MANIPAL ACADEMY
OF HIGHER EDUCATION

INSPIRED BY LIFE

(Deemed-to-be-University under Section 3 of the UGC Act, 1956)

DEPARTMENT OF INFORMATION & COMMUNICATION TECHNOLOGY

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL

CERTIFICATE

This is to certify that Ms./Mr. …………………...…………………………………… Reg. No. …..…………………… Section:……… Roll No: …..…… has satisfactorily completed the lab exercises prescribed for **Computer Network Lab [ICT 3211]** of Third Year **B. Tech. (IT)** Degree at MIT, Manipal, in the academic year 2017-2018.

Date: ……....................................

Signature of the faculty

CONTENTS

Course Objectives

- To write, execute and debug C programs which use Socket API.
- To understand the use of client/server architecture in application development
- To realize the usage of TCP and UDP based sockets and their differences.

Course Outcomes

At the end of this course, students will be able to

- Develop the necessary skills for developing robust and scalable network applications
- Implement next generation protocols required for emerging applications
- Carry out research and development tasks in networking

Evaluation plan

Split up of 60 marks for Regular Lab Evaluation
Execution: 2 Marks Record: 3 Marks Evaluation: 5 Marks
End Semester Lab evaluation: 40 marks (Duration 2 hours)
Program write up: 20 Marks Program execution: 20 Marks

Instructions to the students

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session
2. Be in time and follow the institution dress code
3. Must sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

1. Follow the instructions on the allotted exercises
2. Show the program and results to the instructors on completion of experiments
3. Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Comments should be used to give the statement of the problem.
  - The statements within the program should be properly indented.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- In case a student misses a lab, he/ she must ensure that the experiment is completed before the next evaluation with the permission of the faculty concerned.
- Students missing out the lab for genuine reasons like conference, sports or activities assigned by the Department or Institute will have to take **prior permission** from the HOD to attend **additional lab** (with another batch) and complete it **before** the student goes on leave. The student could be awarded marks for the write up for that day provided he submits it during the **immediate** next lab.

- Students who feel sick should get permission from the HOD for evaluating the lab records. However, attendance will not be given for that lab.
- Students will be evaluated only by the faculty with whom they are registered even though they carry out additional experiments in another batch.
- The presence of the student during the lab end semester exams is mandatory even if the student assumes he has scored enough to pass the examination
- Minimum attendance of 75% is mandatory to write the final exam.
- If the student loses his book, he/she will have to rewrite all the lab details in the lab record.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

The students should NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

LAB NO: 1 **Date:**

INTRODUCTION TO SOCKET PROGRAMMING

Objectives

- To illustrate the significance of socket programming
- To recognize basic socket function calls by performing simple client server operations

Introduction

Computer network is a communication network in which a collection of computers are connected together to facilitate data exchange .The connection between the computers can be wired or wireless. A computer network basically comprises of 5 components as shown in Fig.1.1:

- Sender
- Receiver
- Message
- Transmission medium
- Protocols

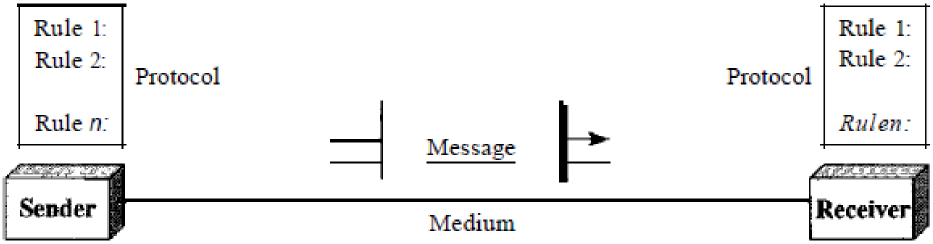


Fig.1.1: Components of data communication

Client Server Architecture

Communication in computer networks follows a client server model as illustrated in Fig. 1.2. Here, a machine (referred as **client**) makes a request to connect to another machine (called as **server**) for providing some service. The services running on the server run on known ports (application identifiers) and the client needs to know the address of the server machine and this port in order to connect to the server. On the other hand, the server does not need to know about the address or the port of the client at the time of connection initiation. The first packet which the client sends as a request to the server contains these information about the client which are further used by the server to send any information. Client **(Active Open)** acts as the active device which makes the first move to establish the connection whereas the server **(Passive Open)** passively waits for such requests from some client.

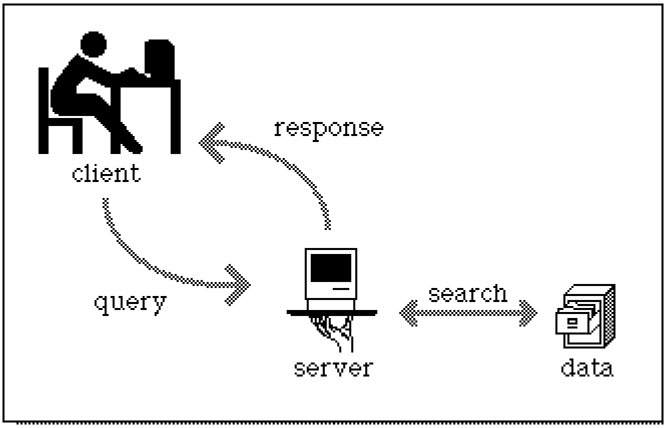


Fig.1.2: Illustration of Client Server Model

Basics of Socket Programming

In UNIX, whenever there is a need for inter process communication within the same machine, we use mechanism like signals or pipes. Similarly, when we desire a communication between two applications possibly running on different machines, we need **sockets**. A network socket is an endpoint for sending or receiving data at a single node in a computer network that supports full duplex transmission.

Sockets are created and used with a set of programming requests or "function calls" sometimes called the sockets Application Programming Interface (API). The most common sockets API is the Berkeley UNIX C interface for sockets which are the interfaces between applications layer and the transport layer that acts as a virtual connection between two processes. Each socket is identified by an address so that processes can connect to them.

Socket Address = IP Address + Port Number

Types of Communication Services

The data transfer between client and server application initiated by socket APIs can be achieved either by using connection oriented or connectionless services.

a. **Connection Oriented Communication :**In connection oriented service a connection has to be established between two devices before starting the communication to allocate the resources needed to aid the data transfer. Then the message transfer takes place until the connection is released. This type of communication is characterized by a high level of reliability in terms of the number and the sequence of bytes. It is analogous to the telephone network.
b. **Connectionless Communication:** In connectionless the data is transferred in one direction from source to destination without checking that destination is still there or not or if it prepared to accept the message. Authentication is not needed in this. It is analogous to the postal service where Packets (letters) are sent at a time to a particular destination.

Based on the two types of communication described above, two kinds of sockets are used:

a. **Stream sockets:** used for connection-oriented communication, when reliability in connection is desired. Protocol used is TCP (Transmission Control Protocol) for data transmission.
b. **Datagram sockets:** used for connectionless communication, when reliability is not as much as an issue compared to the cost of providing that reliability. For e.g. Streaming audio/video is always sent over such sockets so as to diminish network traffic. Protocol used is UDP (User Datagram Protocol) for data transmission.

Socket System Calls for Connection-Oriented Protocol

Steps followed by client to establish the connection:

a. Create a socket
b. Connect the socket to the address of the server
c. Send/Receive data
d. Close the socket connection

Steps followed by server to establish the connection:

1. Create a socket
2. Bind the socket to the port number known to all clients
3. Listen for the connection request
4. Accept connection request. This call typically blocks until a client connects with the server.
5. Send/Receive data
6. Close the socket connection

The sequence of socket function calls to be invoked for a connection oriented client server communication is depicted in Fig. 1.3 and the significance of these function calls is summarized in Table 1.1.

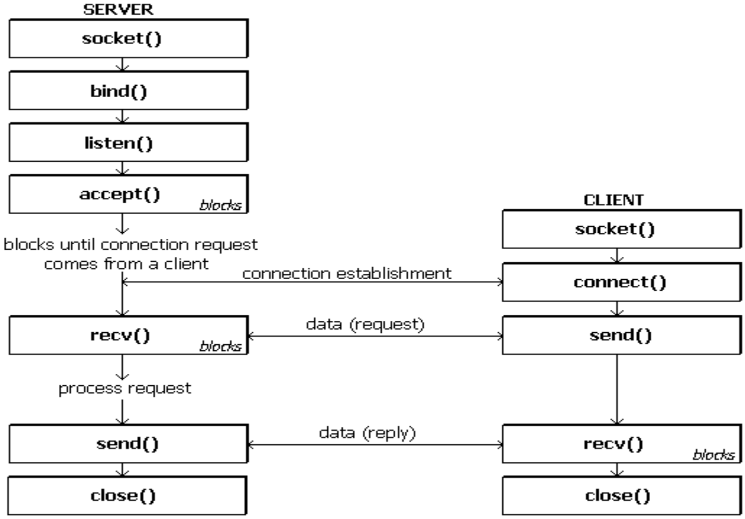


Fig.1.3: Connection oriented Socket Structure

Table 1.1: Significance of each socket function call

Primitive	Meaning
Socket	Create a new communication request
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Socket System Calls for Connectionless Protocol

Steps of establishing a UDP socket communication on the client side are as follows:

1. Create a socket using the socket() function
2. Send and receive data by means of the recvfrom() and sendto() functions.

Steps of establishing a UDP socket communication on the server side are as follows:

1. Create a socket with the socket() function;
2. Bind the socket to an address using the bind() function;
3. Send and receive data by means of recvfrom() and sendto().

Fig. 1.4 shows the interaction between a UDP client and server. The client sends a datagram to the server using the sendto() function which requires the address of the destination as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server calls the recvfrom() function, which waits until data arrives from some client. recvfrom() returns the IP address of the client, along with the datagram, so the server can send a response to the client.

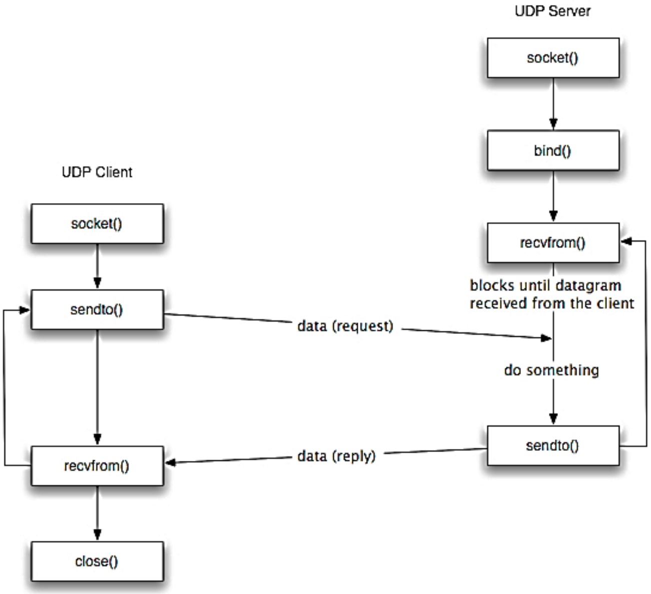


Fig.1. 4: Connectionless Socket Structure

Socket System Calls in detail

Headers

1. **sys/types.h:** Defines the data type of socket address structure in unsigned long.
2. **sys/socket.h:** The socket functions can be defined as taking pointers to the generic socket address structure called sockaddr.
3. **netinet/in.h:** Defines the IPv4 socket address structure commonly called Internet socket address structure called sockaddr_in.

Structures

Various structures are used in Unix Socket Programming to hold information about the address and port, and other information. Most socket functions require a pointer to a socket address structure as an argument.

1. **sockaddr:** This is a generic socket address structure, which will be passed in most of the socket function calls. It holds the socket information. The table 1.2 provides a description of the member fields.

```
struct sockaddr {
        unsigned short sa_family;
        char sa_data[14];
};
```

Table 1.2: Generic socket address structure fields

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sa_data	Protocol-specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address. For the Internet family, we use port number IP address, which is represented by *sockaddr_in* structure.

2. sockaddr_in

This helps to reference to the socket's elements and table 1.3 provides a description of the member fields.

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

Table 1.3: Structure fields of sockaddr_in

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.

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	This value could be set to NULL as this is not being used.

3. in_addr

This structure is used only in the above structure as a structure field and holds 32 bit netid/hostid.

```
struct in_addr {
        unsigned long s_addr;
};
```

The table 1.4 provides a description of the member fields.

Table 1.4: Structure fields of in_addr

Attribute	Values	Description
s_addr	service port	A 32-bit IP address in Network Byte Order.

4. hostent

This structure is used to keep information related to host.

```
struct hostent {
        char *h_name;
        char **h_aliases;
        int h_addrtype;
        int h_length;
        char **h_addr_list
        #define h_addrh_addr_list[0]
};
```

The table 1.5 provides a description of the member fields.

Table 1.5: Structure fields of hostent

Attribute	Values	Description
h_name	ti.com etc.	It is the official name of the host. For example, tutorialspoint.com, google.com, etc.
h_aliases	TI	It holds a list of host name aliases.
h_addrtype	AF_INET	It contains the address family and in case of Internet based application, it will always be AF_INET.
h_length	4	It holds the length of the IP address, which is 4 for Internet Address.
h_addr_list	in_addr	For Internet addresses, the array of pointers h_addr_list[0], h_addr_list[1], and so on, are points to structure in_addr.

NOTE − h_addr is defined as h_addr_list[0] to keep backward compatibility.

5. **servent**

   This particular structure is used to keep information related to service and associated ports.

```
struct servent {
        char  *s_name;
        char  **s_aliases;
        int s_port;
        char  *s_proto;
};
```

The table 1.6 provides a description of the member fields.

Table 1.6: Structure fields of servent

Attribute	Values	Description
s_name	http	This is the official name of the service. For example, SMTP, FTP POP3, etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time this will be set to NULL.
s_port	80	It will have associated port number. For example, for HTTP, this will be 80.
s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Ports and Services

To resolve the problem of identifying a particular server process running on a host, both TCP and UDP have defined a group of ports. A port is always associated with an IP address of a host and the protocol type of the communication, and thus completes the destination or origination address of a communication session. A port is identified for each address and protocol by a 16-bit number, commonly known as the **port number**. Specific port numbers are often used to identify specific services as listed below.

1. **Ports 0-1023** *(well-known ports)*: reserved for privileged services like for ftp: 21 and for telnet: 23.
2. **Ports 1024-49151** *(registered ports)*: vendors use for some applications.
3. **Ports above 49151** *(dynamic/ephemeral/private ports)*: short-lived transport protocol port for Internet Protocol (IP) communications allocated automatically from a predefined range by the IP stack software. After communication is terminated, the port becomes available for use in another session. However, it is usually reused only after the entire port range is used up.

Constructing Messages - Byte Ordering

In computers, addresses and port numbers are stored as integers of type:

u_short sin_port; (16 bit)
in_addr sin_addr; (32 bit)

Unfortunately, not all computers store the bytes that comprise a multibyte value in the same order. There are 2 types of byte ordering.

- **Little Endian (Host byte order)** − in this scheme, low-order byte is stored on the starting address (A) and high-order byte is stored on the next higher address (A + 1).
- **Big Endian (Network byte order)** − in this scheme, high-order byte is stored on the starting address (A) and low-order byte is stored on the next higher address (A + 1).

To allow machines with different byte order conventions communicate with each other, the internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as Network Byte Order. Table 1.7 describes some of the byte order functions available in UNIX programming.

Table 1.7: Byte order conversion function description

Function	Description
htons()	Host to Network Short (16 bit)
htonl()	Host to Network Long (32 bit)
ntohl()	Network to Host Long (32 bit)
ntohs()	Network to Host Short (16 bit)

IP Address Functions

These functions convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures). Some of the commonly used IP address functions are described below.

1. **int inet_aton(const char *strptr, struct in_addr *addrptr)**
   This function call converts the specified string in the Internet standard dot notation to a network address, and stores the address in the structure provided. The converted

address will be in Network Byte Order (bytes ordered from left to right). It returns 1 if the string was valid and 0 on error.

Following is the usage example:

```
#include <arpa/inet.h>
(...)
int retval;
struct in_addr addrptr
memset(&addrptr, '\0', sizeof(addrptr));
retval = inet_aton("68.178.157.132", &addrptr);
(...)
```

2. **in_addr_tinet_addr(const char *strptr)**

   This function call converts the specified string in the Internet standard dot notation to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.

