIPTION

The sendmmsg() system call is an extension of sendmsg(2) that allows the caller to transmit multiple messages on a socket using a single system call. (This has performance benefits for some applications.)

The sockfd argument is the file descriptor of the socket on which data is to be transmitted.

The msgvec argument is a pointer to an array of mmsghdr structures. The size of this array is specified in vlen.

The mmsghdr structure is defined in <sys/socket.h> as:

```
struct mmsghdr {
            struct msghdr msg_hdr;  /* Message header */
            unsigned int  msg_len;  /* Number of bytes transmitted */
        };
```

The msg_hdr field is a msghdr structure, as described in sendmsg(2). The msg_len field is used to return the number of bytes sent from the message in msg_hdr (i.e., the same as the return value from a single sendmsg(2) call).

The flags argument contains flags ORed together. The flags are the same as for sendmsg(2).

A blocking sendmmsg() call blocks until vlen messages have been sent. A nonblocking call sends as many messages as possible (up to the limit specified by vlen) and returns immediately.

On return from sendmmsg(), the msg_len fields of successive elements of msgvec are updated to contain the number of bytes transmitted from the corresponding msg_hdr. The return value of the call indicates the number of elements of msgvec that have been updated.

RETURN VALUE

On success, sendmmsg() returns the number of messages sent from msgvec; if this is less than vlen, the caller can retry with a further sendmmsg() call to send the remaining messages.

- **RECV(2)**

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

DESCRIPTION

The recv(), recvfrom(), and recvmsg() calls are used to receive messages from a socket. They may be used to receive data on both connectionless and connection-oriented sockets.

The only difference between recv() and read(2) is the presence of flags. With a zero flags argument, recv() is generally equivalent to read(2) (but see NOTES). Also, the following call

```
recv(sockfd, buf, len, flags);
```

is equivalent to

```
recvfrom(sockfd, buf, len, flags, NULL, NULL);
```

All three calls return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking, in which case the value -1 is returned and the external variable errno is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

An application can use select(2), poll(2), or epoll(7) to determine when more data arrives on a socket.

recvfrom()

recvfrom() places the received message into the buffer buf. The caller must specify the size of the buffer in len.

If src_addr is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by src_addr. In this case, addrlen is a value-result argument. Before the call, it should be initialized to the size of the buffer associated with src_addr. Upon return, addrlen is updated to contain the actual size of the source address.

The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

If the caller is not interested in the source address, src_addr and addrlen should be specified as NULL.

recv()

The recv() call is normally used only on a connected socket (see connect(2)). It is equivalent to the call:

```
recvfrom(fd, buf, len, flags, NULL, 0);
```

recvmsg()

The recvmsg() call uses a msghdr structure to minimize the number of directly supplied arguments.

RETURN VALUE

These calls return the number of bytes received, or -1 if an error occurred. In the event of an error, errno is set to indicate the error.

When a stream socket peer has performed an orderly shutdown, the return value will be 0 (the traditional "end-of-file" return).

Datagram sockets in various domains (e.g., the UNIX and Internet domains) permit zero-length datagrams. When such a datagram is received, the return value is 0.

The value 0 may also be returned if the requested number of bytes to receive from a stream socket was 0.

- **RECVMMSG(2)**

NAME

recvmmsg - receive multiple messages on a socket

SYNOPSIS

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/socket.h>
int recvmmsg(int sockfd, struct mmsghdr *msgvec, unsigned int vlen,
                 int flags, struct timespec *timeout);
```

DESCRIPTION

The recvmmsg() system call is an extension of recvmsg(2) that allows the caller to receive multiple messages from a socket using a single system call. (This has performance benefits for some applications.) A further extension over recvmsg(2) is support for a timeout on the receive operation.

The sockfd argument is the file descriptor of the socket to receive data from.

The msgvec argument is a pointer to an array of mmsghdr structures. The size of this array is specified in vlen.

The mmsghdr structure is defined in <sys/socket.h> as:

```
          struct mmsghdr {
               struct msghdr msg_hdr;  /* Message header */
               unsigned int  msg_len;  /* Number of received bytes for header
*/
           };
```

The msg_hdr field is a msghdr structure, as described in recvmsg(2). The msg_len field is the number of bytes returned for the message in the entry. This field has the same value as the return value of a single recvmsg(2) on the header.

The timeout argument points to a struct timespec defining a timeout (seconds plus nanoseconds) for the receive operation. (This interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount.) If timeout is NULL, then the operation blocks indefinitely.

A blocking recvmmsg() call blocks until vlen messages have been received or until the timeout expires. A nonblocking call reads as many messages as are available (up to the limit specified by vlen) and returns immediately.

On return from recvmmsg(), successive elements of msgvec are updated to contain information about each received message: msg_len contains the size of the received message; the subfields of msg_hdr are updated as described in recvmsg(2). The return value of the call indicates the number of elements of msgvec that have been updated.

RETURN VALUE

On success, recvmmsg() returns the number of messages received in msgvec; on error, -1 is returned, and errno is set to indicate the error.

<u>RESULT</u>:

The use of system calls used for networking programming in Linux was familiarised and understood.

PROGRAM:

```
// Server (TCP)
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main()
{
    char buf[100];
    int k;
    socklen_t len;
    int sock_desc,temp_sock_desc;
    struct sockaddr_in server,client;

    sock_desc=socket(AF_INET,SOCK_STREAM,0);
    if(sock_desc==-1)
        printf("Error in socketcreation");
    server.sin_family=AF_INET;
    server.sin_addr.s_addr=INADDR_ANY;
    server.sin_port=3003;
    client.sin_family=AF_INET;
    client.sin_addr.s_addr=INADDR_ANY;
    client.sin_port=3003;

    k=bind(sock_desc,(struct sockaddr*)&server,sizeof(server));
    if(k==-1)
        printf("Error in binding");

    k=listen(sock_desc,5);
    if(k==-1)
        printf("Error in listening");

    len=sizeof(client);
    temp_sock_desc=accept(sock_desc,(struct sockaddr*)&client,&len);
    if(temp_sock_desc==-1)
    printf("Error in temporary socket creation");

    k=recv(temp_sock_desc,buf,100,0);
    if(k==-1)
        printf("Error in receiving");

    printf("Message got from client: %s",buf);

    close(temp_sock_desc);
    return 0;
}


// Client (TCP)
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
```

Experiment Number: 3 **Date:**

TRANSMISSION CONTROL PROTOCOL

AIM:

To implement client-server communication using socket programming and TCP as transport layer protocol.

THEORY:

A TCP (transmission control protocol) is a connection-oriented communication. TCP is designed to send the data packets over the network. It ensures that data is delivered to the correct destination. TCP creates a connection between the source and destination node before transmitting the data and keeps the connection alive until the communication is active.

Server-client model is communication model for sharing the resource and provides the service to different machines. Server is the main system which provides the resources and different kind of services when client requests to use it.

ALGORITHM:

Server

```
1. Create a socket with type as SOCK_STREAM to create a tcp socket using socket()
system call.
2. Bind the socket to a specific port using bind() system call.
3. Listen for new connections using the listen() system call.
4. Accept connection from client process into a temporary socket.
5. Read the message from the client using the recv() system call into a buffer.
6. Display the message and close the socket.
```

Client

```
1. Create a socket with type as SOCK_STREAM to create a tcp socket using socket()
system call.
2. Connect to the server using connect() system call.
3. Read the message to be sent from the user.
4. Send the message to the server using send() system call.
5. Close the socket.
```

```
int main()
{
    char buf[100];
    int k;
    int sock_desc;
    struct sockaddr_in client;

    sock_desc=socket(AF_INET,SOCK_STREAM,0);
    if(sock_desc==-1)
        printf("Error in socket creation!");

    client.sin_family=AF_INET;
    client.sin_addr.s_addr=INADDR_ANY;
    client.sin_port=3003;

    k=connect(sock_desc,(struct sockaddr*)&client,sizeof(client));
    if(k==-1)
        printf("Error in connecting to server!");

    printf("\nEnter data to be send: ");
    fgets(buf,100,stdin);

    k=send(sock_desc,buf,100,0);
    if(k==-1)
        printf("Error in sending!");

    close(sock_desc);
    return 0;
}
```

OUTPUT:

```
// Terminal 1
$ gcc -o server server.c
$ ./server
_


// Terminal 2
$ gcc -o client client.c
$ ./client

Enter data to be send: hi programmers!


// Terminal 1
$ gcc -o server server.c
$ ./server
Message got from client: hi programmers!
```

RESULT:

Thus the programs for client-server communication using socket programming for TCP was executed and output was verified successfully.

PROGRAM:

```
// Server (UDP)
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main(int argc,char* argv[])
{
    struct sockaddr_in server,client;

    if(argc!=2)
        printf("Input format not correct!\n");

    int sockfd=socket(AF_INET,SOCK_DGRAM,0);
    if(sockfd==-1)
        printf("Error in socket creation!\n");

    server.sin_family=AF_INET;
    server.sin_addr.s_addr=INADDR_ANY;
    server.sin_port=htons(atoi(argv[1]));

    if(bind(sockfd,(struct sockaddr*)&server,sizeof(server))<0)
        printf("Error in bind()!\n");

    char buffer[100];
    socklen_t server_len=sizeof(server);
    printf("Server waiting...\n");

    if(recvfrom(sockfd,buffer,100,0,(struct sockaddr*)&server,&server_len)<0)
        printf("Error in receiving!\n");

    printf("Got a datagram: %s",buffer);

    return 0;
}


// Client (UDP)
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main(int argc,char* argv[])
{
    struct sockaddr_in server, client;

    if(argc!=3)
        printf("Input format not correct!\n");

    int sockfd=socket(AF_INET,SOCK_DGRAM,0);
    if(sockfd==-1)
        printf("Error in socket creation!\n");
```

Experiment Number: 4 **Date:**

USER DATAGRAM PROTOCOL

AIM:

To implement client-server communication using socket programming and UDP as transport layer protocol.

THEORY:

UDP is a connection-less protocol that, unlike TCP, does not require any handshaking prior to sending or receiving data, which simplifies its implementation. In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive.

Server-client model is communication model for sharing the resource and provides the service to different machines. Server is the main system which provides the resources and different kind of services when client requests to use it.

ALGORITHM:

```
Server
1. Create a socket with type as SOCK_DGRAM to create a udp socket using socket()
system call.
2. Bind the socket to a specific port using bind() system call.
3. Using the recvfrom() system call message sent from the client process into a
buffer.
4. Display the message and close the socket.

Client
1. Create a socket with type as SOCK_DGRAM to create a udp socket using socket()
system call.
2. Read the message to be sent from the user.
3. Send the message to the server using sendto() system call.
4. Close the socket.
```

```
    server.sin_family=AF_INET;
    server.sin_addr.s_addr=INADDR_ANY;
    server.sin_port=htons(atoi(argv[2]));

    char buffer[100];
    printf("Enter a message to sent to server: ");
    fgets(buffer,100,stdin);

    if(sendto(sockfd,buffer,sizeof(buffer),0,(struct
sockaddr*)&server,sizeof(server))<0)
        printf("Error in sending!\n");

    return 0;
}
```

OUTPUT:

```
// Terminal 1
$ gcc -o server server.c
$ ./server 2000
Server waiting...

// Terminal 2
$ gcc -o client client.c
$ ./client localhost 2000
Enter a message to sent to server: Morning!

