DAYANANDA SAGAR UNIVERSITY

Vision

To be a Centre of excellence in education, research & training, innovation & entrepreneurship and to produce citizens with exceptional leadership qualities to serve national and global needs.

Mission

To achieve our objectives in an environment that enhances creativity, innovation and scholarly pursuits while adhering to our vision.

Values

The values that drive DSU and support its vision:

The Pursuit of Excellence

- A commitment to strive continuously to improve ourselves and our systems with the aim of becoming the best in our field.

Fairness

- A commitment to objectivity and impartiality, to earn the trust and respect of society.

Leadership

- A commitment to lead responsively and creatively in educational and research processes.

Integrity and Transparency

- A commitment to be ethical, sincere and transparent in all activities and to treat all individuals with dignity and respect.

Dayananda Sagar University

Laboratory Certificate

*This is to certify that Mr./Ms.*____ *bearing University Seat umber (USN))* ______________________________ *has satisfactorily completed the NETWORKS AND UNIX SYSTEMS PROGRAMMING LAB(16CS374) in above subject prescribed by the University for the* ____________ *semester B.Tech.* _________________*branch of this university during the academic year 20*__________*20*__________

Date: ___________________

____
Signature of the Faculty in charge

Marks	
Maximum	**Obtained**

____
Signature of the Chairman

COMPUTER SCIENCE AND ENGINEERING

STUDENT PERFORMANCE REPORT

Sl No	Date	Particulars	Marks	Sign
1		a) Design and implement a C program to copy a file using command line arguments passed to a user defined function. b) Write a C/C++ POSIX compliant program to check the following limits on the system.		
2		Write a C/C++ POSIX Compliant program that prints POSIX defined configuration options using feature test macros.		
3		Write C Program to demonstrate the inter process communication using FIFO.		
4		Write a C program to add and remove user written modules to or from the Linux kernel.		
5		Write a C Program to illustrate the race condition.		
6		a)Write a C Program to implement mv command using link and unlink APIs. b) Write a C program to implement System function.		
7		Write a C/Java Program to implement IPC using connection oriented (TCP) sockets		
8		Write a C/Java Program to implement IPC using connectionless (UDP) sockets		
9		Simulate three nodes point to point network with duplex links using NS2		
10		Simulate transmission of ping messages over the network topology of 6 nodes and find the number of packets dropped using NS2.		
11		Simulate a four node point-to-point network, and connect the links as follows: n0-n2,n1-n2 and n2-n3. Apply TCP agent between n0-n3 and UDP between n1-n3. Apply relevant applications over TCP and UDP agents. Change the parameters (e.g. Bandwidth and Queue size) and determine the number of packets sent by TCP and UDP.		
12		Simulate an Ethernet LAN using N-nodes (6-10), determine total number of packets dropped due to congestion.		

Internals	**Lab manual with programs and Mini Project**	**Total Marks**
(20)	(20)	(40)

Signature of the Student **Signature of the Staff**

1a.Design and implement a C program to copy a file using command line arguments passed to a user defined function.

Objective:
Apply open, creat, read and write system calls in dealing with files in UNIX.

Description:
Open system call is used to open an already created file. Creat system call is used to create a file. Read and write system calls are used to read from and write to a file respectively.

Algorithm:
```
Step 1: Open the source file in read mode.
Step 2: Create the destination file in write mode.
Step 3: Copy the source file to destination file by reading the characters from the source file and
          writing them to destination file.
Step 4: Check the destination file.
```

Program:

```
#include <fcntl.h>
char buffer[2048];
void copy(int old, int new)
{
        int count;
         while ((count= read(old, buffer, sizeof(buffer))) > 0)
                write(new, buffer, count);
}
int main(int argc, char *argv[])
{
        int fdold, fdnew;
        if (argc!=3)
        {
              printf("\n Need 2 arguments for copy program: \n");
              exit(1);
        }
        fdold= open(argv[1], O_RDONLY);
         if (fdold==-1)
         {
                printf("\n Cannot open file %s", argv[1]);
                exit(1);
         }
         fdnew =  creat(argv[2], 0666);
         if (fdnew==-1)
         {
                 printf("\nCannot create file %", argv[2]);
                 exit(1);
}
copy(fold,fnew);
return 0;
}
```

Output:

1b.Write a C/C++ POSIX compliant program to check the following limits on the system:
(i) Maximum number of characters allowed in a path name
(ii) Maximum number of child processes that may be
created (iii) Maximum number of characters allowed in a
file name
(iv)Maximum number of timers that can be used simultaneously by a process
(v) Maximum number of files that can be opened simultaneously by a process

Objective:
To know the limits on the system confirming to POSIX standard interface.

Description:
Limits: There are three types of limits. They are,
1. Compile-time limits (headers).