Following is the usage example:

```
#include <arpa/inet.h>
(...)
struct sockaddr_ indest;
memset(&dest, '\0', sizeof(dest));
dest.sin_addr.s_addr = inet_addr("68.178.157.132");
(...)
```

3. **char *inet_ntoa(struct in_addr inaddr)**

   This function call converts the specified Internet host address to a string in the Internet standard dot notation.

Following is the usage example −

```
#include <arpa/inet.h>
char *ip;
ip = inet_ntoa(dest.sin_addr);
printf("IP Address is: %s\n",ip);
```

1. **Socket**

int socket (int family, int type, int protocol);

This call returns a socket descriptor that you can use in later system calls or -1 on error.

Parameters

a. **family** − specifies the protocol family and is one of the constants as given n table 1.7

Table 1.7: Different socket address family

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets
AF_KEY	Ket socket

b. **type** − It specifies the kind of socket you want. The possible socket types are listed in Table 1.8.

Table 1.8: Socket Types

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

c. **protocol** − The argument should be set to the specific protocol type given below, or 0 to select the system's default for the given combination of family and type .The default protocol for SOCK_STREAM with AF_INET family is TCP. Other protocol description is given in Table 1.9.

Table 1.9: Different protocols

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Example

```
if ( (sd = socket( AF_INET, SOCK_STREAM, 0 )) < 0 ) {
        cout<<“Socket creation error”;
        exit(-1);
}
```

---

2. Connect

The connect function is used by a TCP client to establish a connection with a TCP server. This call normally **blocks** until either the connection is established or is rejected.

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

This call returns 0 if it successfully connects to the server, otherwise it returns -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **serv_addr** − it is a pointer to struct sockaddr that contains destination (Server) IP address and port.
c. **addrlen** − the length of the address structure pointed to by servaddr. Set it to sizeof(structsockaddr).

---

3. Bind

The bind function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number. This function is called by TCP server only. bind() allows to specify the IP address, the port, both or neither. The table 1.10 summarizes the combinations for IPv4.

int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

This call returns 0 if it successfully binds to the address, otherwise it returns -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **servaddr**− it is a pointer to struct sockaddr that contains the local IP address and port.
c. **addrlen** − Set it to sizeof(structsockaddr).

The <sys/socket.h> header shall define the type **socklen_t**, which is an integer type of width of at least 32 bits.

Table 1.10: IP address and port number combinations

IP Address	IP Port	Result
INADDR_ANY	0	Kernel chooses IP address and port
INADDR_ANY	non zero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	non zero	Process specifies IP address and port

4. Listen

The listen function is called only by a TCP server. It converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.

int listen(int sockfd,int backlog);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

a. **sockfd** - it is a socket descriptor returned by the socket function.
b. **backlog** - it is the maximum number of connections the kernel should queue for this socket.

5. Accept

The accept function is called by a TCP server to return the next completed connection from the front of the completed connection queue.

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);

This call returns a non-negative descriptor on success, otherwise it returns -1 on error. The returned descriptor is assumed to be a client socket descriptor and all read-write operations will be done on this descriptor to communicate with the client.

Parameters

a. **sockfd** − It is a socket descriptor returned by the socket function.
b. **cliaddr** − It is a pointer to structsockaddr that contains client IP address and port.
c. **addrlen** − Set it to sizeof(structsockaddr).

---

6. Send

The send function is used to send data over stream sockets or CONNECTED datagram sockets. If you want to send data over UNCONNECTED datagram sockets, you must use sendto() function.

int send(int sockfd, const void *msg, int len, int flags);

This call returns the number of bytes sent out, otherwise it will return -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **msg** − it is a pointer to the data you want to send.
c. **len** − it is the length of the data you want to send (in bytes).
d. **flags** − it is set to 0.The additional argument flags is used to specify how we want the data to be transmitted.

---

7. Recv

The recv function is used to receive data over stream sockets or CONNECTED datagram sockets. If you want to receive data over UNCONNECTED datagram sockets you must use recvfrom().

int recv(int sockfd, void *buf, int len, unsigned int flags);

This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **buf** − it is the buffer to read the information into.
c. **len** − it is the maximum length of the buffer.
d. **flags** − it is set to 0. The additional argument flags is used to specify how we want the data to be transmitted.

8. Sendto

The sendto function is used to send data over UNCONNECTED datagram sockets. Upon successful completion, sendto() shall return the number of bytes sent. Otherwise, -1 shall be returned and *errno* set to indicate the error.

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

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **msg** − it is a pointer to the data you want to send.
c. **len** − it is the length of the data you want to send (in bytes).
d. **flags** − it is set to 0.
e. **to** − it is a pointer to struct sockaddr for the host where data has to be sent.
f. **tolen** − it is set it to sizeof(struct sockaddr).

9. Recvfrom

The recvfrom function is used to receive data from UNCONNECTED datagram sockets. Upon successful completion, *recvfrom*() shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvfrom*() shall return 0. Otherwise, the function shall return -1 and set *errno* to indicate the error.

size_t recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **buf** − it is the buffer to read the information into.
c. **len** − it is the maximum length of the buffer.
d. **flags** − it is set to 0.
e. **from** − it is a pointer to structsockaddr for the host where data has to be read.
f. **fromlen** − it is set it to sizeof(structsockaddr).
g. **ssize_t** - this data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to size_t, but must be a signed type.

10. Close

The close function is used to close the communication between the client and the server.

int close(int sockfd);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

sockfd − it is a socket descriptor returned by the socket function.

---

11. Shutdown

The shutdown function is used to gracefully close the communication between the client and the server. This function gives more control in comparison to the *close* function.

int shutdown(int sockfd, int how);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **how** − Put one of the numbers −
   i. 0 − indicates that receiving is not allowed,
   ii. 1 − indicates that sending is not allowed, and
   iii. 2 − indicates that both sending and receiving are not allowed. When ‘how’ is set to 2, it's the same thing as close().

Sample exercise

Algorithm to implement an UDP Echo Client/Server Communication

Server:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_DGRAM.
3. Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to SERVER_PORT.
4. Bind the local host address to socket using the bind function.
5. Within an infinite loop, receive message from the client using recvfrom function, print it on the console and send (echo) the message back to the client using sendto function.

Client:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_DGRAM.
3. Initialize the socket parameters. Assign the sin_family to AF_INET, sin_addr to “127.0.0.1", sin_port to dynamically assigned port number.
4. Within an infinite loop, read message from the console and send the message to the server using the sendto function.
5. Receive the echo message using the recvfrom function and print it on the console.

A simple UPD client-server program where a client connects to the server. Server sends a message to the client which is displayed at the client side.

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

main()
{
        int s,r,recb,sntb,x;
        int ca;
        printf("INPUT port number: ");
        scanf("%d", &x);
        socklen_t len;
        struct sockaddr_in server,client;
        char buff[50];

        s=socket(AF_INET,SOCK_DGRAM,0);
        if(s==-1)
        {
                printf("\nSocket creation error.");
                exit(0);
        }
        printf("\nSocket created.");

        server.sin_family=AF_INET;
        server.sin_port=htons(x);
        server.sin_addr.s_addr=htonl(INADDR_ANY);
        len=sizeof(client);
        ca=sizeof(client);

        r=bind(s,(struct sockaddr*)&server,sizeof(server));
        if(r==-1)
        {
                printf("\nBinding error.");
                exit(0);
        }
        printf("\nSocket binded.");

while(1){

        recb=recvfrom(s,buff,sizeof(buff),0,(struct sockaddr*)&client,&ca);
        if(recb==-1)
        {
                printf("\nMessage Recieving Failed");
                close(s);
                exit(0);
        }

        printf("\nMessage Recieved: ");
        printf("%s", buff);

        if(!strcmp(buff,"stop"))
```

```
                break;

        printf("\n\n");
        printf("Type Message: ");
        scanf("%s", buff);

        sntb=sendto(s,buff,sizeof(buff),0,(struct sockaddr*)&client,len);
        if(sntb==-1)
        {
                printf("\nMessage Sending Failed");
                close(s);
                exit(0);
        }

        if(!strcmp(buff,"stop"))
                break;

}
        close(s);
}
```

```
                        /*********Client Code***********/
#include<string.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<fcntl.h>
#include<sys/stat.h>
```

```
main()
{
        int s,r,recb,sntb,x;
        int sa;
        socklen_t len;
        printf("INPUT port number: ");
        scanf("%d", &x);
        struct sockaddr_in server,client;
        char buff[50];
        s=socket(AF_INET,SOCK_DGRAM,0);
        if(s==-1)
        {
                printf("\nSocket creation error.");
                exit(0);
        }
        printf("\nSocket created.");

        server.sin_family=AF_INET;
        server.sin_port=htons(x);
        server.sin_addr.s_addr=inet_addr("127.0.0.1");
        sa=sizeof(server);
        len=sizeof(server);

while(1){

        printf("\n\n");
        printf("Type Message: ");
        scanf("%s", buff);

        sntb=sendto(s,buff,sizeof(buff),0,(struct sockaddr *)&server, len);
        if(sntb==-1)
        {
                close(s);
                printf("\nMessage sending Failed");
                exit(0);
        }
        if(!strcmp(buff,"stop"))
                break;

        recb=recvfrom(s,buff,sizeof(buff),0,(struct sockaddr *)&server,&sa);
        if(recb==-1)
        {
                printf("\nMessage Recieving Failed");
                close(s);
                exit(0);
        }

        printf("\nMessage Recieved: ");
        printf("%s", buff);

        if(!strcmp(buff,"stop"))
                break;
}
        close(s);

}
```

TCP Echo Client/Server Communication

An echo client/server program performs the following:

1. The client reads a line of text from the standard input and writes the line to the server using send() function.
2. The server reads the line from the network input using recv() and echoes the line back to the client using send() function.
3. The client reads the echoed line using recv() and prints it on its standard output.

Fig. 1.5 depicts the echo client/server along with the functions used for input and output.

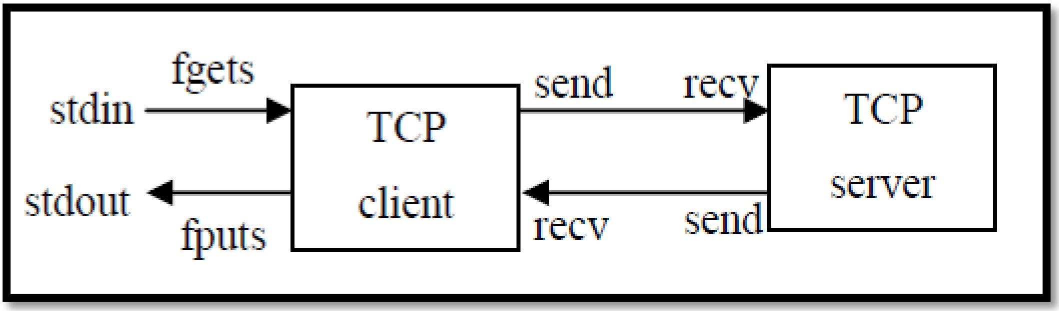


Fig.1.5: Echo client/server

A simple TCP client-server program where a client connects to the server. Server sends a message to the client which is displayed at the client side.

```
                    /*********Client Code***********/
#include<string.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<fcntl.h>
#include<sys/stat.h>

main()
{
        int s,r,recb,sntb,x;
        printf("INPUT port number: ");
        scanf("%d", &x);
        struct sockaddr_in server;
        char buff[50];
        s=socket(AF_INET,SOCK_STREAM,0);
        if(s==-1)
        {
                printf("\nSocket creation error.");
                exit(0);
        }
```

```
    printf("\nSocket created.");

    server.sin_family=AF_INET;
    server.sin_port=htons(x);
    server.sin_addr.s_addr=inet_addr("127.0.0.1");

    r=connect(s,(struct sockaddr*)&server,sizeof(server));
    if(r==-1)
    {
        printf("\nConnection error.");
        exit(0);
    }
    printf("\nSocket connected.");

    printf("\n\n");
    printf("Type Message: ");
    scanf("%s", buff);

    sntb=send(s,buff,sizeof(buff),0);
    if(sntb==-1)
    {
        close(s);
        printf("\nMessage Sending Failed");
        exit(0);
    }

    recb=recv(s,buff,sizeof(buff),0);
    if(recb==-1)
    {
        printf("\nMessage Recieving Failed");
        close(s);
        exit(0);
    }

    printf("\nMessage Recieved: ");
    printf("%s", buff);
    printf("\n\n");
```

```
	close(s);

}
```

/*******SERVER CODE**********/**

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

main()
{
	int s,r,recb,sntb,x,ns,a=0;
	printf("INPUT port number: ");
	scanf("%d", &x);
	socklen_t len;
	struct sockaddr_in server,client;
	char buff[50];

	s=socket(AF_INET,SOCK_STREAM,0);
	if(s==-1)
	{
		printf("\nSocket creation error.");
		exit(0);
	}
	printf("\nSocket created.");

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

	r=bind(s,(struct sockaddr*)&server,sizeof(server));
	if(r==-1)
	{
		printf("\nBinding error.");
```

```
        exit(0);
    }
    printf("\nSocket binded.");

    r=listen(s,1);
    if(r==-1)
    {
        close(s);
        exit(0);
    }
    printf("\nSocket listening.");

    len=sizeof(client);

    ns=accept(s,(struct sockaddr*)&client, &len);
    if(ns==-1)
    {
        close(s);
        exit(0);
    }
    printf("\nSocket accepting.");

    recb=recv(ns,buff,sizeof(buff),0);
    if(recb==-1)
    {
        printf("\nMessage Recieving Failed");
        close(s);
        close(ns);
        exit(0);
    }

    printf("\nMessage Recieved: ");
    printf("%s", buff);

    printf("\n\n");
    scanf("%s", buff);

    sntb=send(ns,buff,sizeof(buff),0);
    if(sntb==-1)
    {
        printf("\nMessage Sending Failed");
        close(s);
        close(ns);
        exit(0);
    }

    close(ns);
    close(s);
}
```

Steps to execute the program

1. Open two terminal windows and open a text file from each terminal with .c extension using command: gedit *filename.c*
2. Type the client and server program in separate text files and save it before exiting the text window.
3. First compile and run the server using commands mentioned below
    a. gcc -o *filename*
    b. ./a.out or ./filename
4. Compile and run the client using the same instructions as listed in 3a & 3b.

Note: The ephemeral port number has to be changed every time the program is executed.

Lab exercises

1. Write two separate C programs (one for server and other for client) using socket APIs for TCP, to implement the client-server model such that the client should send a set of integers along with a choice to search for a number or sort the given set in ascending/descending order or split the given set to odd & even to the server. The server perform the relevant operation according to the choice. Client should continue to send the messages until the user enters selects the choice “exit”.
2. Write two separate C programs (one for server and other for client) using UNIX socket APIs for UDP, in which the client accepts a string from the user and sends it to the server. The server will check if the string is palindrome or not and send the result with the length of the string and the number of occurrences of each vowel in the string to the client. The client displays the received data on the client

screen. The process repeats until user enter the string “Halt”. Then both the processes terminate. (The program should make use of TCP and UDP separately).

Additional exercise

1. Write two separate C programs (one for the Server and the other for Client) using UNIX socket APIs using both connection oriented and connectionless services, in which the server displays the client’s socket address, IP address and port number on the server screen.

[OBSERVATION SPACE – LAB 1]

LAB NO: 2 **Date:**

FILE SERVER OPERATIONS

Objectives

- To illustrate file operations like open, read and write.
- To exemplify the FTP application in a client server communication.

Introduction

The File Transfer Protocol (FTP) is a standard network protocol used to transfer computer files between a client and server on a computer network. In this lab, we learn to implement FTP application in a client server architecture, where the Client on establishing a connection with the Server, sends the name of the file it wishes to access remotely. The Server then sends the contents of the file to the Client. An algorithm to implement this scenario is given below.

Algorithm (TCP):

Server:

1. Include necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attributes. Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to dynamically assigned port number.
4. Bind the server to socket using bind function.
5. Listen to incoming client requests and wait for the client request. On connection request from the client establish a connection using accept() function.
6. Read the files name sent by the client.
7. Open the file, read the file contents to a buffer and send the buffer to the Client.
8. After reading till the end of the file, close the file.
9. Close the connection.

Client:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attribute set. Assign the sin_family to AF_INET, sin_addr to “127.0.0.1", sin_port to dynamically assigned port number.
4. Connect to server using connect () function to initiate the request.
5. Send the file names to server.
6. Receive the buffer from the server and print its contents at the console.
7. Close the connection.

Algorithm (UDP):

Server:

1. Include necessary header files
2. Create a socket using socket function with family AF_INET, type as SOCK_DGRAM.
3. Initialize the socket and set its attributes.Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to dynamically assigned port number.
4. Read the files name sent by the client using recvfrom() function.
5. Open the file, read the file contents to a buffer and send the buffer to the Client using sendto() function.
6. Close the connection.

