COLLEGE OF ENGINEERING
CHENGANNUR

LABORATORY RECORD

YEAR.....

NAME.....

SEMESTER.....BATCH.....

ROLLNO.....REG:NO.....

BRANCH.....

COLLEGE OF ENGINEERING
CHENGANNUR

CSL332-NETWORKING LAB

YEAR.....

NAME.....

SEMESTER.....BATCH.....

ROLLNO.....REG:NO.....

BRANCH.....

CERTIFIED BONAFIDE RECORD OF WORK DONE BY

.....

CHENGANNUR

.....

LECTURER-IN-CHARGE

1.

2.

EXTERNAL EXAMINER

INTERNAL EXAMINER

INDEX

S.no	Date	Name of the Experiment	Page no.	Teacher Remarks/Sign

EXPERIMENT NO:

FAMILIARIZATION OF BASIC NETWORK CONFIGURATION FILES

AIM: Familiarize with basics of networking configuration files and networking commands in Linux.

1) IFCONFIG

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, status of currently active interface is given.
If single argument, displays the status of given interface only.

Options/Arguments:

-a	Displays all interfaces currently active, even if down
-s	Display a short list
-v	Be more verbose for some error conditions

2) PING

PING is used to send ICMP ECHO_REQUEST to network hosts.
It 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.

Options/Arguments:

-a	Audible ping.
-A	Adaptive ping.
-b	Allow pinging a broadcast address.
-c count	Stops after sending *count* ECHO_REQUESTS packets.
-f	Flood ping.
-h	Help
-v	Verbose Output

3) NETSTAT

It prints network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.
By default, it displays a list of open sockets.

Options/Arguments:

-v, --verbose	print some useful information about unconfigured address families.
-c, --continuous	This will cause netstat to print the selected information every second continuously.
-p	Show the PID and name of the program to which each socket belongs.
-l	Show only listening sockets.
-a	Show both listening and non-listening (for TCP this means established connections) sockets.
-c	Print routing information from the route cache.
-f	Print routing information from the FIB.

4) **ARP**

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. ARP with no mode specifier will print the current content of the table.

Options/Arguments:

-v, --verbose	Tell the user what is going on by being verbose.
-n, --numeric	shows numerical addresses instead of trying to determine symbolic host, port or user names.
-a	Use alternate BSD style output format
-e	Use default Linux style output format
-D, --use-device	Instead of a hw_addr, the given argument is the name of an interface.
-i If, --device If	Select an interface. When dumping the ARP cache only entries matching the specified interface will be printed.
-f filename	The address info is taken from file filename.

5) **TELNET**

The **TELNET** command is used to communicate with another host using the TELNET protocol. If telnet is invoked with a host argument, it performs an open command implicitly.

Options/Arguments:

-8	Specifies an 8-bit data path.
-E	Stops any character from being recognized as an escape character.
-L	Specifies an 8-bit data path on output. This causes the BINARY option to be negotiated on output.
-a	Attempt automatic login.
-d	Sets the initial value of the **debug** toggle to TRUE.

6) **FTP**

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.

Options/Arguments:

-p	Use passive mode for data transfers.
-i	Turns off interactive prompting during multiple file transfers.
-n	Restrains **ftp** from attempting "auto-login" upon initial connection.
-e	Disables command editing and history support, if it was compiled into the **ftp** executable. Otherwise, does nothing.
-g	Disables file name globing.
-v	Verbose option forces **ftp** to show all responses from the remote server.
-d	Enables debugging.

7) **FINGER**
The **FINGER** displays information about the system users.

Options/Arguments:

-s Displays the user's login name, real name, terminal name and write status
-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.
-p Prevents the -l option of finger from displaying the contents of the ".plan", ".project" and ".pgpkey" files.
-m Prevent matching of user names.

Difference between Classful and Classless addressing:

Classful Addressing	Classless Addressing
An IP address allocation method that allocates IP addresses according to five major classes.	An IP address allocation method that is designed to replace class full addressing to minimize the rapid exhaustion of IP addresses.
Less practical and useful.	More practical and useful.
Network ID and Host ID changes depending on the classes.	There is no boundary on network ID and host ID.

Various Types of Classful Addressing:

There are 5 classes of IP addresses in network classful addressing:

1) **Class A**
Address ranges from 0.0.0.0 to 127.255.255.255
Network ID is 8 bits and host ID is 24 bits.
2) **Class B**
Address ranges from 128.0.0.0 to 191.255.255.255
Network ID is 16 bits long and host ID is 16 bits long.
3) **Class C**
Address ranges from 192.0.0.0 to 223.255.255.255
Network ID is 24 bits long while host ID is 8 bits long.
4) **Class D**
Reserved for Multicasting.
Address ranges from 224.0.0.0 to 239.255.255.255
This class doesn't possess any subnet mask.

5) **Class E**
Reserved for experimental and research purposes.
Address ranges from 240.0.0.0 to 255.255.255.255
This class also doesn't possess any subnet mask.

RESULT:

Familiarized with basics of Network configuration files and using networking commands in Linux.

EXPERIMENT NO:

FAMILIARIZATION OF SYSTEM CALLS USED FOR LINUX NETWORK PROGRAMMING

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

SOCKET PROGRAMMING:

A socket is a communication connection point (endpoint) that you can name and address in a network. Socket programming shows how to use socket APIs to establish communication links between remote and local processes.