// Terminal 1
$ gcc -o server server.c
$ ./server 2000
Server waiting...
Got a datagram: Morning!
```

RESULT:

Thus the programs for client-server communication using socket programming for UDP was executed and output was verified successfully.

PROGRAM:

```
// Stop and Wait Protocol
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main ()
{
    int i, j = 0, noframes, x, x1 = 10, x2;
    printf ("Enter the number of frames: ");
    scanf ("%d",&noframes);
    printf ("\nnumber of frames is %d",noframes);
    while (noframes > 0)
    {
        printf ("\nsending frame %d", i);
        srand (x1++); //The srand() function sets the starting point for producing
a series of pseudo-random integers
        x = rand () % 10;
        if (x % 2 == 0)
        {
            for (x2=1; x2<2; x2++)
            {
                printf ("\nwaiting for %d seconds\n", x2);
                sleep(x2);
                printf ("Missing Acknowledgement %d", i);
            }
            printf ("\nsending frame %d", i);
            srand (x1++);
            x = rand () % 10;
        }
        printf ("\nack received for frame %d", j);
        noframes -= 1;
        i++;
        j++;
    }
    printf ("\nend of stop and wait protocol\n");
}
```

Experiment Number: 5 (a) **Date:**

STOP AND WAIT PROTOCOL

AIM:

To write a program to simulate the stop and wait protocol.

THEORY:

Sliding window protocols are data link layer protocols for reliable and sequential delivery of data frames. In this protocol, multiple frames can be sent by a sender at a time before receiving an acknowledgment from the receiver. The term sliding window refers to the imaginary boxes to hold frames.

It is the simplest flow control method. In this, the sender will transmit one frame at a time to the receiver. The sender will stop and wait for the acknowledgement from the receiver. When the sender gets the acknowledgement (ACK), it will send the next data packet to the receiver and wait for the disclosure again, and this process will continue as long as the sender has the data to send. The sender and receiver window size is 1.

ALGORITHM:

1. Start the program, declare the variable.
2. Determine the packet size in bytes.
3. In the flow control, the sender sends a single frame to receiver & waits for an acknowledgment.
4. Wait for an event to happen and return its type is event.
5. Fetch a packet from the network layer for transmission on the channel.
6. The next frame is sent by sender only when acknowledgment of previous frame is received.
7. The process of sending a frame & waiting for an acknowledgment continues as long as the sender has data to send.
8. To end up the transmission sender transmits end of transmission (EOT) frame.

OUTPUT:

```
$ gcc main.c
$ ./a.out
Enter the number of frames: 6

number of frames is 6
sending frame 0
ack received for frame 0
sending frame 1
ack received for frame 1
sending frame 2
waiting for 1 seconds
Missing Acknowledgement 2
sending frame 2
ack received for frame 2
sending frame 3
ack received for frame 3
sending frame 4
ack received for frame 4
sending frame 5
waiting for 1 seconds
Missing Acknowledgement 5
sending frame 5
ack received for frame 5

end of stop and wait protocol
```

RESULT:

Thus the program for Stop and Wait protocol was executed and output was verified successfully.

PROGRAM:

```
// Server (Go Back N)
#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<sys/time.h>
#include<netinet/in.h>
#include<string.h>
#include<unistd.h>
#include<arpa/inet.h>
#include<fcntl.h>

int main() {

    int s_sock, c_sock;
    s_sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in server, other;
    memset(&server, 0, sizeof(server));
    memset(&other, 0, sizeof(other));
    server.sin_family = AF_INET;
    server.sin_port = htons(9009);
    server.sin_addr.s_addr = INADDR_ANY;
    socklen_t add;

    if(bind(s_sock, (struct sockaddr*)&server, sizeof(server)) == -1) {
        printf("Binding failed\n");
        return 0;
    }
    printf("Server Up\nGo back n (n=3) used to send 10 messages \n\n");
    listen(s_sock, 10);
    add = sizeof(other);
    c_sock = accept(s_sock, (struct sockaddr*)&other, &add);
    time_t t1,t2;
    char msg[50]="server message: ";
    char buff[50];
    int flag=0;
    fd_set set1,set2,set3;
    struct timeval timeout1,timeout2,timeout3;
    int rv1,rv2,rv3;
    int i=-1;

    qq:
    i=i+1;
    bzero (buff, sizeof (buff));
    char buff2[60];
    bzero (buff2, sizeof (buff2));
    strcpy(buff2,"server message: ");
    buff2[strlen(buff2)] = i+'0';
    buff2[strlen(buff2)] = '\0';
    printf ("Message sent to client: %s \n", buff2);
    write (c_sock, buff2, sizeof (buff2));
    usleep (1000);
    i = i+1;
    bzero (buff2, sizeof (buff2));
    strcpy (buff2, msg);
    buff2[strlen(msg)]=i+'0';
```

Experiment Number: 5 (b) **Date:**

GO BACK N

AIM:

To write a program to simulate the Go Back N.

THEORY:

Sliding window protocols are data link layer protocols for reliable and sequential delivery of data frames. In this protocol, multiple frames can be sent by a sender at a time before receiving an acknowledgment from the receiver. The term sliding window refers to the imaginary boxes to hold frames.

Go-Back-N ARQ protocol is also known as Go-Back-N Automatic Repeat Request. In this, if any frame is corrupted or lost, all subsequent frames have to be sent again. The size of the sender window is N in this protocol. The receiver window size is always 1. If the receiver receives a corrupted frame, it cancels it. The receiver does not accept a corrupted frame. When the timer expires, the sender sends the correct frame again.

PSEUDOCODE:

```
Go-Back-N Sender Algorithm
begin
   frame s;                      //s denotes frame to be sent
   frame t;                      //t is temporary frame
   S_window = power(2,m) – 1;    //Assign maximum window size
   SeqFirst = 0;                 // Sequence number of first frame in window
   SeqN = 0;                     // Sequence number of Nth frame window
   while (true)                  //check repeatedly
   do
      Wait_For_Event();          //wait for availability of packet
      if ( Event(Request_For_Transfer)) then
         //check if window is full
         if (SeqN–SeqFirst >= S_window) then
            doNothing();
         end if;
         Get_Data_From_Network_Layer();
         s = Make_Frame();
         s.seq = SeqN;
         Store_Copy_Frame(s);
         Send_Frame(s);
         Start_Timer(s);
         SeqN = SeqN + 1;
      end if;

    printf ("Message sent to client: %s \n", buff2);
```

```
write (c_sock, buff2, sizeof (buff2));
i=i+1;
usleep (1000);
qqq:
bzero (buff2, sizeof (buff2));
strcpy (buff2, msg);
buff2[strlen (msg)] = i+'0';
printf ("Message sent to client: %s \n", buff2);
write (c_sock, buff2, sizeof(buff2));
FD_ZERO (&set1);
FD_SET (c_sock, &set1);
timeout1.tv_sec = 2;
timeout1.tv_usec = 0;
rv1 = select (c_sock + 1, &set1, NULL, NULL, &timeout1);
if(rv1 == -1)
    perror ("select error ");
else if (rv1 == 0) {
    printf ("Going back from %d: timeout \n",i);
    i = i-3;
    goto qq;
}
else {
    read (c_sock, buff, sizeof (buff));
    printf ("Message from Client: %s\n", buff);
    i++;
    if (i <= 9)
        goto qqq;
}
qq2:
FD_ZERO (&set2);
FD_SET (c_sock, &set2);
timeout2.tv_sec = 3;
timeout2.tv_usec = 0;
rv2 = select (c_sock + 1, &set2, NULL, NULL, &timeout2);
if (rv2 == -1)
    perror ("select error "); // an error accured
else if (rv2 == 0) {
    printf ("Going back from %d:timeout on last 2\n",i-1);
    i=i-2;
    bzero (buff2, sizeof (buff2));
    strcpy (buff2, msg);
    buff2[strlen (buff2)] = i+'0';
    write (c_sock, buff2, sizeof (buff2));
    usleep (1000);
    bzero (buff2, sizeof (buff2));
    i++;
    strcpy (buff2, msg);
    buff2[strlen (buff2)] = i+'0';
    write(c_sock, buff2, sizeof (buff2));
    goto qq2;
} // a timeout occured
else {
    read (c_sock, buff, sizeof (buff));
    printf ("Message from Client: %s\n", buff);
    bzero (buff,sizeof (buff));
    read (c_sock, buff, sizeof (buff));
    printf ("Message from Client: %s\n", buff);
}
```

```
      if ( Event(Frame_Arrival) then
         r = Receive_Acknowledgement();
         if ( AckNo > SeqFirst && AckNo < SeqN ) then
            while ( SeqFirst <= AckNo )
               Remove_copy_frame(s.seq(SeqFirst));
               SeqFirst = SeqFirst + 1;
            end while
            Stop_Timer(s);
         end if
      end if
      // Resend all frames if acknowledgement havn't been received
      if ( Event(Time_Out)) then
         TempSeq = SeqFirst;
         while ( TempSeq < SeqN )
            t = Retrieve_Copy_Frame(s.seq(SeqFirst));

            Send_Frame(t);
            Start_Timer(t);
            TempSeq = TempSeq + 1;
         end while
      end if
end
```

Go-Back-N Receiver Algorithm

```
begin
   frame f;
   RSeqNo = 0;             // Initialise sequence number of expected frame
   while (true)            //check repeatedly
   do
      Wait_For_Event();    //wait for arrival of frame
      if ( Event(Frame_Arrival) then
         Receive_Frame_From_Physical_Layer();
         if ( Corrupted ( f.SeqNo )
            doNothing();
         else if ( f.SeqNo = RSeqNo ) then
            Extract_Data();
            Deliver_Data_To_Network_Layer();
            RSeqNo = RSeqNo + 1;
            Send_ACK(RSeqNo);
         end if
      end if
   end while
end
```

```
    close (c_sock);
    close (s_sock);
    return 0;
}
```

// Client (Go Back N)

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/time.h>
#include<sys/wait.h>
#include<string.h>
#include<unistd.h>
#include<arpa/inet.h>

int main () {

    int c_sock;
    c_sock = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in client;
    memset (&client, 0, sizeof(client));
    client.sin_family = AF_INET;
    client.sin_port = htons (9009);
    client.sin_addr.s_addr = inet_addr ("127.0.0.1");
    if (connect (c_sock, (struct sockaddr *) &client, sizeof (client)) == -1)
    {
        printf ("Connection failed\n");
        return 0;
    }
    printf ("Client -with individual acknowledgement scheme\n\n");
    char msg1[50] = "acknowledgement of: ";
    char msg2[50];
    char buff[100];
    int flag = 1,flg = 1;
    for (int i = 0; i <= 9; i++) {
        flg = 1;
        bzero (buff,sizeof (buff));
        bzero (msg2,sizeof (msg2));
        if(i == 8 && flag == 1){
            printf ("here\n"); //simulating loss
            flag = 0;
            read (c_sock, buff, sizeof (buff));
        }
        int n = read (c_sock, buff, sizeof(buff));
        if (buff[strlen (buff) - 1] != i+'0') { //out of order
            printf ("Discarded as out of order \n");
            i--;
        }
        else {
            printf ("Message received from server: %s \t %d\n", buff, i);
            printf ("Acknowledgement sent for message \n");
            strcpy (msg2, msg1);
            msg2[strlen (msg2)] = i + '0';
            write (c_sock, msg2, sizeof(msg2));
        }
    }
    close(c_sock);
    return 0;
}
```

OUTPUT:

```
// Terminal 1
$ gcc -o server server.c
$ ./server
Server Up
Go back n (n=3) used to send 10 messages


// Terminal 2
$ gcc -o client client.c
$ ./client
Client -with individual acknowledgement scheme

Message received from server: server message: 0          0
Acknowledgement sent for message
Message received from server: server message: 1          1
Acknowledgement sent for message
Message received from server: server message: 2          2
Acknowledgement sent for message
Message received from server: server message: 3          3
Acknowledgement sent for message
Message received from server: server message: 4          4
Acknowledgement sent for message
Message received from server: server message: 5          5
Acknowledgement sent for message
Message received from server: server message: 6          6
Acknowledgement sent for message
Message received from server: server message: 7          7
Acknowledgement sent for message
here
Discarded as out of order
Message received from server: server message: 8          8
Acknowledgement sent for message
Message received from server: server message: 9          9
Acknowledgement sent for message
```

```
// Terminal 1
$ gcc -o server server.c
$ ./server
Server Up
Go back n (n=3) used to send 10 messages

Message sent to client: server message: 0
Message sent to client: server message: 1
Message sent to client: server message: 2
Message from Client: acknowledgement of: 0
Message sent to client: server message: 3
Message from Client: acknowledgement of: 1
Message sent to client: server message: 4
Message from Client: acknowledgement of: 2
Message sent to client: server message: 5
Message from Client: acknowledgement of: 3
Message sent to client: server message: 6
Message from Client: acknowledgement of: 4
Message sent to client: server message: 7
Message from Client: acknowledgement of: 5
Message sent to client: server message: 8
Message from Client: acknowledgement of: 6
Message sent to client: server message: 9
Message from Client: acknowledgement of: 7
Going back from 9:timeout on last 2
Message from Client: acknowledgement of: 8
Message from Client: acknowledgement of: 9
```

RESULT:

The program for Go Back N protocol was executed and output was verified successfully.