Client:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_DGRAM.
3. Initialize the socket and set its attribute set. Assign the sin_family to AF_INET, sin_addr to “127.0.0.1", sin_port to dynamically assigned port number.
4. Send the file names to server using sendto function.
5. Receive the buffer from the server using recvfrom function and print its contents at the console.
6. Close the connection.

Lab exercises

1. Write two separate C programs (one for server and other for client) using UNIX socket APIs for both TCP and UDP to implement the following:
   The user at the client side sends name of the file to the server. If the file is not present, the server sends "File not present" to the client and terminates. Otherwise the following menu is displayed at the client side.
   1. Search
   2. Replace
   3. Reorder
   4. Exit
   - If the user at the client side wants to search a string in file, the users sends to the server option '1' along with the string to be searched. The server searches for the string in the file. If present, it sends the number of times the string has occurred to the client, else the server sends 'String not found' message to the client.
   - If the user wants to replace a string, along with option 2, the two strings 'str1' and 'str2' are sent to the server. The Server searches for str1 in the file and replaces it with 'str2'. After replacing the string, 'String replaced' message is sent to the client. If it is not found 'String not found' command is sent to the client.
   - Option 3 rearranges the entire text in the file in increasing order order of their ASCII value.
   - To terminate the application option '4' is selected

Additional exercise:

1. Write two separate C programs (one for server and other for client) using socket APIs for TCP and UDP, to implement the File Server. The client program will send the name of the text file to the server. If the file is present at the server side, the server should send the contents of the file to the client along with the file size, number of alphabets number of lines, number of spaces, number of digits, and number of other characters present in the text file to the client. If the file is not

present, then the server should send the proper message to the client. Note that the results are always displayed at the client side. Client should continue to send the filenames until the user enters the string ‘stop’.

[OBSERVATION SPACE – LAB 2]

LAB NO: 3 **Date:**

CHAT SERVER

Objectives

- To contrast different modes of communication by implementing half duplex and full duplex chat server in connectionless and connection oriented environment.
- To illustrate the use of fork() system call in socket programming.

Introduction

Writing a chat application with popular web applications stacks has traditionally been very hard. It involves polling the server for changes, keeping track of timestamps, and it's a lot slower than it should be. Sockets have traditionally been the solution around which most real-time chat systems are architected, providing a bi-directional communication channel between a client and a server.

A chat server is a computer dedicated to providing the processing power to handle and maintain chatting and it users. This means that the server can push messages to clients and also support multiple clients to initiate the conversation. To develop a chat server model, it is important to understand the different types of server and the mode of communication supported.

Basic Modes of Communication

Data communication can be either simplex, half duplex or full duplex which are described below.

Simplex Operation

In simplex operation, a network cable or communications channel can only send information in one direction; it's a "one-way street". Simplex operation is used in special types of technologies, especially ones that are asymmetric. For example, one type of satellite Internet access sends data over the satellite only for downloads, while a regular dial-up modem is used for upload to the service provider. In this case, both the satellite link and the dial-up connection are operating in a simplex mode.

Half-Duplex Operation

Technologies that employ half-duplex operation are capable of sending information in both directions between two nodes, but only one direction or the other can be utilized at a time. This is a fairly common mode of operation when there is only a single network medium (cable, radio frequency and so forth) between devices. For example, in conventional Ethernet networks, any device can transmit, but only one may do so at a time. For this reason, regular Ethernet networks are often said to be “half-duplex”.

Full-Duplex Operation

In full-duplex operation, a connection between two devices is capable of sending data in both directions simultaneously. Full-duplex channels can be constructed either as a pair of simplex links or using one channel designed to permit bidirectional simultaneous transmissions. A full-duplex link can only connect two devices, so many such links are required if multiple devices are to be connected together.

Types of Server

There are two types of servers.

- **Iterative Server** − this is the simplest form of server where a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting. In other words, an *iterative* server iterates through each client, handling it one at a time.
- **Concurrent Servers** − this type of server runs multiple concurrent processes to serve many requests at a time because one process may take longer and another client cannot wait for so long. The simplest way to write a concurrent server under UNIX is to *fork* a child process to handle each client separately. An alternative technique is to use threads instead (i.e., light-weight processes).

Some System Calls

System Call: int fork()

- fork() causes a process to duplicate. The child process is an almost-exact duplicate of the original parent process; it inherits a copy of its parent's code, data, stack, open file descriptors, and signal table. However, the parent and child have different process id numbers and parent process id numbers.

- If fork() succeeds, it returns the PID of the child to the parent process, and returns 0 to the child process. If it fails, it returns -1 to the parent process and no child is created.

System Call: int getpid()
int getppid()

getpid() and getppid() return a process's id and parent process's id numbers, respectively. They always succeed. The parent process id number of PID 1 is 1.

Simple fork example to display PID and PPID

```
#include <stdio.h>
main()
{
   int pid;
   printf("I'm the original process with PID %d and PPID %d.\n",
         getpid(),getppid());
   pid=fork();  /* Duplicate. Child and parent continue from here.*/
   if (pid!=0)  /* pid is non-zero, so I must be the parent  */
      {
        printf("I'm the parent process with PID %d and PPID %d.\n",
              getpid(),getppid());
        printf("My child's PID is %d.\n", pid);
      }
   else  /* pid is zero, so I must be the child. */
      {
        printf("I'm the child process with PID %d and PPID %d.\n",
              getpid(),getppid());
      }
   printf("PID %d terminates.\n",pid);  /* Both processes execute this */
}
```

Half Duplex TCP Chat Program

Aim: To implement a half-duplex application, where the Client establishes a connection with the Server. The client and server can send and receive messages one at a time**.**

Algorithm (TCP):

Server:

1. Include necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attributes.Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to dynamically assigned port number.
4. Bind the server to socket using bind function.
5. Listen to incoming client requests and wait for the client request. On connection request from the client establish a connection using accept() function.
6. Receive and display the message sent by the client.
7. Read the message from the console and send it to the client.
8. If the received message is “BYE” terminate the connection (kill the process).
9. Close the socket.

Client:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attribute set. Assign the sin_family to AF_INET, sin_addr to “127.0.0.1", sin_port to dynamically assigned port number.
4. Connect to server using connect () function to initiate the request.
5. Read the message from the console and send it to the server using send call.
6. Receive the message from the server and display it.
7. If the received message is “BYE” terminate the connection (kill the process).
8. Close the socket.

Full Duplex TCP Chat Program

Aim: To implement a full duplex application, where the Client establishes a connection with the Server. The Client and Server can send as well as receive messages at the same time. Both the Client and Server exchange messages.

Algorithm (TCP):

Server:

1. Include necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attributes.Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to dynamically assigned port number.
4. Bind the server to socket using bind function.
5. Listen to incoming client requests and wait for the client request. On connection request from the client establish a connection using accept () function.
6. Fork the process to create child process which is an exact copy of the calling process (parent process).
7. If the process is child receive the message from the client and display it. Else if the process is parent read the message from the console and send it to the client.
8. If the received message is “BYE” terminate the connection (kill the process).
9. Close the socket.

Client:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attribute set. Assign the sin_family to AF_INET, sin_addr to “127.0.0.1", sin_port to dynamically assigned port number.
4. Connect to server using connect () function to initiate the request.
5. Fork the process to create child process which is an exact copy of the calling process (parent process).
6. If the process is child read the message from the console and send it to the server. If the process is parent receive the message from the server and display it.
7. If the received message is “BYE” terminate the connection (kill the process).
8. Close the socket.

Note : Operations for UDP follows the same process but make necessary changes in

the system calls.

Lab exercise

1. Write two separate C programs using UNIX socket APIs illustrate full duplex mode chat application between a single client and server using connection oriented service. Display PID and PPID of both parent and child processes.

2. Write two separate C programs using UNIX socket APIs illustrate half duplex mode chat application between a single client and server connection less service in which the server estimates and prints all permutations of a string sent by the client.

3. Write two separate C programs (one for server and other for client) using socket APIs, to implement the following connection-oriented client-server model.

    i. The user at the client side sends an alphanumeric string to the server.

    ii. The child process at the server sorts the numbers of the alphanumeric string in ascending order. The parent process at the server sorts the characters of the alphanumeric string in descending order.

    iii. Both the processes send the results to the client along with its corresponding process ID.

    Sample Output:
    At the client side:
    Input string: hello451bye7324
    At the server side:
    Output at the child process of the server: 1234457
    Output at the parent process of the server: yollheeb

Additional exercise

1. Write a C program to simulate a menu driven calculator using client server architecture that performs the following.

    The client prompts the user with the options as listed below

    1. Add/Subtract two integers
    2. Find the value of ‘x’ in a linear equation

3. Multiply two matrices
4. Exit

Based on the user input the client prompts the user to enter required data. The client sends the option chosen and the relevant data to the server. The server performs the required operation and sends the result to the client. Note that if option 1 is selected, the server provides result of both addition and subtraction of the two integers.

[OBSERVATION SPACE – LAB 3**]**

LAB NO. 4 **Date:**

DATABASE OPERATIONS & PROTOCOL IMPLEMENTATION (DNS)

Objectives

- To implement client server communication that involves database operations such as insert, delete, edit, search or replace the data.
- To illustrate the significance of Domain Name Server using socket programming.

Introduction

As we are moving towards an era of big data and digitization, handling applications that involve huge database and providing controlled user access is a daunting task. Many applications like banking/e-commerce use complex authentication protocols to provide access to its registered customers. A simple authentication algorithm would be to validate the login credentials with the existing database at the server. An instance of such application can be implemented in a client server architecture using socket programming. To build such an application structures are used. Structure is user defined data type available in C that allows to combine data items of different kinds. They are used to represent a record. Suppose it is necessary to keep track of all the books in a library, then the following attributes might define a book.

- Title
- Author
- Subject
- Book ID

Accessing Structure information:

```
struct Books {                    /******* Define a structure with name Books*****/
char  title[50];
char  author[50];
char  subject[100];
int book_id;
};
int main( ) {
```

```
struct Books Book1;        /* Declare Book1 of type Book */
struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

  /* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);
}
```

Algorithm to implement Library Database Management System (TCP):

Server:

1. Include necessary header files. Create a structure containing all the fields required in the database along with their types.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attributes.Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to dynamically assigned port number.
4. Bind the server to socket using bind function.
5. Listen to incoming client requests and wait for the client request. On connection request from the client establish a connection using accept() call.
6. Based on client’s choice perform Suitable database operations.
7. Communicate the results by sending appropriate messages back to the client informing the success or failure of the requested operation using send()- recv() calls
8. If the client chooses “Exit “option Close the connection.

Client:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize the socket and set its attribute set. Assign the sin_family to AF_INET, sin_addr to “127.0.0.1", sin_port to dynamically assigned port number.
4. Connect to server using connect () function to initiate the request.
5. Select the required option and sends it along with the necessary information to the server.
6. Receive the buffer from the server and print its contents (results) at the console.
7. Close the connection.

Note: For connectionless services repeat the above steps by making suitable changes in the socket calls.

DNS (Domain Name Servers)

Computers and other network devices on the Internet use an IP address to route the client request to required website. It’s impossible for us to remember all the IP addresses of the servers we access every day. Hence we assign a domain name for every server and use a protocol called DNS to turn a user-friendly domain name like "howstuffworks.com" into an Internet Protocol (IP) address like 70.42.251.42 that computers use to identify each other on the network. In other words, DNS is used to map a host name in the application layer to an IP address in the network layer. DNS is a client/server application in which a domain name server, also called a DNS server or name server, manages a massive database that maps domain names to IP addresses. Client requests for address resolution which is defined as mapping a name to an address or an address to a name. It can be done in a recursive fashion or in an iterative fashion.

Algorithm to simulate DNS environment (iterative method)

Server

1. Include header files.
2. Create the socket for the server.
3. Bind the socket to the port.
4. Listen for the incoming client connection.
5. Receive the IP address from the client to be resolved.

6. Get the domain name from the client.
7. Check the existence of the domain in the server database which is a text file.
8. If domain matches then send the corresponding address to the client. Otherwise send a negative response.
9. Stop the program execution.

Client

1. Include header files.
2. Create the socket for the client.
3. Connect the socket to the server.
4. Prompt user to enter the hostname and send the hostname to the server.
5. Display response received.
6. Terminate the program.

Lab exercises

1. Write two separate C programs (one for server and other for client) using socket APIs for TCP and UDP to perform the following. The user at the client side has an option to enter:
    1. Registration Number
    2. Name of the Student
    3. Subject Code.

The Client sends the selected option along with the requisite details to the server. Based on the options received the parent process in the server assigns the task to respective child process.

i. If registration number is sent then the first child process sends Name and Residential Address of the student along with the PID of the child process.
ii. If Name of the Student is received then the second child process sends student enrollment details (Dept., Semester, Section and Courses Registered) along with the PID of the child process.
iii. If Subject Code is entered then the third child process sends the corresponding marks obtained in that subject along with its PID.
iv. The details sent by the server have to be displayed at the client.

2. Write two separate C programs (one for server and other for client) using UNIX socket APIs using connection oriented services to implement DNS Server. Accept

suitable input messages from the user. Assume the server has access to database.txt (can be a structure too). Response is always displayed at the client side.

Additional exercise

1. Create a Book database at the server side and store the following information: title, author, accession number, total pages, and the publisher. Write C programs to implement the following client-server model:
    a. Insert new book information
    b. Delete a book
    c. Display all book information
    d. Search a book (Based on Title or author)
    e. Exit

   At the client side, the user selects the required option and sends it along with the necessary information to the server & server will perform the requested operation. Server should send appropriate messages back to the client informing the success or failure of the requested operation. Client should continue to request the operation until user selects the option “Exit”. To search the book by author name, the client program should send the name of the author to the server. The list of all book details for that author should be sent to the client. If the author name is not found, then server should send appropriate message to the client.

[OBSERVATION SPACE – LAB 4**]**

LAB NO. 5 **Date:**

MULTIPLE CLIENT COMMUNICATION

Objectives

- To implement multiple client communication with concurrent server.
- To implement the services performed by an iterative DNS server to multiple clients using socket programming.

Introduction

Types of Servers

There are two main classes of servers, iterative and concurrent. An *iterative* server iterates through each client, handling it one at a time. A *concurrent* server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the fork function, creating one child process for each client. An alternative technique is to use *threads* instead (i.e., light-weight processes).

Process-based using fork

- Spawn one server process to handle each client connection
- Kernel automatically interleaves multiple server processes
- Each server process has its own private address space

A typical concurrent server using fork has the following structure:

```
pid_tpid;
int listenfd, connfd;
listenfd = socket(...);
/***fill the socket address with server’s well known port***/
bind(listenfd, ...);
listen(listenfd, ...);
for ( ; ; ) {
connfd = accept(listenfd, ...); /* blocking call */
if ( (pid = fork()) == 0 ) {
close(listenfd); /* child closes listening socket */
     /***process the request doing something using connfd ***/
     /* ................. */
close(connfd);
exit(0);  /* child terminates
    }
close(connfd);  /*parent closes connected socket*/
}
}
```

When a connection is established, accept returns, the server calls fork, and the child process services the client (on the connected socket connfd). The parent process waits for another connection (on the listening socket listenfd). The parent closes the connected socket since the child handles the new client. The interactions among client and server are presented in Fig. 6.1.

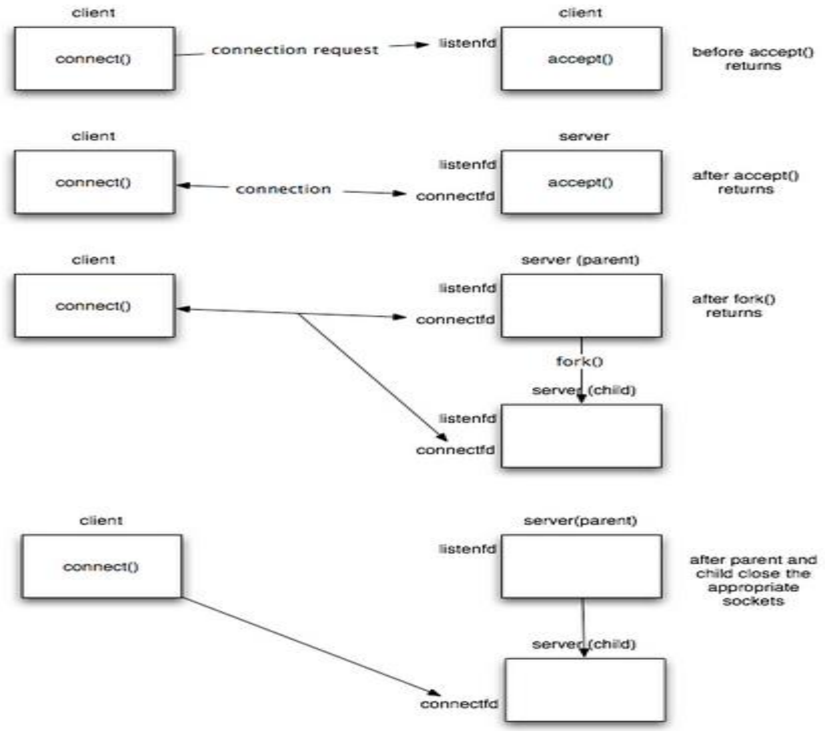


Fig.5.1: Example of interaction among a client and a concurrent server.

