

Experiment No. 1

FORK IMPLEMENTATION

AIM:

Implement a C program in which the main program accepts integers to be sorted and sorts elements using bubble sort. Parent process sorts the integers in ascending order and child process sorts the elements in descending order.

THEORETICAL BACKGROUND:

Fork system call creates a new process, which is called *child process*, which runs concurrently with parent process (the process that called system call fork). When a program calls fork, a duplicate process called the child process is created, which is an exact copy of the parent process. After a new child process created, both parent and the child processes will execute the next instruction following the fork() system call.

The running instance of a program is called process. Each process in a Linux system is identified by its unique process ID, referred as PID. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created. The child process is a new process and has a new process ID distinct from its parent's process ID.

Fork system call provides different return values to the parent and child process. It takes no parameters and returns an integer value. The different values returned by fork() include:

- *Negative Value*: creation of a child process was unsuccessful.
- *Zero*: Returned to the newly created child process.
- *Positive value*: Returned to parent process. The value contains process ID of newly created child process.

Syntax:- pid_t fork();

SYSTEM CALLS USED

- fork()- to create a new process.

ALGORITHM:

Step 1 : Include the necessary header files

Step 2: Declare the integer values : n, temp, A[]

Step 3 :Read number of elements to 'n'

Step 4: Read elements to array, A[]

Step 5: Bubble sort the elements.

Step 6: Create a Process using fork()

Step 7: If pid>0, its Parent Process and display elements in ascending order.

Step 8: Else if pid=0, its Child Process and display elements in descending order.

Step 9: Else process cant be created.

PROGRAM:

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    int n,temp;
    printf("Enter number of elements in the array to be sorted:");
    scanf("%d",&n);
    int A[n];
    printf("Enter the elements:\n");
    for(int i=0;i<n;i++)
        scanf("%d",&A[i]);
    printf("\nThe array is:\n");
    for(int i=0;i<n;i++)
        printf("%d ",A[i]);
    printf("\n");
```

```
for(int i=1;i<n;i++)
{
for(int j=0;j<n-i;j++)
{
if(A[j]>A[j+1])
{
temp=A[j];
A[j]=A[j+1];
A[j+1]=temp;
}
}
}
printf("\nElements are sorted using bubble sort..\n");
pid_t pid=fork();
if(pid>0)
{
printf("\nPARENT PROCESS:\n");
printf("Elements in ascending order:");
for(int i=0;i<n;i++)
printf("%d ",A[i]);
}
else if(pid==0)
{
printf("\nCHILD PROCESS:\n");
printf("Elements in descending order:");
for(int i=0;i<n;i++)
{
for(int j=i+1;j<n;j++)
{
if(A[i]<A[j])
```

```
{
temp=A[j];
  A[j]=A[i];
  A[i]=temp;
}
}
}
for(int i=0;i<n;i++)
printf("%d ",A[i]);
}
else
{
printf("\nFORK CANNOT BE CREATED!!!\n");
}
}
```

OUTPUT:

```
Enter number of elements in the array to be sorted:7
Enter the elements:
123 100 455 576 345 999 678

The array is:
123 100 455 576 345 999 678

Elements are sorted using bubble sort..

PARENT PROCESS:
Elements in ascending order:100 123 345 455 576 678 999
CHILD PROCESS:
Elements in descending order:999 678 576 455 345 123 100
```

Experiment No. 2

INTER PROCESS COMMUNICATION USING PIPE

AIM:

Implement a full duplex communication between parent and child process. Parent process writes filename of a file on pipe1. This filename is read by child process. The child process writes the contents of the file to pipe2. The contents are read from pipe2 by the parent process and is displayed on standard output.

THEORETICAL BACKGROUND:

Pipes are one of the most commonly used mechanisms for IPC (inter process communication). IPC is the mechanism by which two or more processes communicate with each other. The commonly used IPC techniques include shared memory, message queues, pipes and FIFOs.

A Pipe is a communication device that permits unidirectional communication. Data written to the “write end” of the pipe is read back from the “read end”. Pipes are serial devices; the data always read from the pipe in the same order it was written. A pipe is used to communicate between two threads in a single process or between parent and child processes. The pipe() system call creates a pipe.

SYSTEM CALLS USED

- pipe()
 - Syntax: int pipe(int fd[2])
 - pipe() creates a pair of file descriptors.
 - fd[0] is for reading, fd[1] is for writing.
 - Returns zero on success and -1 on error.
- write()
 - Syntax: write(int fd, void *buf, size_t count);
 - Writes to file descriptor. returns upto count bytes from buffer to the file pointed by file referred by file descriptor.
- read()
 - Syntax: read(int fd, void *buf, size_t count);
 - Attempts to read upto count bytes from file descriptor fd into buffer starting at buf.
- open()
 - Syntax: int open(const char *pathname, int flags);
 - Returns file descriptor.

ALGORITHM:

Step 1: Create two pipes

Step 2: Create a child process

Step 3: Parent process write the file name to pipe1

Step 4: Child process read the file contents from pipe1 and write to pipe2.

Step 5: Parent process read the contents from pipe 2 and display it.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

void main()
{
int pipefd1[2],pipefd2[2];
char readbuf[50],writebuf[50],filename[10]="";
    FILE *fp;
int pipe1=pipe(pipefd1);
if(pipe1==0)
    {
printf("PIPE1 CREATED SUCCESSFULLY!!\n");
    }
int pipe2=pipe(pipefd2);
if(pipe2==0)
    {
printf("PIPE2 CREATED SUCCESSFULLY!!\n");
    }
pid_t pid=fork();
if(pid>0)
    {
printf("\nPARENT PROCESS:\n");
```

```
close(pipefd1[0]);
printf("Enter filename:");
scanf("%s",filename);
fp=fopen(filename,"w");
fputs("This is the second experiment of networks lab",fp);
fclose(fp);
write(pipefd1[1],filename,sizeof(filename));
sleep(5);
close(pipefd2[1]);
read(pipefd2[0],readbuf,sizeof(readbuf));
printf("\nThe contents read from PIPE2 is:%s",readbuf);
    }
else if(pid==0)
    {
sleep(3);
printf("\n\nCHILD PROCESS:\n");
close(pipefd1[1]);
read(pipefd1[0],readbuf,sizeof(readbuf));
fp=fopen(readbuf,"r");
fgets(writebuf,sizeof(writebuf),fp);
fclose(fp);
printf("The content from pipe1 is obtained successfully.\n");
close(pipefd2[0]);
write(pipefd2[1],writebuf,sizeof(writebuf));
    }
}
```

OUTPUT:


```
PIPE1 CREATED SUCCESSFULLY!!
PIPE2 CREATED SUCCESSFULLY!!

PARENT PROCESS:
Enter filename:abc

CHILD PROCESS:

The content from pipe1 is obtained successfully.

The contents read from PIPE2 is:This is the second experiment of networks lab

...Program finished with exit code 0
Press ENTER to exit console.█
```

Experiment No. 3

NAMED PIPES

AIM:

Implement full duplex communication between three independent processes using named pipe/fifo. Parent process accepts sentences and writes in pipe1. Second process reads it and find the number of characters, words, sentences and write this output to pipe2. This is read by a third process and is displayed on screen.

THEORETICAL BACKGROUND:

It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.

A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.

Pipes were meant for communication between related processes. Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal and the server program from another terminal? The answer is No. Then how can we achieve unrelated processes communication, the simple answer is Named Pipes. Even though this works for related processes, it gives no meaning to use the named pipes for related process communication.

We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes. The answer is no, we can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication.

- Named pipe is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Creating a FIFO file: In order to create a FIFO file, a function calls i.e. *mkfifo* is used.

```
int mkfifo(const char *pathname, mode_t mode);
```

File mode can also be represented in octal notation such as 0XYZ, where X represents owner, Y represents group, and Z represents others. The value of X, Y or Z can range from 0 to 7. The values for read, write and execute are 4, 2, 1 respectively. If needed in combination of read, write and execute, then add the values accordingly.

Say, if we mention, 0640, then this means read and write (4 + 2 = 6) for owner, read (4) for group and no permissions (0) for others.

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with *errno* variable or *perror()* function.

ALGORITHM:

Step 1- Create two named pipes/FIFO

Step 2- First process accepts sentences and writes in pipe1.

Step 3- Second process reads it and find the number of characters, words, sentences.

Step 4- Write this output to pipe2.

Step 5- Third process read from pipe2 and is displayed on screen.

PROGRAM:

1.

```
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    int fd;
    printf("\nPROCESS:1");
    printf("\n-----\n");
    char *myfifo="/tmp/myfifo";
    mkfifo(myfifo,0666);
    char arr[80];
    while(1)
    {
        fd=open(myfifo,O_WRONLY);
        printf("\nENTER INPUT STRING:\n");
        fgets(arr,80,stdin);
        write(fd,arr,strlen(arr)+1);
        close(fd);
    }
}
```

```
}  
return 0;  
}
```

2.

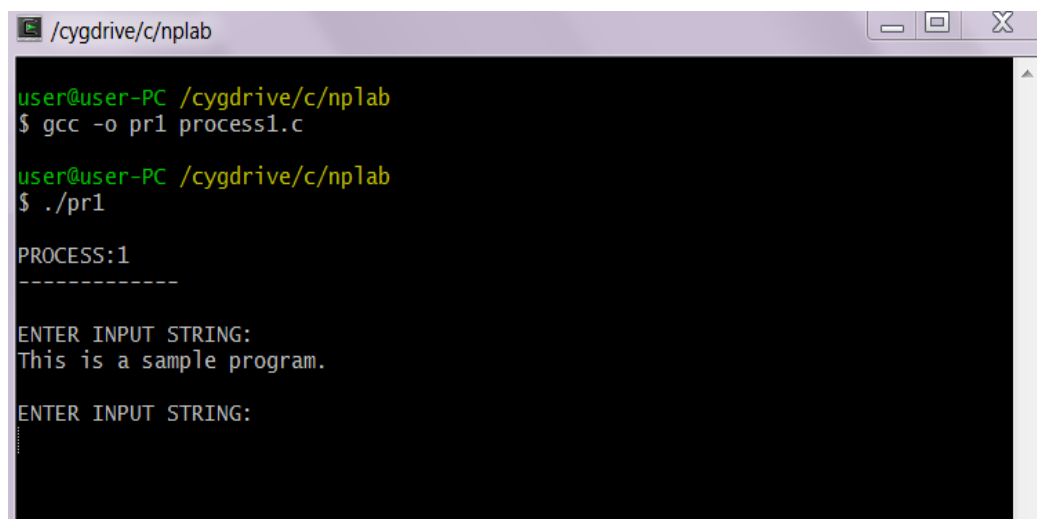
```
#include<stdio.h>  
#include<string.h>  
#include<fcntl.h>  
#include<sys/stat.h>  
#include<sys/types.h>  
#include<unistd.h>  
#include<ctype.h>  
int main()  
{  
int fd1,i,nc=0,nw=0,nl=0;;  
char *myfifo="/tmp/myfifo";  
mkfifo(myfifo,0666);  
char *myfifo2="/tmp/myfifo2";  
mkfifo(myfifo2,0666);  
char str[80];  
printf("\nPROCESS:2");  
printf("\n-----\n");  
while(1)  
{  
fd1=open(myfifo,O_RDONLY);  
read(fd1,str,80);  
printf("RECEIVED INPUT:%s\n",str);  
close(fd1);  
fd1=open(myfifo2,O_WRONLY);  
for(i=0;i<strlen(str);i++)
```

```
{
if(isblank(str[i]))
nw++;
else if(isalnum(str[i]))
nc++;
else if(str[i]=='.')
{
nl++;
nw++;
}
}
printf("\n CALCULATED!!! \n SENDING...\n");
sprintf(str,"\n LINE:%d\n WORD:%d\n CHARACTERS:%d\n",nl,nw,nc);
write(fd1,str,strlen(str)+1);
close(fd1);
}
return 0;
}
```

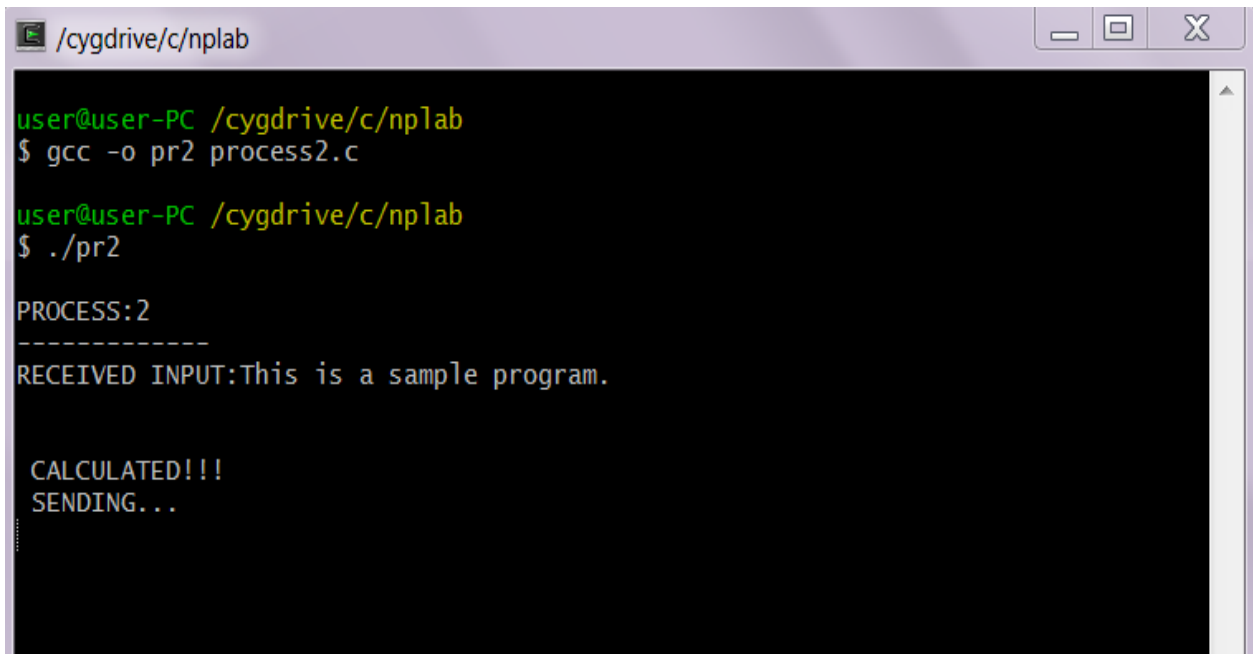
3.

```
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
void main()
{
int fd;
```

```
char recbuf[50];
printf("\nPROCESS:3");
printf("\n-----\n");
while(1)
{
char *myfifo2="/tmp/myfifo2";
mkfifo(myfifo2,0666);
fd=open(myfifo2,O_RDONLY);
read(fd,recbuf,sizeof(recbuf));
close(fd);
puts(recbuf);
}
}
```