PROGRAM:

```
// Server (Selective Repeat)
#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<sys/time.h>
#include<netinet/in.h>
#include<string.h>
#include<unistd.h>
#include<arpa/inet.h>
#include<fcntl.h>

void rsendd(int ch,int c_sock) {
    char buff2[60];
    bzero(buff2,sizeof(buff2));
    strcpy(buff2,"reserver message :");
    buff2[strlen(buff2)]=(ch)+'0';
    buff2[strlen(buff2)]='\0';
    printf("Resending Message to client :%s \n",buff2);
    write(c_sock, buff2, sizeof(buff2));
    usleep(1000);
}

int main() {
    int s_sock, c_sock;
    s_sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in server, other;
    memset(&server, 0, sizeof(server));
    memset(&other, 0, sizeof(other));
    server.sin_family = AF_INET;
    server.sin_port = htons(9009);
    server.sin_addr.s_addr = INADDR_ANY;
    socklen_t add;

    if(bind(s_sock, (struct sockaddr*)&server, sizeof(server)) == -1) {
        printf("Binding failed\n");
        return 0;
    }

    printf("Server Up\nSelective repeat scheme\n\n");
    listen(s_sock, 10);
    add = sizeof(other);
    c_sock = accept(s_sock, (struct sockaddr*)&other, &add);

    time_t t1,t2;
    char msg[50]="server message :";
    char buff[50];
    int flag=0;
    fd_set set1,set2,set3;
    struct timeval timeout1,timeout2,timeout3;
    int rv1,rv2,rv3;
    int tot=0;
    int ok[20];

    memset(ok,0,sizeof(ok));
```

Experiment Number: 5 (c) **Date:**

SELECTIVE REPEAT

AIM:

To write a program to simulate the Selective Repeat protocol.

THEORY:

Sliding window protocols are data link layer protocols for reliable and sequential delivery of data frames. In this protocol, multiple frames can be sent by a sender at a time before receiving an acknowledgment from the receiver. The term sliding window refers to the imaginary boxes to hold frames.

Selective Repeat ARQ is also known as the Selective Repeat Automatic Repeat Request.In this protocol, the size of the sender window is always equal to the size of the receiver window. The size of the sliding window is always greater than 1.If the receiver receives a corrupt frame, it does not directly discard it. It sends a negative acknowledgment to the sender. The sender sends that frame again as soon as on the receiving negative acknowledgment.

PSEUDOCODE:

Selective Repeat Sender Algorithm

```
begin
   frame s; //s denotes frame to be sent
   frame t; //t is temporary frame
   S_window = power(2,m-1); //Assign maximum window size
   SeqFirst = 0; // Sequence number of first frame in window
   SeqN = 0; // Sequence number of Nth frame window
   while (true) //check repeatedly
      do
         Wait_For_Event(); //wait for availability of packet
         if ( Event(Request_For_Transfer)) then
            //check if window is full
            if (SeqN-SeqFirst >= S_window) then
               doNothing();
            end if;
            Get_Data_From_Network_Layer();
            s = Make_Frame();
            s.seq = SeqN;
            Store_Copy_Frame(s);
            Send_Frame(s);
            Start_Timer(s);
            SeqN = SeqN + 1;
         end if;
```

```
while(tot < 9){
        int toti=tot;
        for(int j=(0+toti);j<(3+toti);j++){
            bzero(buff,sizeof(buff));
            char buff2[60];
            bzero(buff2,sizeof(buff2));
            strcpy(buff2,"server message :");
            buff2[strlen(buff2)]=(j)+'0';
            buff2[strlen(buff2)]='\0';
            printf("Message sent to client :%s \t%d\t%d\n",buff2,tot,j);
            write(c_sock, buff2, sizeof(buff2));
            usleep(1000);
        }
        for(int k=0+toti;k<(toti+3);k++){
            qq:
            FD_ZERO(&set1);
            FD_SET(c_sock, &set1);
            timeout1.tv_sec = 2;
            timeout1.tv_usec = 0;

            rv1 = select(c_sock + 1, &set1, NULL, NULL, &timeout1);
            if(rv1 == -1)
                perror("select error ");
            else if(rv1 == 0) {
                printf("Timeout for message :%d \n",k);
                rsendd(k,c_sock);
                goto qq;
            } // a timeout occured
            else {
                read(c_sock, buff, sizeof(buff));
                printf("Message from Client: %s\n", buff);
                if(buff[0]=='n'){
                printf(" corrupt message acknowledgement (msg %d)
\n",buff[strlen(buff)-1]-'0');
                rsendd((buff[strlen(buff)-1]-'0'),c_sock);
                goto qq;}
                else
                tot++;
            }
        }
    }

    close(c_sock);
    close(s_sock);

    return 0;
}
```

// Client (Selective Repeat)

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <string.h>
```

```
         if ( Event(Frame_Arrival) then
            r = Receive_Acknowledgement();

            //Resend frame whose sequence number is with ACK
            if ( r.type = NAK) then
               if ( NAK_No > SeqFirst && NAK_No < SeqN ) then
                  Retransmit( s.seq(NAK_No));
                  Start_Timer(s);
               end if
                  //Remove frames from sending window with positive ACK
               else if ( r.type = ACK ) then
                  Remove_Frame(s.seq(SeqFirst));
                  Stop_Timer(s);
                  SeqFirst = SeqFirst + 1;
               end if
         end if
             // Resend frame if acknowledgement haven't been received
         if ( Event(Time_Out)) then
            Start_Timer(s);
            Retransmit_Frame(s);
         end if
end
```

Selective Repeat Receiver Algorithm

```
begin
   frame f;
   RSeqNo = 0; // Initialise sequence number of expected frame
   NAKsent = false;
   ACK = false;
   For each slot in receive_window
   Mark(slot)=false;
   while (true) //check repeatedly
      do
         Wait_For_Event(); //wait for arrival of frame
         if ( Event(Frame_Arrival) then
            Receive_Frame_From_Physical_Layer();
            if ( Corrupted ( f.SeqNo ) AND NAKsent = false) then
               SendNAK(f.SeqNo);
               NAKsent = true;
            end if
            if ( f.SeqNo != RSeqNo AND NAKsent = false ) then
               SendNAK(f.SeqNo);
               NAKsent = true;
               if ( f.SeqNo is in receive_window ) then
                  if ( Mark(RSeqNo) = false ) then
                     Store_frame(f.SeqNo);
                     Mark(RSeqNo) = true;
                  end if
               end if
```

```
#include <unistd.h>
#include <arpa/inet.h>

int isfaulty(){ //simulating corruption of message
    int d=rand()%4;
    return (d>2);
}

int main() {

    srand(time(0));
    int c_sock;
    c_sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in client;
    memset(&client, 0, sizeof(client));
    client.sin_family = AF_INET;
    client.sin_port = htons(9009);
    client.sin_addr.s_addr = inet_addr("127.0.0.1");

    if(connect(c_sock, (struct sockaddr*)&client, sizeof(client)) == -1) {
        printf("Connection failed");
        return 0;
    }

    printf("\nClient -with individual acknowledgement scheme\n\n");
    char msg1[50]="acknowledgement of ";
    char msg3[50]="negative ack ";
    char msg2[50];
    char buff[100];
    int count=-1,flag=1;

    while(count<8) {
        bzero(buff,sizeof(buff));
        bzero(msg2,sizeof(msg2));

        if(count==7&&flag==1){
            printf("here\n"); //simulate loss
            flag=0;
            read(c_sock,buff,sizeof(buff));
            continue;
        }

        int n = read(c_sock, buff, sizeof(buff));
        char i=buff[strlen(buff)-1];
        printf("Message received from server : %s \n",buff);
        int isfault=isfaulty();
        printf("corruption status : %d \n",isfault);
        printf("Response/acknowledgement sent for message \n");

        if(isfault)
            strcpy(msg2,msg3);
        else {
            strcpy(msg2,msg1);
            count++;
        }
        msg2[strlen(msg2)]=i;
        write(c_sock,msg2, sizeof(msg2));
    }
    close (c_sock);
```

```
               else
                  while ( Mark(RSeqNo))
                     Extract_Data(RSeqNo);
                     Deliver_Data_To_Network_Layer();
                     RSeqNo = RSeqNo + 1;
                     Send_ACK(RSeqNo);
                  end while
            end if
         end if
   end while
end
```

```
    return 0;
}
```

OUTPUT:

```
// Terminal 1
$ gcc -o server server.c
$ ./server
Server Up
Selective repeat scheme

Message sent to client :server message :0       0       0
Message sent to client :server message :1       0       1
Message sent to client :server message :2       0       2
Message from Client: acknowledgement of 0
Message from Client: acknowledgement of 1
Message from Client: acknowledgement of 2
Message sent to client :server message :3       3       3
Message sent to client :server message :4       3       4
Message sent to client :server message :5       3       5
Message from Client: acknowledgement of 3
Message from Client: acknowledgement of 4
Message from Client: negative ack 5
 corrupt message acknowledgement (msg 5)
Resending Message to client :reserver message :5
Message from Client: acknowledgement of 5
Message sent to client :server message :6       6       6
Message sent to client :server message :7       6       7
Message sent to client :server message :8       6       8
Message from Client: acknowledgement of 6
Message from Client: negative ack 7
 corrupt message acknowledgement (msg 7)
Resending Message to client :reserver message :7
Message from Client: acknowledgement of 8
Timeout for message :8
Resending Message to client :reserver message :8
Message from Client: negative ack 8
 corrupt message acknowledgement (msg 8)
Resending Message to client :reserver message :8
Message from Client: acknowledgement of 8
```

```
// Terminal 2
$ gcc -o client client.c
$ ./ client
Client -with individual acknowledgement scheme

Message received from server : server message :0
corruption status : 0
Response/acknowledgement sent for message
Message received from server : server message :1
corruption status : 0
Response/acknowledgement sent for message
Message received from server : server message :2
corruption status : 0
Response/acknowledgement sent for message
Message received from server : server message :3
corruption status : 0
Response/acknowledgement sent for message
Message received from server : server message :4
corruption status : 0
Response/acknowledgement sent for message
Message received from server : server message :5
corruption status : 1
Response/acknowledgement sent for message
Message received from server : reserver message :5
corruption status : 0
Response/acknowledgement sent for message
Message received from server : server message :6
corruption status : 0
Response/acknowledgement sent for message
Message received from server : server message :7
corruption status : 1
Response/acknowledgement sent for message
Message received from server : server message :8
corruption status : 0
Response/acknowledgement sent for message
here
Message received from server : reserver message :8
corruption status : 1
Response/acknowledgement sent for message
Message received from server : reserver message :8
corruption status : 0
Response/acknowledgement sent for message
```

<u>RESULT</u>:

The program for Selective Repeat was executed and output was verified successfully.

PROGRAM:

```
// Distance Vector Routing
#include<stdio.h>
struct node
{
    unsigned dist[20];
    unsigned from[20];
} rt[10];

int main()
{

    int costmat[20][20];
    int nodes, i, j, k, count=0;
    printf ("\n Distance Vector Routing");
    printf ("\n\nEnter the number of nodes : ");
    scanf ("%d",&nodes);    // Enter the nodes
    printf ("\nEnter the cost matrix :\n");
    for (i = 0; i < nodes; i++)
    {
        for (j = 0; j < nodes; j++)
        {
            scanf ("%d", &costmat[i][j]);
            costmat[i][i] = 0;
            rt[i].dist[j] = costmat[i][j];  // initialise the distance equal to
cost matrix
            rt[i].from[j] = j;
        }
    }
    do
    {
        count = 0;
        for (i = 0; i < nodes; i++)  // We choose arbitary vertex k and we
calculate the direct distance from the node i to k using the cost matrix and add
the distance from k to node j
            for (j = 0; j < nodes; j++)
                for( k = 0; k < nodes; k++)
                    if (rt[i].dist[j] > costmat[i][k] + rt[k].dist[j])
                    {   //We calculate the minimum distance
                        rt[i].dist[j] = rt[i].dist[k] + rt[k].dist[j];
                        rt[i].from[j] = k;
                        count++;
                    }
    } while (count != 0);
    for (i = 0; i < nodes; i++)
    {
        printf ("\n\nFor router %d:\n", i+1);
        for(j = 0; j < nodes; j++)
        {
            printf ("\t\nnode %d via %d distance %d ", j + 1, rt[i].from[j] + 1,
rt[i].dist[j]);
        }
    }
```

Experiment Number: 6 (a) **Date:**

DISTANCE VECTOR ROUTING

AIM:

To write a program to implement the Distance Vector Routing.

THEORY:

Distance Vector Routing protocol is a dynamic routing protocol. With this protocol, every router in the network creates a routing table which helps them in determining the shortest path through the network. All the routers in the network are aware of every other router in the network and they keep on updating their routing table periodically. This protocol uses the principle of Bellman-Ford's algorithm.

Distance-vector routing protocols measure the distance by the number of routers a packet has to pass; one router counts as one hop. To determine the best route across a network, routers using a distance-vector protocol exchange information with one another, usually routing tables plus hop counts for destination networks and possibly other traffic information.

PSEUDOCODE:

```
begin
      for all destinations y in N:
            Dx(y) = c(x,y)     // If y is not a neighbour then c(x,y) = ∞
      for each neighbour w
            Dw(y) = ?     for all destination y in N.
      for each neighbour w
            send distance vector Dx = [ Dx(y) : y in N ] to w
      loop
            wait (until I receive any distance vector from some neighbour w)
            for each y in N:
                  Dx(y) = minv{c(x,v)+Dv(y)}
            If Dx(y) is changed for any destination y
                  Send distance vector Dx = [ Dx(y) : y in N ] to all neighbours
      forever
end
```

```
    printf("\n\n");
}
```

OUTPUT:

```
$ gcc main.c
$ ./a.out
Distance Vector Routing

Enter the number of nodes : 3

Enter the cost matrix :
0  2  7
2  0  1
7  1  0


For router 1:

node 1 via 1 distance 0
node 2 via 2 distance 2
node 3 via 2 distance 3

For router 2:

node 1 via 1 distance 2
node 2 via 2 distance 0
node 3 via 3 distance 1

For router 3:

node 1 via 2 distance 3
node 2 via 2 distance 1
node 3 via 3 distance 0
```

RESULT:

The program for Distance Vector Routing was executed and output was verified successfully.

PROGRAM:

```
// Server (SMTP)
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<sys/types.h>
#include<netinet/in.h>
#define BUF_SIZE 256

int main(int argc, char* argv[])
{
    struct sockaddr_in server, client;
    char str[50], msg[20];

    if(argc != 2)
        printf ("Input format not correct!");
    int sockfd = socket (AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1)
        printf ("Error in socket()!");

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons (atoi (argv[1]));

    client.sin_family = AF_INET;
    client.sin_addr.s_addr = INADDR_ANY;
    client.sin_port = htons (atoi (argv[1]));

    if (bind (sockfd, (struct sockaddr *) &server, sizeof(server)) < 0)
        printf ("Error in bind()! \n");

    socklen_t client_len = sizeof (client);
    printf ("Server waiting...");
    sleep (3);
    if (recvfrom (sockfd, str, 100, 0, (struct sockaddr *) &client, &client_len) <
0)
        printf ("Error in recvfrom()!");
    printf ("\nGot message from client: %s", str);

    printf ("\nSending greeting message to client...");
    strcpy (str, "220 127.0.0.1");
    sleep (10);
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client, sizeof
(client)) < 0)
        printf ("Error in sendto()!");
    sleep (3);
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client,
&client_len)) < 0)
        printf ("Error in recvfrom()!");
    if (strncmp (str, "HELLO", 5))
        printf ("\n'HELLO' expected from client...");
```

Experiment Number: 7 **Date:**

SIMPLE MAIL TRANSFER PROTOCOL

AIM:

To implement Simple Mail Transfer Protocol.

THEORY:

SMTP is part of the application layer of the TCP/IP protocol and traditionally operates on port 25. It utilizes a process called “store and forward” which is used to orchestrate sending your email across different networks. Within the SMTP protocol, there are smaller software services called Mail Transfer Agents that help manage the transfer of the email and its final delivery to a recipient’s mailbox. Not only does SMTP define this entire communication flow, it can also support delayed delivery of an email either at the sender site, receiver site, or at any intermediate server.

ALGORITHM:

```
1. Initialize a UDF based socket connection from client to server
2. Client sends greeting messages to server
3. A greeting message with code 220 is sent from server to client
4. HELLO is sent from client to server to introduce itself
5. Server responds with code 250 and its name
6. MAIL FROM is sent from client to server describing name of the sender of the
email.
7. OK from server with code 250 is sent to convey that the sender email address
has been accepted.
8. RCPT TO is sent from client to server describing recipient name.
9. OK from server with 250 is sent to convey acceptance of recipient email
address.
10. Data is sent from Client to Server to denote start of the email body.
11. Go ahead with body of email message is send from server with code 354.
12. Body of email is sent from client to server.
13. OK received from the server.
14. QUIT received from Client to Server.
15. Response to QUIT sent from server with code 221.
```

```
    printf ("\n%s", str);

    printf ("\nSending response...");
    strcpy( str, "250 ok");
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client, sizeof
(client)) < 0)
        printf ("Error in sendto()!");
    sleep (3);
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client,
&client_len)) < 0)
        printf ("Error in recvfrom()!");
    if (strncmp(str, "MAIL FROM", 9))
        printf ("'MAIL FROM' expected from client...");
    printf ("\n%s",str);

    printf ("\nSending response...");
    strcpy (str, "250 ok");
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client,
sizeof(client)) < 0)
        printf ("Error in sendto()!");
    sleep (3);
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client,
&client_len)) < 0)
        printf ("Error in recvfrom()!");
    if (strncmp (str, "RCPT TO", 7))
        printf ("\n'RCPT' TO expected from client...");
    printf ("\n%s", str);

    printf ("\nSending response...");
    strcpy (str, "250 ok");
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client,
sizeof(client)) < 0)
        printf("Error in sendto()!");
    sleep(3);
    if ((recvfrom(sockfd, str, sizeof (str), 0, (struct sockaddr *) &client,
&client_len)) < 0)
        printf ("Error in recvfrom()!");
    if (strncmp (str, "DATA", 4))
        printf ("\n'DATA' expected from client...");
    printf ("\n%s", str);

    printf ("\nSending response...");
    strcpy (str, "354 Go ahead");
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client, sizeof
(client)) < 0)
        printf("Error in sendto()!");
    if ((recvfrom(sockfd, msg, sizeof (str), 0, (struct sockaddr *) &client,
&client_len)) < 0)
        printf ("Error in recvfrom()!");
    printf ("\nmail body received");
    printf ("\n%s", msg);

    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &client,
&client_len)) < 0)
        printf ("Error in recvfrom()!");
    if (strncmp (str, "QUIT", 4))
        printf ("'QUIT' expected from client...");
    printf ("\nSending quit...\n");
    strcpy (str, "221 OK");
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *)& client, sizeof
(client)) < 0)
        printf ("Error in sendto()!");
    close (sockfd);
    return 0;
}
```

// Client (SMTP)

```
#include<string.h>
#include<sys/socket.h>
#include<netdb.h>
#include<stdlib.h>
#include<stdio.h>
#include<netinet/in.h>
#include<sys/types.h>
#include<netinet/in.h>
#define BUF_SIZE 256

int main(int argc, char* argv[])
{
    struct sockaddr_in server, client;
    char str[50] = "hi";
    char mail_f[50], mail_to[50], msg[20], c;
    int t = 0;
    socklen_t l = sizeof (server);
    if (argc != 3)
        printf ("Input format not correct!");
    int sockfd = socket (AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1)
        printf ("Error in socket()!");

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons (atoi (argv[2]));

    client.sin_family = AF_INET;
    client.sin_addr.s_addr = INADDR_ANY;
    client.sin_port = htons (atoi (argv[2]));

    printf ("Sending 'hi' to server...");
    sleep (10);
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr*) &server, sizeof
(server)) < 0)
        printf ("Error in sendto()!");
    if (recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, &l) <
0)
        printf ("Error in recvfrom()!");
    printf ("\ngreeting msg is %s", str);
    if (strncmp (str, "220", 3))
        printf ("\nConnection not established \ncode 220 expected");
    printf ("\nSending 'HELLO'...");
    strcpy (str, "HELLO 127.0.0.1");
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, sizeof
(server)) < 0)
        printf ("Error in sendto()!");
    sleep (3);
    printf ("\nReceiving from server");
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, &l))
< 0)
        printf ("Error in recvfrom()!");
    if (strncmp (str, "250", 3))
        printf ("\nOk not received from server!");
    printf ("\nServer has send %s",str);

    printf ("\n\nEnter FROM address: ");
    scanf ("%s", mail_f);
    strcpy (str, "MAIL FROM ");
    strcat (str, mail_f);
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, sizeof
(server)) < 0)
    printf ("Error in sendto()!");
    sleep (3);
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, &l))
< 0)
    printf ("Error in recvfrom()!");
    if (strncmp(str, "250", 3))
    printf ("\nOk not received from server!");

    printf ("%s", str);
    printf ("\nEnter TO address: ");
    scanf ("%s", mail_to);
    strcpy (str, "RCPT TO ");
    strcat (str, mail_to);
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, sizeof
(server)) < 0)
        printf ("Error in sendto()!");
    sleep (3);
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, &l))
< 0)
        printf ("Error in recvfrom()!");
    if (strncmp (str, "250", 3))
        printf ("\nOk not received from server!");

    printf ("%s", str);
    printf ("\nSending 'DATA' to server...");
    strcpy (str, "DATA");
    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, sizeof
(server)) < 0)
        printf ("Error in sendto()!");
    sleep (3);
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, &l))
< 0)
        printf ("Error in recvfrom()!");
    if (strncmp (str, "354", 3))
        printf ("\nOk not received from server!");
    printf ("%s", str);
    printf ("\nEnter mail body: ");
    while (1)
    {
        c = getchar ();
        if (c == '$')
        {
            msg[t] = '\0';
            break;
        }
        if (c == '\0')
            continue;
        msg[t++] = c;
    }

    if (sendto (sockfd, msg, sizeof (msg), 0, (struct sockaddr *) &server,
sizeof(server)) < 0)
        printf("Error in sendto()!");
    sleep(3);
    printf("\nSending 'QUIT' to server...");
    strcpy(str,"QUIT");

    if (sendto (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server,sizeof
(server)) < 0)
        printf("Error in sendto()!");
    if ((recvfrom (sockfd, str, sizeof (str), 0, (struct sockaddr *) &server, &l))
< 0)
        printf ("Error in recvfrom()!");

    if (strncmp (str, "221", 3))
        printf ("\nOk not received from server!");

    printf ("\nServer has send GOODBYE... \nClosing connection...\n");
    printf ("Bye\n");

    close (sockfd);
    return 0;
}
```

OUTPUT:

```
// Terminal 1
$ gcc -o server server.c
$ ./server 3000
Server waiting...
Got message from client: hi
Sending greeting message to client...
HELLO 127.0.0.1
Sending response...
MAIL FROM studentforum.mec@gmail.com
Sending response...
RCPT TO studentforum.cet@gmail.com
Sending response...
DATA
Sending response...
mail body received

hi programmers!
Sending quit...