2. Runtime limits those are not associated with a file or directory (the sysconf function).

3. Runtime limits those are associated with a file or directory (the pathconf and fpathconf functions).

Sysconf, pathconf, and fpathconf functions: The runtime limits are obtained by calling one of the following three functions.

```
#include long sysconf(int name);
 long pathconf(const char *pathname, int name);
 long fpathconf(int filedes, int name);
All three return: corresponding value if OK, 1 on error
```

Algorithm:
```
Step 1: If the syconf and pathconf functions fail then print error messages.
Step 2: Otherwise display the return value as corresponding limit on the system.
Step 3: End
```

Program:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include<stdio.h>
#include<unistd.h>
int main()
{
  int result;
  #ifdef _POSIX_VERSION
          printf("\nSystem confirms to POSIX %d\n",_POSIX_VERSION);
  #endif
  if((result = pathconf("/",_PC_PATH_MAX)) == -1)
       perror("pathconf");
  else
       printf("Maximum path name length is:
  %d\n",result); if((result = sysconf (_SC_CHILD_MAX))
  == -1)
       perror("sysconf");
  else
```

```
        printf("Maximum number of child processes: %d\n",result);

   if((result = pathconf("/",_PC_NAME_MAX)) == -1)
        perror("pathconf");
   else
        printf("Maximum file name length is: %d\n",result);
   if((result = sysconf(_SC_OPEN_MAX)) == -1)
       perror("sysconf");
   else
       printf("Maximum number of files that can be opened simultaneously:
   %d\n",result); if((result = sysconf(_SC_CLK_TCK))== -1)
            perror("sysconf");
   else
            printf("Number of clock ticks per second: %d\n",result);
return 0;
}
```

Output:

2. Write a C/C++ POSIX Compliant program that prints POSIX defined configuration options using feature test macros.

Objective:

Description:

Algorithm:

Program:

Output:

3. Design and implement a C program that demonstrates inter-process communication between two processes using mkfifo, open, read, write and close APIs

Objective:
To demonstrates inter-process communication between two processes using FIFO.

Description:
FIFO file: It is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer. The size of the buffer is fixed to PIPE_BUF. Data in the buffer is accessed in a first-in-first-out manner. The buffer is allocated when the first process opens the FIFO file for read or write. The buffer is discarded when all processes close their references (stream pointers) to the FIFO file. Data stored in a FIFO buffer is temporary.

The prototype of mkfifo is:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(const char *path_name, mode_t mode);
```

The first argument pathname is the pathname(filename) of a FIFO file to be created. The second argument mode specifies the access permission for user, group and others and as well as the S_IFIFO flag to indicate that it is a FIFO file. On success it returns 0 and on failure it returns –1.

open: This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file. The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information.

The prototype of open function is,

```
#include<sys/types.h>
 #include <sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t permission);
```

- If successful, open returns a nonnegative integer representing the open file descriptor.
- If unsuccessful, open returns –1.
- The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.
- If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non-symbolic link file to which it refers.
- The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.
- Generally the access modes are specified in <fcntl.h>. Various access modes are: O_RDONLY - open for reading file only

  O_WRONLY - open for writing file only

  O_RDWR - opens for reading and writing file.

  There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

  O_APPEND - Append data to the end of file.

  O_CREAT - Create the file if it doesn’t exist

  O_EXCL - Generate an error if O_CREAT is also specified and the file already exists.

  O_TRUNC - If file exists discard the file content and set the file size to zero bytes.

O_NONBLOCK - Specify subsequent read or write on the file should be non-blocking.
O_NOCTTY - Specify not to use terminal device file as the calling process control terminal.

read: The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.

The prototype of read function is:

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fdesc, void *buf, size_t nbyte);
```

- If successful, read returns the number of bytes actually read.
- If unsuccessful, read returns –1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how many bytes of data are to be read from the file.
- The size_t data type is defined in the header and should be the same as unsigned int.
- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.

write:

The write system call is used to write data into a file. The write function puts data to a file in the form of fixed block size referred by a given file descriptor

The prototype of write is

```
#include<sys/types.h>
#include<unistd.h>
ssize_t write(int fdesc, const void *buf, size_t size);
```

- If successful, write returns the number of bytes actually written.
- If unsuccessful, write returns –1.
- The first argument, fdesc is an integer that refers to an opened file.
- The second argument, buf is the address of a buffer that contains data to be written.
- The third argument, size specifies how many bytes of data are in the buf argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (size value)

close:

The close system call is used to terminate the connection to a file from a process.

The prototype of the close is