OUTPUT:

```
/cygdrive/c/nplab
user@user-PC /cygdrive/c/nplab
$ gcc -o pr1 process1.c
user@user-PC /cygdrive/c/nplab
$ ./pr1
PROCESS:1
-----
ENTER INPUT STRING:
This is a sample program.
ENTER INPUT STRING:
```



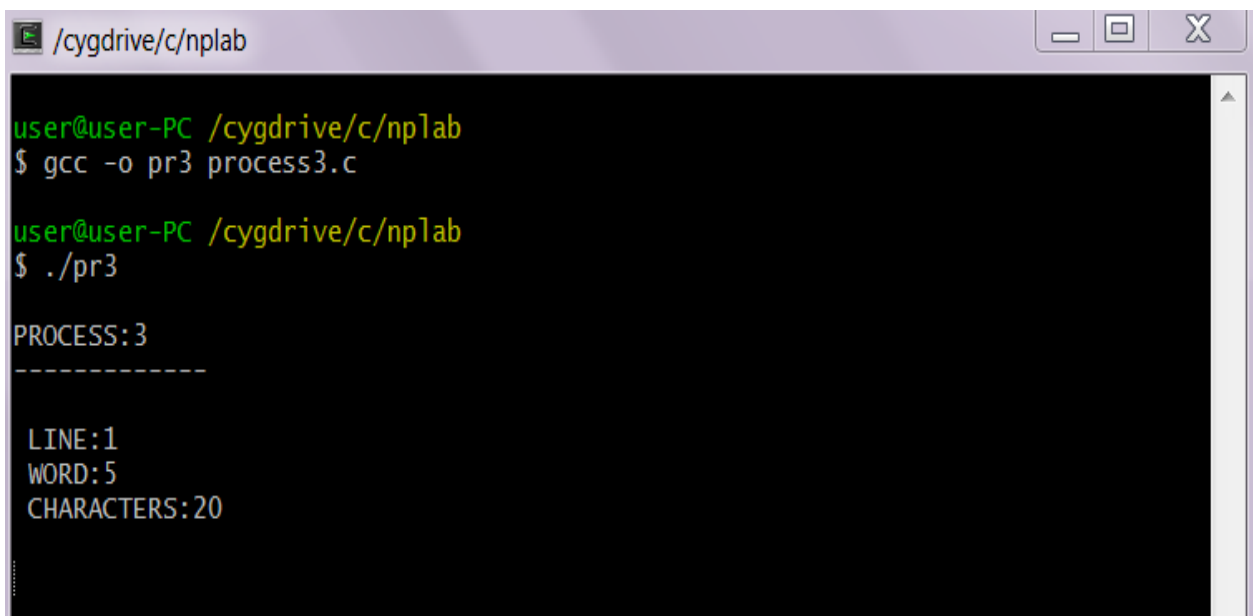
```
/cygdrive/c/nplab

user@user-PC /cygdrive/c/nplab
$ gcc -o pr2 process2.c

user@user-PC /cygdrive/c/nplab
$ ./pr2

PROCESS:2
-----
RECEIVED INPUT:This is a sample program.

CALCULATED!!!
SENDING...
```



```
/cygdrive/c/nplab

user@user-PC /cygdrive/c/nplab
$ gcc -o pr3 process3.c

user@user-PC /cygdrive/c/nplab
$ ./pr3

PROCESS:3
-----

LINE:1
WORD:5
CHARACTERS:20
```

Experiment No. 4

BASICS OF NETWORKING COMMANDS

AIM:

Familiarization of Network configuration files and Networking commands in Linux.

THEORETICAL BACKGROUND:

Networking Commands

Computers are often connected to each other on a network. They send request to each other in form of packets that travels from the host to destination. Linux provides various commands from network configuration and trouble-shooting. These includes:

1. IFCONFIG

ifconfig(Interface Configuration) is a utility in operating system that is used to set or display the IP address and netmask of a network interface. It also provides commands to enable or disable an interface. Many UNIX-like operating system initializes their network interfaces using ifconfig at boot time. ifconfig is also used to view the MTU(Maximum transmission unit).

2. PING(Packet Internet Groper)

ping command is used to ensure that a computer can communicate to a specified device over the network. ping command sends Internet Control Message Protocol(ICMP) Echo request messages in the form of packets

to the destination computer and waits in order to get the response back. Once the packets are received by the destined computer, it starts sending the packets back. This command keeps executing until it is interrupted. ping command provides the details such as

- number of packets transmitted
- number of packets received
- time taken by the packet to return

ping command is generally used for the following purposes:

- measuring the time taken by the packets to return to determine speed of the connection
- to make sure that the network connection between host and the destined computer can be established

3. **NSLOOKUP**

nslookup command queries the DNS in order to fetch the IP address or the domain name from DNS records.

4. **TRACEROUTE**

This command is used to get the route of a packet. In other words, traceroute command is used to determine the path along which a packet travels. It also returns the number of hops taken by the packet to reach the destination. This command prints to the console, a list of hosts through which the packet travels in order to the destination.

5. **HOST**

host command is used to find domain name associated with the IP address or find IP address associated with domain name. The returned IP address is either IPv4 or IPv6.

6. **NETSTAT**

netstat (Network Statistics) is the command that is used to display routing table, connection information, status of ports, etc. This command works with Linux Network Subsystem. This command basically displays the content of /proc/net file defined in linux file system.

7. **DIG Command**

Dig (domain information groper) query DNS related information like A Record, CNAME, MX Record etc. This command is mainly used to troubleshoot DNS related query.

8. **ROUTE Command**

route command shows and manipulates **ip** routing table.

Network Configuration File

Every operating system stores network configuration information in files. Some of these files specify information about the host's address and host name, or unique setup parameters. Other files specify which network services the host will allow, and which other hosts on the network provide services the host may require.

1. ***/etc/hosts File***

- One of the most frequently used network administration files is the */etc/hosts* file (*/etc/inet/hosts* on System V machines).
- The file is a registry of IP addresses and associated host names known to a system.
- At a minimum, it must contain the loop-back address (127.0.0.1) and the IP address for the host.
- The *hosts* file is one of the resources consulted by applications in order to resolve a host name to an IP address when communications are requested.

2. */etc/hostname.if_name File*

- Many versions of UNIX use files in the */etc* directory to aid in the configuration of individual interfaces on the system.
 - For example, Solaris uses files with the generic name */etc/hostname.if_name* to simplify system configuration at boot time.
 - The device name of the network interface is substituted for the *if_name* portion of the file name.
 - For a host with an on-board *hme* Ethernet device, connected to a single network, the */etc/hostname.hme0* file would contain the host name to be used by that interface.
 - Machines connected to multiple networks would have multiple */etc/hostname.if_name* files.
 - Solaris also uses a file called */etc/hostname6.if_name* to configure any Ipv6 interfaces on the system.

3. */etc/services file*

- The */etc/services* file contains a list of network ports and services that correspond to those ports.
 - For example, port 25 is defined as the SMTP port, whereas port 80 is reserved as the hypertext transport protocol daemon (*httpd*) port.
 - To add a new service to a host, the administrator must add a port number and service name pair to the */etc/services* file.

4. */etc/resolv.conf File*

Most versions of UNIX use the information in the */etc/resolv.conf* file to configure the name service client on the host. The file consists of keywords and values. Some of the more common keywords follow.

1. *domain*: DNS domain of this host
2. *nameserver* (up to three allowed): IP address of the name server(s) this host should contact. The preferred name server should be listed first.
3. *search*: List of up to seven domains the system should search when trying to resolve an unqualified host name.

5. */etc/nsswitch.conf File*

The */etc/nsswitch.conf* file, also known as the service switch file, is used to tell the system which order it should try to resolve host names.

6. */etc/networks*

The *networks* file is used to associate symbolic network names with Internet protocol addresses. The *networks* file associates Internet Protocol (IP) network numbers with network names. The format of this file is:

```
# network-name network-number ncnames
```

OUTPUT:

Experiment No. 5

CLIENT-SERVER COMMUNICATION USING TCP

AIM:

Program to implement interprocess communication using TCP.

THEORETICAL BACKGROUND:**TCP**

TCP refers to Transmission Control Protocol. It is one of the main protocols of the Internet protocol suite. TCP enables two hosts to establish a connection and exchange streams of data. It is a reliable protocol since it guarantees delivery of data and guarantees that packets will be delivered in the same order in which they are sent.

SOCKETS

A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. When you create a socket, you must specify three parameters: communication style, namespace, and protocol. A communication style controls how the socket treats transmitted data and specifies the number of communication partners. When

data is sent through a socket, it is packaged into chunks called packets. The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

Socket Stream styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender. A connectionstyle socket is like a telephone call: The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

Creating a connection-oriented socket:

- A connection-oriented server uses the following sequence of function calls:

socket()->bind()->listen()-> accept()-> send()-> recv()-> close()

- A connection-orientated client uses the following sequence of function calls:

socket ()-> connect() ->send()-> recv()-> close()

SYSTEM CALLS USED

1.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

SOCKET TYPE

Stream (SOCK_STREAM):

This type of socket is connection-oriented. Establish 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 considers the data to be a stream of bytes.

2. 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)

3. connect ()

- This function is used to connect to an IP address on a defined port.
- syntax: int connect (int fd, struct sockaddr* serv_addr, int addrlen)
 - fd-> socket file descriptor
 - addrlen -> sizeof (struct sockaddr)
 - serv->pointer to struct sockaddr_in

4.listen ()

- This function is used to wait for incoming connection. Before calling listen(), bind() is to be called. After calling listen (), accept () is to be called in order to accept incoming connection.
- syntax: int listen (int fd, int backlog)
 - fd-> socket file descriptor
 - backlog-> number of allowed connections

5.accept ()

- This function is used to accept an incoming connection.
- syntax: int accept (int fd, void* addr, int *addrlen)
 - fd-> socket file descriptor
 - addr-> ptr to struct sockaddr
 - addrlen-> sizeof (struct sockaddr)

6.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 send
 - len-> length of the data to be send
 - flags-> 0

7.recv ()

- This function receives the data send over the stream sockets. It returns the number of bytes read into the buffer.
- syntax: int recv (int fd, void *buf, int len,unsigned int flags)
 - fd-> socket file descriptor
 - buf-> buffer to read the information into
 - len-> maximum length of the buffer
 - flags-> set to 0

8.close()

- This function is used to close the connection on your socket descriptor.
- syntax: close (fd);
 - fd-> socket file descriptor

ALGORITHM:**Client**

- Step 1 : Include the necessary header files
Step 2: Declare the two integer values : numbytes, sockfd Step 3 : Declare a char buff of size 100
Step 4: Declare a variable servaddr of type sockaddr_in
Step 5: Call the socket function and check if it is successful or not and store the return value in sockfd
Step 6: Initialize the values of serveraddr to 0, using the bzero function Step 7: Initialize the values of sin_family and sin_port of the server address
Step 8: Use the connect function to establish the connection with the server and check if it successful or not
Step 9: Use recv() function to receive the data sent by the server and check if it was successfully received or not
Step 10: Print the data that was received on the screen

Server

- Step 1: Declare two integer values – listenfd and connfd
Step 2: Declare a variable servaddr of type structure sockaddr_in; Step 3: Declare a variable clientaddress of type structure sockaddr_in;
Step 4: Using the socket function create a socket and check if it was successful ornot Step 5: Initialize the values of the servaddr to zero using bzero function
Step 6: Initialize the values of sin_family, sin_port and in_addr (INADDR_ANY)
Step 7: Use the bind() function to get the local protocol address of the server and check if it is successful or not

Step 8: Use the listen() function to wait for a connection and check if it is successful or not Step 9: Use the accept () function to accept a client connection and store the return value in the connfd variable . Check if it is successful or not
Step 10 : Use the str_echo function to process the data send by the client.

PROGRAM:**TCP SERVER:**

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

void main()
{
printf("Server side\n");
char tcpbuffer[50];
int sockfd,newsocket;

struct sockaddr_in addr1,addr2;
addr1.sin_family=AF_INET;
addr1.sin_addr.s_addr=INADDR_ANY;
addr1.sin_port=5000;
int s=sizeof(struct sockaddr_in);
sockfd=socket(AF_INET,SOCK_STREAM,0);
bind(sockfd,(struct sockaddr *)&addr1,sizeof(addr1));
listen(sockfd,5);
newsocket=accept(sockfd,(struct sockaddr *)&addr2,(&s));
do
```

```
{
printf("Connection Established\n");
printf("Receiving message from client: ");
recv(newsocket,tcpbuffer,sizeof(tcpbuffer),0);
printf("%s",tcpbuffer);
printf("\nEnter the message:");
scanf("%s",tcpbuffer);
send(newsocket,tcpbuffer,sizeof(tcpbuffer),0);
}while(strcmp(tcpbuffer,"stop")!=0);