// Terminal 2
$ gcc -o client client.c
$ ./client localhost 3000
Sending 'hi' to server...
greeting msg is 220 127.0.0.1
Sending 'HELLO'...
Receiving from server
Server has send 250 ok

Enter FROM address: studentforum.mec@gmail.com
250 ok
Enter TO address: studentforum.cet@gmail.com
250 ok
Sending 'DATA' to server...354 Go ahead
Enter mail body: hi programmers! $

Sending 'QUIT' to server...
Server has send GOODBYE...
Closing connection...
Bye
```

RESULT:

The programs for client-server communication for SMTP was executed and output was verified successfully.

PROGRAM:

```
// Server (FTP)
#include<stdio.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    FILE *fp;
    int sd, newsd, ser, n, a, cli, pid, bd, port, clilen;
    char name[100], fileread[100], fname[100], ch, file[100], rcv[100];
    struct sockaddr_in servaddr,cliaddr;

    printf ("Enter the port address: ");
    scanf ("%d", &port);

    sd=socket(AF_INET, SOCK_STREAM, 0);
    if(sd < 0)
        printf ("Cant create!\n");
    else
        printf ("Socket is created.\n");

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons(port);
    a = sizeof (servaddr);
    bd = bind (sd, (struct sockaddr *) &servaddr, a);

    if (bd < 0)
        printf ("Cant bind!\n");
    else
        printf ("Binded.\n");

    listen (sd, 5);
    clilen = sizeof (cliaddr);
    newsd = accept (sd, (struct sockaddr *) &cliaddr, &clilen);
    if (newsd < 0)
    {
        printf ("Can't accept!\n");
    }
    else
        printf ("Accepted.\n");

    n = recv (newsd, rcv, 100, 0);
    rcv[n] = '\0';
    fp = fopen (rcv, "r");
    if (fp == NULL)
    {
        send (newsd, "error!", 5, 0);
        close (newsd);
    }
```

Experiment Number: 8 **Date:**

FILE TRANSFER PROTOCOL

AIM:

To implement File Transfer Protocol.

THEORY:

FTP, which stands for File Transfer Protocol, was developed in the 1970s to allow files to be transferred between a client and a server on a computer network. The FTP protocol uses two separate channels — the command (or control) channel and the data channel — to exchange files. The command channel is responsible for accepting client connections and executing other simple commands. It typically uses server port 21. FTP clients will connect to this port to initiate a conversation for file transfer and authenticate themselves by sending a username and password.

After authentication, the client and server will then negotiate a new common server port for the data channel, over which the file will be transferred. Once the file transfer is complete, the data channel is closed. If multiple files are to be sent concurrently, a range of data channel ports must be used. The control channel remains idle until the file transfer is complete. It then reports that the file transfer was either successful or failed.

ALGORITHM:

Server

1. Create socket using socket() system call with address family AF_INET, type SOCK_STREAM and default protocol.
2. Bind server address and port using bind() system call.
3. Wait for client connection to complete accepting connections using accept() system call.
4. Receive the client file using recv() system call.
*5. Using fgets (char *str, int n, FILE *stream) function, we need a line of text from the specified stream and store it into the string pointed by str. It stops when either (n-1) characters are read or when the end of file is reached.*
6. On successful execution is when the file pointer reaches the end of file, file transfer “completed” message is sent by server to accepted client connection.

```
    else
    {
        while (fgets (fileread, sizeof (fileread), fp))
        {
            if(send (newsd, fileread, sizeof (fileread),0) < 0)
            {
                printf ("Cannot send file contents!\n");
            }
            sleep (1);
        }
        if  (!fgets (fileread, sizeof (fileread), fp))
        {
            //when file pointer reaches end of file, file transfer “completed”
message is send to accepted client connection using newsd, socket file descriptor.
            send (newsd, "completed", 999999999, 0);
        }
        return (0);
    }
}
```

// Client (FTP)

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>

int main ()
{
    FILE *fp;
    int csd, n, ser, s, cli, cport, newsd;
    char name[100], rcvmsg[100], rcvg[100], fname[100];
    struct sockaddr_in servaddr;

    printf ("Enter the port: ");
    scanf ("%d", &cport);

    csd = socket (AF_INET, SOCK_STREAM, 0);
    if (csd < 0)
    {
        printf ("Error!!\n");
        exit (0);
    }
    else
        printf ("Socket is created.\n");

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons (cport);

    if (connect (csd, (struct sockaddr *) &servaddr, sizeof (servaddr)) < 0)
        printf ("Error in connection!\n");
    else
        printf ("connected.\n");
```

Client

1. Create socket using socket() system call with address family AF_INET, type SOCK_STREAM and default protocol.
2. Enter the client port number.
3. Fill in the internal socket address structure
4. Connect to the server address using connect() system call.
5. Read the existing and new file name from the user.
6. Send existing file to server using send() system call.
7. Receive feedback from server “completed” regarding file transfer completion.
8. Write file is transferred to the standard output stream.
9. Close the socket connection and file pointer

```
    printf ("Enter the existing file name:\t");
    scanf ("%s", name);
    printf ("Enter the new file name:\t");
    scanf ("%s", fname);
    fp = fopen (fname, "w");
    send (csd, name, sizeof (name), 0);

    while (1)
    {
        s = recv (csd, rcvg, 100, 0);
        rcvg[s] = '\0';
        if (strcmp (rcvg, "error") == 0)
            printf ("File is not available!\n");
        if (strcmp (rcvg, "completed") == 0)
        {
            printf ("File is transferred...\n");
            fclose (fp);
            close (csd);
            break;
        }
        else
            fputs (rcvg, stdout);
        fprintf (fp, "%s", rcvg);
        printf ("\n");
        return 0;
    }
}
```

OUTPUT:

```
// Terminal 1
$ gcc -o server server.c
$ ./server
Enter the port address: 3000
Socket is created.
Binded.
_


// Terminal 2
$ gcc -o client client.c
$ ./client
Enter the port: 3000
Socket is created.
connected.
Enter the existing file name:   existing-file.txt
Enter the new file name:        new-file.txt
Hi Programmers!


// Terminal 1
$ gcc -o server server.c
$ ./server
Enter the port address: 3000
Socket is created.
Binded.
Accepted.
```

RESULT:

The programs for client-server communication for FTP was executed and output was verified successfully.

PROGRAM:

```c
// Leaky Bucket
#include<stdio.h>
#include<stdlib.h>

struct packet {
    int time;
    int size;
};

int main () {

    int i, n, m, k = 0;
    int b_size, b_current, b_outrate;

    printf ("Enter the no. of packets: ");
    scanf ("%d", &n);

    struct packet p[n];

    printf ("\nEnter the packets in the order of their arrival time: \n");
    for (i = 0; i < n; i++) {
        printf ("Enter the time and size: ");
        scanf ("%d %d", &p[i].time, &p[i].size);
    }

    printf ("\nEnter the bucket size: ");
    scanf ("%d", &b_size);
    printf ("Enter the output rate: ");
    scanf ("%d", &b_outrate);

    m = p[n - 1].time;
    i = 1;
    k = 0;
    b_current = 0;
    while (i <= m || b_current != 0) {
        printf("\nAt time %d", i);

        if (p[k].time == i) {
            if (b_size >= b_current + p[k].size) {
                b_current = b_current + p[k].size;
                printf("\n%d byte packet is inserted", p[k].size);
                k = k + 1;
            }
            else {
                printf("\n%d byte packet is discarded", p[k].size);
                k = k + 1;
            }
        }

        if (b_current == 0) {
            printf("\nNo packets to transmit");
        }
        else if (b_current >= b_outrate) {
            b_current = b_current - b_outrate;
            printf("\n%d bytes transferred", b_outrate);
        }
```

Experiment Number: 9 **Date:**

LEAKY BUCKET

AIM:

To write a program to implement the Leaky Bucket.

THEORY:

The Leaky Bucket Algorithm used to control rate in a network. It is implemented as a single-server queue with constant service time. If the bucket (buffer) overflows then packets are discarded. It enforces a constant output rate (average rate) regardless of the burstiness of the input. It does nothing when input is idle. The host injects one packet per clock tick onto the network. This results in a uniform flow of packets, smoothing out bursts and reducing congestion. When packets are the same size (as in ATM cells), the one packet per tick is okay. For variable length packets though, it is better to allow a fixed number of bytes per tick.

ALGORITHM:

1. Start.
2. Set the bucket size on the buffer size.
3. Set the output rate.
4. Transmit the packet such that there is no overflow.
5. Repeat the process of transmission at the output rate until all packets are transmitted (reject packets where its size is greater than the bucket size).
6. Stop.

```
        else {
            printf("\n%d bytes transferred", b_current);
            b_current = 0;
        }

        printf("\nPackets in the bucket %d byte(s)\n", b_current);
        i++;
    }

    return 0;
}
```

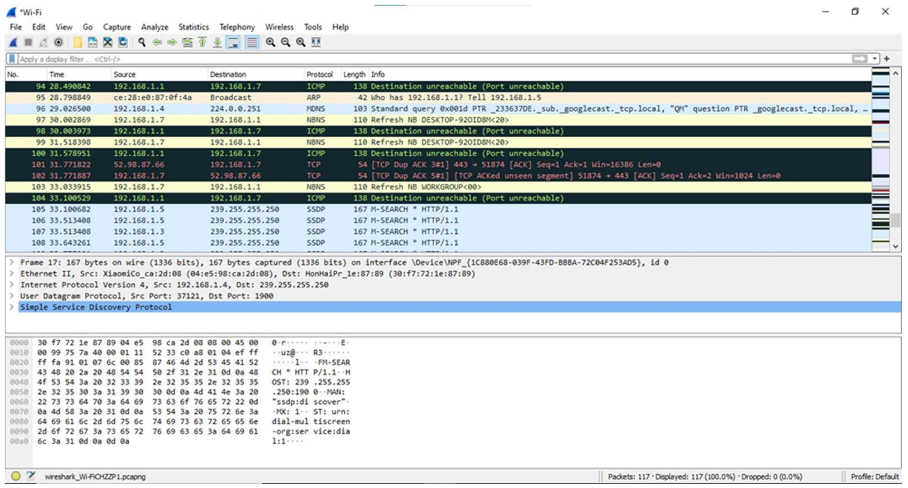
OUTPUT:

```
$ gcc main.c
$ ./a.out
Enter the no. of packets: 3

Enter the packets in the order of their arrival time:
Enter the time and size: 2 600
Enter the time and size: 4 700
Enter the time and size: 5 600

Enter the bucket size: 1000
Enter the output rate: 500

At time 1
No packets to transmit
Packets in the bucket 0 byte(s)

At time 2
600 byte packet is inserted
500 bytes transferred
Packets in the bucket 100 byte(s)

At time 3
100 bytes transferred
Packets in the bucket 0 byte(s)

At time 4
700 byte packet is inserted
500 bytes transferred
Packets in the bucket 200 byte(s)

At time 5
600 byte packet is inserted
500 bytes transferred
Packets in the bucket 300 byte(s)

At time 6
300 bytes transferred
Packets in the bucket 0 byte(s)
```

RESULT:

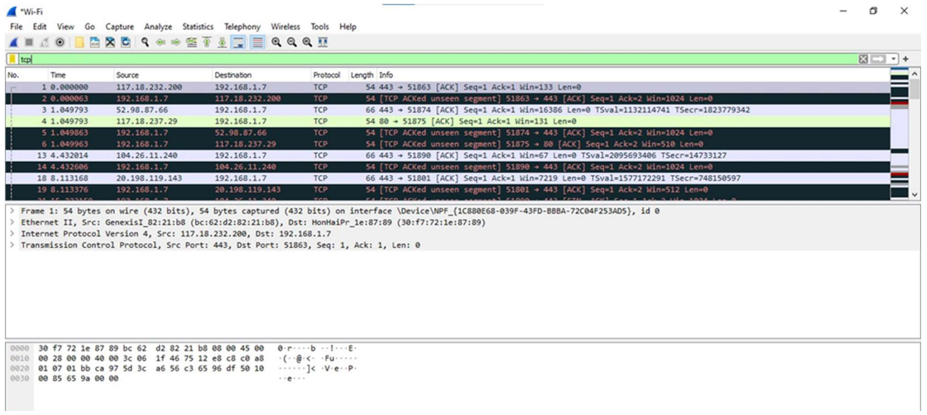
The program for Leaky Bucket was executed and output was verified successfully.