```
#include<unistd.h>
int close(int fdesc);
```

- If successful, close returns 0.
- If unsuccessful, close returns –1.
- The argument fdesc refers to an opened file.
- Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.
- The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

Algorithm:

Step 1: Create the FIFO file (sent as argument to main function) using mkfifo system call.
Step 2: Write the string (last argument to main function) to the FIFO file.
Step 3: Read the string from the FIFO file.
Step 4: Display the string read from the FIFO file.
Step 5: End.

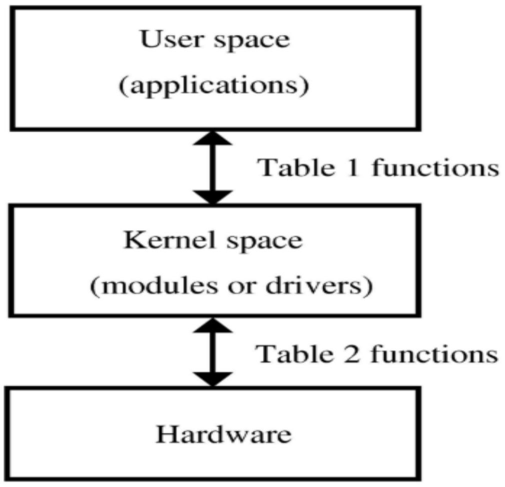
Program:

Output:

4. Write a C program to add and remove user written modules to or from the Linux kernel.

Objective:

Description:
Kernel modules are piece of code that can be loaded and unloaded from kernel on demand. Kernel modules offer an easy way to extend the functionality of the base kernel without having to rebuild or recompile the kernel again. Most of the drivers are implemented as Linux kernel modules. When those drivers are not needed, we can unload only that specific driver, which will reduce the kernel image size. The kernel modules will have a .ko extension. On a normal Linux system, the kernel modules will reside inside /lib/modules/<kernel_version>/kernel/ directory.



When you write kernel modules, it’s important to make the distinction between “user space” and “kernel space”.

- Kernel space. Linux (which is a kernel) manages the machine's hardware in a simple and efficient manner, offering the user a simple and uniform programming interface. In the same way, the kernel, and in particular its device drivers form a bridge or interface between the end-user/programmer and the hardware. Any subroutines or functions forming part of the kernel (modules and device drivers, for example) are considered to be part of kernel space.
- User space. End-user programs, like the UNIX shell or other GUI based applications (kpresenter for example), are part of the user space. Obviously, these applications need to interact with the system's hardware. However, they don’t do so directly, but through the kernel supported functions as shown in the figure above.

Algorithm:
1. Add the header files:
2. Add the description of the module
3. Create a Kernel Module:
4. Compile our Kernel Module
5. Insert the Module:
6. Load a Kernel Module
7. Display the content of the Module:
8. Remove module from kernel

Program:

Output:

5. **Design and implement a C program to illustrate the race condition using parent and child processes.**

Objective:
To demonstrate race condition between two processes.

Description:
A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. It is also defined as; an execution ordering of concurrent flows that results in undesired behavior is called a race condition-a software defect and frequent source of vulnerabilities.
Race condition is possible in runtime environments, including operating systems that must control access to shared resources, especially through process scheduling.

Avoid Race Condition:
If a process wants to wait for a child to terminate, it must call one of the wait functions. If a process wants to wait for its parent to terminate, a loop of the following form could be used

```
while( getppid() != 1 )
 sleep(1);
```

The problem with this type of loop (called polling) is that it wastes CPU time, since the caller is woken up every second to test the condition. To avoid race conditions and to avoid polling, some form of signaling is required between multi processes.

fork() function:
An existing process can create a new one by calling the fork function.

```
#include<unistd.h>
 pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, 1 on error.

- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent. Example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment.

Algorithm:

Program:

Output:

6 a. Design and implement a C program to imitate the mv command in Unix using link and unlink APIs.

Objective:
To use link and unlink APIs to implement mv command.

Description:
unlink - delete a name and possibly the file it refers to

```
#include <unistd.h>
int unlink(const char *pathname);
```

unlink() deletes a name from the file system. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.
If the name was the last link to a file but any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed.
If the name referred to a symbolic link the link is removed. If the name referred to a socket, fifo or device the name for it is removed but processes which have the object open may continue to use it. On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

link - make a new name for a file

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

link() creates a new link (also known as a hard link) to an existing
file. If *newpath* exists it will *not* be overwritten.
This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the `original'. On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

Hard links, as created by **link**(), cannot span filesystems. Use **symlink**() if this is required.

POSIX.1-2001 says that **link**() should dereference *oldpath* if it is a symbolic link. However, Linux does not do so: if *oldpath* is a symbolic link, then *newpath* is created as a (hard) link to the same symbolic link file (i.e., *newpath* becomes a symbolic link to the same file that *oldpath* refers to). Some other implementations behave in the same manner as Linux.

Hard links and Symbolic Links
link: The link function creates a new hard link for the existing file. The prototype of the link function is
#include<unistd.h>

```
int link(const char *cur_link, const char *new_link);
```

- If successful, the link function returns 0. If unsuccessful, link returns –1.
- The first argument cur_link, is the pathname of existing file.
- The second argument new_link is a new pathname to be assigned to the same file.
- If this call succeeds, the hard link count will be increased by 1.
- The UNIX ln command is implemented using the link API.
-

Symlink:
A symbolic link is created with the symlink function.
The prototype of the symlink function
is #include <unistd.h>

```
int symlink(const char *actualpath , const char *sympath);
```

- If successful, the link function returns 0.
- If unsuccessful, link returns –1.
- A new directory entry, sympathy, is created that points to actualpath.
- It is not required that actualpath exists when the symboliclink is created.
- Actualpath and sympath need not reside in the same file system.

Algorithm:

Program:

Output:

6b. Design and implement a C program to implement the System function.

Objective:
To illustrate system function in UNIX platform.

Description:
When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk. There are six **exec functions:**

```
#include<unistd.h>
 int execl(const char *pathname, const char *arg0,... /* (char *)0 */);
int execv(const char *pathname, char *const argv [ ]);
int execle(const char *pathname, const char *arg0,... /*(char *)0, char *const envp */);
int execve(const char *pathname, char *const argv[ ], char *const envp[]);
 int execlp(const char *filename, const char *arg0, ... /* (char *)0 */);
int execvp(const char *filename, char *const argv [ ]); All six return:
-1 on error, no return on success.
```

The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument.

- If filename contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- The next difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- The final difference is the passing of the environment list to the new program.
- The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

Algorithm:

Program:

Output:

7. Write C/Java programs for both server and client to demonstrate interprocess communication using connection oriented (TCP) sockets. Client requests a file from server. The file received from server should be displayed on the client's terminal. Otherwise an appropriate error message should be printed on client side.

Objective:
To design and implement C/Java programs for client and server using TCP sockets.