The processes that use a socket can reside on the same system or different systems on different networks. Sockets are useful for both stand-alone and network applications. Sockets allow you to exchange information between processes on the same machine or across a network, distribute work to the most efficient machine, and they easily allow access to centralized data.

SOCKET TYPE:

- Stream (SOCK-STREAM)

This type of socket is connection-oriented. It establishes an end-to-end connection by using the bind (), listen (), accept () and connect () functions. SOCK-STREAM sends data without errors or duplication and receives the data in the sending order. SOCK-STREAM builds flow control to avoid data overruns. It does not impose record boundaries on the data and considers the data to be a stream of bytes.

- Datagram (SOCK-DGRAM)

In Internet Protocol Terminology, the basic unit of data transfer is a datagram. The datagram socket is connectionless. It establishes no end-to-end connection with the transport providers (Protocol). The socket Sends datagram as independent packets with no guarantee of delivery. Datagrams can arrive out of order. For some transport providers, each diagram can use a different route through the network.

Creating a connection-oriented socket:

A connection-oriented server uses the following sequence of functions calls socket(),bind(), connect(), listen(), accept(), send(), recv(), close().
A connection client uses the follwinng sequence of function calls socket(), gethostbyname(), connect(), send(), recv(), close().

Creating a connectionless socket:

A connection client illustrates the socket APIs that are written for the user datagram.

Protocol (UDP)

The server uses the following sequence of functions calls socket(), bind(), sendto(), recvfrom(), close(). The client example uses the following sequence of functions calls socket(), gethostbyname(), sendto(), recvfrom(), close().

Functions and their parameters:

- **socket()**

This function gives a socket descriptor that can be used in later system calls.
Syntax:

int socket (int domain, int type, int protocol)
domain → AF_INET or AF_UNIX
type → the type of socket needed (stream or Datagram)
SOCK_STREAM for stream socket
SOCK_DGRAM for Datagram socket
Protocol → 0

- **bind()**

This function associates a socket with a port.
Syntax:

int bind(int fd, struct sockaddr *my_addr, int addrlen)
fd → socket descriptor
my_addr → ptr to structure sockaddr
addrlen → sizeof(struct sockaddr)

- **connect()**

This function is used to connect to an IP address on a defined port.
Syntax:

int connect(int fd, struct sockaddr *addr, int addrlen)
fd → socket file descriptor
addr → The address/port to bind to
addrlen → sizeof(struct sockaddr)

- **listen()**

This function is used to wait for an incoming connection Before calling listen(), bind() is to be called. After Calling listen (), accept () is to be called to accept the incoming connection.
Syntax:

int listen(int fd, int backlog)
fd → Socket file descriptor
backlog → number of allowed connections

- **accept()**

This function is used to accept incoming connections.
Syntax:

int accept (int fd, void *addr, int addrlen)
fd → socket file descriptor
addr → ptr to struct sockaddr

- **send()**

This function is used to send data over stream sockets. It returns the number of bytes sent out.
Syntax:

int send (int fd, const void *msg, int len, int flags)
fd → socket descriptor
msg → ptr to the data to be sent
flags → 0

- **recv()**

This function receives the data sent over the stream sockets and returns the number of bytes read into the buffer.
Syntax:

int recv(int fd, void *buf, int len, unsigned int flag)
fd → socket file descriptor
buf → buffer to read into the buffer
len → maximum length of the buffer
flags → set to 0.

- **sendto()**

This function serves the same purpose as send() function except that it is used for the datagram Socket. It also returns the member of bytes send out.
Syntax:

int sendto(int fd, void *msg, int len, unsigned int flags, const struct sockaddr *to, int token)
fd → socket file descriptor
msg → ptr to the daa to be sent
len → length of the data to be sent
flags → set to 0
to → ptr to a struct sockaddr
token → sizeof(struct sockaddr)

- **recvfrom()**

This function serves the same purpose as the recv() function except that it is used for datagram sockets. It also returns the number of bytes received.

Syntax:

int recvfrom (int fd, const void *buf, int len, unsigned int flags, const struct sockaddr *from, int *fromlen)
fd → socket file descriptor
buf → buffer to read information into
len → maximum length of the buffer
flags → set to 0
from → ptr to struct sockaddr from
fromlen → size of (ptr to an int that should be initialized to struct sockaddr)

- **close()**

This function is used to close that socket on your socket descriptor.
Syntax:

close (fd);
fd → Socket file descriptor

Client-Server Communication using TCP

Description:

Develop a program to implement interprocess communication using stream Sockets with the help of interfaces provided by the standard library.

Data Structures and Functions:

Data Structures used here is *struct sockaddr*. This structure holds socket address information for many types of sockets.

```
struct sockaddr
{
        unsigned short sa_family;       //address family AF_XXX
        varchar sa_data[14];            //14 bytes of protocol address
};
```

sa_family can be a variety of things but it will be AF_INET for everything we do in this document sa_data contains a destination address and port number for the socket. To deal with struct sock adds, programmers created a parallel structure.

```
struct sockaddr_in ("in" for "internet")
struct sockaddr_in
{
        short int sin_family;           //address family
        unsigned short int sin_post;    // port number
        struct in_addr sin_addr;        //internet address
        unsigned char sin-zero[8]:      // same size as struct sockaddr
};
```

This structure makes it easy to reference elements the socket address. Note that sin-zero (which is included to pad the structure to the length of a struct sockaddr) should be set to all zeroes with the function memset. Finally, the sin port and sin addr must be in Network Byte Order.