OUTPUT:


*Wi-Fi
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time       Source             Destination      Protocol Length Info
  94 28.490842  192.168.1.1        192.168.1.7      ICMP     138 Destination unreachable (Port unreachable)
  95 28.798849  ce:28:e0:87:0f:4a  Broadcast        ARP       42 Who has 192.168.1.1? Tell 192.168.1.5
  96 29.026500  192.168.1.4        224.0.0.251      MDNS     103 Standard query 0x001d PTR _233637DE._sub._googlecast._tcp.local, "QM" question PTR _googlecast._tcp.local, …
  97 30.002869  192.168.1.7        192.168.1.1      NBNS     110 Refresh NB DESKTOP-920ID8M<20>
  98 30.003973  192.168.1.1        192.168.1.7      ICMP     138 Destination unreachable (Port unreachable)
  99 31.518398  192.168.1.7        192.168.1.1      NBNS     110 Refresh NB DESKTOP-920ID8M<20>
 100 31.578951  192.168.1.1        192.168.1.7      ICMP     138 Destination unreachable (Port unreachable)
 101 31.771822  52.98.87.66        192.168.1.7      TCP       54 [TCP Dup ACK 3#1] 443 → 51874 [ACK] Seq=1 Ack=1 Win=16386 Len=0
 102 31.771887  192.168.1.7        52.98.87.66      TCP       54 [TCP Dup ACK 5#1] [TCP ACKed unseen segment] 51874 → 443 [ACK] Seq=1 Ack=2 Win=1024 Len=0
 103 33.033915  192.168.1.7        192.168.1.1      NBNS     110 Refresh NB WORKGROUP<00>
 104 33.100529  192.168.1.1        192.168.1.7      ICMP     138 Destination unreachable (Port unreachable)
 105 33.100682  192.168.1.5        239.255.255.250  SSDP     167 M-SEARCH * HTTP/1.1
 106 33.513408  192.168.1.5        239.255.255.250  SSDP     167 M-SEARCH * HTTP/1.1
 107 33.513408  192.168.1.3        239.255.255.250  SSDP     167 M-SEARCH * HTTP/1.1
 108 33.643261  192.168.1.5        239.255.255.250  SSDP     167 M-SEARCH * HTTP/1.1

> Frame 17: 167 bytes on wire (1336 bits), 167 bytes captured (1336 bits) on interface \Device\NPF_{1C880E68-039F-43FD-BBBA-72C04F253AD5}, id 0
> Ethernet II, Src: XiaomiCo_ca:2d:08 (04:e5:98:ca:2d:08), Dst: HonHaiPr_1e:87:89 (30:f7:72:1e:87:89)
> Internet Protocol Version 4, Src: 192.168.1.4, Dst: 239.255.255.250
> User Datagram Protocol, Src Port: 37121, Dst Port: 1900
> Simple Service Discovery Protocol

0000  30 f7 72 1e 87 89 04 e5  98 ca 2d 08 08 00 45 00   0·r····· ··-···E·
0010  00 99 75 7a 40 00 01 11  52 33 c0 a8 01 04 ef ff   ··uz@··· R3······
0020  ff fa 91 01 07 6c 00 85  87 46 4d 2d 53 45 41 52   ·····l·· ·FM-SEAR
0030  43 48 20 2a 20 48 54 54  50 2f 31 2e 31 0d 0a 48   CH * HTT P/1.1··H
0040  4f 53 54 3a 20 32 33 39  2e 32 35 35 2e 32 35 35   OST: 239 .255.255
0050  2e 32 35 30 3a 31 39 30  30 0d 0a 4d 41 4e 3a 20   .250:190 0··MAN:
0060  22 73 73 64 70 3a 64 69  73 63 6f 76 65 72 22 0d   "ssdp:di scover"·
0070  0a 4d 58 3a 20 31 0d 0a  53 54 3a 20 75 72 6e 3a   ·MX: 1·· ST: urn:
0080  64 69 61 6c 2d 6d 75 6c  74 69 73 63 72 65 65 6e   dial-mul tiscreen
0090  2d 6f 72 67 3a 73 65 72  76 69 63 65 3a 64 69 61   -org:ser vice:dia
00a0  6c 3a 31 0d 0a 0d 0a                               l:1····

wireshark_Wi-FiCHZZP1.pcapng        Packets: 117 · Displayed: 117 (100.0%) · Dropped: 0 (0.0%)        Profile: Default
```

Tcp filter


*Wi-Fi
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp

```
No.  Time      Source           Destination      Protocol Length Info
   1 0.000000  117.18.232.200   192.168.1.7      TCP      54 443 → 51863 [ACK] Seq=1 Ack=1 Win=133 Len=0
   2 0.000063  192.168.1.7      117.18.232.200   TCP      54 [TCP ACKed unseen segment] 51863 → 443 [ACK] Seq=1 Ack=2 Win=1024 Len=0
   3 1.049793  52.98.87.66      192.168.1.7      TCP      66 443 → 51874 [ACK] Seq=1 Ack=1 Win=16386 Len=0 TSval=1132114741 TSecr=1823779342
   4 1.049793  117.18.237.29    192.168.1.7      TCP      54 80 → 51875 [ACK] Seq=1 Ack=1 Win=131 Len=0
   5 1.049863  192.168.1.7      52.98.87.66      TCP      54 [TCP ACKed unseen segment] 51874 → 443 [ACK] Seq=1 Ack=2 Win=1024 Len=0
   6 1.049963  192.168.1.7      117.18.237.29    TCP      54 [TCP ACKed unseen segment] 51875 → 80 [ACK] Seq=1 Ack=2 Win=510 Len=0
  13 4.432014  104.26.11.240    192.168.1.7      TCP      66 443 → 51890 [ACK] Seq=1 Ack=1 Win=67 Len=0 TSval=2095693406 TSecr=14733127
  14 4.432606  192.168.1.7      104.26.11.240    TCP      54 [TCP ACKed unseen segment] 51890 → 443 [ACK] Seq=1 Ack=2 Win=1024 Len=0
  18 8.113168  20.198.119.143   192.168.1.7      TCP      66 443 → 51801 [ACK] Seq=1 Ack=1 Win=7219 Len=0 TSval=1577172291 TSecr=748150597
  19 8.113376  192.168.1.7      20.198.119.143   TCP      54 [TCP ACKed unseen segment] 51801 → 443 [ACK] Seq=1 Ack=2 Win=512 Len=0

> Frame 1: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface \Device\NPF_{1C880E68-039F-43FD-BBBA-72C04F253AD5}, id 0
> Ethernet II, Src: GenexisI_82:21:b8 (bc:62:d2:82:21:b8), Dst: HonHaiPr_1e:87:89 (30:f7:72:1e:87:89)
> Internet Protocol Version 4, Src: 117.18.232.200, Dst: 192.168.1.7
> Transmission Control Protocol, Src Port: 443, Dst Port: 51863, Seq: 1, Ack: 1, Len: 0