Description:
Most interprocess communication uses the *client server model*. These terms refer to the two processes which will be communicating with each other. One of the two processes, the *client*, connects to the other process, the *server*, typically to make a request for information. A good analogy is a person who makes a phone call to another person. Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Notice also that once a connection is established, both sides can send and receive information.
The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a *socket*. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps involved in establishing a socket on the *client* side are as follows:

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

The steps involved in establishing a socket on the *server* side are as follows:

1. Create a socket with the socket() system call
2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the listen() system call
4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
5. Send and receive data

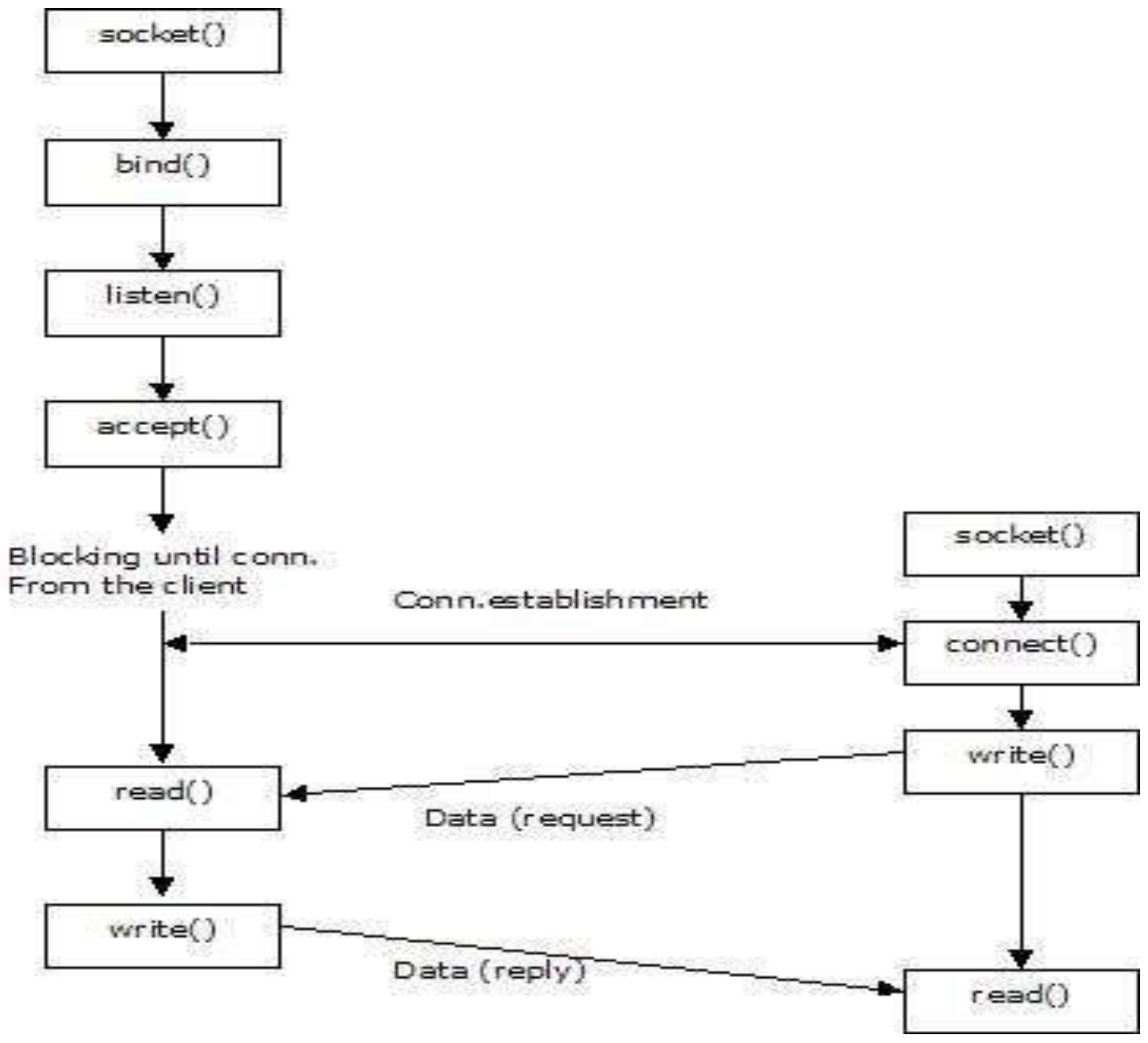
An example of a simple and ubiquitous client-server application would be that of a web-server. A client (Internet Explorer, or Netscape) sends out a request for a particular web page, and the web-server (which may be geographically distant, often in a different continent!) receives and processes this request, and sends out a reply, which in this case, is the web page that was requested. The web page is then displayed on the browser (client).

Sockets (also called Berkeley Sockets, owing to their origin) can simply be defined as end-points for communication. To provide a rather crude visualization, we could imagine the client and server hosts being connected by a pipe through which data-flow takes place, and each end of the pipe can now be regarded as an “end-point”. Thus, a socket provides us with an abstraction, or a logical end point for communication. There are different types of sockets. Stream Sockets, of type SOCK STREAM are used for connection oriented, TCP connections, whereas Datagram Sockets of type SOCK DGRAM are used for UDP based applications. Apart from these two, other socket types like SOCK RAW and SOCK SEQPACKET are also defined.