TCP Iterative Server Example

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/
#define LISTENQ 8 /*maximum number of client connections */
int main (intargc, char **argv)
{
int listenfd, connfd, n;
socklen_t clilen;
char buf[MAXLINE];
struct sockaddr_incliaddr, servaddr;

 //creation of the socket
listenfd = socket (AF_INET, SOCK_STREAM, 0);

 //preparation of the socket address
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

bind (listenfd, (structsockaddr *) &servaddr, sizeof(servaddr));

listen (listenfd, LISTENQ);

printf("%s\n","Server running...waiting for connections.");

for ( ; ; ) {
clilen = sizeof(cliaddr);
connfd = accept (listenfd, (structsockaddr *) &cliaddr, &clilen);
printf("%s\n","Received request...");
while ( (n = recv(connfd, buf, MAXLINE,0)) > 0)  {
printf("%s","String received from and resent to the client:");
puts(buf);
send(connfd, buf, n, 0);
  }

if (n < 0) {
perror("Read error");
exit(1);
 }
close(connfd);

 }
 //close listening socket
close (listenfd);
}
```

TCP Concurrent server Using Fork

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/
#define LISTENQ 8 /*maximum number of client connections*/

int main (intargc, char **argv)
{
Int listenfd, connfd, n;
```

```
pid_t childpid;
socklen_t clilen;
char buf[MAXLINE];
struct sockaddr_in cliaddr, servaddr;

 //Create a socket
 //If sockfd<0 there was an error in the creation of the socket
if ((listenfd = socket (AF_INET, SOCK_STREAM, 0)) <0) {
perror("Problem in creating the socket");
exit(2);
 }
 //preparation of the socket address
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

 //bind the socket
bind (listenfd, (structsockaddr *) &servaddr, sizeof(servaddr));

 //listen to the socket by creating a connection queue, then wait for clients
listen (listenfd, LISTENQ);
printf("%s\n","Server running...waiting for connections.");

for ( ; ; ) {

clilen = sizeof(cliaddr);
  //accept a connection
connfd = accept (listenfd, (structsockaddr *) &cliaddr, &clilen);

printf("%s\n","Received request...");
if ( (childpid = fork ()) == 0 ) {//if it’s 0, it’s child process
printf ("%s\n","Child created for dealing with client requests");
  //close listening socket
close (listenfd);

while ( (n = recv(connfd, buf, MAXLINE,0)) > 0)  {
printf("%s","String received from and resent to the client:");
puts(buf);
send(connfd, buf, n, 0);
}

if (n < 0)
printf("%s\n", "Read error");
exit(0);
 }
 //close socket of the server
close(connfd);
}
}
```

Lab exercises

1. Write a single server and multiple client program to illustrate multiple clients communicating with a concurrent server. The client1 on establishing successful connection sends “Institute Of” string to the server along with its socket address. The client2 on establishing successful connection sends “Technology” string to the server along with its socket address. The server opens a text file having the keyword “ Manipal” , append the keywords” Institute of “ and “ Technology “ and displays “ Manipal Institute of Technology” along with the socket addresses of the clients . If the number of clients connected exceeds 2, the server sends “terminate session” to all clients and the program terminates.
2. Write a single server multiple client program to illustrate multiple clients communicating with a single iterative server. The client on establishing successful connection prompts the user to enter 2 strings which is sent to the server along with client socket address. The server checks whether the strings are anagrams or not and sends an appropriate message to the client. The result obtained is then

displayed on the client side. The server displays the date and time along with client socket address that it is connected to it at any instant.

Additional exercise

1. Write C program to simulate travel ticket reservation system. Where the server displays the number of seats available and the number of seats booked of two different source and destination locations. Multiple clients try to connect to server and sends the number of seats to be booked as entered by the user. The server database has to be updated and the client should terminate its session after successful seat reservation. Note that if the requested number of seats are unavailable the server sends appropriate message to the client and the client program terminates. Price of ticket need not be taken into consideration.

[OBSERVATION SPACE – LAB 5**]**

LAB NO: 6 **Date:**

APPLICATION DEVOLPMENT

Objectives

- To apply the socket programming concept in developing the real world applications

Lab exercises

1. Write two separate C programs (one for server and other for client) using socket APIs, to implement the following connection-oriented client-server model for "BANKING APPLICATION"

   To login, the user at client side sends username and password (can be alphanumeric) to the server. The server maintains a file (database) that has a list of username, its corresponding password in encrypted form (Use Caser Cipher for encryption where each letter is replaced by a letter which is a shift of 3 of the original letter. Eg. 'a' is replaced by 'd') and the current account balance. On receiving the credentials from the client, the server first encrypts the password and validates it against the file contents)

   i. If the username is incorrect, the server displays 'Incorrect Username 'and sends this message to the client.

   ii. Otherwise, the server displays 'Incorrect Password' and sends this message to the client.

   On Successful login the server sends a menu with following options.

   i. Debit
   ii. Credit
   iii. View Balance
   iv. EXIT

   Based on the Choice of the client transactions are done (must be reflected in the database) and the application Quits on choosing option iv.

2. Demonstrate a TCP Chat System between two PCS.

3. Write two separate C programs (one for server and other for client) using socket APIs, to implement the following client-server model. The user at the client side sends a filename containing a text to the server. The server checks the existence

of the file. If present it performs the following operations:

i. The child process at the server converts the text to uppercase.

ii. The parent process at the server replaces each letter in the text with its equivalent digit. Map a to 1 b to 2 and so on.

iii. Both the processes should write the results onto the file with their process IDs.

iv. Client reads the results from the file.

Additional exercises

1. Write two separate C programs (one for server and the other for client) using socket APIs, to implement the following **connection-oriented** client -server model for Movie ticket booking application.

   The server maintains a database (file/structure) consisting of ‘Movie Names’, ‘Movie timings’ and ‘Seats Available’. At the client side the following menu is displayed

   1. Book Tickets
   2. Exit.

   The option selected by the user is sent to the server. If option ‘1’ is selected, a list of ‘Movie Names’ is sent to the client by the server.

   i. The user at the client side selects a movie name which is displayed at the server side.

   ii. The server sends ‘Movie timings’ and ‘Seats available’ to the user at the client.

   iii. The user sends movie timings and the required number of seats to the server. If seats are available the server decrements the number of seats available and sends the message ‘Seats booked’ to the user at client. If Seats are unavailable the Server sends the message ‘Seats Unavailable’ to the client. The updated database must be displayed at the server side. This process should continue until user selects option 2 (Exit).

[OBSERVATION SPACE – LAB 6**]**

LAB NO. 7 **Date:**

INTRODUCTION To NS-2 – WIRED NETWORK

Objectives

- To implement simple network scenario using TCL script in NS-2.
- To interpret different agents and their applications like FTP over TCP and CBR over UDP.

Introduction

The Network Simulator -2 (NS - Version 2) is discrete event packet level open source network simulation tool. Network simulator is a package of tools that simulates the behavior of networks such as creating network topologies, log events that happen under any load, analyze the events and understand the network. The primary use of NS is in network researches to simulate various types of wired/wireless local and wide area networks; to implement network protocols such as TCP and UPD, traffic source behavior such as FTP, Telnet, Web, CBR and VBR, router queue management mechanism such as Drop Tail, RED and CBQ, routing algorithms such as Dijkstra and many more. NS-2 is written in C++ and Otcl to separate the control and data path implementations. The simulator supports a class hierarchy in C++ (the compiled hierarchy) and a corresponding hierarchy within the Otcl interpreter (interpreted hierarchy). Fig. 7.1 shows the link between C++ and Otcl. In general the output of a tcl script will be a trace file or animation of trace file using NAM.

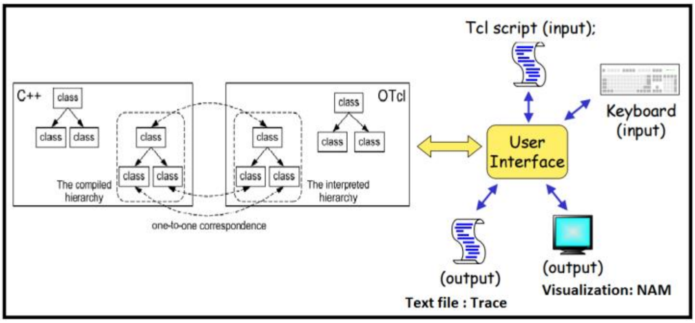


Fig. 7.1: Link between C++ and Otcl

The reason why NS-2 uses two languages is that different tasks have different requirements: For example, simulation of protocols requires efficient manipulation of bytes and packet headers making the run-time speed very important. On the other hand, in network studies where the aim is to vary some parameters and to quickly examine a number of scenarios the time to change the model and run it again is more important.

In NS-2, C++ is used for detailed protocol implementation and in cases where every packet of a flow has to be processed. For instance, if you want to implement a new queuing discipline, then C++ is the language of choice. Otcl, on the other hand, is suitable for configuration and setup. Otcl runs quite slowly, but it can be changed very quickly making the construction of simulations easier. In NS-2, the compiled C++ objects can be made available to the Otcl interpreter. In this way, the ready-made C++ objects can be controlled from the OTcl level.

Strengths and Weaknesses of C++ and OTcl

C++: C++ is a compiled programming language. A C++ program needs to be compiled (i.e., translated) into the executable machine code. Since the executable is in the form of machine code, C++ program is very fast to run. However, the compilation process can be quite annoying. Every tiny little change like adding “int x = 0;” will take few seconds.

OTcl: OTcl is an interpreted programming language. An OTcl program can run on the fly without the need for compilation. Upon execution, the interpreter translates OTcl instructions to machine code understandable to the operating system line by line. Therefore, OTcl codes run more slowly than C++ codes do. The upside of OTcl codes is that every change takes effect immediately.

Simple NS 2 Script To Print Hello World

```
set ns [new Simulator]
$ns at 1 "puts \"hello world\""
$ns at 1.5 "exit"
$ns run
open the terminal and execute the hello.tcl by
        $ ns hello.tcl
```

Tcl script

The algorithm for writing a Tcl script is given below

Begin:

1. Create instance of network simulator object
2. Turn on NAM
3. Define a 'Finish' procedure – Here the instruction to open NAM has to be included
4. Create network
    - i. Create nodes
    - ii. Assign node position (NAM)
    - iii. Create link between nodes (Simplex/Duplex)
    - iv. Define colors for data flow (NAM)
    - v. Set the Queue size for node (Optional)
    - vi. Monitor the queue (NAM)
    - vii. Attach agent for every node - Transport Connection (TCP/UDP)
    - viii. Set up Traffic Application (FTP/CBR)
5. Schedule the events
6. Call finish procedure
7. Print the necessary output data
8. Run the Simulation
9. Visualize using NAM or analyze the trace file

End

Optionally, procedures can be written to compute various parameters and display it on the screen. To enhance the visualization of the network scenario, the node attributes like node shape, node color etc, can be modified in the tcl script. Fig. 7.2 visualizes the sample network that can be created using tcl script.

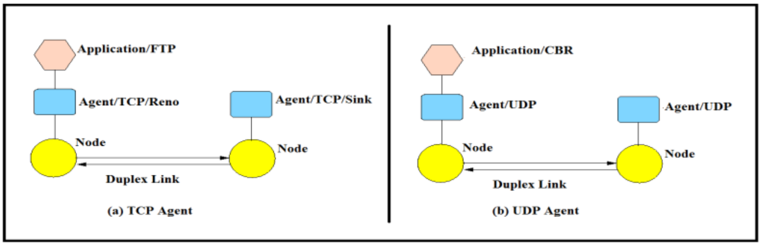


Fig. 7.2: Classes invoked in general to create duplex link connected nodes (a) TCP Agent is connected to nodes with one sink (b) UDP Agent is connected to nodes with one receiver.

Commands used in Tcl script

Table 7.1 briefs some important commands of Tcl script that can be used in NS-2

Table 7.1: Tcl Commands and its description

Command/Example	**Description**	**Additional Information**
set a 5	The variable ‘a’ is assigned the value “5”.	
set b [expr $a/5]	The result of the command [expr $a/5] is used as an argument to another command, which in turn assigns a value to the variable b.	The “$” sign is used to obtain a value contained in a variable and square brackets is an indication of a command

proc sum {a b} {
expr $a + $b
}
OR
proc sum {} {
global a b
expr $a + $b
}
	The first argument to ‘proc’ is the name of the procedure and the second argument contains the list of the argument names to that procedure which can be null.	One can define new procedures with the ‘proc’ command.

set testfile [open test.dat r]	Open a file named ‘test.dat in read mode. The variable ‘testfile’ is the argument.	
gets $testfile list	Store the first line of the file pointed by variable ‘testfile’ in a list	
set first [lindex $list	Obtain the elements of the list	Numbering of elements starts from 0
set tstfile [open test.dat w] puts $tstfile “test1”	Open the file ‘test.dat’ in write mode and write the string ‘test1’ to the file	

exec out.nam	The ‘out.nam’ is executed. The command ‘exec’ creates a subprocess and waits for it to complete.	The exec command is particularly useful when one wants to call a tcl-script from within another tcl script/NAM file
set ns [new Simulator]	A new simulator object must be created at the beginning of the script. The simulator object has member functions that enable creating the nodes and the links, connecting agents etc	When using functions belonging to this class, the command begins with “$ns”, since ns was defined to be a handle to the Simulator object.
set n0 [$ns node]	The member function of the Simulator class, called “node” creates a node and assigns it to the handler n0 that can later be used when referring to the nodes.	
$ns at 0.0 “cbr0 start” $ns at 50.0 “ftp1start”	Schedule events, such as the starting or stopping times of the clients or data transfer.	

Links

Links are required to complete the topology. In NS-2, the output queue of a node is implemented as part of the link, so when creating links the user also has to define the queue-type. The Fig. 7.3 shows the construction of a simplex link in NS-2. If a duplex-link is created, two simplex links will be created, one for each direction. In the link, the packet is first enqueued at the queue. After this, it is either dropped, passed to the Null Agent and freed there, or dequeued and passed to the Delay object which simulates the link delay.

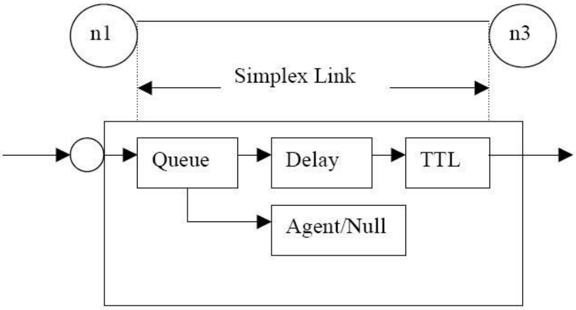


Fig.7.3: Link in NS-2

The below line tells the simulator object to connect the nodes n0 and n1 with a duplex link with the bandwidth 1Megabit, a delay of 10ms and a DropTail queue. The values for bandwidth can be given as a pure number or by using qualifiers k (kilo), M (Mega), b (bit) and B (byte).