close(newsocket);
close(sockfd);
}
```

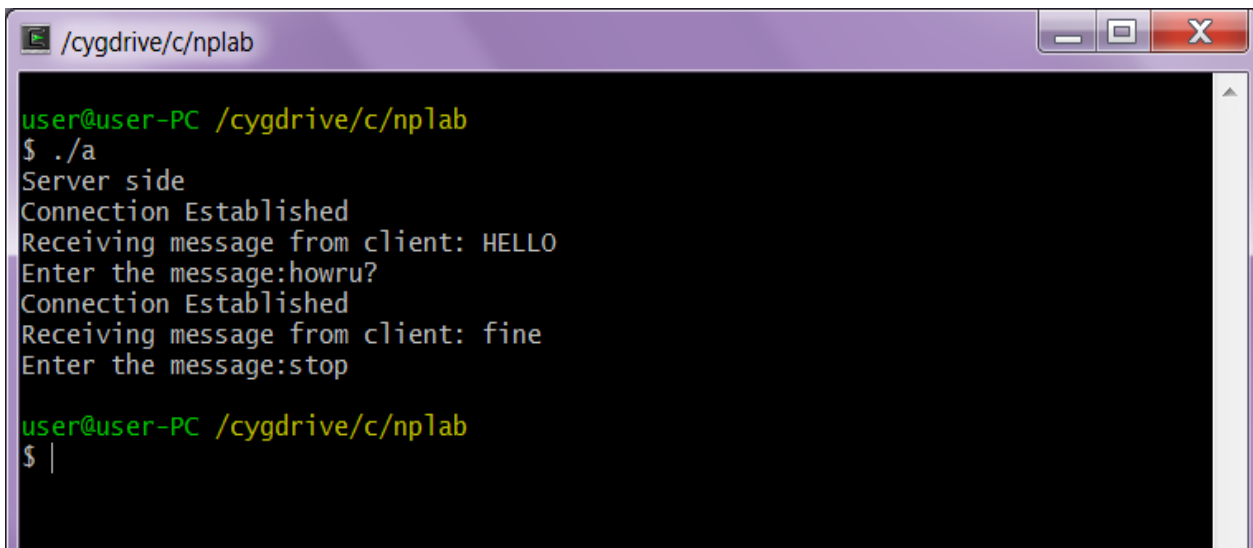
TCP CLIENT:

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

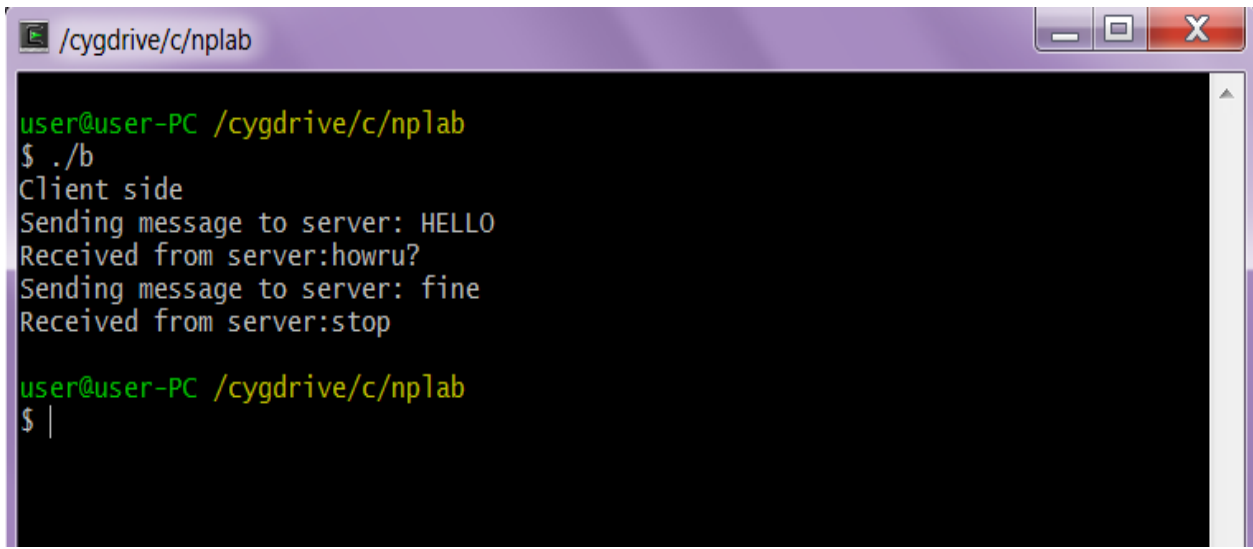
```
void main()
{
printf("Client side!!!\n");
char tcpbuffer[50];
int sockfd;
sockfd=socket(AF_INET,SOCK_STREAM,0);
struct sockaddr_in addr1;
```



```
addr1.sin_family=AF_INET;
addr1.sin_addr.s_addr=INADDR_ANY;
addr1.sin_port=5000;
connect(sockfd,(struct sockaddr *)&addr1,sizeof(addr1));
do
{
printf("Sending message to server: ");
scanf("%s",tcpbuffer);
send(sockfd,tcpbuffer,sizeof(tcpbuffer),0);
recv(sockfd,tcpbuffer,sizeof(tcpbuffer),0);
printf("Received from server:%s\n",tcpbuffer);
}while(strcmp(tcpbuffer,"stop")!=0);
close(sockfd);
}
```

OUTPUT:

```
/cygdrive/c/nplab
user@user-PC /cygdrive/c/nplab
$ ./a
Server side
Connection Established
Receiving message from client: HELLO
Enter the message:howru?
Connection Established
Receiving message from client: fine
Enter the message:stop
user@user-PC /cygdrive/c/nplab
$ |
```



A terminal window titled "/cygdrive/c/nplab" with standard Windows window controls (minimize, maximize, close). The terminal output shows a client-side program execution. The prompt is "user@user-PC /cygdrive/c/nplab". The user enters "./b", which runs a program. The program outputs "Client side", then "Sending message to server: HELLO", then "Received from server:howru?", then "Sending message to server: fine", and finally "Received from server:stop". The prompt returns to "user@user-PC /cygdrive/c/nplab" and the user enters a dollar sign followed by a vertical bar "\$ |".

```
user@user-PC /cygdrive/c/nplab
$ ./b
Client side
Sending message to server: HELLO
Received from server:howru?
Sending message to server: fine
Received from server:stop

user@user-PC /cygdrive/c/nplab
$ |
```

Experiment No. 6**CLIENT-SERVER COMMUNICATION USING UDP****AIM:**

Program to implement client-server communication using UDP.

THEORETICAL BACKGROUND:**UDP**

UDP refers to user datagram protocol. User Datagram Protocol (UDP) is part of the Internet Protocol suite used by programs running on different computers on a network. UDP is used to send short messages called datagrams but overall, it is an unreliable, connectionless protocol. UDP provides two services not provided by the IP layer. It provides port numbers to help distinguish different user requests.

SOCKETS

Datagram styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered. The system guarantees only “best effort,” so packets may disappear or arrive in a different order than shipping. A datagram-style socket behaves more like postal mail. The sender specifies the receiver’s address for each individual message

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 provider (protocol). The socket sends datagrams as independent packets with no guarantee of delivery. Datagrams can arrive out of order. For some transport providers, each datagram can use a different route through the network.

Creating a connectionless socket:

A connectionless client illustrates the socket APIs that are written for User Datagram Protocol (UDP).

The server uses the following sequence of function calls:

socket ()-> bind()-> sendto()-> recvfrom()-> close()

The client uses the following sequence of function calls:

socket ()-> sendto()-> recvfrom()-> close().

SYSTEM CALLS USED

1.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

SOCKET TYPE

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 provider (protocol). The socket sends datagrams as independent packets with no guarantee of delivery. Datagrams can arrive out of order. For some transport providers, each datagram can use a different route through the network.

2. 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)

3. sendto ()

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

syntax:

`int sendto (int fd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen)`

- fd-> socket file descriptor
- msg-> pointer to the data to be send
- len-> length of the data to be send
- flags-> 0
- to-> pointer to a struct sockaddr
- tolen-> sizeof (struct sockaddr)

3. recvfrom ()

- This function serves the same purpose as `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 the information into
- len-> maximum length of the buffer
- flags-> set to 0
- from-> pointer to struct sockaddr
- fromlen-> sizeof (ptr to an int that should be initialized to struct sockaddr)

ALGORITHM:

Client

- Step 1 : Include the necessary header files
- Step 2: Declare a variable serveraddr of type sockaddr_in
- Step 3: Call the socket function and check if it is successful or not and store the return value in sockfd
- Step 4: Initialize the values of serveraddr to 0, using the bzero function
- Step 5: Initialize the values of sin_family and sin_port of the serveraddress
- Step 6: Use the dg_cli() function for sending and receiving data.
- Step 7: Use sendto() function to send the data to the server.
- Step 8: Use recvfrom() function to receive the data sent by the server and check if it was successfully received or not
- Step 9: Print the data that was received on the screen

Server

- Step 1 : Declare integer variable – sockfd
- Step 2 : Declare a variable servaddr of type structure sockaddr_in; Step 3 : Declare a variable cliaddr of type structure sockaddr_in;
- Step 4 : Using the socket function create a socket and check if it was successful or not .
- Step 5 : Initialize the values of the servaddr to zero using bzero function
- Step 6 : Initialize the values of sin_family, sin_port and in_addr (INADDR_ANY)
- Step 7 : Use the bind() function to get the local protocol address of the server and check if it is successful or not
- Step 8 : Use the dg_echo() function for process the data send by the client.
- Step 9 : Use the recvfrom () function to accept a client connection and receive the data.
- Step 10 : Use the sendto() function to send some data to the client and check if it was successfully sent or not .

PROGRAM:

UDP SERVER:

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

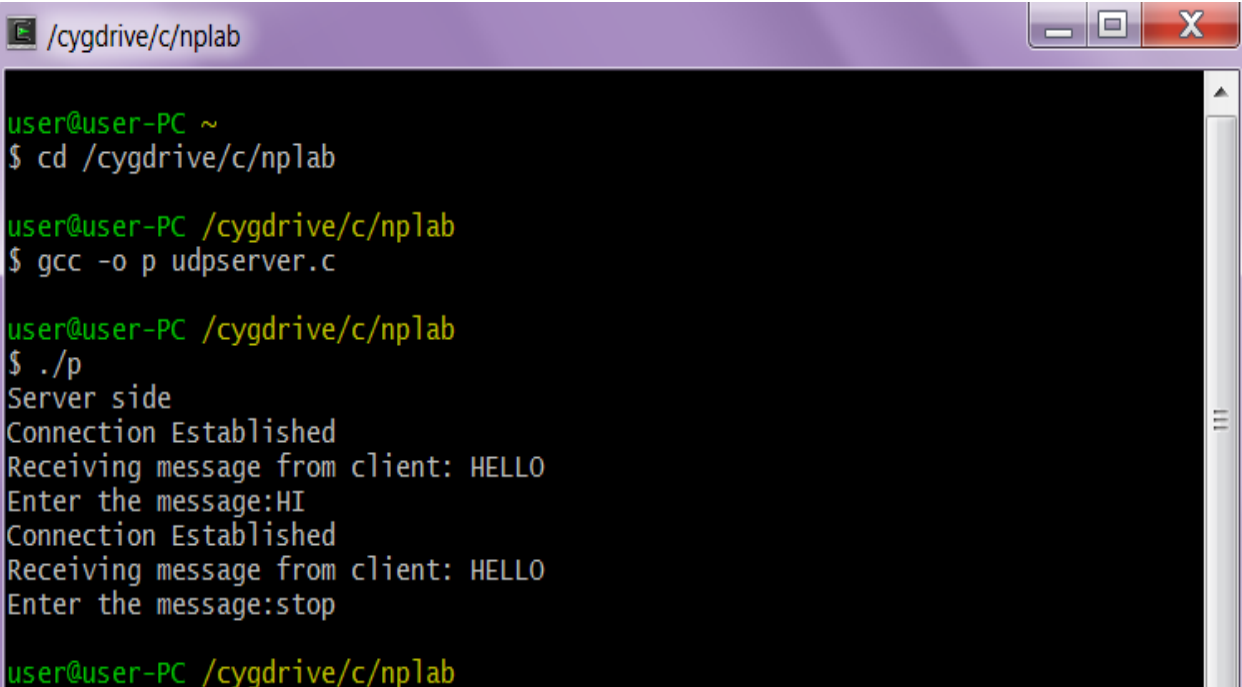
```
{
printf("Server side\n");
char udpbuffer[50];
int sockfd;
struct sockaddr_in addr;
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=INADDR_ANY;
addr.sin_port=5000;
int s=sizeof(struct sockaddr_in);
sockfd=socket(AF_INET,SOCK_DGRAM,0);
bind(sockfd,(struct sockaddr *)&addr,sizeof(addr));
do
{
printf("Connection Established\n");
printf("Receiving message from client: ");
recvfrom(sockfd,udpbuffer,sizeof(udpbuffer),0,(struct sockaddr *)&addr,&s);
puts(udpbuffer);
printf("\nEnter the message:");
gets(udpbuffer);
sendto(sockfd,udpbuffer,sizeof(udpbuffer),0,(struct sockaddr *)&addr,s);
}while(strcmp(udpbuffer,"stop")!=0);
close(sockfd);
}
```

UDP CLIENT:

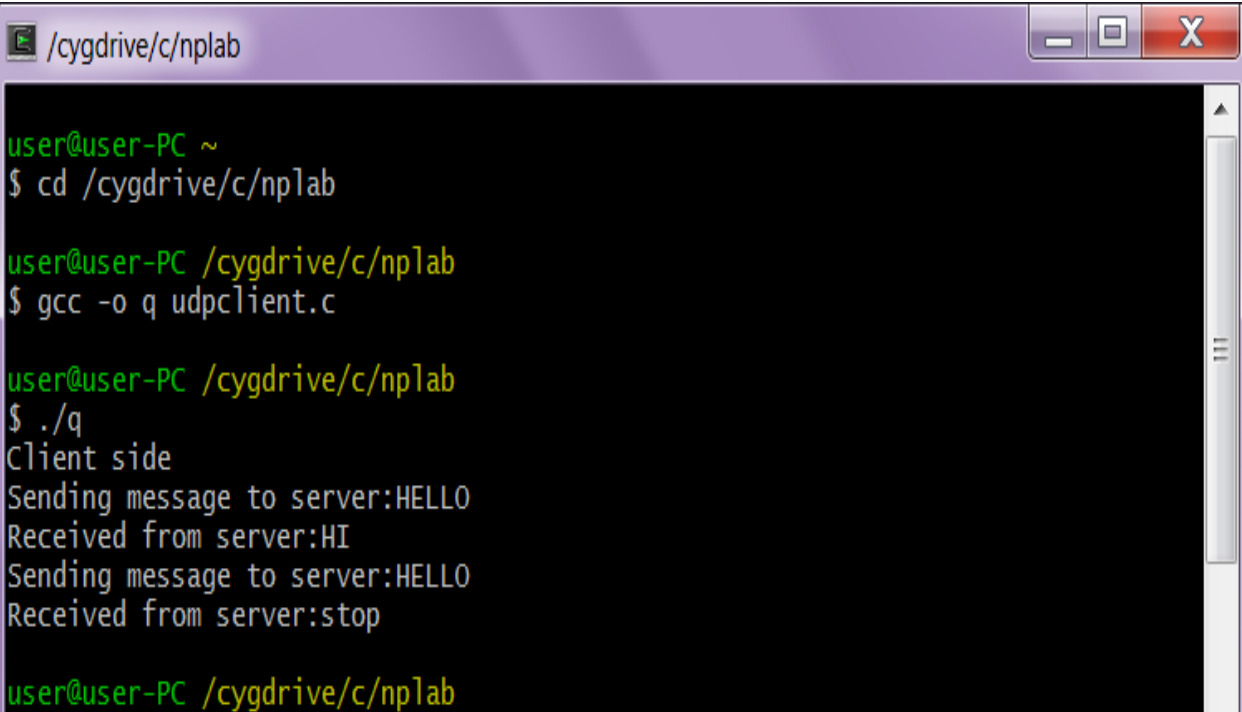
```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
```

```
void main()
{
printf("Client side\n");
char udpbuffer[50];
int sockfd;
sockfd=socket(AF_INET,SOCK_DGRAM,0);
struct sockaddr_in addr;
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=INADDR_ANY;
addr.sin_port=5000;
```

```
int s=sizeof(struct sockaddr_in);
connect(sockfd,(struct sockaddr*)&addr,sizeof(addr));
do
{
printf("Sending message to server:");
scanf("%s",udpbuffer);
sendto(sockfd,udpbuffer,sizeof(udpbuffer),0,(struct sockaddr *)&addr,s);
recvfrom(sockfd,udpbuffer,sizeof(udpbuffer),0,(struct sockaddr *)&addr,&s);
printf("Received from server:%s\n",udpbuffer);
}while(strcmp(udpbuffer,"stop")!=0);
close(sockfd);
}
```

OUTPUT:

```
/cygdrive/c/nplab
user@user-PC ~
$ cd /cygdrive/c/nplab
user@user-PC /cygdrive/c/nplab
$ gcc -o p udpserver.c
user@user-PC /cygdrive/c/nplab
$ ./p
Server side
Connection Established
Receiving message from client: HELLO
Enter the message:HI
Connection Established
Receiving message from client: HELLO
Enter the message:stop
user@user-PC /cygdrive/c/nplab
```



```
/cygdrive/c/nplab

user@user-PC ~
$ cd /cygdrive/c/nplab

user@user-PC /cygdrive/c/nplab
$ gcc -o q udpclient.c

user@user-PC /cygdrive/c/nplab
$ ./q
Client side
Sending message to server:HELLO
Received from server:HI
Sending message to server:HELLO
Received from server:stop

user@user-PC /cygdrive/c/nplab
```


Experiment No. 7

IMPLEMENTATION OF DNS

AIM:

THEORETICAL BACKGROUND:

The **Domain Name System (DNS)** is a hierarchical naming system for computers, services, or any resource participating in the Internet. The Domain Name System distributes the responsibility of assigning domain names and mapping those names to IP addresses.

ALGORITHM:

Server:

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port where the service will be defined to be used by client.
- Bind the address and port using *bind()* system call.
- Server executes *listen()* system call to indicate its willingness to receive connections.
- Accept the next completed connection from the client process by using an *accept()* system call.
- Receive the request from the Client using *recv()* system call.
- For the domain-name thus received from the Client, obtain the corresponding IP address using appropriate logic.
- Send the result (in the buffer) of the request made by the client using *send()* system call.

Client:

- Include appropriate header files.

- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- To obtain the IP address for the domain name. Send request to the server consisting of the domain-name using *send()* system call.
- Receive the result of the request made to the server using *recv()* system call.

PROGRAM:**SERVER:**

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

void main()
{
printf("--DNS SERVER--\n\n");
FILE *fp;
struct sockaddr_in server,client;
int s,n;
char b1[100],b2[100],a[100];
s=socket(AF_INET,SOCK_DGRAM,0);
server.sin_family=AF_INET;
server.sin_port=5000;
server.sin_addr.s_addr=INADDR_ANY;
bind(s,(struct sockaddr*)&server,sizeof(server));
n=sizeof(struct sockaddr_in);
```

```
while(1)
{
strcpy(b2,"");
fp=fopen("dns.txt","r");
recvfrom(s,b1,sizeof(b1),0,(struct sockaddr*)&client,&n);
while(!feof(fp))
{
fscanf(fp,"%s",a);
if(strcmp(a,b1)==0)
{
fscanf(fp,"%s",b2);
break;
}
}
if(strcmp(b2,"")==0)
{
strcpy(b2,"Not found!");
}
fclose(fp);
sendto(s,b2,sizeof(b2),0,(struct sockaddr*)
&client,n);
printf("sent ip=%s for domain %s\n",b2,b1);
}
close(s);
}
```

CLIENT:

```
#include<stdio.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<sys/socket.h>
```

```
#include<arpa/inet.h>
#include<netinet/in.h>
#include<unistd.h>
void main()
{
printf("--DNS CLIENT--\n");
struct sockaddr_in server,client;
int s,n;
char b1[100],b2[100];
s=socket(AF_INET,SOCK_DGRAM,0);
server.sin_family=AF_INET;
server.sin_port=5000;
server.sin_addr.s_addr=INADDR_ANY;
n=sizeof(struct sockaddr_in);
connect(s,(struct sockaddr*)
&server,sizeof(server));
printf("\nEnter address:");
scanf("%s",b1);
sendto(s,b1,sizeof(b1),0,(struct
sockaddr*)&server,n);
recvfrom(s,b2,sizeof(b2),0,NULL,NULL);
printf("ip:= %s\n",b2);
close(s);
}
```

TEXT FILE(dns.txt):

```
www.microsoft.com 93.170.52.20
www.google.com 192.168.0.7
www.yahoo.com 192.168.0.16
www.gmail.com 66.102.13.191
```

OUTPUT:

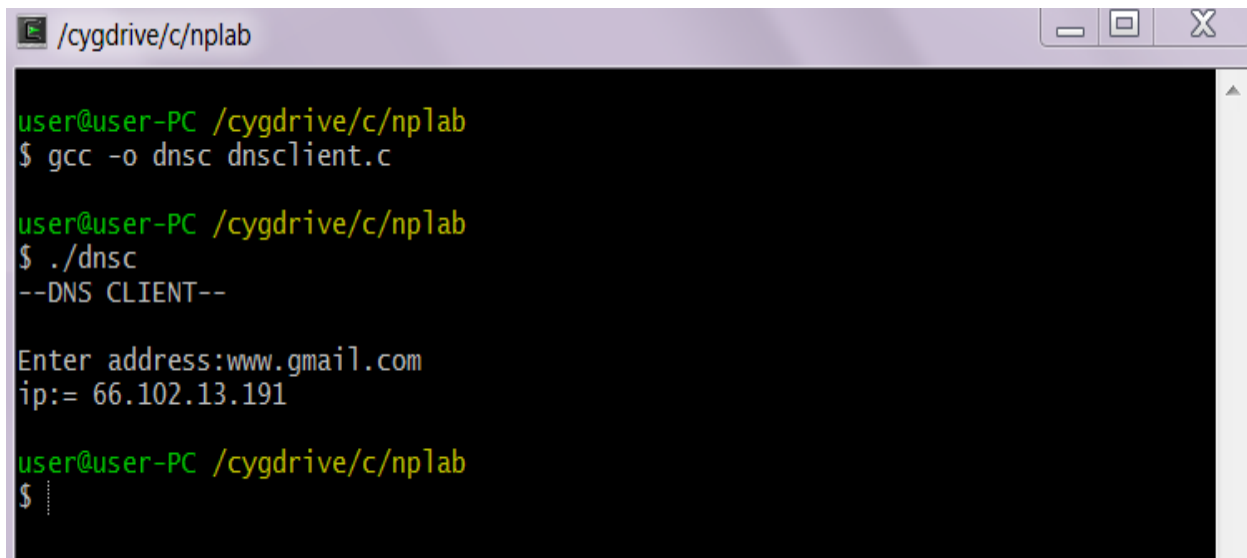
```
/cygdrive/c/nplab

user@user-PC /cygdrive/c/nplab
$ gcc -o dnss dnsserver.c

user@user-PC /cygdrive/c/nplab
$ ./dnss
--DNS SERVER--

sent ip=66.102.13.191 for domain www.gmail.com

user@user-PC /cygdrive/c/nplab
$
```



```
/cygdrive/c/nplab

user@user-PC /cygdrive/c/nplab
$ gcc -o dnsc dnscclient.c

user@user-PC /cygdrive/c/nplab
$ ./dnsc
--DNS CLIENT--

Enter address:www.gmail.com
ip:= 66.102.13.191

user@user-PC /cygdrive/c/nplab
$
```

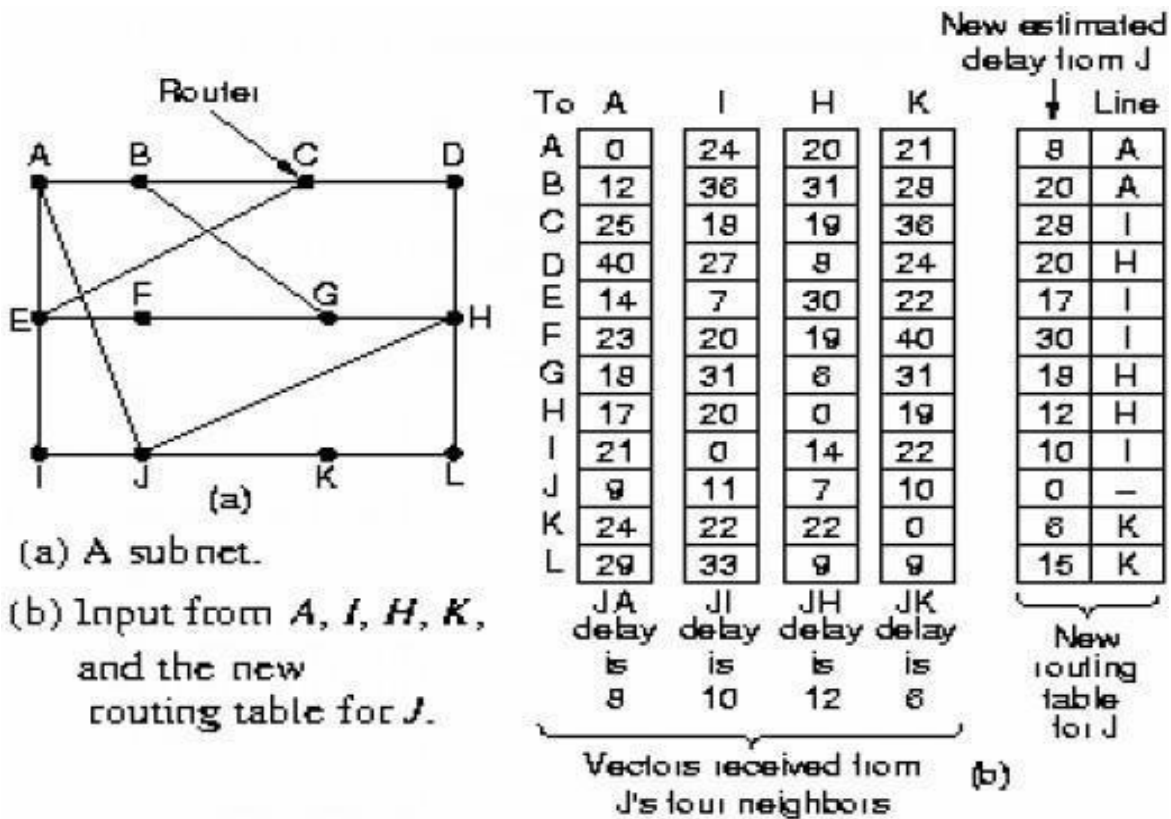
Experiment No. 8

IMPLEMENTATION OF DISTANCE VECTOR ROUTING

AIM:

To implement Distance vector routing

THEORETICAL BACKGROUND:



In

computer communication theory relating to packet-switched networks, a distance vector routing protocol is one of the two major classes of routing protocols, the other major class being the link-state protocol. A distance-vector routing protocol uses the Bellman-Ford algorithm to calculate paths.

A distance-vector routing protocol requires that a router informs its neighbors of topology changes periodically and, in some cases, when a change is detected in the topology of a network. Compared to link-state protocols, which require a router to inform all the nodes in a network of topology changes, distance-vector routing protocols have less computational complexity and message overhead.

Distance Vector means that Routers are advertised as vector of distance and direction. 'Direction' is represented by next hop address and exit interface, whereas 'Distance' uses metrics such as hop count.

Routers using distance vector protocol do not have knowledge of the entire path to a destination. Instead DV uses two methods:

1. Direction in which or interface to which a packet should be forwarded.
2. Distance from its destination.

Examples of distance-vector routing protocols include Routing Information Protocol Version 1 & 2, RIPv1 and RIPv2 and IGRP. EGP and BGP are not pure distance-vector routing protocols because a distance-vector protocol calculates routes based only on link costs whereas in BGP, for example, the local route preference value takes priority over the link cost.

ALGORITHM:

PROGRAM:

```
#include<stdio.h>
struct NODE
{
int distance[10],next[10];
}p[10];
void main()
{
int i,n,j,k,dis[10][10];
printf("DISTANCE VECTOR ROUTING");
printf("\n.....\n");
printf("Enter number of nodes:");
scanf("%d",&n);
printf("\nEnter the cost\n");
for(i=0;i<n;i++)
```

```
{
for(j=0;j<n;j++)
{
printf("Distance from %d to %d:",i,j);
scanf("%d",&dis[i][j]);
p[i].distance[j]=dis[i][j];
p[i].next[j]=j;
}
}
for(i=0;i<n;i++)
{
printf("\nRouting table(%d)\n",i);
printf("\n.....\n");
printf("Destination\tCost\tNext\n");
for(j=0;j<n;j++)
{
printf("%d\t%d\t%d\n",j,p[i].distance[j],p[i].next[j]);
}
printf("\n");
}
printf("\nCalculating optimal path.....");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
for(k=0;k<n;k++)
{
if(p[i].distance[j]>(dis[i][k]+p[k].distance[j]))
{
p[i].distance[j]=p[i].distance[k]+p[k].distance[j];
p[i].next[j]=k;
}
}
}
}
for(i=0;i<n;i++)
{
printf("\nOPTIMIZED ROUTING TABLE of %d\n",i);
printf("\n.....\n");
```



```
printf("\nDestination\tCost\tNext\n");
for(j=0;j<n;j++)
{
printf("%d\t%d\t%d\n",j,p[i].distance[j],p[i].next[j]);
}
printf("\n");
}
}
```

OUTPUT:

```
DISTANCE VECTOR ROUTING
.....
Enter number of nodes:4

Enter the cost
Distance from 0 to 0:0
Distance from 0 to 1:2
Distance from 0 to 2:9999
Distance from 0 to 3:1
Distance from 1 to 0:2
Distance from 1 to 1:0
Distance from 1 to 2:3
Distance from 1 to 3:7
Distance from 2 to 0:9999
Distance from 2 to 1:3
Distance from 2 to 2:0
Distance from 2 to 3:11
Distance from 3 to 0:1
Distance from 3 to 1:7
Distance from 3 to 2:11
Distance from 3 to 3:0
```

Routing table(0)

.....

Destination	Cost	Next
0	0	0
1	2	1
2	9999	2
3	1	3

Routing table(1)

.....

Destination	Cost	Next
0	2	0
1	0	1
2	3	2
3	7	3

Routing table(2)

.....

Destination	Cost	Next
0	9999	0
1	3	1
2	0	2
3	11	3

Routing table(3)

.....

Destination	Cost	Next
0	1	0
1	7	1
2	11	2
3	0	3

Calculating optimal path.....

OPTIMIZED ROUTING TABLE of 0

.....

Destination	Cost	Next
0	0	0
1	2	1
2	5	1
3	1	3

OPTIMIZED ROUTING TABLE of 1

.....

Destination	Cost	Next
0	2	0
1	0	1
2	3	2
3	3	0

OPTIMIZED ROUTING TABLE of 2

.....

Destination	Cost	Next
0	5	1
1	3	1
2	0	2
3	6	1

```
OPTIMIZED ROUTING TABLE of 3
```

```
.....
```

Destination	Cost	Next
0	1	0
1	3	0
2	6	0
3	0	3

```
...Program finished with exit code 0  
Press ENTER to exit console.
```

Experiment No. 9

CONCURRENT TIME SERVER USING UDP

AIM:

To implement concurrent time server application using UDP to execute the program at remote server. Client sends a time request to the server. Server sends its system time to the client. Client displays the result.