Functions used here are:

System calls that allow accessing the network the functionality of a Unix box are as given below. When you call one of these functions, the kernel takes over and does all the work for you automatically.

Server Side:

socket(), bind(), connect(), listen (), accept(), send(), recv(), close()

Client-Side:

socket(), gethostbyname(), connect(), send(), recv(), close()

RESULT:

Familiarized and understood the use and functioning of system calls used for network programming in Linux.

PROGRAM

Server.c

```
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <fcntl.h>
#include <unistd.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 Socket creation\n");

    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\n");

    k=listen(sock_desc,5);

    if(k==-1)
            printf("Error in listening\n");

    len=sizeof(client);
    temp_sock_desc=accept(sock_desc,(struct sockaddr*) &client,&len);

    if(temp_sock_desc==-1)
            printf("Error in temporary socket creation\n");
    for(;;)
    {
        bzero(buf,sizeof(buf));
            read(temp_sock_desc,buf,sizeof(buf));
```

EXPERIMENT NO:

CLIENT SERVER COMMUNICATION USING TCP(TRANSMISSION CONTROL PROTOCOL)

AIM: Program to implement client-server communication using socket programming and TCP as transport layer protocol

ALGORITHM:

Server Side:

1. Create a socket using the socket() function.
2. Bind the socket to a specific IP address and port using the bind() function.
3. Start listening for incoming connections using the listen() function.
4. Accept a client connection using the accept() function. This will create a temporary socket for communication.
5. Enter a loop to handle client requests:
    a. Read data from the client using the read() function.
    b. If the received message is "exit" or "Exit," exit the loop.
    c. Print the received message from the client.
    d. Prompt for a message to send to the client.
6. Send the message to the client using the write() function.
7. Close the temporary socket and the server socket.

Client Side:

1. Create a socket using the socket() function.
2. Connect to the server using the connect() function, providing the server's IP address and port.
3. Enter a loop for communication:
    a. Prompt for a message to send to the server.
    b. If the message is "exit" or "Exit," exit the loop.
    c. Send the message to the server using the write() function.
    d. Receive the server's response using the read() function.
    e. Print the received message from the server.
4. Close the client socket.

```
   printf("\n[From Client]:%s",buf);
            if(strncmp("exit",buf,4)==0 || strncmp("Exit",buf,4)==0)
            {
                    printf("Got Exit Request\nExiting!!!\n");
                    exit(0);
            }
            printf("[To Client]:");
            fgets(buf,100,stdin);
            write(temp_sock_desc,buf,sizeof(buf));
            }
  return 0;
}
```

Client.c

```
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <unistd.h>
int main()

{
  char buf[100];
  int k;
  socklen_t len;
  int sock_desc;
  struct sockaddr_in server,client;
  sock_desc=socket(AF_INET, SOCK_STREAM,0);

  if(sock_desc==-1)
            printf("Error in Socket creation\n");

  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\n");
  for(;;)
  {
                    bzero(buf,sizeof(buf));
                    printf("\n[To Server]:");
                    fgets(buf,100,stdin);
            if(strncmp("exit",buf,4)==0 || strncmp("Exit",buf,4)==0)
                    {
```

```
            write(sock_desc,buf,sizeof(buf));
            printf("Exit Request Sent!\nExiting!!!\n");
            exit(0);
            }
write(sock_desc,buf,sizeof(buf));
read(sock_desc,buf,sizeof(buf));
printf("[From Server]:%s",buf);
}
close(sock_desc);
return 0;

  }
```

OUTPUT:

Server.c	**Client.c**
Socket created successfully... Socket binded successfully... Server listening... [From Client]: Hello server! [To Client]: Welcome client! [From Client]: How are you? [To Client]: I'm doing well, thank you! [From Client]: What is the current time? [To Client]: The current time is 10:30 AM. [From Client]: exit Got Exit Request Exiting!!!	Socket Creation Successful Connected to Server [To Server]: Hello server! [From Server]: Welcome client! [To Server]: How are you? [From Server]: I'm doing well, thank you! [To Server]: What is the current time? [From Server]: The current time is 10:30 AM. [To Server]: exit Exit Request Sent! Exiting!!!

RESULT:

Program to implement client-server communication using socket programming and TCP as transport layer protocol has been implemented and executed, output has been verified.