$ns duplex-link $n0 $n2 15Mb 10ms DropTail

The Final Syntax of link Creation is as follows :
$ns <link_type> $n0 $n1 <bandwidth> <delay> <queue_type>
<link_type>: duplex-link, simplex-link
<queue_type>:

- *DropTail*
- *RED – Random Early Discard*
- *FQ – Fair Queuing*
- *DRR – Difict Round Robin*
- *SFQ – Stochastic Fair Queuing, etc*

Code So far considering the simple topology with two nodes as shown below:

```
#Create a simulator object
set ns [new Simulator]

#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 trace file
        close $nf
             #Execute nam on the trace file
        exec nam out.nam &
        exit 0
}

#Create two nodes
set n0 [$ns node]
set n1 [$ns node]

#Create a duplex link between the nodes
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

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

#Run the simulation
$ns run
```

*** BUT WE HAVE NO TRAFFIC HERE.**

Agents, applications and traffic sources

In NS-2, data is always being sent from one 'agent' to another. The most common agents used in NS-2 are UDP and TCP agents. In case of a TCP agent, several types are available. The most common agent types are:

- Agent/TCP – a Tahoe TCP sender
- Agent/TCP/Reno – a Reno TCP sender
- Agent/TCP/Sack1 – TCP with selective acknowledgement

The most common applications and traffic sources provided by NS-2 are:

- Application/FTP – produces bulk data that TCP will send
- Application/Traffic/CBR – generates packets with a constant bit rate
- Application/Traffic/Exponential – during off-periods, no traffic is sent. During on-periods, packets are generated with a constant rate. The length of both on and off-periods is exponentially distributed.
- Application/Traffic/Trace – Traffic is generated from a trace file, where the sizes and inter-arrival times of the packets are defined.

In addition to these ready-made applications, it is possible to generate traffic by using the methods provided by the class Agent. For example, if one wants to send data over UDP, the method *send(int nbytes)* can be used at the tcl-level provided that the udp-agent is first configured and attached to some node.

Below is a complete example of how to create a CBR traffic source using UDP as a transport protocol and attach it to node n0:

```
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packet_size_ 1000
$udp0 set packet_size_ 1000
$cbr0 set rate_ 1000000
```

An FTP application using TCP as a transport protocol can be created and attached to node n1 in much the same way:

```
set tcp1 [new Agent/TCP]
$ns attach-agent $n1 $tcp1
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$tcp1 set packet_size_ 1000
```

The UDP and TCP classes are both child-classes of the class Agent. With the expressions [new Agent/TCP] and [new Agent/UDP] the properties of these classes can be combined with the new objects udp0 and tcp1. These objects are then attached to nodes n0 and n1. Next, the application is defined and attached to the transport protocol. Finally, the configuration parameters of the traffic source are set. In case of CBR, the traffic can be defined by parameters rate_ (or equivalently interval_, determining the inter-arrival time of the packets), packetSize_ and random_. With the random_ parameter, it is possible to add some randomness in the inter-arrival times of the packets. The default value is 0, meaning that no randomness is added. Attachment of UDP and TCP agent to node is depicted in Fig.7.4 and 7.5 respectively.

Traffic Sinks

If the information flows are to be terminated without processing, the udp and tcp sources have to be connected with traffic sinks. A TCP sink is defined in the class Agent/TCPSink and a UDP sink is defined in the class Agent/Null.

A UDP sink can be attached to n2 and connected with udp0 in the following way:

```
set null [new Agent/Null]

$ns attach-agent $n2 $null

$ns connect $udp0 $null
```

A standard TCP sink that creates one acknowledgement per a received packet can be attached to n3 and connected with tcp1 with the commands:

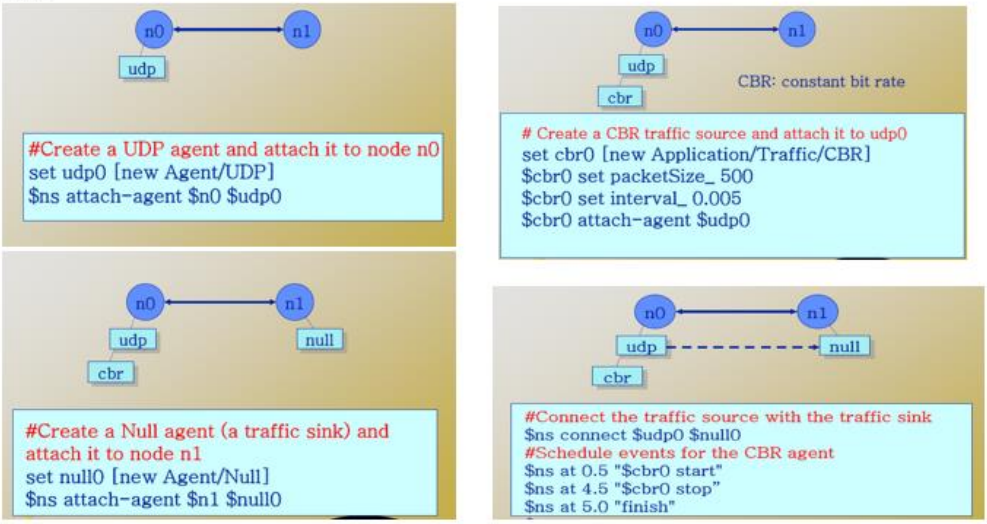
```
set sink [new Agent/Sink]

$ns attach-agent $n3 $sink

$ns connect $tcp1 $sink
```

There is also a shorter way to define connections between a source and the destination with the command:

```
$ns create-connection <srctype> <src> <dsttype> <dst> <pktclass>
```

For example, to create a standard TCP connection between n1 and n3 with a class ID of 1: *$ns create-connection TCP $n1 TCPSink $n3 1*

UDP



```
#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
```

```
# Create a CBR traffic source and attach it to udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

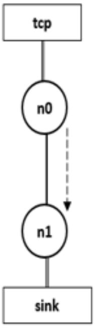
```
#Create a Null agent (a traffic sink) and
attach it to node n1
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0
```

```
#Connect the traffic source with the traffic sink
$ns connect $udp0 $null0
#Schedule events for the CBR agent
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
$ns at 5.0 "finish"
```

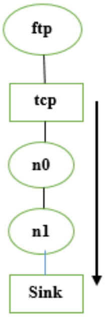
Fig.7.4: UDP Agent attachment to node

TCP

```
# defining agents
set tcp [new Agent/TCP]
set tcpsink [new Agent/TCPSink]
$ns attach-agent $n0 $tcp
$ns attach-agent $n1 $tcpsink
$ns connect $tcp $tcpsink
```



```
#Creating Traffic
#FTP
set ftp [new Application/FTP]
$ftp attach-agent $tcp
#Telnet
set telnet [new Application/Telnet]
$telnet attach-agent $tcp
```



Marking Flows

```
#Define different colors for data flows (for NAM)
$ns color 1 Blue
$ns color 2 Red
#marking the flows
$tcp set fid_ 1
$udp set fid_ 2
```

Additional Functions

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

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

Creating Toplogies

```
#to gain more control in the NAM layout the topology can be
set.
$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
```

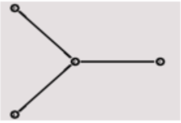


Fig. 7.5: TCP Agent attachment to node and additional functions in Tcl script.

Controlling the simulation

After the simulation topology is created, agents are configured etc., the start and stop of the simulation and other events have to be scheduled. The simulation can be started and stopped with the commands

```
$ns at $simtime “finish”
$ns run
```

The first command schedules the procedure finish at the end of the simulation, and the second command actually starts the simulation. The finish procedure has to be defined to flush the trace buffer, close the trace files and terminate the program with the exit routine. It can optionally start NAM (a graphical network animator), post processing information and plot this information.

The finish procedure has to contain at least the following elements:

```
proc finish {} {
    global ns trace_all
    $ns flush-trace
    close $trace_all
    exit 0
}
```

Support software

Nam – VINT/LBL Network Animator

Nam is a Tcl/TK based animation tool for viewing network simulation traces and real world packet trace data. It supports topology layout, packet level animation, and various data inspection tools.

The first step to use Nam is to produce a trace file. The trace file should contain topology information, for example nodes, links and packet traces. During an NS-2 simulation, user can produce topology configurations, layout information and packet traces using tracing events in NS2.

When the trace file is generated, it is ready to be animated by Nam. Upon startup, Nam will read the trace file, create topology, pop up a window, do layout if it is necessary and then pause at the time of the first packet in the trace file. Nam provides control over many aspects of animation through its user interface. Nam does animation using the following building blocks: node, link, queue, packet, agent and monitor. Fig. 7.6 shows the screenshot of NAM window and features available.

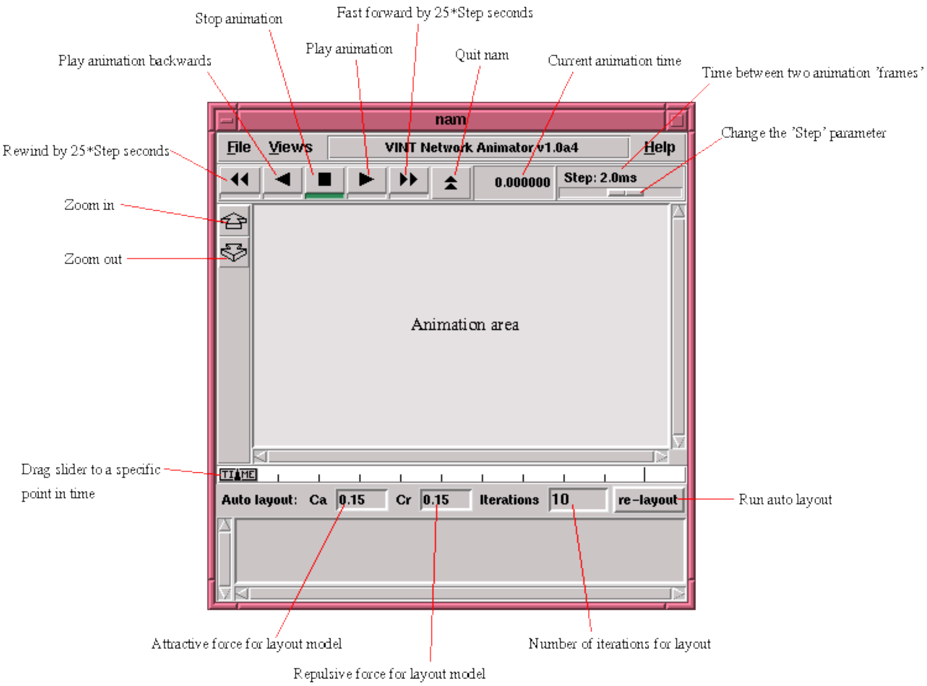

Stop animation
Fast forward by 25*Step seconds
Play animation
Play animation backwards
Quit nam
Current animation time
Time between two animation 'frames'
Change the 'Step' parameter
Rewind by 25*Step seconds
Zoom in
Zoom out
nam
File
Views
VINT Network Animator v1.0a4
Help
0.000000
Step: 2.0ms
Animation area
Drag slider to a specific point in time
TIME
Auto layout:
Ca
0.15
Cr
0.15
Iterations
10
re-layout
Run auto layout
Attractive force for layout model
Number of iterations for layout
Repulsive force for layout model

Fig.7.6: Nam window

Commands that can be used on NAM

$ns color fid color

This is used set color of the packets for a flow specified by the flow id (fid). This member function of "Simulator" object is for the NAM display, and has no effect on the actual simulation.

$ns simplex-link-op/duplex-link-op [node-instance1] [node-instance2] orient [orientation]

Position of links can be defined for better representation of network in NAM using command as mentioned above. Note that the specified orientation applied to second node position respect to first.

Ex: $ns duplex-link-op $n0 $n1 orient down

Steps to create and execute Tcl script

1. Open a text editor and write the codes there and save it with .tcl file extension (‘gedit filename.tcl’)
2. To execute the file, open terminal and navigate to the folder where the saved file is present and run **ns filename**
3. Step 2 automatically creates the .nam and .tr files in the same folder. Optionally, to execute the nam file following command can be used in the terminal
   nam filename.nam

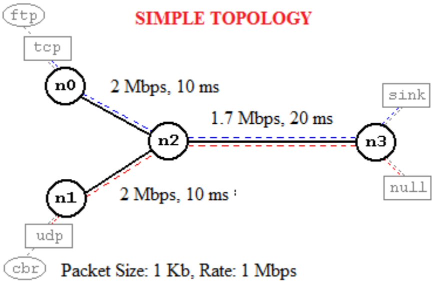
Additional Options

- Set a Routing Protocol: $ns rtproto *name_of_routing_protocol (DV for distance vector and LS for Link State)*
  Note: For Static routes do not use this command.
- Simulate Node Failure/Restoration: $ns rtmodel-at *[time]* [down/up] *[node_id]*
  Examples:
  $ns rtmodel-at 10.2 down $n16 #Node failure
  $ns rtmodel-at 43.5 up $n16 #Node restoration
- Simulate Link Failure/Restoration: $ns rtmodel-at *[time]* [down/up] *[node_id_from node_id_to]*

Sample Program

Given below is a complete code:

```
#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
```

Lab exercises

1. Consider a network with five nodes n0, n1, n2, n3, n4 forming a star topology. The node n4 is at the center. Node n0 is a UDP source, which transmits packets to node n3 (a UDP sink) through the node n4. Node n1 is another traffic source, and sends UDP packets to node n2 through n4. The duration of the simulation time is 10 seconds. Write a TCL script to simulate this scenario.

2. Write a TCL script to set up a simple TCP_FTP, with four nodes, where n0, n2 are source nodes and n1, n3 are sink nodes. Duplex-link is of 1Mb 10ms. Assign

different flow colors. Arrange these four nodes in square layout. Demonstrate link failure scenario between n0 and n1.

Additional exercise

1. Write a TCL script to set up a LAN, with 12 nodes connected to each other as shown in Fig.7.5. Assign different colors for the flow and nodes. Assign TCP traffic from any one server node to all other nodes outside server pool.

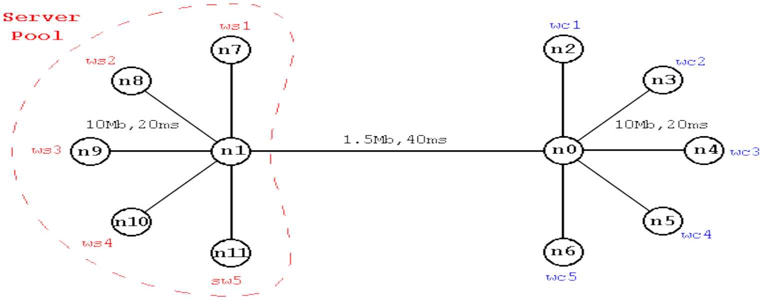


Fig.7.7: Topology for additional exercise

[OBSERVATION SPACE – LAB 7]

LAB NO: 8 **Date:**

WIRELESS SIMULATION

Objectives

- To interpret different parameters needed to configure a wireless node.
- To simulate a wireless network scenario and calculate the network traffic parameters.

Introduction

NS-2 is more prominently used to simulate wireless network scenario. It supports wireless sensor networks, Vehicular Adhoc Network (VANETs) and all IEEE 802.11 services. The major difference between a wired and a wireless network scenario creation in NS-2 is that, in wireless network scenario, every node should have a transmission range in wireless network created using god object. Every wireless node position is defined using random generation or grid model or any such equivalent models and node movement is scheduled as an event.

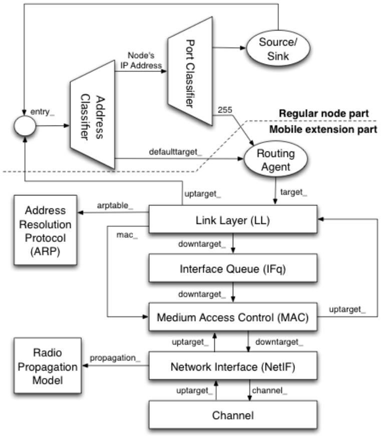


Fig.8.1: Architecture of mobile node.

Mobile node in NS-2 is an extension of a regular node. Architecture of Mobile node in NS-2 is shown in Fig. 8.1.The part below the dotted line in Fig: 8.1 is the extension consisting of several blocks as described in Table 8.1.

Table 8.1: Parameters associated with mobile node in NS-2

Parameter	Description
Routing agent	It tells the node how a packet should be transmitted. It works with routing protocol responsible for propagating routing information throughout the networks.
Link layer	Models bandwidth, packet transmission time, propagation delay and so on.
Address Resolution Protocol (ARP)	This module translates hardware addresses into network (i.e., IP) addresses
Interface queue	This module models buffer management.
Medium Access Control (MAC)	Models the MAC layer.
Network interface	It is in this module where actual transmission takes place. It works with the radio propagation model to simulate packet transmission error.
Channel	A mobile node puts the packet being transmitted on this module. The destination node reads the channel and picks up the packet belonging to itself from this channel

Steps for mobile networking simulation

1. Create Simulator object
2. Create god
3. Create topography object
4. Configure attributes for wireless node
5. Attach agent & application
6. Connect them

General Operation Director

Two nodes can transmit data to each other if those nodes lies in range of each other. If a node moves away from range of another node in this case connection will be terminated. For this purpose NS-2 has general operation director (GOD).

GOD (general operation director) keeps information of each node in wireless simulation. For every wireless simulation GOD need to be created. At time of GOD creation number of nodes needs to be specified so that GOD can reserve memory space for those nodes for keeping their information.

Syntax:

```
create-god <number_of_nodes>
```

Topology

To determine area in which mobile host can move, topography class is used.

Syntax:

```
  # Create object of Topography class
  set topo [new Topography]
  # Specify boundary of area in which node can move
  $topo load_Flatgrid <X> &tl;Y> res
```

Size of grid is specified in x and y direction and res passed as grid resolution.

Wireless Node Configuration

In NS-2, node can be configured to support wireless transmission using **node-config** function of Simulator class.

Syntax :

```
$ns node-config -addressingType <usually flat or hierarchical used for wireless topologies> \
                -adhocRouting   <adhoc routing protocol like DSR, AODV, DSDV etc> \
                -llType         <LinkLayer> \
                -macType        <MAC type like Mac/802_11> \
                -propType       <Propagation model like Propagation/TwoRayGround> \
                -ifqType        <interface queue type like Queue/DropTail/PriQueue> \
                -ifqLen         <interface queue length like 50> \
                -phyType        <network inteface type like Phy/WirelessPhy> \
                -antType        <antenna type like Antenna/OmniAntenna> \
                -channel        <Channel type like Channel/WirelessChannel> \
                -topoInstance   <the topography instance> \
                -wiredRouting   <turning wired routing ON or OFF> \
                -mobileIP       <setting the flag for mobileIP ON or OFF> \
                -energyModel    <EnergyModel type> \
                -initialEnergy  <specified in Joules> \
                -rxPower        <specified in W> \
                -txPower        <specified in W> \
                -agentTrace     <tracing at agent level turned ON or OFF> \
                -routerTrace    <tracing at router level turned ON or OFF> \
                -macTrace       <tracing at mac level turned ON or OFF> \
                -movementTrace  <mobilenode movement logging turned ON or OFF>
```

Options for node configuration

Table 8.2: Options for node Configuration for mobile node in NS-2

Option	Available Values	Remarks
addressType	flat, hierarchical	
MPLS	ON,OFF	Multi protocol Label Switching
wiredRouting	ON, OFF	
llType	LL, LL/Sat	Link Layer
macType	Mac/802_11, Mac/Csma/Ca, Mac/Sat, Mac/Sat/UnslottedAloha, Mac/Tdma	Medium Access Control
ifqType	Queue/DropTail, Queue/DropTail/PriQueue	Interface Queue type
phyType	Phy/wirelessPhy,Phy/Sat	Physical Layer Type
adhocRouting	Diffusion/Rate, Diffusion/Prob, Dsdv, Dsr, Flooding, Omnimcast,Aodv,Tora,Puma	adhocrouting protocol
propType	Propagation/TwoRayGround, Propagation/Shadowing	Propagation Type
antType	Antenna/OmniAntenna,	Antenna type
Channel	Channel/WirelessChannel, Channel/Sat	Channel to be used
mobileIP	ON,OFF	to set the IP for Mobile or not
energyModel	EnergyModel	energy model to be enabled or not
initialEnergy	<joule>	in terms of joules (Ex: 3.24)
txPower		Power in terms of Watts (0.32)
rxPower		Power in terms of Watts (0.1)

idlePower		Power in terms of Watts (0.02)
agentTrace	ON, OFF	Tracing to be on or off
routerTrace	ON, OFF	Tracing to be on or off
macTrace	ON, OFF	Tracing to be on or off
movementTrace	ON, OFF	Tracing to be on or off
errProc	Uniform Error Proc	
toraDebug	ON, OFF	

Mobile Node Movement

For moving a node during a simulation following commands are used.

Random motion

Node can move randomly if their random-motion property is on.

Syntax :

```
                    $[node-instance] random-motion 0 or 1
```

Random motion will be turned on if 1 is specified, in case of 0 it will be off.

Setting Position

Position(X,Y,Z) of node can be set in ns2. Z coordinates of position is 0. Following command is used for setting position

Syntax :

$[node-instance] set X_ <x-coordinates> $[node-instance] set Y_ <y-coordinates> $[node-instance] set Z_ <z-coordinates>

Setting Movement

Future position can be set by following command.

Syntax :

$[simulator-instance] at $[node-instance] setdest <X> <Y> <sp> # Example $ns at 1.2 $n0 setdest 300 200 20 # In above command node start moving to position 300,200 at time 1.2 with 20 speed.

When above command executed, node starts moving to specified position with *sp* speed**.**

Setting Radius of Node

Range of wireless node can changed using following command.

# Syntax: [node-instance] radius <r> $n0 radius 20 # above command will set radius for node 0 as 20

The radius denotes the node's range. All mobile nodes that fall within the circle of radius with the node at its center are considered as neighbours.

Setting distance between nodes

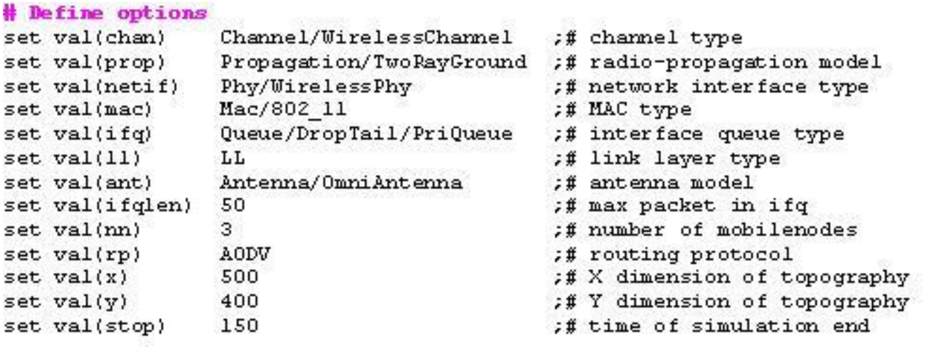
Distance between nodes can be changed for routing protocols by using following command

# Syntax # [god-instance] set-dist <node_number> <node_number> <hop> $god set-dist 0 1 2

Above command will set distance between node 0 and 1 as 2 hops.

Sample Program

#TCP Wireless communication between nodes in NS2



```
# Define options
set val(chan)      Channel/WirelessChannel     ;# channel type
set val(prop)      Propagation/TwoRayGround    ;# radio-propagation model
set val(netif)     Phy/WirelessPhy             ;# network interface type
set val(mac)       Mac/802_11                  ;# MAC type
set val(ifq)       Queue/DropTail/PriQueue     ;# interface queue type
set val(ll)        LL                          ;# link layer type
set val(ant)       Antenna/OmniAntenna         ;# antenna model
set val(ifqlen)    50                          ;# max packet in ifq
set val(nn)        3                           ;# number of mobilenodes
set val(rp)        AODV                        ;# routing protocol
set val(x)         500                         ;# X dimension of topography
set val(y)         400                         ;# Y dimension of topography
set val(stop)      150                         ;# time of simulation end
```

```
set ns [new Simulator]
#creating trace file and nam file
set tracefd [open bhapith.tr w]
set windowVsTime2 [open win.tr w]
set namtrace [open bhapith.nam w]

$ns trace-all $tracefd
$ns namtrace-all-wireless $namtrace $val(x) $val(y)

# set up topography object
set topo       [new Topography]

$topo load_flatgrid $val(x) $val(y)

create-god $val(nn)
```

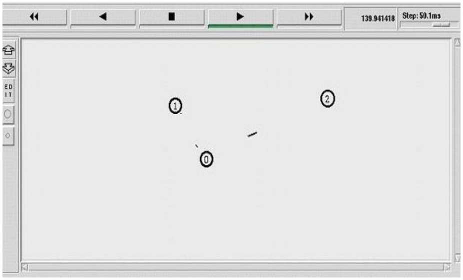
```
# configure the nodes
        $ns node-config -adhocRouting $val(rp) \
                   -llType $val(ll) \
                   -macType $val(mac) \
                   -ifqType $val(ifq) \
                   -ifqLen $val(ifqlen) \
                   -antType $val(ant) \
                   -propType $val(prop) \
                   -phyType $val(netif) \
                   -channelType $val(chan) \
                   -topoInstance $topo \
                   -agentTrace ON \
                   -routerTrace ON \
                   -macTrace OFF \
                   -movementTrace ON

      for {set i 0} {$i < $val(nn) } { incr i } {
          set node_($i) [$ns node]   }

# Provide initial location of mobilenodes
$node_(0) set X_ 5.0
$node_(0) set Y_ 5.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 490.0
$node_(1) set Y_ 285.0
$node_(1) set Z_ 0.0

$node_(2) set X_ 150.0
$node_(2) set Y_ 240.0
$node_(2) set Z_ 0.0

# Generation of movements
$ns at 10.0 "$node_(0) setdest 250.0 250.0 3.0"
$ns at 15.0 "$node_(1) setdest 45.0 285.0 5.0"
$ns at 19.0 "$node_(2) setdest 480.0 300.0 5.0"

# Set a TCP connection between node_(0) and node_(1)
set tcp [new Agent/TCP/Newreno]
$tcp set class_ 2
```

```
set sink [new Agent/TCPSink]
$ns attach-agent $node_(0) $tcp
$ns attach-agent $node_(1) $sink
$ns connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns at 10.0 "$ftp start"

set tcp [new Agent/TCP/Newreno]
$tcp set class_ 2
set sink [new Agent/TCPSink]
$ns attach-agent $node_(1) $tcp
$ns attach-agent $node_(2) $sink
$ns connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns at 10.0 "$ftp start"

# Printing the window size
proc plotWindow {tcpSource file} {
global ns
set time 0.01
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
puts $file "$now $cwnd"
$ns at [expr $now+$time] "plotWindow $tcpSource $file" }
$ns at 10.0 "plotWindow $tcp $windowVsTime2"

# Define node initial position in nam
for {set i 0} {$i < $val(nn)} { incr i } {
# 30 defines the node size for nam
$ns initial_node_pos $node_($i) 30
}

# Telling nodes when the simulation ends
for {set i 0} {$i < $val(nn) } { incr i } {
   $ns at $val(stop) "$node_($i) reset";
}

# ending nam and the simulation
```

```
$ns at $val(stop) "$ns nam-end-wireless $val(stop)"
$ns at $val(stop) "stop"
$ns at 150.01 "puts \"end simulation\" ; $ns halt"
proc stop {} {
    global ns tracefd namtrace
    $ns flush-trace
    close $tracefd
    close $namtrace
exec nam bhapith.nam &
}
```

```
$ns run
```

Output:



Lab exercises

1. Set up a wireless network with mobile node N2 between N0 and N3. As the nodes N0 and N3 moves towards each other they exchange packets. As they move out of each other’s range they drop some packets. Vary the transmission range of the mobile nodes and the location to visualize packet transfer in different scenario.

2. Simulate a hierarchical layout of wireless sensor nodes as shown in Fig.8.2. There is only one sink node ‘n0’ and all other nodes are source. Define the packet flow such that layer 1 nodes send data to layer 2 nodes which in turn send data to sink node. Assign suitable position in NAM and implement packet flow between any two source nodes with sink.

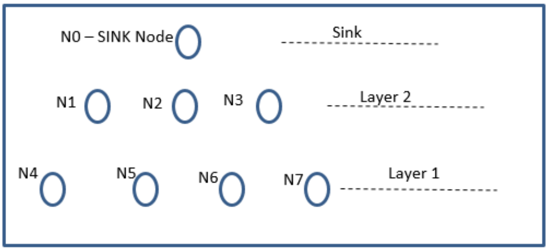


Fig. 8.2: Hierarchical layout of wireless nodes

Additional exercise

1. Simulate a wired-cum-wireless scenario consisting of 2 wired nodes connected to wireless domain through a base-station node. Assume TCP packet transfer between nodes and base station for this scenario.

[OBSERVATION SPACE – LAB 8]

LAB NO: 9 **Date:**

ANALYSIS USING TRACE FILE

Objectives

- To interpret the trace file generated by executing a wired Tcl script in NS-2.
- To interpret the trace file generated by executing a wireless Tcl script in NS-2.
- To calculate various network performance measurement parameters using AWK scripts.

Introduction

The output after executing a Tcl script is generally visualization of the network scenario using NAM file. Alternately, the output can be trace file which holds a log of entire simulation. Trace files can be used to measure some of the performance measurement metrics such as those listed below:

Latency: It can take a long time for a packet to be delivered across intervening networks. In reliable protocols where a receiver acknowledges delivery of each chunk of data, it is possible to measure this as round-trip time.

Packet loss: In some cases, intermediate devices in a network will lose packets. This may be due to errors, to overloading of the intermediate network, or to intentional discarding of traffic in order to enforce a particular service level.

Retransmission: When packets are lost in a reliable network, they are retransmitted. This incurs two delays: First, the delay from re-sending the data; and second, the delay resulting from waiting until the data is received in the correct order before forwarding it up the protocol stack.

Throughput: The amount of traffic a network can carry is measured as throughput, usually in terms such as kilobits per second. Throughput is analogous to the number of lanes on a highway, whereas latency is analogous to its speed limit.

Different parameters can together or independently determine how well a network would perform. A few such are mentioned below:

Bandwidth: It is the maximum data transfer rate which a link allows. It is expressed in bits per seconds (bps).

Propagation Delay: It is the amount of time required to for a packet to travel from one node to another. If the propagation delay is high then throughput will be low i.e they are inversely proportional to each other.

Queue type and queue size: The queue of a node is implemented as a part of a link whose input is that node to handle the overflow at the queue. But if the buffer capacity of

the output queue is exceeded then the last packet arrived is dropped. We do set the buffer capacity by using queue size.
This section introduces the interpretation of trace file and monitors.

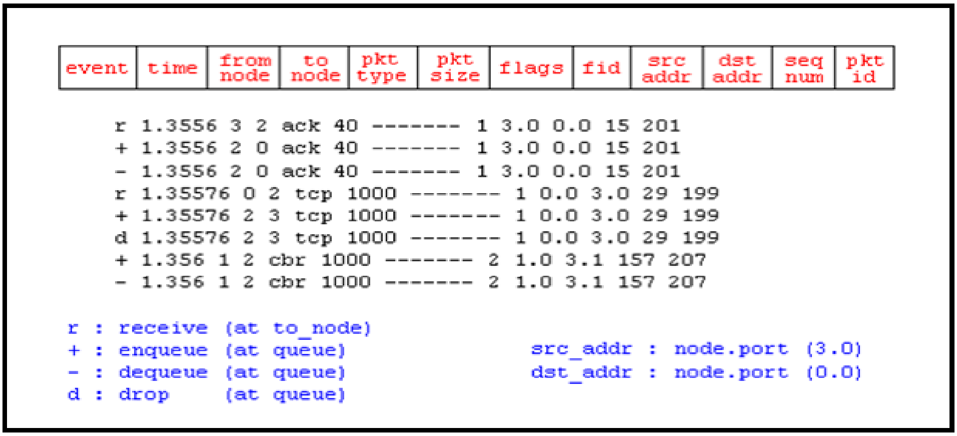
Tracing and monitoring

In order to be able to calculate the results from the simulations, the data have to be collected somehow. NS-2 supports two primary monitoring capabilities: traces and monitors. The *traces* enable recording of packets whenever an event such as packet drop or arrival occurs in a queue or a link. The *monitors* provide a means for collecting quantities, such as number of packet drops or number of arriving packets in the queue. The monitor can be used to collect these quantities for all packets or just for a specified flow (a flow monitor).

Tracing Objects NS-2 can produce visualizations trace as well as ASCII file corresponding to the events that are registered on the network. While tracing NS inserts four objects: EnqT, DeqT, RecvT & DrpT. EnqT registers information regarding the arrival of packet and is queued at the input queue of the link. When overflow of a packet occurs, then the information of the dropped packet is registered in DrpT. DeqT holds the information about the packet that is dequeued instantly. RecvT hold the information about the packet that has been received instantly.

Structure of Trace files

Fig. 9.1 shows the format of a trace file and its description is given in table 9.1



event	time	from node	to node	pkt type	pkt size	flags	fid	src addr	dst addr	seq num	pkt id

```
r 1.3556 3 2 ack 40 ------- 1 3.0 0.0 15 201
+ 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
- 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
r 1.35576 0 2 tcp 1000 ------- 1 0.0 3.0 29 199
+ 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
d 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
+ 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
- 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
```

```
r : receive (at to_node)
+ : enqueue (at queue)            src_addr : node.port (3.0)
- : dequeue (at queue)            dst_addr : node.port (0.0)
d : drop    (at queue)
```

Fig. 9.1: Format of trace file

Table 9.1: Description of various fields in trace

Field Name	**Description**
Event	It gives four possible symbols '+' '-' 'r' 'd'. These four symbols correspond respectively to enqueue, dequeued, received and dropped.
Times	Specifies the time at which the event occurs
From node	The input node of the link at which the event occurs
To node	The output node at which the event occurs
Pkt Type	Information about the packet type i.e., whether the packet is UDP or TCP
Pkt Size	Specifies the size of the packet
Flags	Provides information about some flags
Fid	flow id (fid) for IPv6 that a user can set for each flow in a Tcl script.
Src addr	Source Address
Dst addr	Destination Address
Seq. No.	Network layer protocol's packet sequence number
Pkt id	Unique id of the packet

Syntax for creating trace files

# Open the trace file

1. set nf [open out.tr w]
2. $ns trace-all $n

which means we are opening a newtrace file named as "out" and also telling that data must be stored in .tr [trace] format.

"nf" is the file handler that we are used here to handle the trace file

"w" means write i.e the file out.tr is opened for writing,
>> "r" means reading and "a" means appending

Line number 2 tells the simulator to trace each packet on every link in the topology and for that we give file handler nf for the simulator ns.