THEORETICAL BACKGROUND:**ALGORITHM:****Server:**

1. Include appropriate header files.
2. Initialise necessary variables, arrays and structure pointer sockaddr_in.
3. Create a socket and assign socket address by binding.
4. Wait for client requests.
5. Perform next 3 steps every time client request is received.
6. Call fork().
7. Retrieve current system time and store it in array.
8. Send the message(array) to the client.

Client :

1. Include appropriate header files.
2. Initialize necessary variables, arrays and structure pointer
3. Create a socket and assign socket address by binding.
4. Perform next 4 steps until termination occurs
5. Send request for time to server.
6. Receive data from the server.
7. Display time received.
8. Send stop message to stop connection

PROGRAM:

SERVER:

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

void main()
{
printf("Server side!!!\n");
char buffer[50];
int sockfd;
struct sockaddr_in addr;
    addr.sin_family=AF_INET;
    addr.sin_addr.s_addr=INADDR_ANY;
    addr.sin_port=5000;
int s=sizeof(struct sockaddr_in);
sockfd=socket(AF_INET,SOCK_DGRAM,0);
bind(sockfd,(struct sockaddr *)&addr,sizeof(addr));
printf("Connection established...\n");
do
{
recvfrom(sockfd,buffer,sizeof(buffer),0,(struct sockaddr *)&addr,&s);
printf("Message from client:%s\n",buffer);
pid_t pid=fork();
if(pid==0)
{
time_t rawtime;
struct tm *timeinfo;
```

```
time(&rawtime);
timeinfo=localtime(&rawtime);
sprintf(buffer,"Time is %d:%d:%d",timeinfo->tm_hour,timeinfo->tm_min,timeinfo->tm_sec);
sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr *)&addr,s);
printf("Time sent to client..\n");
}
}while(strcmp(buffer,"stop")!=0);
close(sockfd);
}
```

CLIENT-1:

```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
void main()
{
printf("Client 1!!!\n");
char buffer[50];
int sockfd;
sockfd=socket(AF_INET,SOCK_DGRAM,0);
struct sockaddr_in addr;
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=INADDR_ANY;
addr.sin_port=5000;
int s=sizeof(struct sockaddr_in);
connect(sockfd,(struct sockaddr *)&addr,sizeof(addr));
do
```

```
{
printf("Enter message for server:");
scanf("%s",buffer);
if(strcmp(buffer,"stop")==0)
break;
sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr *)&addr,s);
recvfrom(sockfd,buffer,sizeof(buffer),0,(struct sockaddr *)&addr,&s);
printf("Received from server:%s\n",buffer);
}while(strcmp(buffer,"stop")!=0);
close(sockfd);
}
```

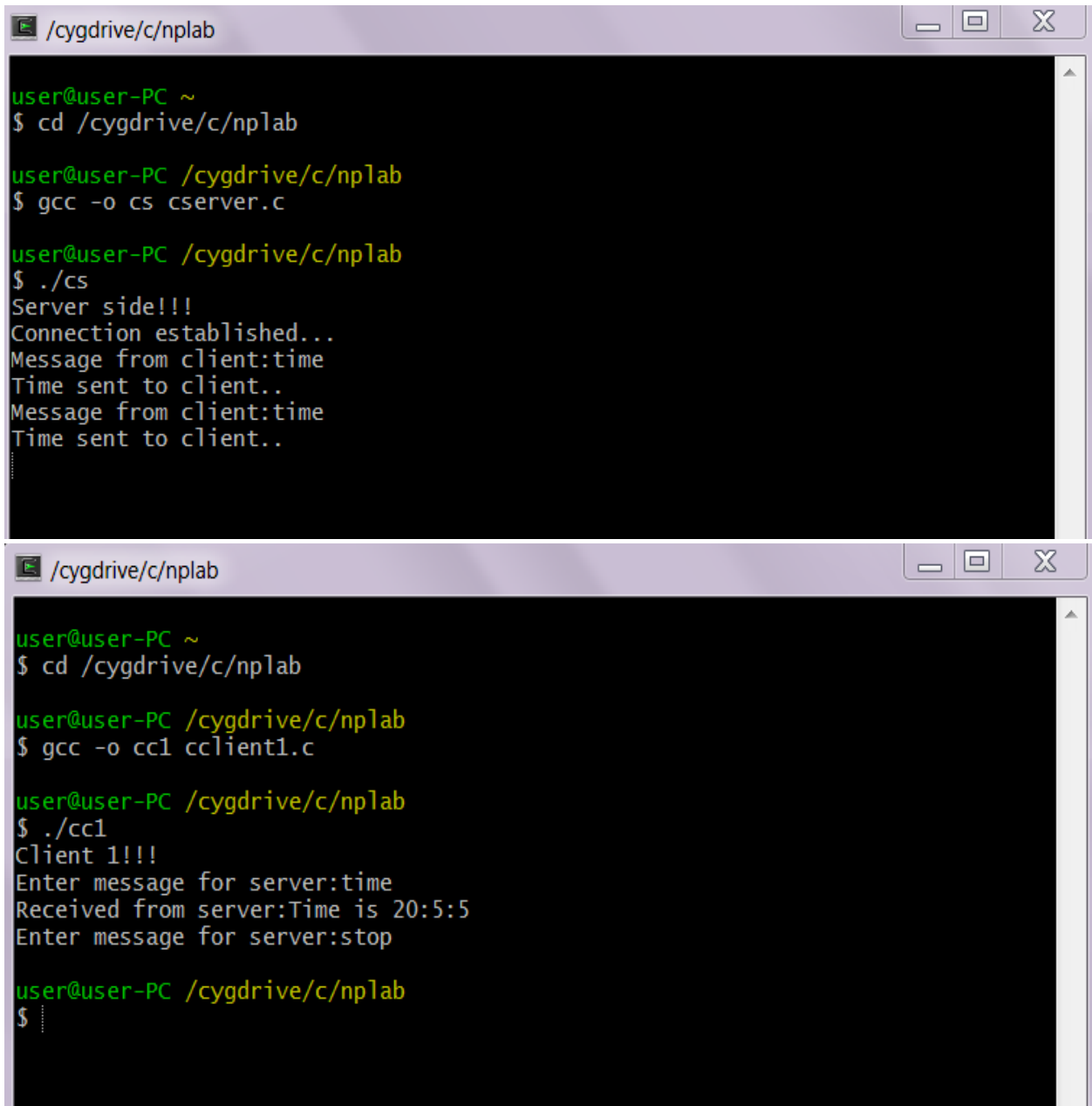
CLIENT-2:

```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
void main()
{
printf("Client 2!!!\n");
char buffer[50];
int sockfd;
sockfd=socket(AF_INET,SOCK_DGRAM,0);
struct sockaddr_in addr;
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=INADDR_ANY;
addr.sin_port=5000;
int s=sizeof(struct sockaddr_in);
```



```
connect(sockfd,(struct sockaddr*)&addr,sizeof(addr));
do
{
printf("Enter message for server:");
scanf("%s",buffer);
if(strcmp(buffer,"stop")==0)
break;
sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr*)&addr,s);
recvfrom(sockfd,buffer,sizeof(buffer),0,(struct sockaddr*)&addr,&s);
printf("Received from server:%s\n",buffer);
}while(strcmp(buffer,"stop")!=0);
close(sockfd);
}
```

OUTPUT:



```
/cygdrive/c/nplab
user@user-PC ~
$ cd /cygdrive/c/nplab

user@user-PC /cygdrive/c/nplab
$ gcc -o cs cserver.c

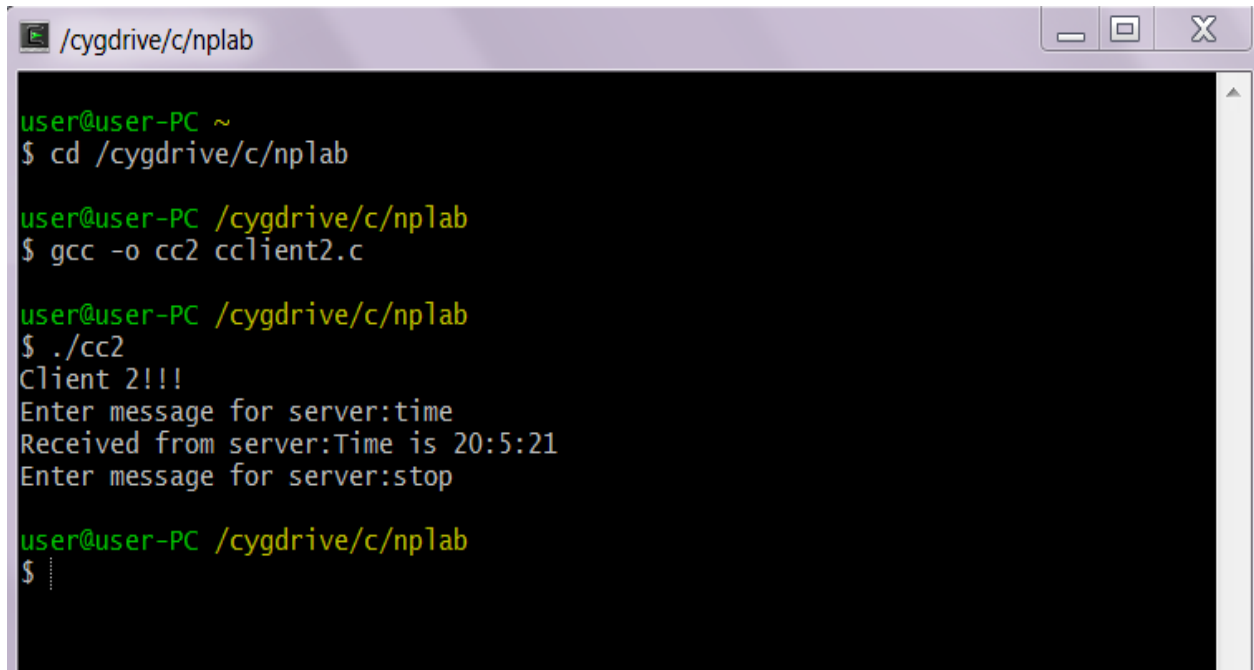
user@user-PC /cygdrive/c/nplab
$ ./cs
Server side!!!
Connection established...
Message from client:time
Time sent to client..
Message from client:time
Time sent to client..
:

/cygdrive/c/nplab
user@user-PC ~
$ cd /cygdrive/c/nplab

user@user-PC /cygdrive/c/nplab
$ gcc -o cc1 cclient1.c

user@user-PC /cygdrive/c/nplab
$ ./cc1
Client 1!!!
Enter message for server:time
Received from server:Time is 20:5:5
Enter message for server:stop

user@user-PC /cygdrive/c/nplab
$
```



```
/cygdrive/c/nplab  
user@user-PC ~  
$ cd /cygdrive/c/nplab  
user@user-PC /cygdrive/c/nplab  
$ gcc -o cc2 cclient2.c  
user@user-PC /cygdrive/c/nplab  
$ ./cc2  
Client 2!!!  
Enter message for server:time  
Received from server:Time is 20:5:21  
Enter message for server:stop  
user@user-PC /cygdrive/c/nplab  
$
```

Experiment No. 10

SMTP USING TCP

AIM:

THEORETICAL BACKGROUND:

ALGORITHM:

Server:

1. START
2. Create a socket and bind it.
3. Listen for incoming connection requests from clients.
4. Upon accepting request, create a new socket descriptor for communication.
5. Receive to,from,subject and body of the mail from client.
6. Extract the domain name from to and from mail address and display it.
7. STOP

Client:

1. START
2. Create a socket and bind it.
3. Initiate connection with server
4. When established , send to,from,subject and body of the mail to the server
5. STOP

PROGRAM:

SERVER:

```
#include<stdio.h>
#include<netinet/in.h>
```

```
#include<sys/socket.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
void main()
{
    char to[20],from[20],subject[20],msg[20],todom[20],fromdom[20];
    printf("Server side!!!!\n");
    int sockfd,newsocket,k;
    struct sockaddr_in addr1,addr2;
    addr1.sin_family=AF_INET;
    addr1.sin_addr.s_addr=INADDR_ANY;
    addr1.sin_port=5000;
    int s=sizeof(struct sockaddr_in);
    sockfd=socket(AF_INET,SOCK_STREAM,0);
    bind(sockfd,(struct sockaddr*)&addr1,sizeof(addr1));
    listen(sockfd,5);
    newsocket=accept(sockfd,(struct sockaddr*)&addr2,(&s));
    printf("Connection Established\n");
    printf("Receiving data from client:");
    recv(newsocket,to,sizeof(to),0);
    recv(newsocket,from,sizeof(from),0);
    recv(newsocket,subject,sizeof(subject),0);
    recv(newsocket,msg,sizeof(msg),0);
    k=0;
    for(int i=0;i<strlen(to);i++)
    {
        if(to[i]=='@')
        {
            for(int j=i+1;j<strlen(to);j++)
```

