**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** - 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** - 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 familiarise and understand the use of system calls used for networking 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.

## **RESULT:**

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

**Experiment Number: 3 (a)** **Date:**

# STOP AND WAIT PROTOCOL

## AIM:

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

## 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.

## RESULT:

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

**Experiment Number: 3 (b)** **Date:**

# GO BACK N

## AIM:

To write a program to simulate the Go Back N.

## 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;
      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
```

**RESULT:**

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

**Experiment Number: 3 (c)** **Date:**

# SELECTIVE REPEAT

## AIM:

To write a program to simulate the Selective Repeat protocol.

## 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;
         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
               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
```

**RESULT:**

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

**Experiment Number: 4** **Date:**

# DISTANCE VECTOR ROUTING

## AIM:

To write a program to implement the Distance Vector Routing.

## ALGORITHM:

```
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
```

## RESULT:

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

**Experiment Number: 5** **Date:**

# LINK STATE ROUTING

## AIM:

To write a program to implement the Link State Routing.

## ALGORITHM:

```
begin
	N = {A}		// A is a root node.

	for all nodes v
		if v adjacent to A then
			D(v) = c(A, v)
		else
			D(v) = infinity

	loop
		find w not in N such that D(w) is a minimum.
		Add w to N
		Update D(v) for all v adjacent to w and not in N:
			D(v) = min(D(v), D(w) + c(w, v))
		Until all nodes in N
	end
end
```

## RESULT:

The program for Link State Routing was executed and output was verified successfully.

**Experiment Number: 6** **Date:**

# LEAKY BUCKET

## AIM:

To write a program to implement the Leaky Bucket.

## 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.

## RESULT:

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