```
# Define finish procedure
proc finish {} {
global ns nf
$ns flush-trace
close nf
exit 0
}
```

Here, the trace data is flushed into the file by using command $ns flush-trace and then file is closed. In order to generate a trace file, we have to create a trace file in OTcl script. The trace file can be viewed with the cat command:

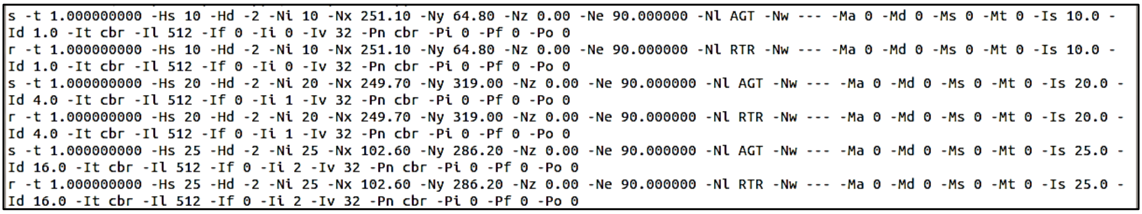
cat out.tr

While running the NS-2 program via terminal, trace file is generated in the selected directory (folder or directory where the program stored). Tracing all events from a simulation to a specific file and then calculating the desired quantities from this file for instance by using Perl or awk and Matlab is an easy and suitable way when the topology is relatively simple and the number of sources is limited. However, with complex topologies and many sources this way of collecting data can become too slow. The trace file will also consume a significant amount of disk space.

New Trace file format for wireless scenario

Fig.9.2 shows the sample trace file format for wireless scenario using the command

$ns use-newtrace



```
s -t 1.000000000 -Hs 10 -Hd -2 -Ni 10 -Nx 251.10 -Ny 64.80 -Nz 0.00 -Ne 90.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 10.0 -
Id 1.0 -It cbr -Il 512 -If 0 -Ii 0 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
r -t 1.000000000 -Hs 10 -Hd -2 -Ni 10 -Nx 251.10 -Ny 64.80 -Nz 0.00 -Ne 90.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 10.0 -
Id 1.0 -It cbr -Il 512 -If 0 -Ii 0 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
s -t 1.000000000 -Hs 20 -Hd -2 -Ni 20 -Nx 249.70 -Ny 319.00 -Nz 0.00 -Ne 90.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 20.0 -
Id 4.0 -It cbr -Il 512 -If 0 -Ii 1 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
r -t 1.000000000 -Hs 20 -Hd -2 -Ni 20 -Nx 249.70 -Ny 319.00 -Nz 0.00 -Ne 90.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 20.0 -
Id 4.0 -It cbr -Il 512 -If 0 -Ii 1 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
s -t 1.000000000 -Hs 25 -Hd -2 -Ni 25 -Nx 102.60 -Ny 286.20 -Nz 0.00 -Ne 90.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 25.0 -
Id 16.0 -It cbr -Il 512 -If 0 -Ii 2 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
r -t 1.000000000 -Hs 25 -Hd -2 -Ni 25 -Nx 102.60 -Ny 286.20 -Nz 0.00 -Ne 90.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 25.0 -
Id 16.0 -It cbr -Il 512 -If 0 -Ii 2 -Iv 32 -Pn cbr -Pi 0 -Pf 0 -Po 0
```

Fig. 9.2: Screenshot of wireless trace format

Description of each field is given below in brief:

s - send packet
r - received packet
d - packet dropped
f - packet forwarded
c - collision of packet at MAC level
t - time at which packet tracing started
Hs - ID of the hop
Hd - ID of the next hop towards destination
Ni - Node ID
Nx,Ny,Nz - Coordinates that the nodes situated
Ne - Node energy level
Nl - Trace level
Nw - Reason of the event
AGT - Agent
RTR -Routing
END - DROP End of Simulation

MAC LAYER Information
Ma - MAC Layer duration
Md - Destn. Ethernet Address
Ms - Source Ethernet Address
Mt - Ethernet Type

PACKET Information
P arp - address resolution protocol-Po - ARP Request / Reply
Pm - Source MAC Address
Ps - Source Address
Pa - Destination MAC Address
Pd - Destination Address
Pn - Nodes Transversed
Pq – Flag

PACKET Information
Pi - Route Request Sequence Number/ Sequence Number
Pp - Flag
Pl - Reply Length
Pe - src of source routing
Pw - Error Report Flag
Pc - Report to whom

COL - DROP MAC COLLISION
DUP - DROP MAC DUPLICATE
DERR - DROP MAC PACKET ERROR
RET - DROP MAC RETRY COUNT EXCEED
STA - DROP MAC INVALID STATE
BSY - DROP MAC BUSY
NRTE - DROP RTR - NO ROUTE
LOOP - DROP RTR ROUTE LOOP
TTL - DROP RTR TTL has reached Zero
TOUT - DROP-RTR-QTIME OUT Expired
Is - Source address of source port
Id - Destination address of destination port
Il - Packet Size
If - Flow ID
Ii - Unique ID
Iv - TTL value next hop into

Pb - Link error from link a to link b
P cbr - CBR data
Pf - How many level packet leave
Po - Optimal Number of Forward
P TCP - TCP flow
Ps - seq. number
Pu - acknowledgement
Pf - Packet Failure

Writing simple AWK scripts for calculating result

AWK Scripts are very good in processing the data from the log (trace files) which we get from NS2. Command to run the awk script in Linux is
gawk –f filename.awk filename.tr

A sample AWK script to compute throughput is given in next section.

```
BEGIN {
recvdSize = 0
startTime = 400
stopTime = 0
}

{
event = $11
time = $2
node_id = $3
pkt_size = $8
level = $4

# Store start time
if (level == "AGT" &;& event == "s" && pkt_size >= 512) {
   if (time <; startTime) {
   startTime = time
   }
}
# Update total received packets' size and store packets arrival time
if (level == "AGT" &;& event == "r" && pkt_size >= 512) {
if (time >; stopTime) {
stopTime = time
}
# Rip off the header
hdr_size = pkt_size % 512
pkt_size -= hdr_size
# Store received packet's size
recvdSize += pkt_size
}
}
END {
printf("Average Throughput[kbps] = %.2f\t\t
StartTime=%.2f\tStopTime=%.2f\n",(recvdSize/(stopTime-
startTime))*(8/1000),startTime,stopTime)
 }
```

Lab Exercises

1. Write a NS-2 program to simulate both TCP and UDP packets in a single network with 10 mobile nodes. Node 0 is the source and all other nodes are destination. Create a unicast packet from node 0 to node 2 (TCP) and node 1 (UDP). Calculate packet drop ratio, number of packets delivered to the destination successfully from the trace file. [Use suitable Linux/Unix command to count the number of entries]
2. Consider a dumbbell topology with eight nodes as shown as in the Fig.9.3. Consider nodes 2 and 3 to be two routers connecting two different networks. Assume node 0 is running a FTP application (over TCP) and sending data to node 6. Node 1 is sending CBR data node 7. Assume all the links except 2-3 has a bandwidth of 1 Mb, propagation delay of 10ms and queue type as DropTail. (All are duplex links). The link 2-3 has a propagation delay of 10 ms. Vary it's bandwidth from 0.5 Mb to 2.5 Mb in steps of 0.25Mb.
   Compute the throughput for node 3 in each case using AWK scripts. [When the bandwidth of the link 2-3 is much lower than the sum of bandwidths of the other links in the network, it act as a bottleneck.]

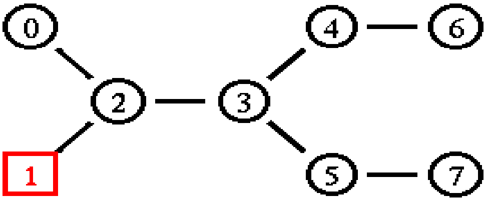


Fig. 9.3: Topology of LAN

Additional exercise

1. Set up the topology as shown in Fig. 9.4 with 7 nodes, and demonstrate the working of link failure (static). The link between 1 and 2 breaks at 1.0ms and comes up at 2.0ms. Assume that the source node 0 transmits packets to node 3. Assume your own parameters for bandwidth, delay etc. Compute the packet drop ratio.

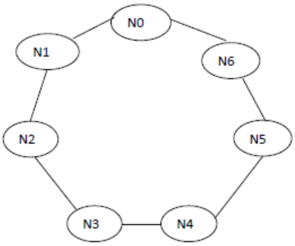


Fig. 9.4: Ring topology

Useful Command:

```
grep " 3 2 " out.tr > outa.tr
grep -c " 3 2 " out.tr > outa.tr
```

where,
-c : count of lines that satisfy the cases
" 3 2 " : case to filter out
out.tr : input trace file
outa.tr : output trace file

[OBSERVATION SPACE – LAB 9]

LAB NO: 10 **Date:**

GRAPHICAL ANALYSIS

Objectives

1. To compare performance of various routing protocols using NS-2 for wireless network scenario using GNUPLOT.
2. To generate Tcl scripts using NSG.

Introduction

Any network performance analysis can be done easily using graphical method. GNUPLOT is one of the widely used tool to plot graph using scripts in LINUX platform. To use grnuplot, type ‘gnuplot ‘ from the LINUX prompt. The syntax for plotting a graph is shown below

Gnuplot> plot “filename” using X-axis_col_no **:** *Y-axis_col_no title ‘sample_text’ with lines*

Plots may be displayed in one of eight styles: lines, points, linespoints, impulses, dots, steps, fsteps, histeps, errorbars, xerrorbars, yerrorbars, xyerrorbars, boxes, boxerrorbars, boxxyerrorbars, financebars, candlesticks or vector. Use any one of the following command to plot the graph available in a data file ‘d1.dat’

Plot a function. To plot a function instead of a data file, try

plot sin(x) with lines

Plot in a specified range. To plot a data set in the range 0 < x < 10 and 0.2 < y < 0.6, try

plot [0,10] [0.2,0.6] "d1.dat" with lines

Plot with points. To plot with points, or to plot with lines and points, use the following forms

plot "d1.dat" with points

plot "d1.dat" with linespoints

Plot different columns of datafile. For example, to take x from column 4 and y from column 2, plot "data.dat" using 4:2 with lines

Plot several functions at once. To plot d1.dat against the line y(x)=exp(-x), try

plot "d1.dat" with lines, exp(x) with lines

Change the Plot Title. To change the name that appears in the 'key' in the corner of the plot, try

plot "d1.dat" title 'Sample Data' with lines

Performance Analysis of different routing protocols

In a wireless network, routing algorithms are used by each node to find an optimum route to the destination. A routing protocol specifies how routers communicate with each other, disseminating information that enables them to select routes between any two nodes on a computer network. Routing algorithms determine the specific choice of route. Each router has a priori knowledge only of networks attached to it directly. A routing protocol shares this information first among immediate neighbors, and then throughout the network. This way, routers gain knowledge of the topology of the network.

Every network routing protocol performs three basic functions:

1. *discovery* - identify other routers on the network
2. *route management* - keep track of all the possible destinations (for network messages) along with some data describing the pathway of each
3. *path determination* - make dynamic decisions for where to send each network message

A few routing protocols (called *link state protocols*) enable a router to build and track a full map of all network links in a region while others (called *distance vector protocols)* allow routers to work with less information about the network area. Gnuplot can be used to measure the performance of various routing protocols with varying number of nodes in the network and the link stability.

Generating tcl script using NS G

Assigning movement and packet flow with large number of nodes in a network is cumbersome. NS – G is used to generate the tcl script for wired or wireless scenario. Some of the main features of NS2 Scenarios Generator (NSG) are as mentioned below:

- Creating Wired and Wireless nodes just by drag and drop.
- Creating Simplex and Duplex links for wired network.
- Creating Grid, Random and Chain topologies.
- Creating TCP and UDP agents. Also supports TCP Tahoe, TCP Reno, TCP New-Reno and TCP Vegas.
- Supports Ad Hoc routing protocols such as DSDV, AODV, DSR and TORA.
- Supports FTP and CBR applications.
- Supports node mobility.
- Setting the packet size, start time of simulation, end time of simulation, transmission range and interference range in case of wireless networks, etc.
- Setting other network parameters such as bandwidth, etc for wireless scenarios

Fig. 10.1 shows the GUI of NSG. Suitable mobility can be generated using the menu driven options in NSG and the tcl script can be run using NS-2 commands.

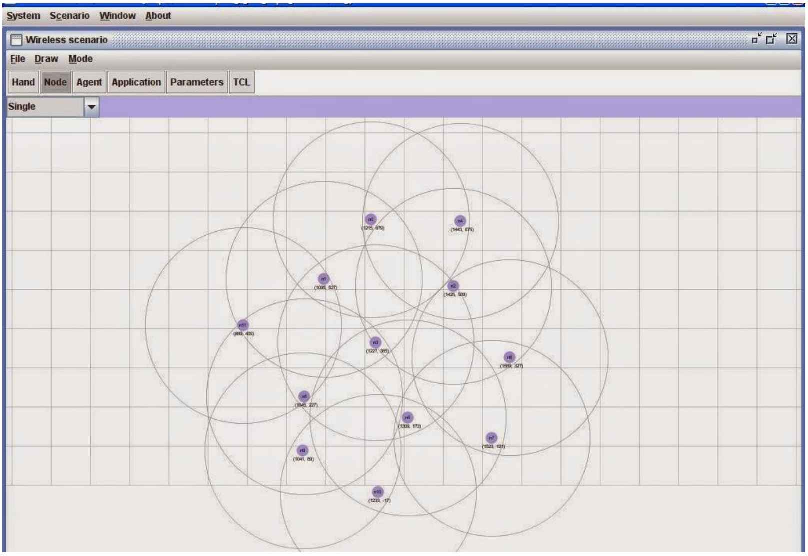


Fig. 10.1: GUI of NSG

Lab exercise

1. Implement a wireless network scenario in NS-2 with 10 nodes communicating with each other using AODV, DSDV and DSR routing protocols. Plot the graph of packet delivery ratio for each routing protocol. Repeat the same using 15 nodes and 20 nodes in same geographical layout. Compare the performance of each of the routing protocols by plotting a graph of throughput and packet delivery ratio v/s number of nodes in the network.
2. Consider a source node, a destination node, and an intermediate router (respectively as shown in Fig.10.2). The link between nodes S and R (Link-1) has a bandwidth of 1Mbps and 50ms latency. The link between nodes R and D (Link-2) has a bandwidth of 100kbps and 5ms latency. Vary the source data rate (40kbs, 80 kbps, 120 kbps and 160 kbps) and answer the following using graphical approach.
   a. At what point does it get congested?
   b. How do the throughput and loss vary as a function of the source data rate?

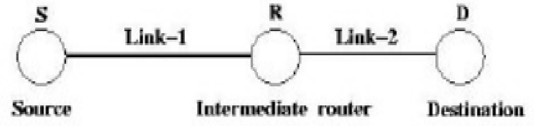


Fig. 10.2: Topology for additional exercise

Additional exercise

1. Create a wireless network scenario to simulate a VANET (Vehicular Adhoc Network) where the 20 vehicles are communicating with each other at different time intervals. Set the node positions to illustrate a two lane street in a plane surface of 100 X 100 unit grid as shown in Fig. 10.3. As the nodes are allowed to move in their respective lanes, they are allowed to exchange data with vehicles from same lane for the duration that the node exists in the network. Calculate the total number of packets sent and received by each node and plot a graph to depict the same.

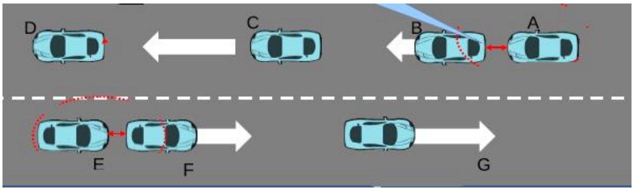


Fig. 10.3: Vehicle to Vehicle communication scenario.

[OBSERVATION SPACE – LAB 10]

LAB NO: 11 **Date:**

ADDITION AND CONFIRGURATION OF NEW PACKAGES TO NS-2

Objectives

- To extend the existing protocols by modifying backend dependent .cc files.
- To integrate multiple modules, additional packages and configure the existing packages.

Introduction

NS-2 can be used to simulate majority of the real time applications and networks like satellite communication, wireless sensor network, personal area network GSM, UMTS and underwater sensor networks. Each application uses its own packages consisting of set of protocols and needs some configuration in the corresponding NS-2 version.

Addition of new packages or modification of existing files

In order to modify any existing files related to any protocol in NS-2, it is necessary to understand the dependencies of the .cc files. Since the NS-2 library maps every .cc file to an OTcl file, to reflect the modifications done in .cc files, ‘make’ command is used. When certain packages are to be installed, it is necessary to add the corresponding files to the LIBRARY PATH in the bashrc file. To reflect the modifications it is necessary to use the following set of commands

- ./configure
- Make clean
- Make
- Make install

Adding a malicious node in NS2 in AODV Protocol

To illustrate the modifications that can be made in NS-2, process of defining and simulating a malicious node in AODV protocol in explained in this section. Adhoc on-demand Distance Vector (AODV) is one of the commonly used reactive on demand routing protocols in Mobile Adhoc NETwork (MANET). AODV is a reactive enhancement of the DSDV protocol. The route discovery process involves ROUTE REQUEST (RREQ) and ROUTE REPLY (RREP) packets. Fig.11.1 shows the dependencies of various files and its reference flow during AODV protocol implementation in NS-2.

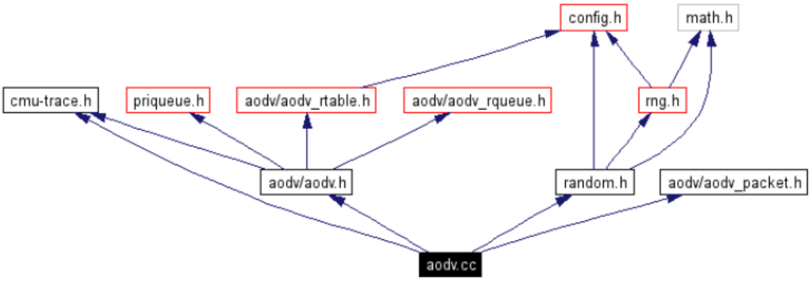


Fig.11.1: AODV.cc dependencies in NS-2

The node which is declared as malicious will simply drop the router packet

Two files have to be modified to define a malicious node.

1. aodv.h
2. aodv.cc

aodv.h file changes

Declare a boolean variable malicious as shown below in the protected scope in the class AODV

bool malicious;

aodv.cc file changes

1. Initialize the malicious varible with a value "false". Declare it inside the constructor as shown below