```
{
if(to[j]=='.')
{
todom[k]='\0';
break;
}
todom[k]=to[j];
k++;
}
break;
}
}
k=0;
for(int i=0;i<strlen(from);i++)
{
if(from[i]=='@')
{
for(int j=i+1;j<strlen(from);j++)
{
if(from[j]=='.')
{
fromdom[k]='\0';
break;
}
fromdom[k]=from[j];
k++;
}
break;
}
}
```

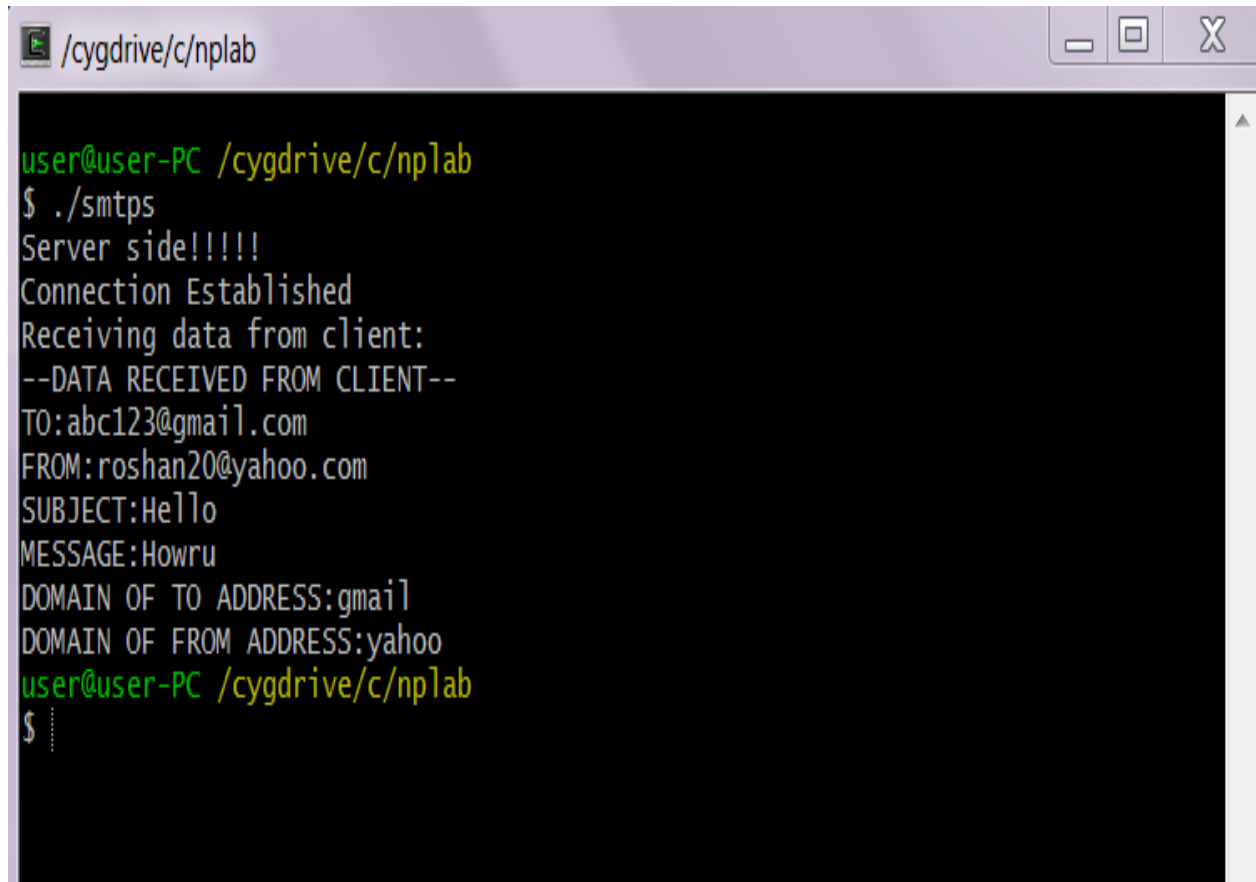
```
printf("\n--DATA RECEIVED FROM CLIENT--\n");
printf("TO:%s\n",to);
printf("FROM:%s\n",from);
printf("SUBJECT:%s\n",subject);
printf("MESSAGE:%s\n",msg);
printf("DOMAIN OF TO ADDRESS:%s\n",todom);
printf("DOMAIN OF FROM ADDRESS:%s",fromdom);
close(newsocket);
close(sockfd);
}
```

CLIENT:

```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
void main()
{
printf("Client side!!!!\n");
char to[20],from[20],subject[20],msg[20];
int sockfd;
sockfd=socket(AF_INET,SOCK_STREAM,0);
struct sockaddr_in addr1;
addr1.sin_family=AF_INET;
addr1.sin_addr.s_addr=INADDR_ANY;
addr1.sin_port=5000;
connect(sockfd,(struct sockaddr*)&addr1,sizeof(addr1));
printf("Enter details:\n");
printf("To:");
```

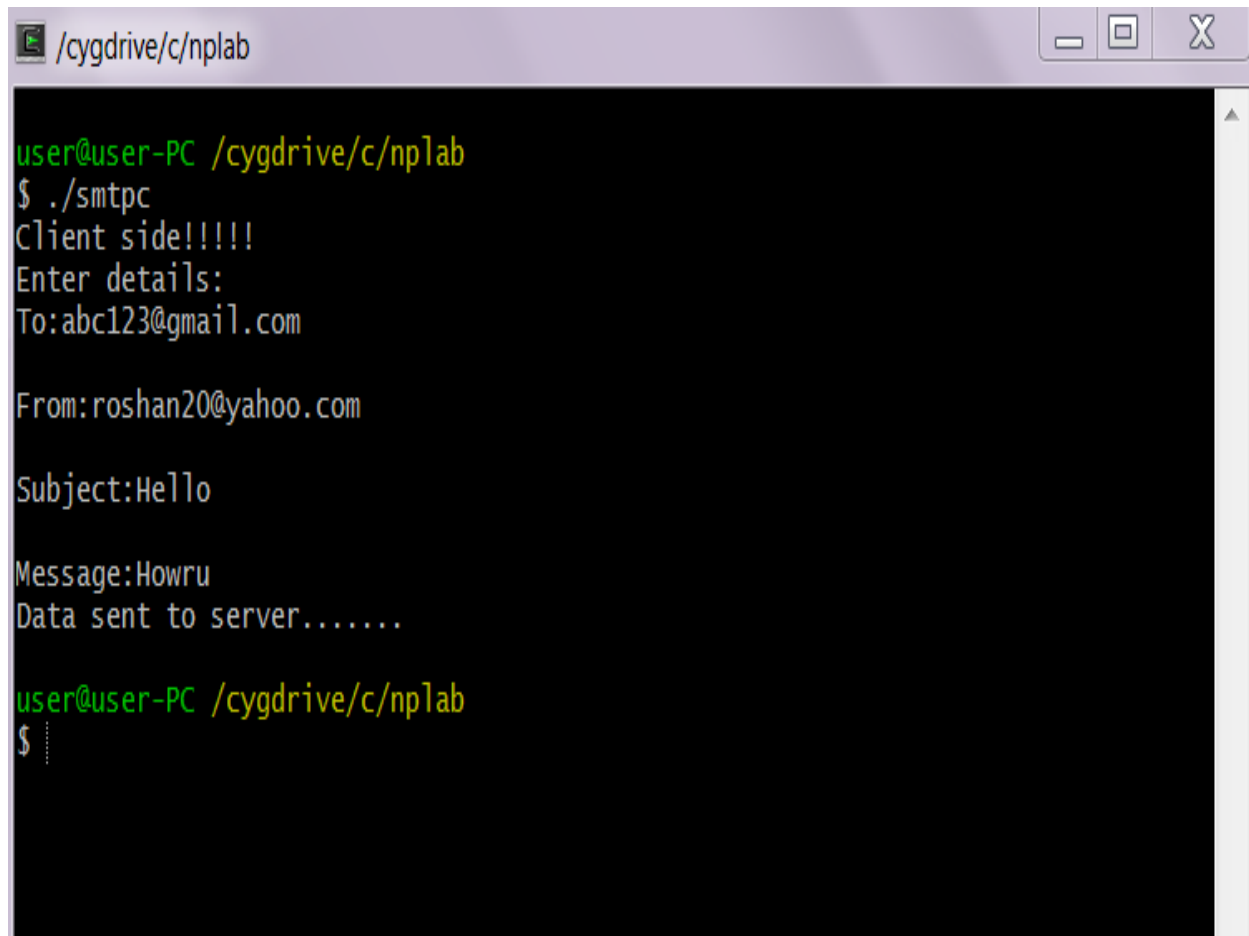
```
scanf("%s",to);
send(sockfd,to,sizeof(to),0);
printf("\nFrom:");
scanf("%s",from);
send(sockfd,from,sizeof(from),0);
printf("\nSubject:");
scanf("%s",subject);
send(sockfd,subject,sizeof(subject),0);
printf("\nMessage:");
scanf("%s",msg);
send(sockfd,msg,sizeof(msg),0);
printf("Data sent to server.....\n");
close(sockfd);
}
```

OUTPUT:



The screenshot shows a terminal window titled "/cygdrive/c/nplab". The prompt is "user@user-PC /cygdrive/c/nplab". The user enters the command "\$./smtps". The program outputs "Server side!!!!", "Connection Established", and "Receiving data from client:". It then displays the received email data in a structured format: "--DATA RECEIVED FROM CLIENT--", "TO:abc123@gmail.com", "FROM:roshan20@yahoo.com", "SUBJECT:Hello", and "MESSAGE:Howru". It also shows domain extraction: "DOMAIN OF TO ADDRESS:gmail" and "DOMAIN OF FROM ADDRESS:yahoo". Finally, the prompt returns to "user@user-PC /cygdrive/c/nplab" with a cursor on a new line.

```
user@user-PC /cygdrive/c/nplab
$ ./smtps
Server side!!!!
Connection Established
Receiving data from client:
--DATA RECEIVED FROM CLIENT--
TO:abc123@gmail.com
FROM:roshan20@yahoo.com
SUBJECT:Hello
MESSAGE:Howru
DOMAIN OF TO ADDRESS:gmail
DOMAIN OF FROM ADDRESS:yahoo
user@user-PC /cygdrive/c/nplab
$
```



The screenshot shows a Cygwin terminal window with the title bar "/cygdrive/c/nplab". The terminal output is as follows:

```
user@user-PC /cygdrive/c/nplab
$ ./smtpc
Client side!!!!
Enter details:
To:abc123@gmail.com

From:roshan20@yahoo.com

Subject:Hello

Message:Howru
Data sent to server.....

user@user-PC /cygdrive/c/nplab
$ .....
```

Experiment No. 11**FTP USING TCP****AIM:**

To implement a file server which will provide the file requested by client. If it exists, contents of file is given. If not, server sends appropriate message to the client. The client must write the file contents to a new file.

THEORETICAL BACKGROUND:**ALGORITHM:****Server**

- START
- Create a socket and bind it.
- Listen for incoming requests from clients.
- Perform next 4 steps for all requests.
- Initiate communication and point creation upon accepting the request.
- Call fork() to serve multiple requests parallelly.
- Accept the file name and search for it.
- If found, transfer the file , else send error message to client.
- STOP

Client

- START
- Create a socket and bind it.
- Initiate connection with server
- When established , send the request for file
- Receive the response
- If the response is an error message, display 'file doesn't exist'. Else go to next step
- Open a new file and write the content into it.
- If user needs another file, repeat previous 4 steps or else terminate.
- STOP

PROGRAM:

CLIENT:

```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
void main()
{
    printf("Client side\n");
    char buffer[50];
    int sockfd;
    sockfd=socket(AF_INET,SOCK_STREAM,0);
    struct sockaddr_in addr1,addr2;
    addr2.sin_family=AF_INET;
    addr2.sin_addr.s_addr=INADDR_ANY;
    addr2.sin_port=3008;
    connect(sockfd,(struct sockaddr *)&addr2,sizeof(addr2));
    printf("Enter the name of the file whose content is to be accessed: ");
    scanf("%s",buffer);
    send(sockfd,buffer,50,0);
    recv(sockfd,buffer,50,0);
    printf("Received from server:%s\n",buffer);
    close(sockfd);
}
```

SERVER:

```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
void main()
{
    printf("Server side\n");
    char buffer[50];
    char buffer2[50];
```

```
int sockfd,newsocket;
char c;
int i;
FILE *f1;
struct sockaddr_in addr1,addr2;
addr1.sin_family=AF_INET;
addr1.sin_addr.s_addr=INADDR_ANY;
addr1.sin_port=3008;
int s=sizeof(struct sockaddr_in);
sockfd=socket(AF_INET,SOCK_STREAM,0);
bind(sockfd,(struct sockaddr *)&addr1,sizeof(addr1));
printf("Connection Established\n");
do
{
    listen(sockfd,5);
    newsocket=accept(sockfd,(struct sockaddr *)&addr2,(&s));
    printf("Receiving file name from client: ");
    i=0;
    recv(newsocket,buffer,sizeof(buffer),0);
    printf("%s",buffer);
    if(access(buffer,F_OK)!=-1)
    {
        f1=fopen(buffer,"r");
        c=fgetc(f1);
        while(!feof(f1))
        {
            buffer2[i]=c;
            i++;
            c=fgetc(f1);
        }
    }
    else
        strcpy(buffer2,"File not found");
    printf("\nSending required information to client");
    send(newsocket,buffer2,sizeof(buffer2),0);
}
while(strcmp(buffer,"stop")!=0);
close(newsocket);
close(sockfd);
```

}

OUTPUT:

Experiment No. 12

LEAKY BUCKET ALGORITHM

AIM:

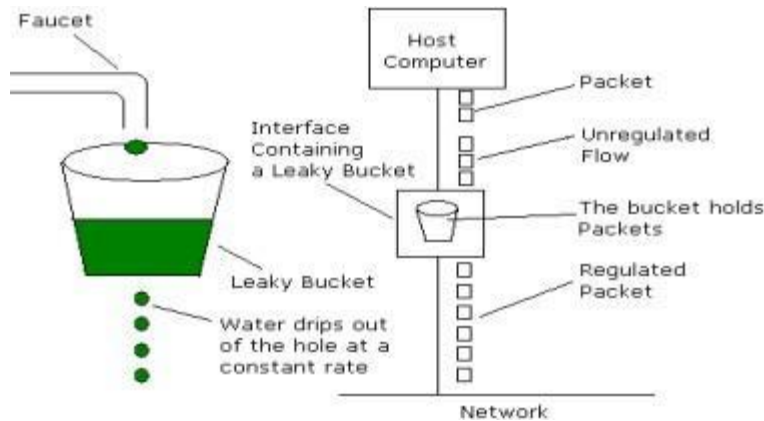
Implementation of leaky bucket Algorithm

THEORETICAL BACKGROUND:

The congesting control algorithms are basically divided into two groups: open loop and closed loop. Open loop solutions attempt to solve the problem by good design, in essence, to make sure it does not occur in the first place. Once the system is up and running, midcourse corrections are not made. Open loop algorithms are further divided into ones that act at source versus ones that act at the destination. In contrast, closed loop solutions are based on the concept of a feedback loop if there is any congestion. Closed loop algorithms are also divided into two sub categories: explicit feedback and implicit feedback. In explicit feedback algorithms, packets are sent back from the point of congestion to warn the source. In implicit algorithm, the source deduces the existence of congestion by making local observation, such as the time needed for acknowledgment to come back. The presence of congestion means that the load is (temporarily) greater than the resources (in part of the system) can handle. For subnets that use virtual circuits internally, these methods can be used at the network layer. Another open loop method to help manage congestion is forcing the packet to be transmitted at a more predictable rate. This approach to congestion management is widely used in ATM networks and is called traffic shaping. The other method is the leaky bucket algorithm.

Each host is connected to the network by an interface containing a leaky bucket, that is, a finite internal queue. If a packet arrives at the queue when it is full, the packet is discarded. In other words, if one or more process are already queued, the new packet is unceremoniously discarded. This arrangement can be built into the hardware interface or simulated by the host operating system. In fact it is nothing other than a single server queuing system with constant service time.

The host is allowed to put one packet per clock tick onto the network. This mechanism turns an uneven flow of packet from the user process inside the host into an even flow of packet onto the network, smoothing out bursts and greatly reducing the chances of congestion.

**ALGORITHM:**

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

PROGRAM:

```
#include<stdio.h> int
main(){
    int incoming, outgoing, buck_size, n, store = 0;
    printf("Enter bucket size, outgoing rate and no of inputs: ");
    scanf("%d %d %d", &buck_size, &outgoing, &n);

    while (n != 0) {
        printf("Enter the incoming packet size : ");
        scanf("%d", &incoming);
        printf("Incoming packet size %d\n", incoming);
        if (incoming <= (buck_size - store)){
```