Socket structure is defined as:

```
struct sockaddr_in {
        short int           sin_family;
        unsigned short int  sin_port;
        struct in_addr      sin_addr;
        unsigned char       sin_zero[8];
};
```

sin_port	Service Port	A 16-bit port number in Network Byte Order.
sin_addr	IP Address	A 32-bit IP address in Network Byte Order.
sin_zero	Not Used	You just set this value to NULL as this is not being used.



socket() system call syntax:

int sd = socket (int domain, int type, int protocol);

The socket() system call creates a socket and returns a socket file descriptor to the socket created. The descriptor is of data type int. Here, domain is the address family specification, type is the socket type and the protocol field is used to specify the protocol to be used with the address family specified. The address family can be one of AF INET (for Internet protocols like TCP, UDP, which is what we are going to use) or AF UNIX (for Unix internal protocols).

The bind() system call is used to specify the association <Local-Address, Local-Port>. It is used to bind either connection oriented or connectionless sockets. The bind() function basically associates a name to an unnamed socket. “Name”, here refers to three components - The address family, the host address, and the port number at which the application will provide its service.

The syntax and arguments taken by the bind system call is given below:

int result = bind(int sd, struct sockaddr *address, int addrlen);

Here, sd is the socket file descriptor returned by the socket() system call before, and name points to the sockaddr structure, and addrlen is the size of the sockaddr structure. Like all other socket system calls, upon success, bind() returns 0. In case of error, bind() returns -1. After creation of the socket, and binding to a local port, the server has to wait on incoming connection requests.

The listen() system call is used to specify the queue or backlog of waiting (or incomplete) connections. The syntax and arguments taken by the listen() system call is given below:

int result = listen(int sd, int backlog);

Here, sd is the socket file descriptor returned by the socket() system call before, and backlog is the number of incoming connections that can be queued. Upon success, listen() returns 0. In case of error, listen() returns -1. After executing the listen() system call, a server waits for incoming connections.

An actual connection setup is completed by a call to accept(). accept() takes the first connection request on the queue, and creates another socket descriptor with the same properties as sd (the socket descriptor returned earlier by the socket() system call). The new socket descriptor handles communications with the new client while the earlier socket descriptor goes back to listening for new connections. In a sense, the accept() system call completes the connection, and at the end of a successful accept(), all elements of the

four tuple (or the five tuple - if you consider “protocol” as one of the elements) of a connection are filled. The “four-tuple” that we talk about here is <Local Addr, Local Port, Remote Addr, Remote Port>. This

combination of fields is used to uniquely identify a flow or a connection. The fifth tuple element can be the protocol field. No two connections can have the same values for all the four (or five) fields of the tuple.

accept() syntax:

int newsd = accept(int sd, void *addr, int *addrlen);

Here, sd is the socket file descriptor returned by the socket() system call before, and addr is a pointer to a structure that receives the address of the connecting entity, and addrlen is the length of that structure. Upon success, accept() returns a socket file descriptor to the new socket created. In case of error, accept() returns -1. In the case of a connection oriented protocol (like TCP/IP), the connect() system call results in the actual connection establishment of a connection between the two hosts. In case of TCP, following this call, the three-way handshake to establish a connection is completed. Note that the client does not necessarily have to bind to a local port in order to call connect(). Clients typically choose ephemeral port numbers for their end of the connection. Servers, on the other hand, have to provide service on well-known (premeditated).

port numbers. connect() syntax:

int result = connect(int sd, struct sockaddr *servaddr, int addrlen);

Here, sd is the socket file descriptor returned by the socket() system call be fore, servaddr is a pointer to the server’s address structure (port number and IP address). addrlen holds the length of this parameter and can be set to sizeof(struct sockaddr). Upon success, connect() returns 0. In case of error, connect() returns -1. After connection establishment, data is exchanged between the server and client using the system calls send(), recv(), sendto() and recvfrom().

The syntax of the system calls are as below:
int nbytes = send(int sd, const void *buf, int len, int flags); int
nbytes = recv(int sd, void *buf, int len, unsigned int flags);
int nbytes = sendto(int sd, const void *buf, int len, unsigned int flags, const struct sockaddr *to, int tolen);
int nbytes = recvfrom(int sd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);
Here, sd is the socket file descriptor returned by the socket() system call before, buf is the buffer to be sent or received, flags is the indicator specifying the way the call is to be made (usually set to 0). sendto() and recvfrom() are used in case of connectionless sockets, and they do the same function as send() and recv() except that they take more arguments (the "to" and "from" addresses - as the socket is connectionless) Upon success, all these calls return the number of bytes written or read. Upon failure, all the above calls return -1. recv() returns 0 if the connection was closed by the remote side. The close() system call is used to close the connection. In some cases, any remaining data that is queued is sent out before the close() is executed.

The close() system call prevents further reads or writes to the socket. close() syntax:
int result = close (int sd);

Algorithm:

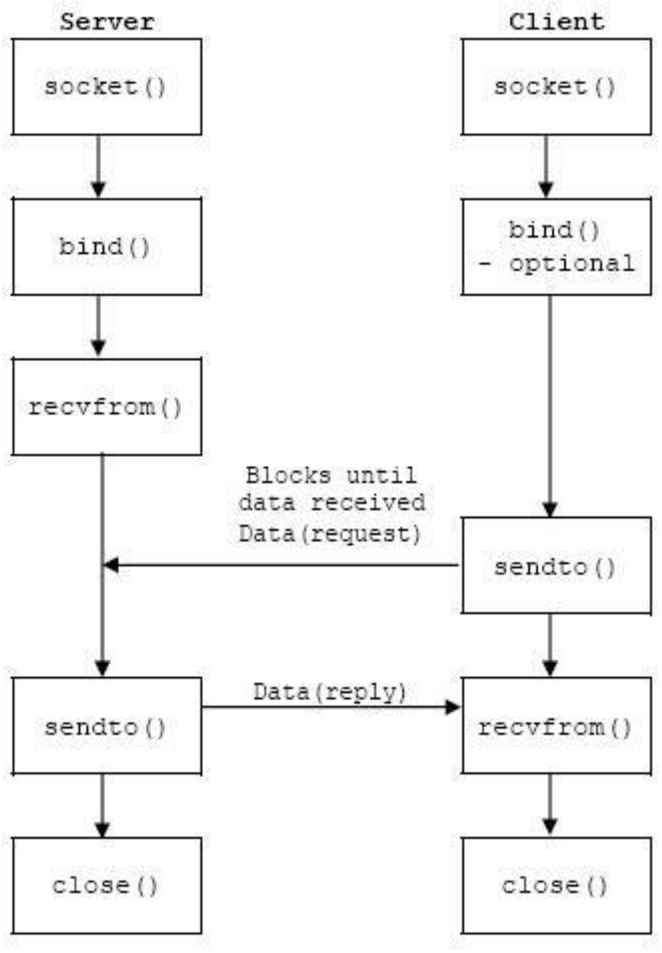
Program:

Output:

8. Write C/Java programs for both server and client to demonstrate inter-process communication using connectionless (UDP) sockets. Client requests a file from server. The file received from server should be displayed on the client's terminal. Otherwise an appropriate error message should be printed on client side.

Objective:
To design and implement C/Java programs for client and server using UDP sockets.

Description:
A TCP connection is like telephone calls where we dial up to connect to the person with whom we want to communicate. The connection remains intact throughout the communication process even if we are not sending any signals. UDP, on the other hand, is more like mail carried through the postal service that assumes they may arrive out of sequence, lost in transit, or duplicate datagrams were detected at the receiving end. Here there are no connecting and accept system calls used in client and server respectively. The system calls sendto and recvfrom are used by client and server respectively. The steps involved are shown in the figure below.



For more details refer to: https://www.cs.rutgers.edu/~pxk/417/notes/sockets/udp.html

Algorithm:

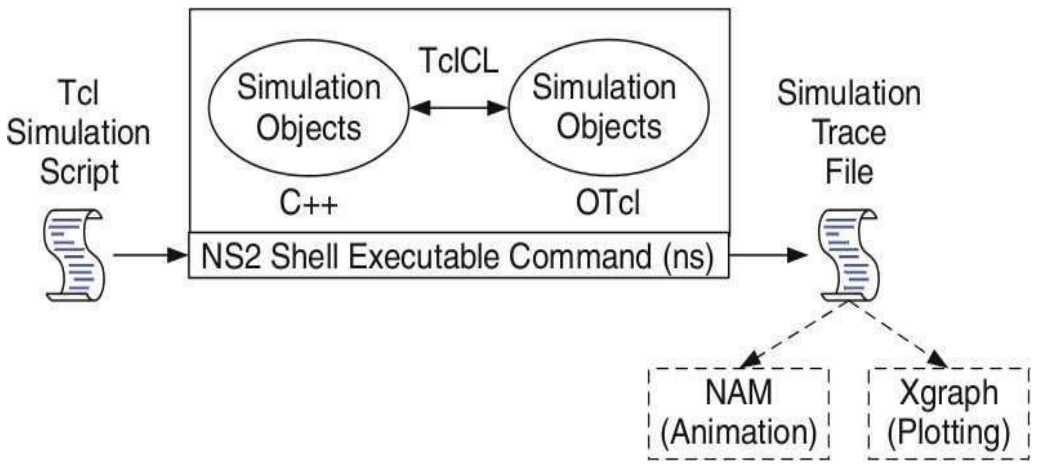
Program:

Output:

INTRODUCTION TO NS-2:

- Widely known as NS2, is simply an event driven simulation tool.
- Useful in studying the dynamic nature of 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.

Basic Architecture of NS2



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]
```

- **Simple Arithmetic**

```
expr 7.2 / 4
```

- **Procedures**

```
proc Diag {a b} {  set c [expr sqrt($a * $a + $b * $b)]
puts —Diagonal of a 3, 4 right triangle is [Diag 3 4]

Output: Diagonal of a 3, 4 right triangle is 5.0
```

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

NS Simulator Preliminaries.

1. Initialization and termination aspects of the ns simulator.
2. Definition of network nodes, links, queues and topology.
3. Definition of agents and of applications.
4. The nam visualization tool.
5. Tracing and random variables.

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:

To set Trace file:

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

To set NAM file

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

The above creates a 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 ―tracefile 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.The last line tells the simulator to record all simulation traces in NAM input format. It also gives the file name that the trace will be written to later by the command $ns flush-trace. In our case, this will be the filepointed at by the pointer ―$namfile, i.e the file ―out.tr. The termination of the program is done using a ―finish procedure.

#Define a finish procedure

The word proc declares a procedure in this case called finish and without arguments. The word global is used to tell that we are using variables declared outside the procedure. The simulator method ―flush-trace" will dump the traces on the respective files. The tcl command ―close" closes the trace files defined before and exec executes the nam program for visualization. The command exit will ends the application and return the number 0 as status to the system. Zero is the default for a clean exit. Other values can be used to say that is a exit because something fails.

At the end of ns program we should call the procedure ―finish‖ and specify at what time the termination should occur. For example,