PROGRAM

Server.c

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<stdlib.h>
#include<netdb.h>

int main(int argc,char *argv[])
{
	struct sockaddr_in server,client;
	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]));
	if(bind(sockfd,(struct sockaddr*) &server,sizeof(server))<0)
		printf("Error in bind()\n");
	char buffer[100];
	socklen_t server_len = sizeof(server);
	for(;;)
	{
		int k = recvfrom(sockfd,buffer,100,0,(struct sockaddr *) &server,&server_len);
		if(k<0)
			printf("Error in recvfrom()");
		if(strncmp("exit",buffer,4)==0||strncmp("Exit",buffer,4)==0)
		{
			printf("Received exit request from client!\nExiting!!!");
			break;
		}
		else
		{
			printf("\nMessage from Client:%s",buffer);
			printf("Enter data to send to client:");
			fgets(buffer,100,stdin);
			printf("\n");
			k = sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr*) &server,sizeof(server));
			if(k<0)
				printf("Error in sendto()");
		}
	}
	return 0;
}
```

EXPERIMENT NO:

CLIENT SERVER COMMUNICATION USING UDP(USER DATAGRAM PROTOCOL)

AIM: Program to implement client-server communication using socket programming and UDP as transport layer protocol

ALGORITHM:

Server:

1. Create a socket using the socket() function with the domain set to AF_INET and the type set to SOCK_DGRAM.
2. Initialize the server address structure with the appropriate values:
   a. Set the family to AF_INET.
   b. Set the IP address to INADDR_ANY.
   c. Set the port number by converting the command line argument to an integer and using htons() to convert it to network byte order.
3. Bind the socket to the server address using the bind() function.
4. Enter a loop to continuously receive messages from the client:
   a. Use the recvfrom() function to receive a message from the client and store it in a buffer.
   b. Check if an exit request was received. If so, break out of the loop.
   c. Otherwise, process the received message as desired.
   d. Use the sendto() function to send a response to the client if needed.
5. Close the socket and terminate the program.

Client:

1. Create a socket using the socket() function with the domain set to AF_INET and the type set to SOCK_DGRAM.
2. Initialize the server address structure with the appropriate values:
   a. Set the family to AF_INET.
   b. Set the IP address to the server's IP address.
   c. Set the port number by converting the command line argument to an integer and using htons() to convert it to network byte order.
3. Create a buffer to store messages.
4. Enter a loop to continuously send messages to the server:
   a. Prompt the user to enter a message to send.
   b. Read the message from the user and store it in the buffer.
   c. Check if an exit request was entered. If so, send the request to the server and break out of the loop.
   d. Otherwise, use the sendto() function to send the message to the server.
   e. Use the recvfrom() function to receive a response from the server if needed.
   f. Process and display the response as desired.
5. Close the socket and terminate the program.

Client.c

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<stdlib.h>
#include<netdb.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);
    int k;
    if(sockfd==-1)
            printf("Error in socket()\n");
    server.sin_family=AF_INET;
    server.sin_addr.s_addr=INADDR_ANY;
    server.sin_port = htons(atoi(argv[2]));
    socklen_t server_len = sizeof(server);
    socklen_t client_len = sizeof(client);
    char buffer[100];
    for(;;)
    {
            printf("\nEnter a message to be sent to server:");
            fgets(buffer,100,stdin);
            if(strncmp("exit",buffer,4)==0||strncmp("Exit",buffer,4)==0)
            {
                    k = sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr*) &server,sizeof(server));
                    if(k<0)
                            printf("Error in exit request");
                    printf("Exit Request Sent\nExiting");
                    break;
            }
            else
            {
                    k = sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr*) &server,sizeof(server));
                    if(k<0)
                            printf("\nError in sendto");
                    k = recvfrom(sockfd,buffer,100,0,(struct sockaddr *) &client,&client_len);
                    if(k<0)
                            printf("\nError in recvfrom()");
                    printf("Message from Server:%s\n",buffer);
            }
    }
    return 0;
}
```

OUTPUT:

Server.c

```
Message from Client: Hello server!
Enter data to send to client: Welcome client!

Message from Client: How are you?
Enter data to send to client: I'm doing well, thank you!

Message from Client: What is the current time?
Enter data to send to client: The current time is 10:30 AM.

Message from Client: exit
Received exit request from client!
Exiting!!!
```

Client.c

```
Enter a message to be sent to server: Hello server!
Message from Server: Welcome client!

Enter a message to be sent to server: How are you?
Message from Server: I'm doing well, thank you!

Enter a message to be sent to server: What is the current time?
Message from Server: The current time is 10:30 AM.

Enter a message to be sent to server: exit
Exit Request Sent
Exiting
```

RESULT:

Program to implement client-server communication using socket programming and UDP as transport layer protocol has been executed successfully, and the output is obtained.

PROGRAM

Server.c

```
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <fcntl.h>
#include <unistd.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 Socket creation\n");
        printf("Socket created successfully...\n");

        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\n");
        printf("Socket binded successfully...\n");

        k=listen(sock_desc,5);

        if(k==-1)
                printf("Error in listening\n");
        printf("Server listening...\n");

        len=sizeof(client);
        temp_sock_desc=accept(sock_desc,(struct sockaddr*) &client,&len);

        if(temp_sock_desc==-1)
                printf("Error in temporary socket creation\n");
        printf("Server accepted client\n\n");
```

EXPERIMENT NO:

STOP & WAIT ARQ PROTOCOL

AIM: Program to simulate sliding window protocol – Stop and Wait ARQ Protocol.

ALGORITHM:

For Server:

1. Create a socket using the socket() function.
2. Initialize the server address structure with the server IP address, port number, and other necessary details.
3. Bind the socket to the server address using the bind() function.
4. Listen for incoming connections using the listen() function.
5. Accept a client connection using the accept() function, which creates a temporary socket for communication with the client.
6. Receive the total number of frames from the client using the recv() function.
7. Loop over the number of frames:
    a. Receive a frame from the client using the recv() function.
    b. Print the received frame information.
    c. Sleep for 1 second.
    d. Send an acknowledgement for the frame back to the client using the send() function.
8. Close the temporary socket and the main server socket.