```
        store += incoming;
        printf("Bucket buffer size %d out of %d\n", store, buck_size);
    } else {
        printf("Dropped %d no of packets\n", incoming - (buck_size - store));
        printf("Bucket buffer size %d out of %d\n", store, buck_size);
        store = buck_size;
    }
    store = store - outgoing;
    printf("After outgoing %d packets left out of %d in buffer\n", store, buck_size);
    n--;
}
}
```

OUTPUT:

Enter bucket size, outgoing rate and no of inputs: 50 100 3 Enter

the incoming packet size : 50

Incoming packet size 50 Bucket buffer
size 50 out of 50

After outgoing -50 packets left out of 50 in buffer Enter
the incoming packet size : 100

Incoming packet size 100 Bucket buffer
size 50 out of 50

After outgoing -50 packets left out of 50 in buffer Enter
the incoming packet size : 20

Incoming packet size 20 Bucket buffer
size -30 out of 50

After outgoing -130 packets left out of 50

Experiment No. 13

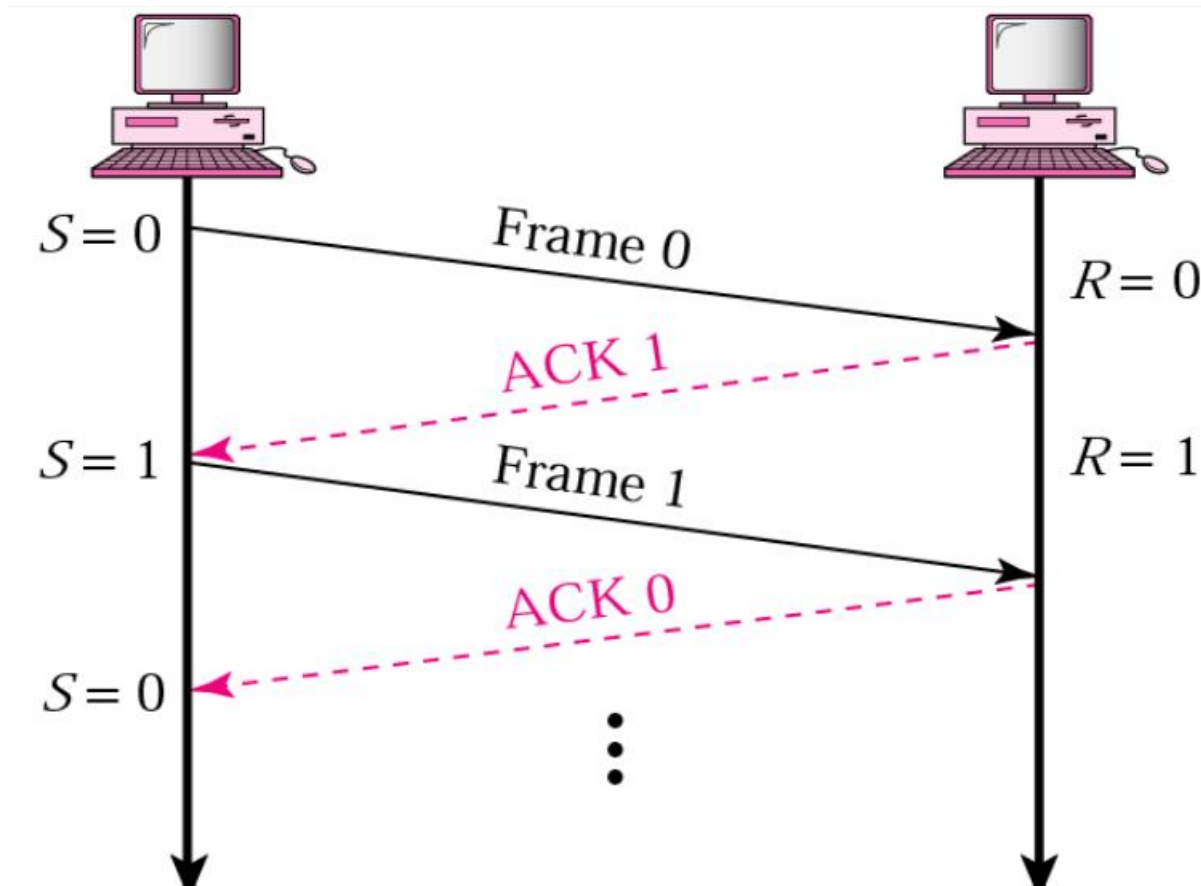
SLIDING WINDOW PROTOCOLS

AIM:

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

THEORETICAL BACKGROUND:

Stop-and-wait Protocol is a flow control protocol used in the data link layer for transmission of data in noiseless channels. Sender keeps on sending messages to the Receiver. In order to prevent the receiver from overwhelming, there is a need to tell the sender to slow down the transmission of frames. We can make use of feedback from the receiver to the sender. Frame 0 sends to receiver, ACK 1 will be sent back to sender; frame 1 goes to receiver, ACK 0 will be back to sender, and so on.



ALGORITHM:

1. Start the program
2. Generate a random number that gives the total number of frames to be transmitted.
3. Transmit the first frame
4. Receive the acknowledgement for the first frame
5. Transmit the next frame
6. Find the remaining frames to be sent.
7. If an acknowledgement is not received for a particular frame, retransmit that frame alone again.
8. Repeat the steps 5 to 7 till the number of remaining frames to be sent becomes zero.
9. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>
void main()
{
int n, x, x1 = 10, x2, i, j, k;
printf("\nEnter no.of frames : ");
scanf("%d", &n);
i = 1;
j = 1;
srand(time(0));
while(n > 0)
```

```
{
printf("\n-> Sending Frame %d", i);
x = rand() % 10;
if(x % 2 == 0)
{
for(x2 = 1; x2 < 2; x2++)
{
printf("\n\tWaiting for acknowledgement...");
sleep(3);
printf("Missing Acknowledgement");
}
printf("\n\n\tSending frame %d again\n", i);
srand(x1++);
x = rand() % 10;
}
printf("\n\tAcknowledgement received for frame %d\n", j);
n -= 1;
i++;
j++;
}
printf("\n");
}
```

OUTPUT:

Enter no.of frames : 12

-> Sending Frame 1

Acknowledgement received for frame 1

-> Sending Frame 2

Waiting for acknowledgement...Missing Acknowledgement

Sending frame 2 again

Acknowledgement received for frame 2

-> Sending Frame 3

Waiting for acknowledgement...Missing Acknowledgement

Sending frame 3 again

Acknowledgement received for frame 3

-> Sending Frame 4

Waiting for acknowledgement...Missing Acknowledgement

Sending frame 4 again

Acknowledgement received for frame 4

-> Sending Frame 5

Waiting for acknowledgement...Missing Acknowledgement

Sending frame 5 again

Acknowledgement received for frame 5

-> Sending Frame 6

Acknowledgement received for frame 6

-> Sending Frame 7

Acknowledgement received for frame 7

-> Sending Frame 8

Acknowledgement received for frame 8

-> Sending Frame 9

Waiting for acknowledgement...Missing Acknowledgement

Sending frame 9 again

Acknowledgement received for frame 9

-> Sending Frame 10

Waiting for acknowledgement...Missing Acknowledgement

Sending frame 10 again

Acknowledgement received for frame 10

-> Sending Frame 11

Waiting for acknowledgement...Missing Acknowledgement

Sending frame 11 again

Acknowledgement received for frame 11

-> Sending Frame 12

Acknowledgement received for frame 12

Experiment No. 14

SELECTIVE REPEAT PROTOCOL

AIM:

Implementation of Selective Repeat Protocol.

THEORETICAL BACKGROUND:

Selective repeat protocol, also called Selective Repeat ARQ (Automatic Repeat reQuest), is a data link layer protocol that uses sliding window method for reliable delivery of data frames. Here, only the erroneous or lost frames are retransmitted, while the good frames are received and buffered.

It uses two windows of equal size: a sending window that stores the frames to be sent and a receiving window that stores the frames received by the receiver. The size is half the maximum sequence number of the frame. For example, if the sequence number is from 0 – 15, the window size will be 8.

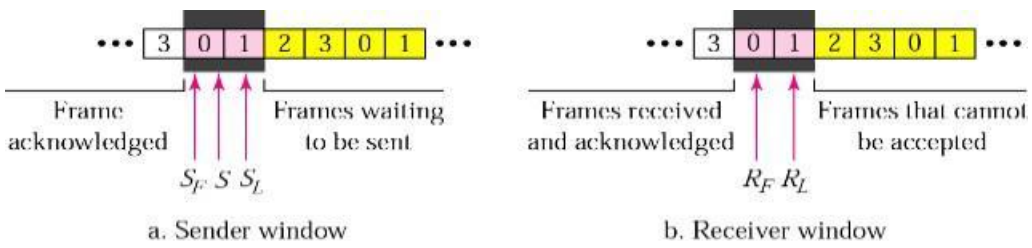
Working Principle

Selective Repeat protocol provides for sending multiple frames depending upon the availability of frames in the sending window, even if it does not receive acknowledgement for any frame in the interim. The maximum number of frames that can be sent depends upon the size of the sending window.

The receiver records the sequence number of the earliest incorrect or un-received frame. It then fills the receiving window with the subsequent frames that it has received. It sends the sequence number of the missing frame along with every acknowledgement frame.

The sender continues to send frames that are in its sending window. Once, it has sent all the frames in the window, it retransmits the frame whose sequence number is given by the acknowledgements. It then continues sending the other frames.

The control variables in Selective Repeat ARQ are same as in Go-Back-N ARQ: SF, SL and S. But the sender sliding window size changed into 2^{m-1} . Receiver sliding window has 2 control variables, RF and RL.



Selective Repeat ARQ receiver slide window [1]

Sender-site Selective Repeat algorithm

```

1  $S_w = 2^{W-1}$ ;
2  $S_f = 0$ ;
3  $S_n = 0$ ;
4
5 while (true) //Repeat forever
6 {
7   WaitForEvent();
8   if(Event(RequestToSend)) //There is a packet to send
9   {
10    if( $S_n - S_f \geq S_w$ ) //If window is full
11      Sleep();
12    GetData();
13    MakeFrame( $S_n$ );
14    StoreFrame( $S_n$ );
15    SendFrame( $S_n$ );
16     $S_n = S_n + 1$ ;
17    StartTimer( $S_n$ );
18  }
19 }

```

```

20 if(Event(ArrivalNotification)) //ACK arrives
21 {
22   Receive(frame); //Receive ACK or NAK
23   if(corrupted(frame))
24     Sleep();
25   if (FrameType == NAK)
26     if (nakNo between  $S_f$  and  $S_n$ )
27     {
28       resend(nakNo);
29       StartTimer(nakNo);
30     }
31   if (FrameType == ACK)
32     if (ackNo between  $S_f$  and  $S_n$ )
33     {
34       while( $s_f < \text{ackNo}$ )
35       {
36         Purge( $s_f$ );
37         StopTimer( $s_f$ );
38          $S_f = S_f + 1$ ;
39       }
40     }
41 }
42
43 if(Event(TimeOut(t))) //The timer expires
44 {
45   StartTimer(t);
46   SendFrame(t);
47 }
48 }

```

Receiver-site Selective Repeat algorithm

```

1  $R_n = 0$ ;
2 NakSent = false;
3 AckNeeded = false;
4 Repeat(for all slots)
5   Marked(slot) = false;
6
7 while (true) //Repeat forever
8 {
9   WaitForEvent();
10
11   if(Event(ArrivalNotification)) //Data frame arrives
12   {
13     Receive(Frame);
14     if(corrupted(Frame) && (NOT NakSent))
15     {
16       SendNAK( $R_n$ );
17       NakSent = true;
18       Sleep();
19     }
20     if( $\text{seqNo} < R_n$ ) && (NOT NakSent)
21     {
22       SendNAK( $R_n$ );

```

```

23     NakSent = true;
24     if (( $\text{seqNo}$  in window) && (!Marked(seqNo)))
25     {
26       StoreFrame(seqNo);
27       Marked(seqNo) = true;
28       while(Marked( $R_n$ ))
29       {
30         DeliverData( $R_n$ );
31         Purge( $R_n$ );
32          $R_n = R_n + 1$ ;
33         AckNeeded = true;
34       }
35       if(AckNeeded);
36       {
37         SendAck( $R_n$ );
38         AckNeeded = false;
39         NakSent = false;
40       }
41     }
42   }
43 }
44 }

```

ALGORITHM:

1. Start the program
2. Generate a random number that gives the total number of frames to be transmitted.
3. Read window size
3. Transmit the window size number of frames
4. Receive the acknowledgement for those frames
5. Transmit the next window size number of frames
6. Find the remaining frames to be sent.
7. If an acknowledgement is not received for a particular frame, retransmit that frame alone again.
8. Repeat the steps 5 to 7 till the number of remaining frames to be sent becomes zero.
9. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>
void main()
{
int n, ws, flag = 0, k,i = 1, z = 0, tt = 0, x;
int t[100] = {0};
printf("Enter number of frames :");
scanf("%d", &n);
printf("\nEnter window size :");
scanf("%d", &ws);
```

```
srand(time(0));
while(i <= n)
{
    z = 0;
    flag = 1;
    for(k = i; k < i + ws && k <= n; k++)
    {
        if(t[k] == 0)
        {
            printf("\nSending frame %d", k);
            tt++;
        }
    }
    for(k = i; k < i+ws && k <= n; k++)
    {
        if(t[k] == 1)
        {
            if(flag)
            z = z + 1;
            continue;
        }
        x = rand() % 2;
        if(x == 0)
        {
            printf("\nAcknowledgement of frame %d not received", k);
            printf("\nRetransmitting frame %d", k);
            flag = 0;
        }
        else
        {

```

```
printf("\nAcknowledgement of frame %d received", k);
t[k] = 1;
if(flag == 1)
z = z + 1;
}
}
i = z + i;
}
printf("\n");
}
```

OUTPUT:

Enter number of frames :12

Enter window size :3

Sending frame 1

Sending frame 2

Sending frame 3

Acknowledgement of frame 1 received

Acknowledgement of frame 2 received

Acknowledgement of frame 3 received

Sending frame 4

Sending frame 5

Sending frame 6

Acknowledgement of frame 4 received

Acknowledgement of frame 5 received

Acknowledgement of frame 6 not received

Retransmitting frame 6

Sending frame 6

Sending frame 7

Sending frame 8

Acknowledgement of frame 6 received

Acknowledgement of frame 7 received

Acknowledgement of frame 8 received

Sending frame 9

Sending frame 10

Sending frame 11

Acknowledgement of frame 9 not received

Retransmitting frame 9

Acknowledgement of frame 10 received

Acknowledgement of frame 11 not received

Retransmitting frame 11

Sending frame 9

Sending frame 11

Acknowledgement of frame 9 not received

Retransmitting frame 9

Acknowledgement of frame 11 received

Sending frame 9

Acknowledgement of frame 9 received

Sending frame 12

Acknowledgement of frame 12 received

Experiment No. 15**GO BACK N****AIM:****THEORETICAL BACKGROUND:****ALGORITHM:**

1. Start the program
2. Generate a random number that gives the total number of frames to be transmitted.
3. Read window size
3. Transmit the window size number of frames
4. Receive the acknowledgement for those frames
5. Transmit the next window size number of frames
6. Find the remaining frames to be sent.
7. If an acknowledgement is not received for a particular frame, retransmit that frame onwards again.
8. Repeat the steps 5 to 7 till the number of remaining frames to be sent becomes zero.
9. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>
void main()
```

```
{
int n, nf, tt = 0, i, j, k;
srand(time(0));
printf("\nEnter no. of frames : ");
scanf("%d", &nf);
printf("Enter window size : ");
scanf("%d", &n);
i = 1;
while(i <= nf)
{
j = 0;
for(k = i; k < i + n && k <= nf; k++)
{
printf("\nSending frame %d", k);
tt++;
}
for(k = i; k < i + n && k <= nf; k++)
{
int l = rand() % 2;
if(!l)
{
printf("\nAcknowledgement for frame %d received", k);
j++;
}
else
{
printf("\nFrame %d not received", k);
printf("\nRetransmitting Window");
break;
}
}
```

```
}  
i = i + j;  
}  
printf("\n");  
}
```