```
AODV::AODV(nsaddr_t id):Agent(PT_AODV)...
{…..
malicious = false;
}
```

2. Add the following statement to the aodv.cc file in the "if(argc==2)" statment.

```
if(strcmp(argv[1], "malicious") == 0) {
    malicious = true;
   return TCL_OK;
}
```

3. Implement the behavior of the malicious node by setting the following code in the rt_resolve(Packet *p) function. The malicious node will simply drop the packet as indicated below.

```
if(malicious==true)
{
drop(p,DROP_RTR_ROUTE_LOOP);
}
```

Once done, recompile ns2 using ‘make’ command. Then, check the malicious behavior using the Tcl Script by setting any one node as malicious node. The command to set the malicious node is

```
$ns at 0.0 "[$n2 set ragent_] malicious"
```

Lab Exercises

1. Add a malicious node in aodv.cc file and implement a wireless network scenario with 10 non-malicious nodes and 1 to 3 malicious nodes. Use AWK scripts and calculate the number of packets sent, received and packet drop ratio by varying the number of malicious nodes in the network.
2. Modify the appropriate .cc file to measure the power of transmitting node in a wireless network scenario. Illustrate the same using suitable number of nodes and plot a graph of power level used by the transmitting node against the number of receivers.

Additional exercise

1. Consider a wired network with 5 nodes connected in a star topology. The node at the center is the server node. Modify the appropriate .cc file to demonstrate message encryption and decryption. Any message sent by a node has to be encrypted and stored in the server node. Any message received by the node has to be decrypted and verified. Using suitable AWK scripts, calculate time taken to encrypt and decrypt the messages.

[OBSERVATION SPACE – LAB 11]

LAB NO: 12 **Date:**

SIMULATION OF A SATELLITE NETWORK USING NS-2

Objective:

- To illustrate the simulation of basic satellite network with ns2

Introduction

A satellite works as an attendant around the orbit, collect information and relay them to destination. The geostationary satellite appears to be fixed from a ground observer and is useful for telecommunication, television broadcasting, weather forecasting etc. A low earth orbit satellite is closer to the earth and it provide different types of communication service in connection with the small ground terminals. In this experiment we will learn the simulation of how two terminals on ground are communicated with each other using geostationary satellite and low earth orbit satellite.

Simulating a Satellite network in ns2

Configure the terminals and satellite objects.

```
# Set these global options for the satellite terminals
global opt
set opt(chan)       Channel/Sat
set opt(bw_up)      2Mb         # uplink bandwidth
set opt(bw_down)    2Mb         # downlink bandwidth
set opt(phy)        Phy/Sat
set opt(mac)        Mac/Sat
set opt(ifq)        Queue/DropTail
set opt(qlim)       50          # queue size (pkts)
set opt(ll)     LL/Sat
set opt(wiredRouting)   OFF;
set opt(alt)            780;    # Polar satellite altitude (Iridium)
set opt(inc)            86.4;     # Orbit inclination w.r.t. Equator
```

Specifically, the array opt defines the type of the objects that make up a terminal node and configure their attributes. Then, configure the satellite object and terminals.

There are three different types of nodes in satellite network:

Geostationary satellite nodes

Terminal nodes (placed on the Earth's surface)

Non-geostationary satellite nodes

Geostationary satellite nodes

A geostationary satellite is specified by its longitude above the equator. The longitude ranges from [-180,180] degrees. Two types of geostationary nodes exist: ``geo" (for processing satellites) and ``geo-repeater" (for bent-pipe satellites).

```
# Configure and create the satellite node “n1”
$ns node-config -satNodeType geo (or ``geo-repeater") \
            -phyType Phy/Repeater \
            -channelType $opt(chan) \
            -downlinkBW $opt(bw_down) \
            -wiredRouting $opt(wiredRouting)
set n1 [$ns node]
$n1 set-position $lon; # in decimal degrees
```

Terminal nodes

A terminal node[i] is specified by its latitude and longitude. Latitude ranges from [-90, 90] and longitude ranges from [-180, 180], with negative values corresponding to south and west, respectively.

```
# Configure and create the terminal node “n2”
$ns node-config -satNodeType terminal \
             -llType $opt(ll) \         #Link layer
             -ifqType $opt(ifq) \       #interface queue type
             -ifqLen $opt(qlim) \       #interface queue length
```

```
        -macType $opt(mac) \        #MAC type
        -phyType $opt(phy) \        #network interface type
        -channelType $opt(chan) \       #channel type
        -downlinkBW $opt(bw_down) \     #downlink bandwidth
        -wiredRouting $opt(wiredRouting)

set n2 [$ns node]
$n2 set-position $lat $lon; # in decimal degrees
```

Polar orbiting satellite nodes (Non-geostationary satellite)

Satellite orbits are usually specified by six parameters: altitude, semi-major axis, and eccentricity, right ascension of ascending node, inclination, and time of perigee passage.

The polar orbiting satellites in ns have purely circular orbits, so the orbits include parameters altitude, inclination, longitude, alpha and plane. Altitude is specified in kilometers above the Earth's surface.

Inclination can range from [0.180]degrees with 90 corresponding to pure polar orbits and angles greater than 90 degrees corresponding to “retrograde" orbits. The ascending node refers to the point where the footprint of the satellite orbital track crosses the equator moving from south to north.

Longitude of ascending node specifies the earth-centric longitude at which the satellite's nadir point crosses the equator moving south to north. Longitude of ascending node can range from [-180,180] degrees.

Alpha, specifies the initial position of the satellite along this orbit, starting from the ascending node. For example, an alpha of 180 degrees indicates that the satellite is initially above the equator moving from north to south. Alpha can range from [0,360] degrees. Plane, is specified when creating polar satellite nodes- all satellites in the same plane are given the same plane index.

```
# Configure and create the polar orbitting satellite nodes “n3”
# Nodes 0-99 are satellite nodes; 100 and higher are earth terminals
```

```
$ns node-config -satNodeType polar \
        -llType $opt(ll) \      #Link layer
        -ifqType $opt(ifq) \        #interface queue type
        -ifqLen $opt(qlim) \        #interface queue length
        -macType $opt(mac) \        #MAC type
        -phyType $opt(phy) \        #network interface type
        -channelType $opt(chan) \   #channel type
        -downlinkBW $opt(bw_down) \ #downlink bandwidth
        -wiredRouting $opt(wiredRouting)

set alt $opt(alt)
set inc $opt(inc)

set n3 [$ns node]
$n3 set-position $alt $inc $lon $alpha $plane           #set the parameters
```

Satellite links

Satellite links transmit and receive interfaces must be connected to different channels, and there is no ARP implementation in satellite links.

Network interfaces can be added with the following instproc:

$node add-interface $type $ll $qtype $qlim $mac $mac_bw $phy

The add-interface is eithere a add-gsl or add-isl. The following parameters must be provided:

type: It is used to identify the different types of links: geo or polar for links from a terminal to a geo or polar satellite, respectively, gsl and gsl-repeater for links from a satellite to a terminal, and intraplane, interplane, and crossseam ISLs.

ll:The link layer type (class LL/Sat is currently the only one defined).

qtype:The queue type (e.g., class Queue/DropTail).

qlim:The length of the interface queue, in packets.

mac:The MAC type. Currently, two types are defined: class Mac/Sat- a basic MAC for links with only one receiver (i.e., it does not do collision detection), and Class Mac/Sat/UnslottedAloha- an implementation of unslotted Aloha.

mac_bw:The bandwidth of the link is set by this parameter, which controls the transmission time how fast the MAC sends.

phy:The physical layer- currently two Phys (Class Phy/Sat and Class Phy/Repeater) are defined. The class Phy/Sat just pass the information up and down the stack- as in the wireless code, a radio propagation model could be attached at this point. The class Phy/Repeater pipes any packets received on receive interface straight through to a transmit interface.

An ISL(Inter Satellite Link) can be added between two nodes using the following instproc:

$ns add-isl $ltype $node1 $node2 $bw $qtype $qlim

This creates two channels, and appropriate network interfaces on both nodes, and attaches the channels to the network interfaces. The bandwidth of the link is set to bw. The linktype (ltype) must be specified as either intraplane, interplane, or crossseam.

A GSL(Ground to Satellite Link) involves adding network interfaces and a channel on board the satellite, and then defining the correct interfaces on the terrestrial node and attaching them to the satellite link, as follows:

*$node add-gsl $type $ll $qtype $qlim $mac $bw_up $phy *

[$node_satellite set downlink_] [$node_satellite set uplink_]

The type must be either geo or polar. The command setups a bidirectional connection between satellite node and the terminal node using parameters previously indicated opt.

Handoffs

Satellite handoff modelling is used in LEO satellite network simulations. There are two types of links to polar orbiting satellites that must be handed off: GSLs to polar satellites, and cross seam ISLs.

Each terminal connected to a polar orbiting satellite runs a timer. When the time expires, the Handoff Manager to check the current position of the satellite. If the satellite has fallen below the elevation mask of the terminal, the handoff manager detaches the terminal from that satellite's up and down links, and searches for another possible satellite. If it finds a suitable polar satellite, it connects its network interfaces to that satellite's uplink and downlink channels, and restarts the handoff timer. If it does not find a suitable satellite, it restarts the timer and tries again later. If any link changes occur, the routing agent is notified.

The elevation mask and handoff timer interval are set as follows:

HandoffManager/Term set elevation_mask_ 10; # degrees

HandoffManager/Term set term_handoff_int_ 10; # seconds

Handoffs may be randomized to avoid phase effects by setting the following variable:

HandoffManager set handoff_randomization_ 0; # 0 is false, 1 is true

If handoff_randomization_ is true, then the next handoff interval is a random variate picked from a uniform distribution across.

The satellite handoff interval is set by the following command:

HandoffManager/Sat set sat_handoff_int_ 10; # seconds

Interplane and crossseam ISLs are deactivated near the poles, because the pointing requirements for the links are too severe as the satellite draw close to one another. Shutdown of these links is governed by a parameter:

HandoffManager/Sat set latitude_threshold_ 70; # degrees

If crossseam ISLs exist, there are certain situations in which the satellites draw too close to one another in the mid-latitudes (if the orbits are not close to being pure polar orbits). The occurence of this orbital overlap is checked with the following parameter:

HandoffManager/Sat set longitude_threshold_ 10; # degrees

Routing

The routing genie is a class SatRouteObject and is created and invoked with the following OTcl commands:

set satrouteobject_ [new SatRouteObject]

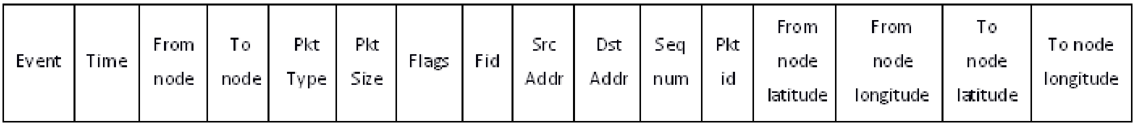
$satrouteobject_ compute_routes

Where the call to compute_routes is performed after all of the links and nodes in the simulator have been instantiated.

Structure of trace files in Satellite network

Satellite frame structure

The trace is organized in 16 fields as follows:



Event	Time	From node	To node	Pkt Type	Pkt Size	Flags	Fid	Src Addr	Dst Addr	Seq num	Pkt id	From node latitude	From node longitude	To node latitude	To node longitude

The first 12 fields are similar to conventional ns tracing.

The last four fields log the geographic latitude and longitude of the node logging the trace (the latitude and longitude correspond to the nadir point of the satellite).

Ex:

```
+ 1.0000 66 26 cbr 210 ------- 0 66.0 67.0 0 0 37.90 -122.30 48.90 -120.94
```

In this case, node 66 is at latitude 37.90 degrees, longitude -122.30 degrees, while node 26 is a LEO satellite whose sub-satellite point is at 48.90 degrees latitude, -120.94 degrees longitude (negative latitude corresponds to south, while negative longitude corresponds to west).

To enable tracing of all satellite links in the simulator, use the following commands before instantiating nodes and links:

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

Then use the following line after all node and link creation (and all error model insertion, if any) to enable tracing of all satellite links:

```
$ns trace-all-satlinks $f
```

Lab Exercise

1. Consider a multipurpose geostationary satellite (MGS-1) and two satellite terminals, one at location ‘A’ and the other at location ‘B’. The position of ‘A’ is 13.9 degree latitude north and 100.9 degree longitude east. The position of ‘B’ is 33.8 degree latitude north and 44.4 degree longitude east. MGS-1is used to provide television broadcasting from A to B. MGS-1 is positioned at 93.5 degrees longitude East. The traffic consists of a FTP source and a CBR stream. The simulation lasts for 50 secs. Analyze the trace file and find the end-to-end delay between two terminals.

[OBSERVATION SPACE – LAB 12]

REFERENCES

1. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX Environment: Second Edition, Addison Wesley Professional, 2005

2. Andrew S. Tanenbaum and David J Wetherall, Computer Networks: Fifth Edition, Prentice Hall, 2010

3. Teerawat Issariyakul and Ekram Hossain. Introduction to Network Simulator NS-2: Second Edition, Springer, 2011

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION

ADDITIONAL PAGES FOR OBSERVATION