For Client:

1. Create a socket using the socket() function.
2. Initialize the client address structure with the client IP address, port number, and other necessary details.
3. Connect to the server using the connect() function.
4. Prompt the user to enter the total number of frames.
5. Send the total number of frames to the server using the send() function.
6. Loop over the number of frames:
    a. Send a frame to the server using the send() function.
    b. Print the sent frame information.
    c. Sleep for 1 second.
7. Receive an acknowledgement from the server for the frame using the recv() function.
8. Close the socket.

```
        k = recv(temp_sock_desc,buf,100,0);
        int nof = atoi(buf);
        for(int i=0;i<nof;i++)
        {       if(i%2==0){
                        k=recv(temp_sock_desc,buf,100,0);
                        printf("Frame received from Client:Frame [0]\n");
                        sleep(1);
                }
                else
                {
                        k=recv(temp_sock_desc,buf,100,0);
                        printf("Frame received from Client:Frame [1]\n");
                        sleep(1);
                }
                printf("Acknowledgement for Frame [%d]\n",(i%2==0)?1:0);
                k=send(temp_sock_desc,buf,100,0);
                printf("\n");
        }
        return 0
}
```

Client.c

```
  #include<sys/socket.h>
  #include<netinet/in.h>
  #include<stdio.h>
  #include<string.h>
  #include<stdlib.h>
  #include <unistd.h>
  int main()
  {
        char buf[100];
        int k,nof;
        socklen_t len;
        int sock_desc;
        struct sockaddr_in server,client;
        sock_desc=socket(AF_INET, SOCK_STREAM,0);

        if(sock_desc==-1)
                printf("Error in Socket creation\n");

        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\n");
```

```
        printf("Socket Creation Successful\nConnected to Server\n");
        printf("Enter the total number of frames:");
        scanf("%d",&nof);
        printf("\n");
        sprintf(buf,"%d", nof);
        k = send(sock_desc,buf,100,0);
        for(int i=0;i<nof;i++)
        {
                if(i%2==0){
                        printf("Frame [0] Sent\n");
                        k=send(sock_desc,buf,100,0);
                        sleep(1);
                }
                else
                {
                        printf("Frame [1] Sent\n");
                        k = send(sock_desc,buf,100,0);
                        sleep(1);
                }
                printf("Acknowledgement received from server for Frame [%d]\n",(i%2==0)?1:0);
                k=recv(sock_desc,buf,100,0);
                printf("\n");
        }
        close(sock_desc);
        return 0;
}
```

OUTPUT:

Server.c	**Client.c**
Socket created successfully... Socket binded successfully... Server listening... Server accepted client Frame received from Client: Frame [0] Acknowledgement for Frame [1] Frame received from Client: Frame [1] Acknowledgement for Frame [0] Frame received from Client: Frame [0] Acknowledgement for Frame [1] Frame received from Client: Frame [1] Acknowledgement for Frame [0]	Socket Creation Successful Connected to Server Enter the total number of frames: 5 Frame [0] Sent Acknowledgement received from server for Frame [0] Frame [1] Sent Acknowledgement received from server for Frame [1] Frame [0] Sent Acknowledgement received from server for Frame [0] ... Frame [1] Sent Acknowledgement received from server for Frame [1]

RESULT:

Program to simulate Stop and Wait ARQ protocol has been simulated and executed successfully and output has been obtained.

PROGRAM

Server.c

```
#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("\tServer Up\n Go 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';
```

EXPERIMENT NO:

GO-BACK-N ARQ

AIM: Program to implement Go-Back--N ARQ flow control protocol.

ALGORITHM:

Server:

1. Create a socket using the socket() function.
2. Set up the server address and port.
3. Bind the socket to the server address using the bind() function.
4. Listen for incoming connections using the listen() function.
5. Accept a client connection using the accept() function.
6. Initialize variables and buffers.
7. Send messages to the client in a loop.
    a. Construct the message to be sent.
    b. Send the message to the client using the write() function.
    c. Check for acknowledgment from the client using the select() function.
    d. If acknowledgment is received within the timeout, read the acknowledgment message from the client.
    e. Repeat the process for the next message.
    f. If acknowledgment is not received within the timeout, go back to the previous message and repeat the process.
8. Close the client and server sockets.

Client:

1. Create a socket using the socket() function.
2. Set up the server address and port.
3. Connect to the server using the connect() function.
4. Initialize variables and buffers.
5. Receive messages from the server in a loop.
    a. Read the message from the server using the read() function.
    b. Check if the message is received in order.
    c. If the message is in order, print the message and construct the acknowledgment message.
    d. Send the acknowledgment message to the server using the write() function.
    e. Repeat the process for the next message.
6. Close the client socket.

```
    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';
    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 ");
    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;
    }
    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);
    }

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

Client.c

```
#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");
        return 0;
    }
    printf("\n\tClient -with individual acknowledgement scheme\n\n");
    char msg1[50] = "acknowledgementof-";
    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");
            flag = 0;
            read(c_sock, buff, sizeof(buff));
        }
        int n = read(c_sock, buff, sizeof(buff));
        if (buff[strlen(buff) - 1] != i + '0')
        {
            printf("Discarded as out of order \n");
            i--;
        }
        else
        {
            printf("Message received from server : %s \n", buff);
            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:

Server:	**Client:**
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: acknowledgementof-0 Message sent to client :server message :3 Message from Client: acknowledgementof-1 Message sent to client :server message :4 Message from Client: acknowledgementof-2 Message sent to client :server message :5 Message from Client: acknowledgementof-3 Message sent to client :server message :6 Going back from 6:timeout Message sent to client :server message :4 Message sent to client :server message :5 Message sent to client :server message :6 Message from Client: acknowledgementof-4 Message sent to client :server message :7 Message from Client: acknowledgementof-5 Message sent to client :server message :8 Message from Client: acknowledgementof-6 Message sent to client :server message :9 Message from Client: acknowledgementof-7 Going back from 9:timeout on last 2 Message from Client: acknowledgementof-8 Message from Client: acknowledgementof-9	Client -with individual acknowledgement scheme Message received from server : server message :0 Acknowledgement sent for message Message received from server : server message :1 Acknowledgement sent for message Message received from server : server message :2 Acknowledgement sent for message Message received from server : server message :3 Acknowledgement sent for message Discarded as out of order Discarded as out of order Discarded as out of order Message received from server : server message :4 Acknowledgement sent for message Message received from server : server message :5 Acknowledgement sent for message Message received from server : server message :6 Acknowledgement sent for message Message received from server : server message :7 Acknowledgement sent for message here Discarded as out of order Message received from server : server message :8 Acknowledgement sent for message Message received from server : server message :9 Acknowledgement sent for message

RESULT:

Program to implement Go-back N ARQ protocol has been executed successfully and output has been verified.

PROGRAM

Server.c

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netdb.h>
#include<time.h>

int main(int argc, char *argv[])
{
int num;
struct sockaddr_in server, client;
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]));
if (bind(sockfd, (struct sockaddr *)&server, sizeof(server)) < 0)
    printf("Error in bind()\n");
socklen_t server_len = sizeof(server);
printf("Server is running on 127.0.0.1\n");
while(1)
{
    recvfrom(sockfd, &num,sizeof(num), 0, (struct sockaddr *)&server, &server_len);
    time_t current_time = time(NULL);
    printf("Client asked for time:%s\n",ctime(&current_time));
    sendto(sockfd, &current_time, sizeof(current_time), 0, (struct sockaddr *)&server, sizeof(server));
    exit(0);
}
return 0;
}
```

EXPERIMENT NO:

TIME-SERVER

AIM: Program to implement a Concurrent Time Server application using UDP to execute the program at a remote server. Client sends a time request to the server, server sends its system time back to the client. Client displays the result.

ALGORITHM:

Server:

1. Create a UDP socket using the socket() function.
2. Set up the server address structure (server sockaddr_in) with the desired IP address and port number.
3. Bind the socket to the server address using the bind() function.
4. Enter a loop to continuously receive client requests.
5. Inside the loop:
    a. Use the recvfrom() function to receive the request from the client.
    b. Get the current time using time() function.
    c. Send the current time back to the client using sendto() function.
6. Exit the loop and close the socket.

Client:

1. Create a UDP socket using the socket() function.
2. Set up the server address structure (server sockaddr_in) with the server's IP address and port number.
3. Send a request (number 1) to the server using sendto() function.
4. Get the current time using time() function and store it as start_time.
5. Start a clock using clock() function and store it as start.
6. Use the recvfrom() function to receive the current time from the server.
7. Stop the clock using clock() function and store it as end.
8. Calculate the round-trip time (RTT) as the difference between the current time and start_time.
9. Add half of the RTT to the received current_time to account for the delay caused by network communication.
10. Calculate the difference between end and start and divide it by 2 to get the half of the clock difference.
11. Display the server's time (current_time) and the delay (diff) in milliseconds.

Client.c

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netdb.h>
#include<time.h>

int main(int argc, char *argv[])
{
    time_t current_time,rtt;
    int num = 1;
    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()\n");
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(atoi(argv[2]));
    socklen_t client_len = sizeof(client);
    sendto(sockfd, &num, sizeof(num), 0, (struct sockaddr *)&server, sizeof(server));
    time_t start_time = time(NULL);
    clock_t start = clock();
    recvfrom(sockfd, &current_time, sizeof(current_time), 0, (struct sockaddr *)&client, &client_len);
    clock_t end = clock();
    rtt = time(NULL) - start_time;
    current_time = current_time+(rtt/2);
    double diff = (end-start)/2;
    printf("Server's Time:%sDelayed by %.3f milliseconds\n",ctime(&current_time),diff);
    return 0;
}
```

OUTPUT:

Server: Server is running on 127.0.0.1 Client asked for time:Sun Jun 18 17:03:40 2023
Client: Server's Time:Sun Jun 18 17:03:40 2023 Delayed by 19.000 milliseconds

RESULT:

Program to implement a concurrent time server application using UDP to execute the program at a remote server has been executed successfully and output has been obtained.