OUTPUT:

```
Enter no. of frames : 12  
Enter window size : 3  
Sending frame 1  
Sending frame 2  
Sending frame 3  
Acknowledgement for frame 1 received  
Acknowledgement for frame 2 received  
Acknowledgement for frame 3 received  
Sending frame 4  
Sending frame 5  
Sending frame 6  
Frame 4 not received  
Retransmitting Window  
Sending frame 4  
Sending frame 5  
Sending frame 6  
Acknowledgement for frame 4 received  
Acknowledgement for frame 5 received  
Frame 6 not received  
Retransmitting Window  
Sending frame 6  
Sending frame 7
```


Sending frame 8

Acknowledgement for frame 6 received

Acknowledgement for frame 7 received

Frame 8 not received

Retransmitting Window

Sending frame 8

Sending frame 9

Sending frame 10

Acknowledgement for frame 8 received

Frame 9 not received

Retransmitting Window

Sending frame 9

Sending frame 10

Sending frame 11

Acknowledgement for frame 9 received

Acknowledgement for frame 10 received

Frame 11 not received

Retransmitting Window

Sending frame 11

Sending frame 12

Frame 11 not received

Retransmitting Window

Sending frame 11

Sending frame 12

Frame 11 not received

Retransmitting Window

Sending frame 11

Sending frame 12

Acknowledgement for frame 11 received

Frame 12 not received

Retransmitting Window

Sending frame 12

Frame 12 not received

Retransmitting Window

Sending frame 12

Acknowledgement for frame 12 received

Experiment No. 16

FAMILIARIZATION OF NS2

AIM:

THEORETICAL BACKGROUND:

NS2 stands for Network Simulator Version 2. It is an open-source event-driven simulator designed specifically for research in computer communication networks. Simulation of wired as well as wireless network functions and protocols (e.g., routing algorithms, TCP, UDP) can be done using NS2. In general, NS2 provides users with a way of specifying such network protocols and simulating their corresponding behaviors.

Some features of NS2 are

- It is a discrete event simulator for networking research.
- It provides substantial support to simulate bunch of protocols like TCP, FTP, UDP, https and DSR.
- It simulates wired and wireless network.
- It is primarily Unix based.
- Uses TCL as its scripting language.
- Otcl: Object oriented support
- Tclcl: C++ and otcl linkage
- Discrete event scheduler

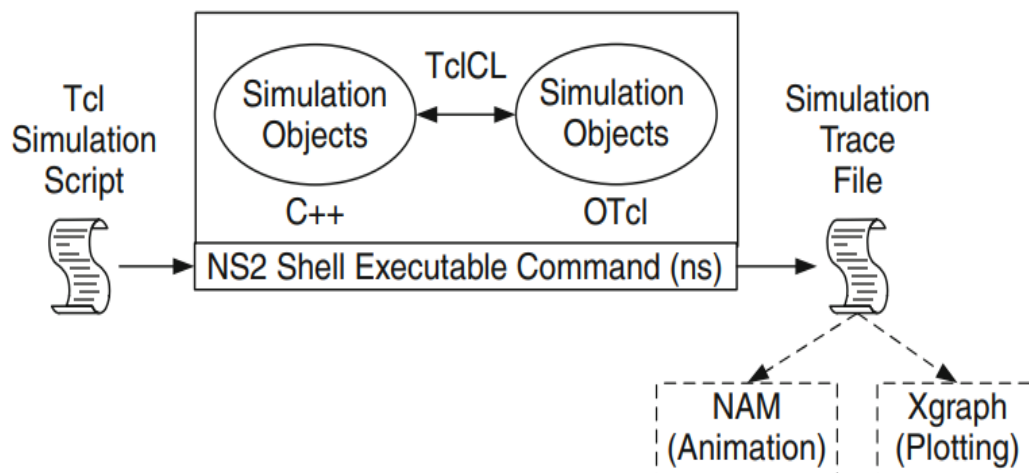


Fig 1: Basic Architecture of NS2

NS2 provides users with an executable command “ns” which takes one input argument, the name of a Tcl simulation scripting file. In most cases, a simulation trace file is created and is

used to plot graph and/or to create animation. Figure 1 shows the basic architecture of NS2 consists of two key languages: CCC and Object-oriented Tool Command Language (OTcl). While the CCC defines the internal mechanism (i.e., a backend) of the simulation, the OTcl sets up simulation by assembling and configuring the objects as well as scheduling discrete events (i.e., a frontend). The CCC and the OTcl are linked together using TclCL. Mapped to a CCC object, variables in the OTcl domains are sometimes referred to as handles. Conceptually, a handle is just a string (e.g., “_o10”) in the OTcl domain and does not contain any functionality. Instead, the functionality (e.g., receiving a packet) is defined in the mapped CCC object (e.g., of class Connector). In the OTcl domain, a handle acts as a frontend which interacts with users and other OTcl objects. It may define its own procedures and variables to facilitate the interaction. Note that the member procedures and variables in the OTcl domain are called instance procedures (instprocs) and instance variables (instvars), respectively.

Tcl scripting

Tcl is a general purpose scripting language. [Interpreter]

- Tcl runs on most of the platforms such as Unix, Windows, and Mac.
- The strength of Tcl is its simplicity.
- It is not necessary to declare a data type for variable prior to the usage.

Basics of TCL

Syntax: command arg1 arg2 arg3

Hello World!

puts stdout{Hello, World!} Hello, World!

Variables Command Substitution

set a 5 set len [string length foobar]

set b \$a set len [expr [string length foobar] + 9]

Wired TCL Script Components

Create the event scheduler

Open new files & turn on the tracing

Create the nodes

Setup the links

Configure the traffic type (e.g., TCP, UDP, etc)

Set the time of traffic generation (e.g., CBR, FTP)

Terminate the simulation

Initialization and Termination of TCL Script in NS-2

An ns simulation starts with the command

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

Which is thus the first line in the tcl script. This line declares a new variable as using the set command,

you can call this variable as you wish, In general people declares it as ns because it is an instance of

the Simulator class, so an object the code[new Simulator] is indeed the installation of the class Simulator using the reserved word new.

In order to have output files with data on the simulation (trace files) or files used for visualization

(nam files), we need to create the files using —open command:

#Open the Trace file

```
set tracefile1 [open out.tr w]
$ns trace-all $tracefile1
```

#Open the NAM trace file

```
set namfile [open out.nam w]
$ns namtrace-all $namfile
```

The above creates a dta trace file called out.tr and a nam visualization trace file called out.nam. Within the tcl script, these files are not called explicitly by their names, but instead by pointers that are

declared above and called —tracefile1 and —namfile respectively. Remark that they begins with a #

symbol. The second line open the file —out.tr to be used for writing, declared with the letter —w. The

third line uses a simulator method called trace-all that have as parameter the name of the file where the

traces will go.

Define a “finish” procedure

```
Proc finish { } {
global ns tracefile1 namfile
$ns flush-trace
Close $tracefile1
Close $namfile
Exec nam out.nam &
Exit 0
}
```

Definition of a network of links and nodes

The way to define a node is

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

Once we define several nodes, we can define the links that connect them. An example of a definition

of a link is:

```
$ns duplex-link $n0 $n2 10Mb 10ms DropTail
```

Which means that \$n0 and \$n2 are connected using a bi-directional link that has 10ms of propagation

delay and a capacity of 10Mb per sec for each direction.

To define a directional link instead of a bi-directional one, we should replace —duplex-link by —simplex-link.

In ns, an output queue of a node is implemented as a part of each link whose input is that node. We

should also define the buffer capacity of the queue related to each link. An example would be:

```
#set Queue Size of link (n0-n2) to 20
$ns queue-limit $n0 $n2 20
```

FTP over TCP

TCP is a dynamic reliable congestion control protocol. It uses Acknowledgements created by the

destination to know whether packets are well received.

There are number variants of the TCP protocol, such as Tahoe, Reno, NewReno, Vegas. The type of

agent appears in the first line:

```
set tcp [new Agent/TCP]
```

The command \$ns attach-agent \$n0 \$tcp defines the source node of the tcp connection.

The command set sink [new Agent /TCPSink] Defines the behavior of the destination node of TCP

and assigns to it a pointer called sink.

Setup a UDP connection

```
set udp [new Agent/UDP]
```

```
$ns attach-agent $n1 $udp
```

```
set null [new Agent/Null]
```

```
$ns attach-agent $n5 $null
```

```
$ns connect $udp
```

To install NS2 in Ubuntu

Step 1: Open terminal

Step 2: `sudo apt-get install -y nam`

Press enter and continue installation of nam

Step 3: `sudo apt-get install -y ns2`

Press enter and continue installation of ns2

ALGORITHM:

PROGRAM:

This network consists of 4 nodes (n0, n1, n2, n3) as shown in above figure. The duplex links between n0 and n2, and n1 and n2 have 2 Mbps of bandwidth and 10 ms of delay. The duplex link between n2 and n3 has 1.7 Mbps of bandwidth and 20 ms of delay. Each node uses a DropTail queue, of which the maximum size is 10. A "tcp" agent is attached to n0, and a connection is established to a tcp "sink" agent attached to n3. As default, the maximum size of a packet that a "tcp" agent can generate is 1KByte. A tcp "sink" agent generates and sends ACK packets to the sender (tcp agent) and frees the received packets. A "udp" agent that is attached to n1 is connected to a "null" agent attached to n3. A "null" agent just frees the packets received. A "ftp" and a "cbr" traffic generator are attached to "tcp" and "udp" agents respectively, and the "cbr" is configured to generate 1 KByte packets at the rate of 1 Mbps. The "cbr" is set to start at 0.1 sec and stop at 4.5 sec, and "ftp" is set to start at 1.0 sec and stop at 4.0 sec.

```
#Create a simulator object
```

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

```
#Define different colors for data flows (for NAM)
```

```
$ns color 1 Blue
```

```
$ns color 2 Red
```

```
#Open the NAM trace file
```

```
set nf [open out.nam w]
```

```
$ns namtrace-all $nf
```

```
#Define a 'finish' procedure
proc finish {} {
    global ns nf
        $ns flush-trace
        #Close the NAM trace file
    close $nf
        #Execute NAM on the trace file
    exec nam out.nam &
    exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns duplex-link $n2 $n3 1.7Mb 20ms DropTail

#Set Queue Size of link (n2-n3) to 10
$ns queue-limit $n2 $n3 10

#Give node position (for NAM)
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right

#Monitor the queue for link (n2-n3). (for NAM)
$ns duplex-link-op $n2 $n3 queuePos 0.5

#Setup a TCP connection
set tcp [new Agent/TCP]
$tcp set class_ 2
$ns attach-agent $n0 $tcp
```



```
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink
$ns connect $tcp $sink
$tcp set fid_ 1
```

```
#Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP
```

```
#Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $null
$ns connect $udp $null
$udp set fid_ 2
```

```
#Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 1mb
$cbr set random_ false
```

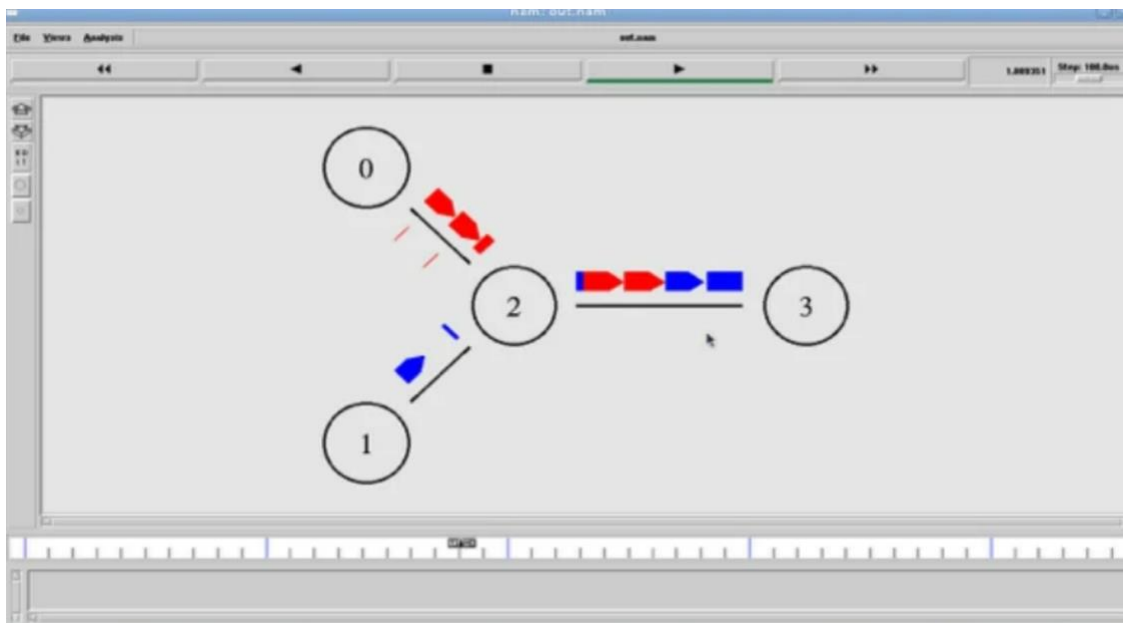
```
#Schedule events for the CBR and FTP agents
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 4.0 "$ftp stop"
$ns at 4.5 "$cbr stop"
```

```
#Detach tcp and sink agents (not really necessary)
$ns at 4.5 "$ns detach-agent $n0 $tcp ; $ns detach-agent $n3 $sink"
```

```
#Call the finish procedure after 5 seconds of simulation time
$ns at 5.0 "finish"
```

```
#Print CBR packet size and interval  
puts "CBR packet size = [$cbr set packet_size_]"  
puts "CBR interval = [$cbr set interval_]"
```

```
#Run the simulation  
$ns run
```

OUTPUT:**Experiment CO mapping**

Experiment Title	CO1	CO2	CO3	CO4	CO5	CO6	CO7	CO8	CO9