0000  30 f7 72 1e 87 89 bc 62  d2 82 21 b8 08 00 45 00   0·r····b ··!···E·
0010  00 28 00 00 40 00 3c 06  1f 46 75 12 e8 c8 c0 a8   ·(··@·<· ·Fu·····
0020  01 07 01 bb ca 97 5d 3c  a6 56 c3 65 96 df 50 10   ······]< ·V·e··P·
0030  00 85 65 9a 00 00                                  ··e···
```

Experiment Number: 10 **Date:**

WIRESHARK

AIM:

To understand Wireshark tool and explore its features like filters, flow graphs, statistics and protocol hierarchy.

THEORY:

Wireshark is a free and open-source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development. It can parse and display the fields, along with their meanings as specified by different networking protocols. Wireshark uses pcap to capture packets, so it can only capture packets on the types of networks that pcap supports. Wireshark can color packets based on rules that match particular fields in packets, to help the user identify the types of traffic at a glance.

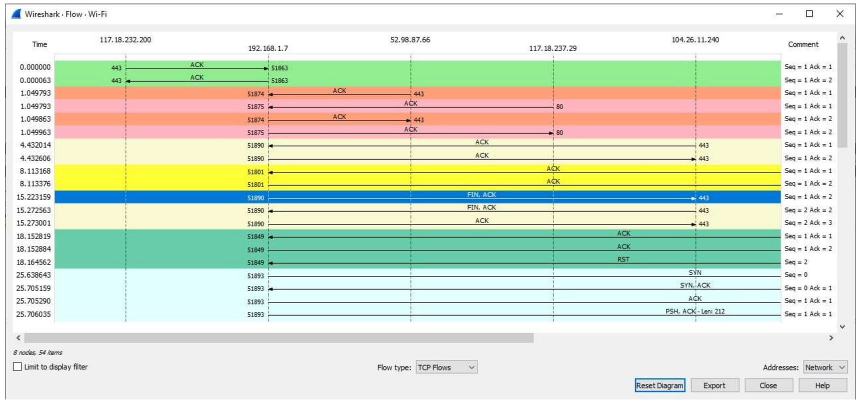
Filters: Wireshark share a powerful filter engine that helps remove the noise from a packet trace and lets you see only the packets that interest you. If a packet meets the requirements expressed in your filter, then it is displayed in the list of packets. Display filters let you compare the fields within a protocol against a specific value, compare fields against fields, and check the existence of specified fields or protocols.

Flow Graph: The Flow Graph window shows connections between hosts. It displays the packet time, direction, ports and comments for each captured connection. You can filter all connections by ICMP Flows, ICMPv6 Flows, UIM Flows and TCP Flows. Flow Graph window is used for showing multiple different topics. Each vertical line represents the specific host, which you can see in the top of the window. The numbers in each row at the very left of the window represent the time packet. The numbers at the both ends of each arrow between hosts represent the port numbers.

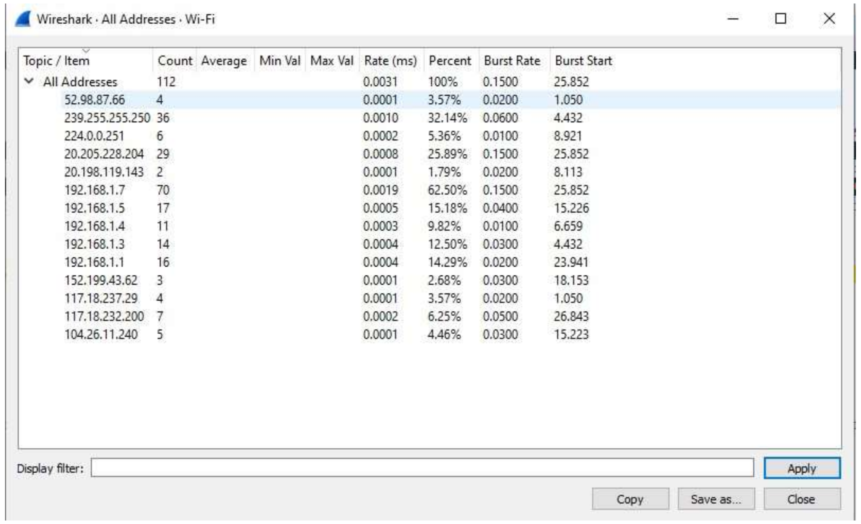
Statistics: Wireshark provides a wide range of network statistics. These statistics range from general information about the loaded capture file (like the number of captured packets), to statistics about specific protocols (e.g. statistics about the number of HTTP requests and responses captured). General statistics involve Summary about the capture file like: packet counts, captured time period, Protocol Hierarchy of the captured packets, Conversations like traffic between specific Ethernet/IP/… addresses etc.

Protocol Hierarchy: This is a tree of all the protocols in the capture. Each row contains the statistical values of one protocol. Two of the columns (Percent Packets and Percent Bytes) serve double duty as bar graphs. If a display filter is set it will be shown at the bottom.

Flow graphs


Wireshark · Flow · Wi-Fi
Time
117.18.232.200
192.168.1.7
52.98.87.66
117.18.237.29
104.26.11.240
Comment
0.000000 443 ACK 51863 Seq = 1 Ack = 1
0.000063 443 ACK 51863 Seq = 1 Ack = 2
1.049793 51874 ACK 443 Seq = 1 Ack = 1
1.049793 51875 ACK 80 Seq = 1 Ack = 1
1.049863 51874 ACK 443 Seq = 1 Ack = 2
1.049963 51875 ACK 80 Seq = 1 Ack = 2
4.432014 51890 ACK 443 Seq = 1 Ack = 1
4.432606 51890 ACK 443 Seq = 1 Ack = 2
8.113168 51801 ACK Seq = 1 Ack = 1
8.113376 51801 ACK Seq = 1 Ack = 2
15.223159 51890 FIN, ACK 443 Seq = 1 Ack = 2
15.272563 51890 FIN, ACK 443 Seq = 2 Ack = 2
15.273001 51890 ACK 443 Seq = 2 Ack = 3
18.152819 51849 ACK Seq = 1 Ack = 1
18.152884 51849 ACK Seq = 1 Ack = 2
18.164562 51849 RST Seq = 2
25.638643 51893 SYN Seq = 0
25.705159 51893 SYN, ACK Seq = 0 Ack = 1
25.705290 51893 ACK Seq = 1 Ack = 1
25.706035 51893 PSH, ACK - Len: 212 Seq = 1 Ack = 1
8 nodes, 54 items
Limit to display filter
Flow type: TCP Flows
Addresses: Network
Reset Diagram
Export
Close
Help

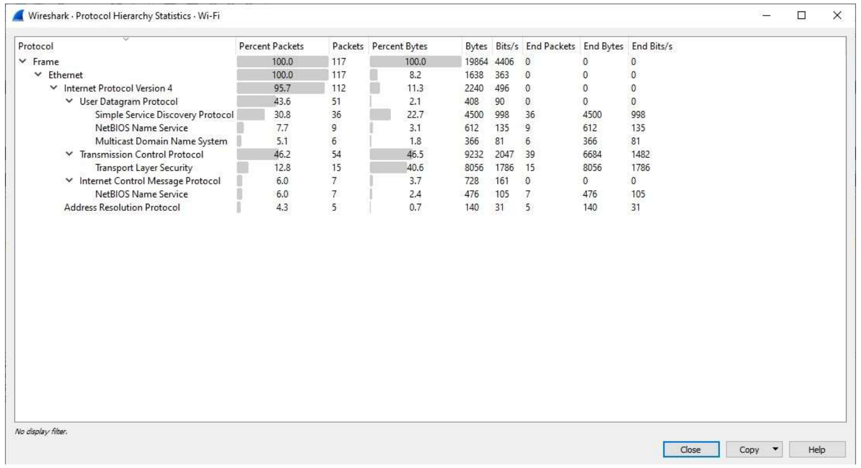
Statistics


Wireshark · All Addresses · Wi-Fi

Topic / Item	Count	Average	Min Val	Max Val	Rate (ms)	Percent	Burst Rate	Burst Start
All Addresses	112				0.0031	100%	0.1500	25.852
52.98.87.66	4				0.0001	3.57%	0.0200	1.050
239.255.255.250	36				0.0010	32.14%	0.0600	4.432
224.0.0.251	6				0.0002	5.36%	0.0100	8.921
20.205.228.204	29				0.0008	25.89%	0.1500	25.852
20.198.119.143	2				0.0001	1.79%	0.0200	8.113
192.168.1.7	70				0.0019	62.50%	0.1500	25.852
192.168.1.5	17				0.0005	15.18%	0.0400	15.226
192.168.1.4	11				0.0003	9.82%	0.0100	6.659
192.168.1.3	14				0.0004	12.50%	0.0300	4.432
192.168.1.1	16				0.0004	14.29%	0.0200	23.941
152.199.43.62	3				0.0001	2.68%	0.0300	18.153
117.18.237.29	4				0.0001	3.57%	0.0200	1.050
117.18.232.200	7				0.0002	6.25%	0.0500	26.843
104.26.11.240	5				0.0001	4.46%	0.0300	15.223

Display filter:

Protocol Hierarchy



Wireshark · Protocol Hierarchy Statistics · Wi-Fi

Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s
Frame	100.0	117	100.0	19864	4406	0	0	0
Ethernet	100.0	117	8.2	1638	363	0	0	0
Internet Protocol Version 4	95.7	112	11.3	2240	496	0	0	0
User Datagram Protocol	43.6	51	2.1	408	90	0	0	0
Simple Service Discovery Protocol	30.8	36	22.7	4500	998	36	4500	998
NetBIOS Name Service	7.7	9	3.1	612	135	9	612	135
Multicast Domain Name System	5.1	6	1.8	366	81	6	366	81
Transmission Control Protocol	46.2	54	46.5	9232	2047	39	6684	1482
Transport Layer Security	12.8	15	40.6	8056	1786	15	8056	1786
Internet Control Message Protocol	6.0	7	3.7	728	161	0	0	0
NetBIOS Name Service	6.0	7	2.4	476	105	7	476	105
Address Resolution Protocol	4.3	5	0.7	140	31	5	140	31

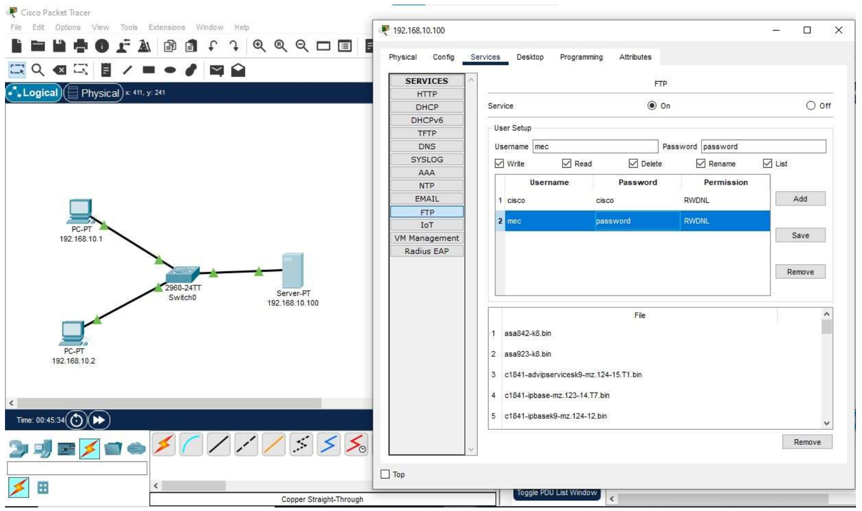
No display filter.

RESULT:

The experiment was executed successfully.

OUTPUT:

FTP



192.168.10.100

FTP

Service [✓ On] Off

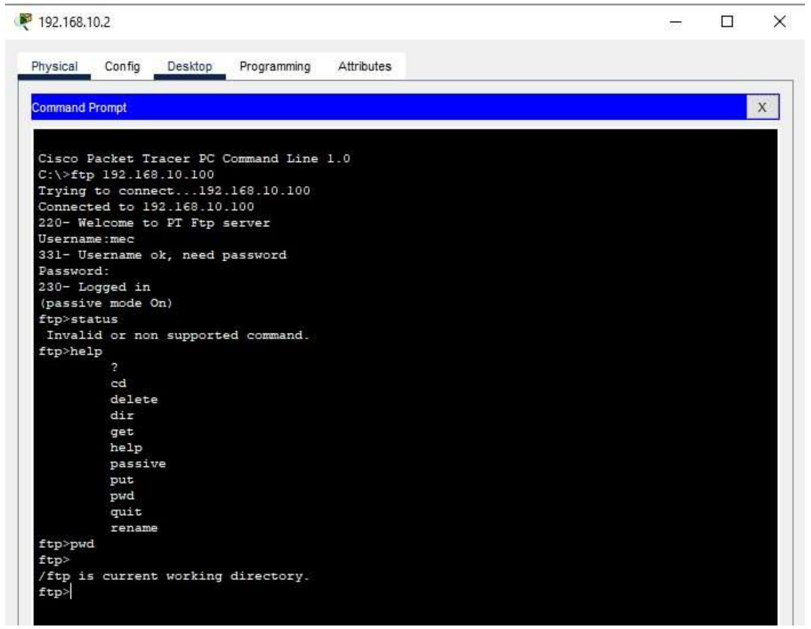
User Setup

Username mec Password password

[✓ Write] [✓ Read] [✓ Delete] [✓ Rename] [✓ List]

	Username	Password	Permission
1	cisco	cisco	RWDNL
2	mec	password	RWDNL

	File
1	asa842-k8.bin
2	asa923-k8.bin
3	c1841-advipservicesk9-mz.124-15.T1.bin
4	c1841-ipbase-mz.123-14.T7.bin
5	c1841-ipbasek9-mz.124-12.bin