```
$ns at 125.0 "finish"
```

will be used to call ―finish at time 125sec.Indeed,the at method of the simulator allows us to schedule events explicitly.
The simulation can then begin using the command

```
$ns run
```

Definition of a network of links and nodes
The way to define a node is

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

The node is created which is printed by the variable n0. When we shall refer to that node in the script we shall thus write $n0.
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. The definition of the link then includes the way to handle overflow at that queue. In our case, if the buffer capacity of the output queue is exceeded then the last packet to arrive is dropped. Many alternative options exist, such as the RED (Random Early Discard) mechanism, the FQ (Fair Queuing), the DRR (Deficit Round Robin), the stochastic Fair Queuing (SFQ) and the CBQ (which including a priority and a round-robin scheduler).

Agents and Applications
We need to define routing (sources, destinations) the agents (protocols) the application that use them.
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 $null
  $udp set fid_2
```

#Setup a CBR over UDP connection

```
  set cbr [new Application/Traffic/CBR]
   $cbr attach-agent $udp
   $cbr set packetsize_ 100
    $cbr set rate_ 0.01Mb
    $cbr set random_ false
```

The command $ns attach-agent $n4 $sink defines the destination node.

The command $ns connect $tcp $sink finally makes the TCP connection between the source and destination nodes.

TCP has many parameters with initial fixed defaults values that can be changed if mentioned explicitly. For example, the default TCP packet size has a size of 1000bytes.This can be changed to another value, say 552bytes, using the command

```
$tcp set packetSize_ 552.
```

When we have several flows, we may wish to distinguish them so that we can identify them with different colors in the visualization part. This is done by the command $tcp set fid_ 1 that assigns to the TCP connection a flow identification of ―1.We shall later give the flow identification of ―2‖ to the UDP connection.

Structure of Trace Files

When tracing into an output ASCII file, the trace is organized in 12 fields as follows in fig shown below, The meaning of the fields are:

Event	Time	From Node	To Node	PKT Type	PKT Size	Flags	Fid	Src Addr	Dest Addr	Seq Num	Pkt id

1. The first field is the event type. It is given by one of four possible symbols r, +, -, d which correspond respectively to receive (at the output of the link), enqueued, dequeued and dropped.
2. The second field gives the time at which the event occurs.
3. Gives the input node of the link at which the event occurs.
4. Gives the output node of the link at which the event occurs.
5. Gives the packet type (eg CBR or TCP)
6. Gives the packet size
7. Some flags
8. This is the flow id (fid) of IPv6 that a user can set for each flow at the input OTcl script one can further use this field for analysis purposes; it is also used when specifying stream color for the NAM display.
9. This is the source address given in the form of ―node.port‖.

XGRAPH

The xgraph program draws a graph on an x-display given data read from either data file or from standard input if no files are specified. It can display upto 64 independent data sets using different colors and line styles for each set. It annotates the graph with a title, axis labels, grid lines or tick marks, grid labels and a legend.

Syntax:

```
Xgraph [options] file-name
```

Awk-

Awk is a programmable, pattern-matching, and processing tool available in UNIX. It works equally well with text and numbers.
Awk is not just a command, but a programming language too. In other words, awk utility is a pattern scanning and processing language. It searches one or more files to see if they contain lines that will match specified patterns and then perform associated actions, such as writing the line to the standard output or incrementing a counter each time it finds a match.

Syntax:

```
awk option 'selection_criteria {action}' file(s)
```

Here, selection_criteria filters input and select lines for the action component to act upon. The selection_criteria is enclosed within single quotes and the action within the curly braces. Both the selection_criteria and action forms an awk program.

Example: $ awk „/manager/ {print}‟ emp.lst

Experiment No: 9

THREE NODE POINT TO POINT NETWORK

Aim: Implement three nodes point – to – point network with duplex links between them. Set the queue size, vary the bandwidth and find the number of packets dropped.

```
set ns [new Simulator]                                # Letter S is capital
set nf [open lab1.nam w]                              # open a nam trace file in write mode
$ns namtrace-all $nf                                  # nf nam filename
set tf [open lab1.tr w]                               # tf trace filename
$ns trace-all $tf

proc finish { } {
        global ns nf tf
        $ns flush-trace                               # clears trace file contents
        close $nf
        close $tf
        exec nam lab1.nam &
        exit 0
}
set n0 [$ns node]                                     # creates 3 nodes
set n2 [$ns node]
set n3 [$ns node]

$ns duplex-link $n0 $n2 200Mb 10ms DropTail           # establishing links
$ns duplex-link $n2 $n3 1Mb 1000ms DropTail
$ns queue-limit $n0 $n2 10

set udp0 [new Agent/UDP]                              # attaching transport layer protocols
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]                # attaching application layer protocols
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

set null0 [new Agent/Null]                            # creating sink(destination) node
$ns attach-agent $n3 $null0
$ns connect $udp0 $null0

$ns at 0.1 "$cbr0 start"
$ns at 1.0 "finish"
$ns run
```

AWK file: (Open a new editor using “vi command” and write awk file and save with “.awk” extension)

```
BEGIN {
 c=0;
}
{
  if($1= ="d")
 {c++;
         printf("%s\t%s\n",$5,$1
         1);
  }
 }
END {
 printf("The number of packets dropped is
%d\n",c); }
```

Steps for execution

Open gedit editor and type program. Program name should have the extension “ **.tcl**

[root@localhost ~]# gedit lab1.tcl

Save the program and close the file.