PROGRAM

```
#include <stdio.h>

int costMatrix[20][20], n;

struct routers
{
   int distance[20];
   int adjNodes[20];
} node[20];

void readCostMatrix()
{
   int i, j;
   printf("\nEnter cost matrix\n");
   for (i = 0; i < n; ++i)
   {
      for (j = 0; j < n; ++j)
      {
         scanf("%d", &costMatrix[i][j]);
         costMatrix[i][i] = 0;
         node[i].distance[j] = costMatrix[i][j];
         node[i].adjNodes[j] = j;
      }
   }
}

void calcRoutingTable()
{
   int i, j, k;
   for (i = 0; i < n; ++i)
   {
      for (j = 0; j < n; ++j)
      {
         for (k = 0; k < n; ++k)
         {
            if (node[i].distance[j] > costMatrix[i][k] + node[k].distance[j])
            {
               node[i].distance[j] = node[i].distance[k] + node[k].distance[j];
               node[i].adjNodes[j] = k;
            }
         }
      }
   }
}
```

EXPERIMENT NO:
DISTANCE VECTOR ROUTING ALGORITM

AIM: Program to implement and simulate algorithm for Distance Vector Routing protocol.

ALGORITHM:

1. Import the necessary header file: stdio.h.
2. Declare a global integer array costMatrix of size 20x20 and an integer variable n to store the number of nodes.
3. Define a structure routers with two integer arrays: distance and adjNodes.
4. Declare a global array node of type routers with size 20 to store the distance and adjacent nodes for each router.
5. Define the function readCostMatrix:
    a. Declare integer variables i and j.
    b. Print a message asking the user to enter the cost matrix.
    c. Use nested loops to read and store the cost matrix values in the costMatrix array.
    d. Set the distance from a node to itself as 0 in the cost matrix.
    e. Initialize the distance and adjNodes arrays of each node in the node array.
6. Define the function calcRoutingTable:
    a. Declare integer variables i, j, and k.
    b. Use nested loops to iterate through all possible paths between nodes.
    c. Check if the cost of the path from node X to node Y is greater than the cost of the path from node X to node Z plus the cost from node Z to node Y.
    d. If the above condition is true, update the distance and adjNodes arrays for node X and Y with the minimum cost and path.
7. Define the function displayRoutes:
    a. Declare integer variables i and j.
    b. Use loops to iterate through each router and display the routing table.
    c. Print the node number, the next hop (adjacent node), and the distance for each destination node.
8. Define the main() function:
    a. Declare integer variables i and j.
    b. Print a message asking the user to enter the number of nodes.
    c. Read and store the number of nodes in the variable n.
    d. Call the readCostMatrix() function to read the cost matrix from the user.
    e. Call the calcRoutingTable() function to calculate the routing table for each node.
    f. Call the displayRoutes() function to display the routing table for each router.
9. End the main() function.

```
void displayRoutes()
{
   int i, j;
   for (i = 0; i < n; ++i)
   {
      printf("\nRouter %d\n", i + 1);
      for (j = 0; j < n; ++j)
      {
         printf("Node %d via %d : Distance %d\n", j + 1, node[i].adjNodes[j] + 1, node[i].distance[j]);
      }
      printf("\n");
   }
}

int main()
{
   int i, j;
   printf("Number of nodes: ");
   scanf("%d", &n);
   readCostMatrix();
   calcRoutingTable();
   displayRoutes();
   return 0;
}
```

OUTPUT:

```
~$./a.out
Number of nodes: 3

Enter cost matrix
0 2 999
8 0 5
7 1 2

Router 1
Node 1 via 1 : Distance 0
Node 2 via 2 : Distance 2
Node 3 via 2 : Distance 7


Router 2
Node 1 via 1 : Distance 8
Node 2 via 2 : Distance 0
Node 3 via 3 : Distance 5


Router 3
Node 1 via 1 : Distance 7
Node 2 via 2 : Distance 1
Node 3 via 3 : Distance 0
```

RESULT:

Program to implement and simulate distance vector routing algorithm has been executed successfully and the output has been obtained.

PROGRAM

Server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define PORT 8080
#define MAX_BUFFER_SIZE 1024

int main() {
    int server_socket, client_socket, read_size;
    struct sockaddr_in server, client;
    char buffer[MAX_BUFFER_SIZE];


    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);


    if (bind(server_socket, (struct sockaddr*)&server, sizeof(server)) < 0) {
        perror("Binding failed");
        exit(EXIT_FAILURE);
    }

    listen(server_socket, 3);

    printf("FTP server listening on port %d...\n", PORT);

    while (1) {

        int client_address_size = sizeof(struct sockaddr_in);
        client_socket = accept(server_socket, (struct sockaddr*)&client, (socklen_t*)&client_address_size);
        if (client_socket < 0) {
            perror("Accept failed");
            exit(EXIT_FAILURE);