```
Cisco Packet Tracer PC Command Line 1.0
C:\>ftp 192.168.10.100
Trying to connect...192.168.10.100
Connected to 192.168.10.100
220- Welcome to PT Ftp server
Username:mec
331- Username ok, need password
Password:
230- Logged in
(passive mode On)
ftp>status
 Invalid or non supported command.
ftp>help
         ?
         cd
         delete
         dir
         get
         help
         passive
         put
         pwd
         quit
         rename
ftp>pwd
ftp>
/ftp is current working directory.
ftp>
```

Experiment Number: 11 **Date:**

NETWORK WITH MULTIPLE SUBNETS

AIM:

To Study of Cisco Packet Tracer and configure FTP server, DHCP server and DNS server in a wired network using required network devices.

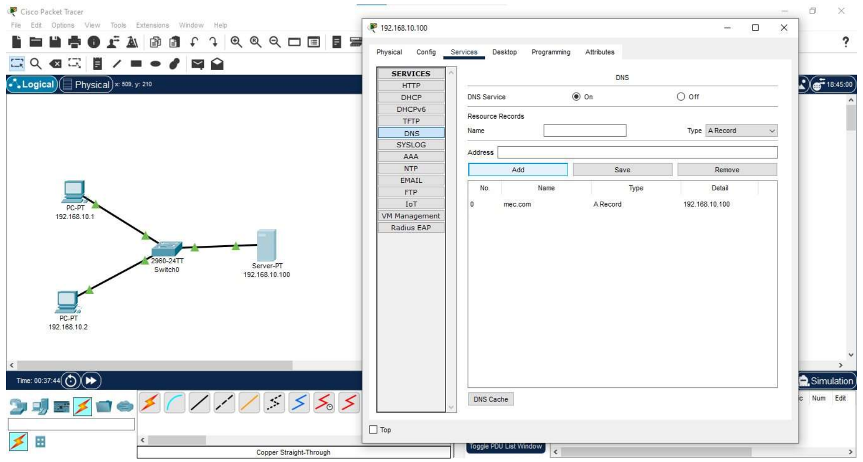
THEORY:

Packet Tracer is a cross-platform visual simulation tool designed by Cisco Systems that allows users to create network topologies and imitate modern computer networks. The software allows users to simulate the configuration of Cisco routers and switches using a simulated command line interface. Packet Tracer supports an array of simulated Application Layer protocols, as well as basic routing with RIP, OSPF, EIGRP and BGP.

DNS: The Domain Name System (DNS) is the hierarchical and decentralized naming system used to identify computers reachable through the Internet or other Internet Protocol (IP) networks. The resource records contained in the DNS associate domain names with other forms of information. These are most commonly used to map human-friendly domain names to the numerical IP addresses computers need to locate services and devices using the underlying network protocols.

DHCP: The Dynamic Host Configuration Protocol (DHCP) is a network management protocol used on Internet Protocol networks for automatically assigning IP addresses and other communication parameters to devices connected to the network using a client–server architecture. It employs a connectionless service model, using the User Datagram Protocol (UDP). It is implemented with two UDP port numbers, 67 is the destination port of a server, and 68 is used by the client.

DNS



192.168.10.100

Physical Config Services Desktop Programming Attributes

SERVICES: HTTP, DHCP, DHCPv6, TFTP, DNS, SYSLOG, AAA, NTP, EMAIL, FTP, IoT, VM Management, Radius EAP

DNS

DNS Service [✓ On] Off

Resource Records

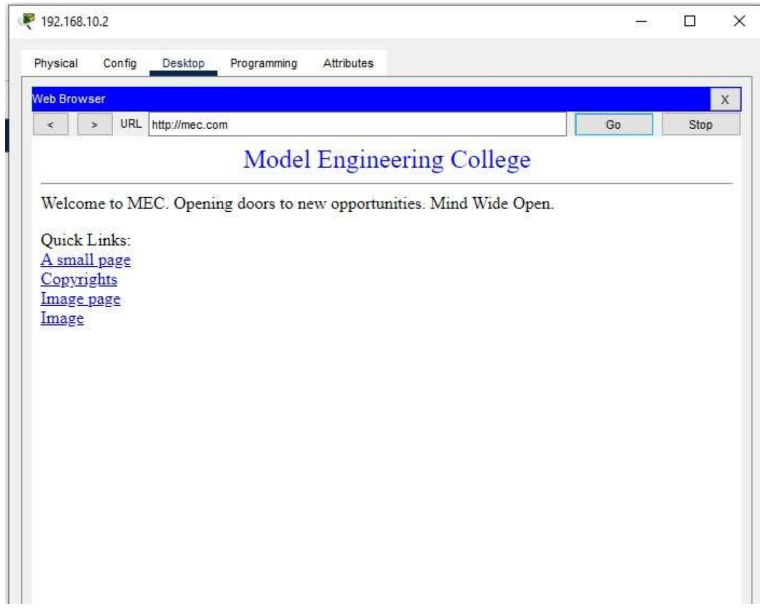
Name Type A Record

Address

Add Save Remove

No.	Name	Type	Detail
0	mec.com	A Record	192.168.10.100

DNS Cache



192.168.10.2

Physical Config Desktop Programming Attributes

Web Browser

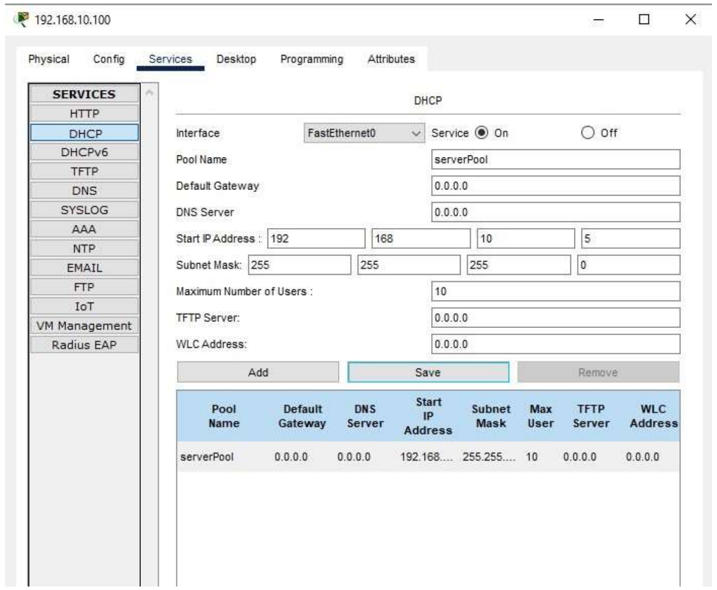
URL http://mec.com Go Stop

Model Engineering College

Welcome to MEC. Opening doors to new opportunities. Mind Wide Open.

Quick Links:
A small page
Copyrights
Image page
Image

DHCP


192.168.10.100

DHCP

Interface: FastEthernet0 — Service [✓ On] Off

Pool Name: serverPool

Default Gateway: 0.0.0.0

DNS Server: 0.0.0.0

Start IP Address: 192 . 168 . 10 . 5

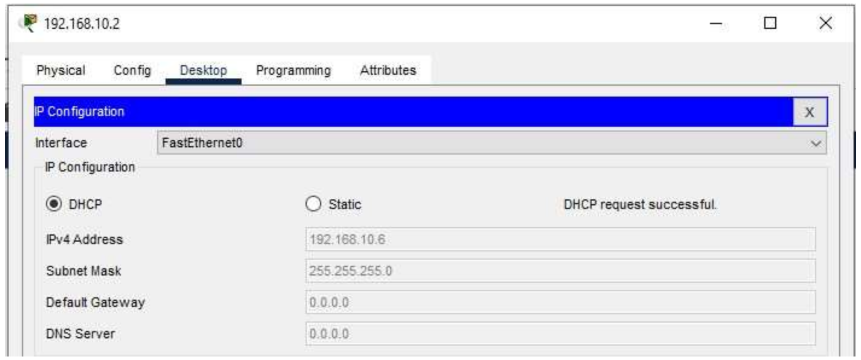
Subnet Mask: 255 . 255 . 255 . 0

Maximum Number of Users: 10

TFTP Server: 0.0.0.0

WLC Address: 0.0.0.0

Pool Name	Default Gateway	DNS Server	Start IP Address	Subnet Mask	Max User	TFTP Server	WLC Address
serverPool	0.0.0.0	0.0.0.0	192.168....	255.255....	10	0.0.0.0	0.0.0.0


192.168.10.2

IP Configuration

Interface: FastEthernet0

IP Configuration

[✓ DHCP] Static — DHCP request successful.

IPv4 Address: 192.168.10.6

Subnet Mask: 255.255.255.0

Default Gateway: 0.0.0.0

DNS Server: 0.0.0.0

<u>RESULT:</u>

The experiment was executed successfully.

PROGRAM:

link-state.tcl

```
set ns [new Simulator]
$ns rtproto LS
set nf [open ls1.tr w]
$ns trace-all $nf
set nr [open ls2.nam w]
$ns namtrace-all $nr

proc finish {} {
    global ns nf nr
    $ns flush-trace
    close $nf
    close $nr
    exec nam ls2.nam
    exit 0
}
for {set i 0} {$i<12} {incr i} {
    set n$i [$ns node]
}

$ns duplex-link $n0 $n1 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n2 $n3 1Mb 10ms DropTail
$ns duplex-link $n3 $n4 1Mb 10ms DropTail
$ns duplex-link $n4 $n5 1Mb 10ms DropTail
$ns duplex-link $n5 $n6 1Mb 10ms DropTail
$ns duplex-link $n6 $n7 1Mb 10ms DropTail
$ns duplex-link $n7 $n8 1Mb 10ms DropTail
$ns duplex-link $n8 $n0 1Mb 10ms DropTail
$ns duplex-link $n0 $n9 1Mb 10ms DropTail
$ns duplex-link $n1 $n10 1Mb 10ms DropTail
$ns duplex-link $n9 $n11 1Mb 10ms DropTail
$ns duplex-link $n10 $n11 1Mb 10ms DropTail
$ns duplex-link $n11 $n5 1Mb 10ms DropTail

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
set null0 [new Agent/Null]
$ns attach-agent $n5 $null0

set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $udp1
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005

set null1 [new Agent/Null]
$ns attach-agent $n5 $null1
```

Experiment Number: 12 **Date:**

NS2 SIMULATOR

AIM:

To study of NS2 and simulate Link State Protocol and Distance Vector Routing protocol in it.

THEORY:

NS (Network Simulator) is a discrete event simulator targeted at networking research. Ns provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. It is an object-oriented, discrete event-driven simulator written in C++ and Otcl/Tcl. NS-2 can be used to implement network protocols such as TCP and UDP, traffic source behavior such as FTP, Telnet, Web, CBR, and VBR, router queues management mechanism such as Drop Tail, RED, and CBQ and routing algorithms.Install NS-2 using this command :

```
sudo apt-get install ns2
```

Nam (Network Animator) is an animation tool to graphically represent the network and packet traces. Use this command:

```
sudo apt-get install nam
```

Basic Commands:

```
set a 8
set b [expr $a/8]
```

In the first line, the variable a is assigned the value 8. In the second line, the result of the command [expr $a/8], which equals 1, is then used as an argument to another command, which in turn assigns a value to the variable b. The “$” sign is used to obtain a value contained in a variable and square brackets are an indication of command substitution.

Define new procedures with proc command :

```
proc factorial fact {
    if {$fact <= 1} {
       return 1
    }
    expr $fact * [factorial [expr $fact-1]]
}
```

To open a file for reading :

```
set testfile [open hello.data r]
```

Similarly, put command is used to write data into the file.

```
set testfile [open hello.data w]
puts $testfile “hello1”
```

```
$ns connect $udp0 $null0
$ns connect $udp1 $null1

$ns rtmodel-at 10.0 down $n11 $n5
$ns rtmodel-at 30.0 up $n11 $n5
$ns rtmodel-at 15.0 down $n7 $n6
$ns rtmodel-at 20.0 up $n7 $n6

$ns at .1 "$cbr1 start"
$ns at .2 "$cbr0 start"
$ns at 45.0 "$cbr1 stop"
$ns at 45.1 "$cbr0 stop"

$ns at 50.0 "finish"
$ns run
```

AWK Program for LS

```
BEGIN {
    print " performance evaluation"
    send=0
    recv=0
    dropped=0
    rout=0
}
{
    if($1=="+" && $3=="0"||"1" && $5=="cbr")
    {
        send++
    }
    if($1=="r" && $4=="5" && $5=="cbr")
    {
            recv++
    }
    if($1=="d")
    {
        dropped++
    }
    if($1=="r" && $5=="rtProtoLS")
    {
        rout++
    }
}

END {
    print "No of packets Send :" send
    print "No of packets Received :" recv
    print "No of packets dropped :" dropped
    print "No of routing packets :" rout
    NOH=rout/recv
    PDR=recv/send
    print "Normalised overhead :" NOH
    print "Packet delivery ratio :" PDR
}
```

To call sub processes within another process, exec is used, which creates a subprocess and waits for it to complete.

```
exec rm $testfile
```

To be able to run a simulation scenario, a network topology must first be created. In ns2, the topology consists of a collection of nodes and links.

```
set ns [new Simulator]
```

The simulator object has member functions that enable the creation of the nodes and define the links between them. The class simulator contains all the basic functions. Since ns was defined to handle the Simulator object, the command $ns is used for using the functions belonging to the simulator class.

In the network topology nodes can be added in the following manner:

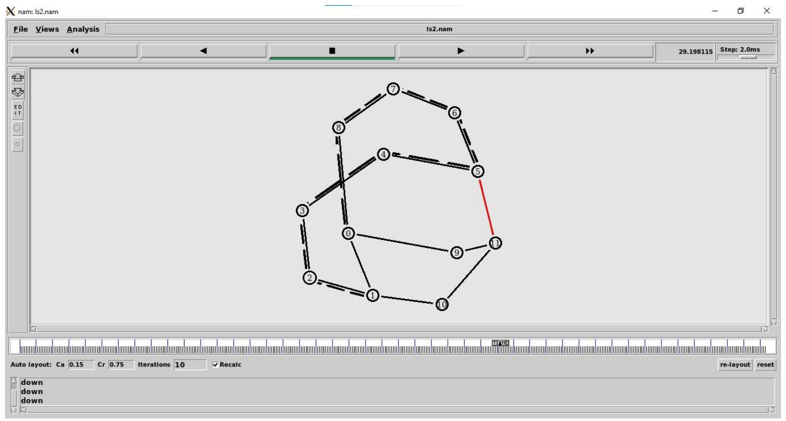
```
set n0 [$ns node]
set n1 [$ns node]
```

Traffic agents (TCP, UDP, etc.) and traffic sources (FTP, CBR, etc.) must be set up if the node is not a router. It enables the creation of CBR traffic source using UDP as transport protocol or an FTP traffic source using TCP as a transport protocol.

CBR traffic source using UDP:

```
set udp0 [new Agent/UDP]

$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packet_size_ 512
```

OUTPUT:



Auto layout: Ca 0.15 Cr 0.75 Iterations 10 ✓ Recalc

```
down
down
down
```

RESULT:

The experiment was executed successfully.