Open gedit editor and type **awk** program. Program name should have the extension “**.awk** ”

[root@localhost ~]# gedit lab1.awk

Save the program and close the file.

Run the simulation program

[root@localhost~]# ns lab1.tcl

Here **“ns”** indicates network simulator. We get the topology shown in the snapshot.

Now press the play button in the simulation window and the simulation will begins.

After simulation is completed run **awk file** to see the output ,

[root@localhost~]# awk –f lab1.awk lab1.tr

To see the trace file contents open the file as ,

[root@localhost~]# gedit lab1.tr

Trace file contains 12 columns:

Event type, Event time, From Node, To Node, Packet Type, Packet Size, Flags (indicated by - -------), Flow ID, Source address, Destination address, Sequence ID, Packet ID

Output:

Experiment No 10: TRANSMISSION OF PING MESSAGE

Aim: Implement transmission of ping messages/trace route over a network topology consisting of 6 nodes and find the number of packets dropped due to congestion.

```
set ns [ new Simulator ] set
nf [ open lab2.nam w ]
$ns namtrace-all $nf
set tf [ open lab2.tr w ]
$ns trace-all $tf

set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

$ns duplex-link $n0 $n4 1005Mb 1ms DropTail
$ns duplex-link $n1 $n4 50Mb 1ms DropTail
$ns duplex-link $n2 $n4 2000Mb 1ms DropTail
$ns duplex-link $n3 $n4 200Mb 1ms DropTail
$ns duplex-link $n4 $n5 1Mb 1ms DropTail

set p1 [new Agent/Ping] # letters A and P should be capital
$ns attach-agent $n0 $p1
$p1 set packetSize_ 50000
$p1 set interval_ 0.0001

set p2 [new Agent/Ping] # letters A and P should be
capital $ns attach-agent $n1 $p2

set p3 [new Agent/Ping] # letters A and P should be capital
$ns attach-agent $n2 $p3
$p3 set packetSize_ 30000
$p3 set interval_ 0.00001

set p4 [new Agent/Ping] # letters A and P should be
capital $ns attach-agent $n3 $p4

set p5 [new Agent/Ping] # letters A and P should be
capital $ns attach-agent $n5 $p5

$ns queue-limit $n0 $n4 5
$ns queue-limit $n2 $n4 3

$ns queue-limit $n4 $n5 2 Agent/Ping
instproc recv {from rtt} {
$self instvar node_
puts "node [$node_ id] received answer from $from with round trip time $rtt msec"
}


# please provide space between $node_ and id. No space between $ and from. No space between
and $ and rtt */
```

```
$ns connect $p1 $p5
$ns connect $p3 $p4

proc finish { } {
global ns nf tf
$ns flush-trace
close $nf
close $tf
exec nam lab2.nam &
exit 0
}
$ns at 0.1 "$p1 send"
$ns at 0.2 "$p1 send"
$ns at 0.3 "$p1 send"
$ns at 0.4 "$p1 send"
$ns at 0.5 "$p1 send"
$ns at 0.6 "$p1 send"
$ns at 0.7 "$p1 send"
$ns at 0.8 "$p1 send"
$ns at 0.9 "$p1 send"
$ns at 1.0 "$p1 send"

$ns at 0.1 "$p3 send"
$ns at 0.2 "$p3 send"
$ns at 0.3 "$p3 send"
$ns at 0.4 "$p3 send"
$ns at 0.5 "$p3 send"
$ns at 0.6 "$p3 send"
$ns at 0.7 "$p3 send"
$ns at 0.8 "$p3 send"
$ns at 0.9 "$p3 send"
$ns at 1.0 "$p3 send"
$ns at 2.0 "finish"
$ns run
```

AWK file: *(Open a new editor using "gedit command" and write awk file and save with ".awk" extension)*

```
BEGIN {
drop=0;
 }
 if($1= ="d" )
  {
   drop++;
   }
   }
   END {
printf("Total number of %s packets dropped due to congestion =%d\n",$5,drop);

}
```

<u>Steps for execution</u>

- *Open gedit editor and type program. Program name should have the extension* ***.tcl [root@localhost ~]# gedit lab2.tcl***
- *Save the program and close the file.*
- *Open gedit editor and type* ***awk*** *program. Program name should have the extension “****.awk*** *”*

  [root@localhost ~]# gedit lab2.awk
- *Save the program and close the file.*
- *Run the simulation program*

  [root@localhost~]# ns lab2.tcl
- *Here* ***“ns”*** *indicates network simulator. We get the topology shown in the snapshot.*
- *Now press the play button in the simulation window and the simulation will begins.*
- *After simulation is completed run* ***awk file*** *to see the output ,*

  [root@localhost~]# awk –f lab2.awk lab2.tr
- *To see the trace file contents open the file as ,*

  [root@localhost~]# gedit lab2.tr}

Output:

Experiment No: 11

Simulate a four node point-to-point network, and connect the links as follows : n0-n2,n1-n2 and n2-n3. Apply TCP agent between n0-n3 and UDP between n1-n3. Apply relevant applications over TCP and UDP agents. Change the parameters (e.g. Bandwith and Queue size) and determine the number of packets sent by TCP and UDP.

Aim:

Program:

AWK File:

Output:

Experiment no 12:

Simulate an Ethernet LAN using N-nodes (6-10), determine the total number of packets dropped due to congestion.

Aim:

Program

Output