        }
```

EXPERIMENT NO:
FILE TRANSFER PROTOCOL

AIM: Program to implement File Transfer Protocol.

ALGORITHM:

Server Algorithm:

1. Create a socket using socket() function.
2. Initialize the server address structure (struct sockaddr_in) with the server's IP address, port number, and address family.
3. Bind the socket to the server address using bind() function.
4. Listen for incoming connections using listen() function.
5. Print a message indicating that the server is listening on the specified port.
6. Enter a continuous loop to accept incoming client connections:
    a. Accept a client connection using accept() function, which returns a new socket descriptor for the client.
    b. Print the client's IP address and port number.
    c. Open the file in write mode to save the received data.
    d. Enter a loop to receive data from the client:
        i. Receive data from the client using recv() function.
        ii. Write the received data to the file using fwrite() function.
    e. Close the file.
    f. Print a message indicating that the file has been received and saved.
    g. Close the client socket.
7. Close the server socket.

Client Algorithm:

1. Create a socket using socket() function.
2. Initialize the server address structure (struct sockaddr_in) with the server's IP address, port number, and address family.
3. Connect to the server using connect() function.
4. Print a message indicating a successful connection to the server.
5. Open the file to be sent in read mode.
6. Enter a loop to read and send data from the file:
    a. Read data from the file using fread() function.
    b. Send the data to the server using send() function.
7. Close the file.
8. Print a message indicating that the file has been sent successfully.
9. Close the client socket.

```
        printf("Connected to client: %s:%d\n", inet_ntoa(client.sin_addr), ntohs(client.sin_port));

        FILE* file = fopen("received_file.txt", "w");
        if (!file) {
            perror("File opening failed");
            exit(EXIT_FAILURE);
        }

        while ((read_size = recv(client_socket, buffer, sizeof(buffer), 0)) > 0) {
            fwrite(buffer, sizeof(char), read_size, file);
        }

        fclose(file);
        printf("File received and saved as 'received_file.txt'\n");

        close(client_socket);
    }

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

Client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>


#define SERVER_IP "127.0.0.1"
#define PORT 8080
#define MAX_BUFFER_SIZE 1024

int main() {
    int socket_desc;
    struct sockaddr_in server;
    char buffer[MAX_BUFFER_SIZE];

    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_desc == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr(SERVER_IP);
```

```
    server.sin_port = htons(PORT);

    if (connect(socket_desc, (struct sockaddr*)&server, sizeof(server)) < 0) {
        perror("Connection failed");
        exit(EXIT_FAILURE);
    }

    printf("Connected to FTP server at %s:%d\n", SERVER_IP, PORT);

    FILE* file = fopen("file_to_send.txt", "r");
    if (!file) {
        perror("File opening failed");
        exit(EXIT_FAILURE);
    }

    while (!feof(file)) {
        size_t read_size = fread(buffer, sizeof(char), sizeof(buffer), file);
        if (send(socket_desc, buffer, read_size, 0) < 0) {
            perror("Send failed");
            exit(EXIT_FAILURE);
        }
    }

    fclose(file);
    printf("File sent successfully\n");

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

OUTPUT

~$ ./client Connected to FTP server at 127.0.0.1:8080 File sent successfully	**~$.**/server FTP server listening on port 8080... Connected to client: 127.0.0.1:56606 File received and saved as 'received_file.txt'
file_to_send.txt hello world	**recieved.txt** hello world

RESULT:

Program to implement the File Transfer Protocol has been executed successfully and the output has been obtained.

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>

#define OUTPUT_RATE 1

int main() {
   int BUCKET_CAPACITY;
   printf("Enter the bucket capacity:");
   scanf("%d",&BUCKET_CAPACITY);
   int bucket_size = 0;  // Current bucket size
   while (true) {
      int incoming_packets = rand() % 5;
      printf("Incoming packets: %d\n", incoming_packets);
      bucket_size += incoming_packets;
      if (bucket_size > BUCKET_CAPACITY) {
         printf("Bucket overflow! Dropping packets: %d\n", bucket_size - BUCKET_CAPACITY);
         bucket_size = BUCKET_CAPACITY;
      }
      if (bucket_size > 0) {
         int outgoing_packets = bucket_size < OUTPUT_RATE ? bucket_size : OUTPUT_RATE;

         printf("Outgoing packets: %d\n", outgoing_packets);
         bucket_size -= outgoing_packets;
      }
      sleep(1);
   }
   return 0;
}
```

OUTPUT:

```
Enter the bucket capacity: 10
Incoming packets: 2
Outgoing packets: 1
Incoming packets: 4
Bucket overflow! Dropping packets: 1
Outgoing packets: 1
Incoming packets: 3
Bucket overflow! Dropping packets: 4
Outgoing packets: 1
Incoming packets: 1
Bucket overflow! Dropping packets: 6
Outgoing packets: 1
...
```

EXPERIMENT NO:
LEAKY BUCKET ALGORITHM

AIM: Program to implement congestion control using a leaky bucket algorithm.

ALGORITHM:

1. Prompt the user to input the bucket capacity.
2. Initialize the bucket size to 0.
3. Enter an infinite loop:
    a. Generate a random number of incoming packets.
    b. Display the number of incoming packets.
    c. Add the incoming packets to the bucket size.
    d. If the bucket size exceeds the bucket capacity:
        i. Calculate the number of packets to be dropped (bucket size - bucket capacity).
        ii. Display the number of dropped packets.
        iii. Set the bucket size to the maximum capacity.
    e. If the bucket size is greater than 0:
        i. Calculate the number of outgoing packets (minimum of bucket size and output rate).
        ii. Display the number of outgoing packets.
        iii. Subtract the outgoing packets from the bucket size.
    f. Sleep the program execution for 1 second.
4. End of the loop.
5. End of the program

RESULT:

Program to implement congestion control using leaky bucket algorithm has been executed successfully and output has been obtained.