MANIPAL INSTITUTE OF TECHNOLOGY

BENGALURU

(A constituent unit of MAHE, Manipal)

B.TECH. FIFTH SEMESTER

Information Technology
(B. Tech CSE (CYBER SECURITY))

COMPUTER NETWORKS LAB

CSE_3162

LABORATORY MANUAL

CONTENTS

Course Objectives

• To develop skills in network monitoring and analysis using various tools.

• To understand development of network applications.

• To design and deploy computer networks.

Course Outcomes

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

• Learn to Develop Network Application Programs.

• Analyze Network Traffic using Network Monitoring Tool

Evaluation plan

Sl. No	Components	Marks	Duration	Comment
1.	Continuous Evaluation	24	Regular Lab	Regular Labs The assessment is based on punctuality, program execution, neatness of Lab Record, viva-voce etc. Journal : 4 Viva:4 (3 evaluations)
2.	Test	24	2 hours	Write-Up:8 M, Execution:10 M Viva: 6
3.	Mini Project	12	Regular Lab	The suggested topic list is in Appendix-1
4.	Total Internal	60		
5.	End Semester	40	2 hours	
6.	Total	100		

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Record and the required stationery to every lab session.
2. Be on time and follow the institution's 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

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- 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.
- Lab records should be complete with proper design, Algorithms, and Flowcharts, related to the experiment they perform.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalties in evaluation.
- The exercises for each week are divided into three sets:
  - Solved exercise.
  - Lab exercises - to be completed during lab hours.
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examinations are **not** necessarily limited to the questions in the manual but may involve some variations and/or combinations of the questions.

Instructions for Mini Project:

- All the students must take up a mini-project and submit the project report.
- Mini projects can be taken up in a group of a maximum comprising of 2 or 3 students.
- Students may refer to Appendix-I for list of suggested topics for Mini Project.

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

Aim: Socket Programming in 'C' using TCP -Iterative Client-Server Programs

Objectives:

- To familiarize yourself with application-level programming with sockets.
- To understand principles of Inter-Process Communication with Unix TCP Sockets.
- To Learn to write Network programs using C programming language.

Prerequisites:

- Knowledge of the C programing language and Linux Networking APIs
- Knowledge of Basic Computer Networking

I. Sockets

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else. To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

Types of Sockets

There are four types of sockets available to the users. The first two are most commonly used and the last one is rarely used.

- **Stream Sockets:** Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order - "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
- **Datagram Sockets:** Delivery in a networked environment is not guaranteed. They're

connectionless because you don't need to have an open connection as in Stream Sockets - you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

- **Raw Sockets:** These provide users access to the underlying communication protocols, which support socket abstractions. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

Types of Servers

There are two types of servers you can have:

- **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.
- **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.

How to implement a client

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. Both the processes establish their own sockets. The steps involved in establishing a socket on the client side are as follows:

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

How to implement a Server Program in C using Linux APIs?

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

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

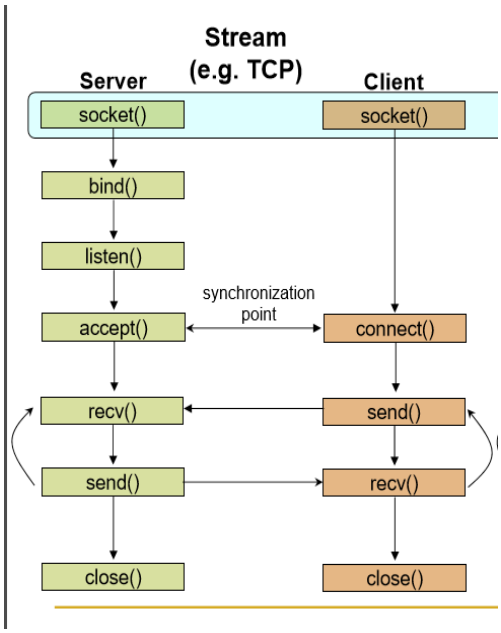

Figure 1.1 TCP client server interactions

Basic data structures used in Socket programming:

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. Structures defined in this chapter are related to Internet. Protocol Family.

o Socket Descriptor

- A simple file descriptor in Unix. Data type is integer.

o Socket Address

- This construct holds the information for socket address.

Table 1.1 System calls used in socket programming

Primitive	Meaning
Socket	Create a new communication endpoint
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

syntax

```
struct sockaddrs {
unsigned short sa_family; // address family,
AF_xxx or //PF_xxx char sa_data[14];
                                    // 14 bytes of
protocol address
 };
```

- AF stands for Address Family and PF stands for Protocol Family.

Table 2.2 Address Family

Name	Purpose
AF_UNIX, AF_LOCAL	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols
AF_IPX	IPX - Novell protocols

- struct sockaddr_in
  - This construct holds the information about the address family, port number,

    Internet address,and the size of the struct sockaddr.
- struct sockaddr_in

{
short int sin_family; // Address family unsigned short int sin_port; // Port number struct in_addr sin_addr; // Internet address
};

- The IP address structure, in_addr, is defined as follows struct in_addr
{
unsigned long int s_addr;
};

System calls used.

1. Socket creation in C using *socket()*

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

- *sockid* is socket descriptor, an integer (like a file-handle)
- *family* is the communication domain, like PF_INET for IPv4 protocols and Internet

addresses or PF_UNIX for Local communication and File addresses.

- *Type* defines communication type such as SOCK_STREAM or SOCK_DGRAM.
- *protocol* specifies protocol used. It take values like IPPROTO_TCP or

IPPROTO_UDP but usually set to 0 (i.e., use default protocol).

If the return value *sockid* is negative values, it means there is problem in socket creation.

NOTE: socket call does not specify where data will be coming from, nor where it will be

going to – it just creates the interface!

2. Assign address to socket using *bind()*

bind() associates and reserves a port for use by the socket.

Syntax:

int status = bind(sockid, &addrport, size);

- *Sockid* is a integer describing socket descriptor
- *addrport* is struct sockaddr which contains the (IP) address and port of the

machine ,, for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming

interface

- *size* specifies the size (in bytes) of the addrport structure
- *Status* will be assigned -1 returns on failure.

3. Listening to connection requests using *listen()*

This system call instructs TCP protocol implementation to listen for connections Syntax:

int status = listen(sockid, queueLimit);

- *Sockid* is socket descriptor which is created using *socket()*
- *Queuelemit* is an integer which specifies number of active participants that can "wait" for a connection
- *Status* will be assigned -1when returns on failure.

Note: The listening socket (sockid) is never used for sending and receiving. It is used by the server only as a way to get new sockets.

4. Establish Connection using *connect()*

The client establishes a connection with the server by calling *connect()*

Syntax:

int status = connect(sockid, &foreignAddr, addrlen);

- *sockid* is socket descriptor to be used in connection
- *foreignAddr is* struct sockaddr which contains address of the passive participant
- *addrlen is* sizeof(*foreignAddr*)

Status will be assigned -1 when returns on failure

Note: *connect()* is blocking where as *listen()* is non blocking.

5. Accept incoming Connection using *accept()*

The server gets a socket for an incoming client connection by calling accept() Syntax:

int newsockid = accept(sockid, &clientAddr, &addrLen);

- *newsockid* is an integer, the new socket is created in server which is client specific and this new socket is used for data-transfer between server and client.
- *Sockid* is the socket created using socket system call, which is used only to

listen to incoming requests from clients.

- *clientAddr* is in the form of *struct sockaddr*, address of the active participant.
- *addrLen* is size of *clientAddr* parameter.

Note: *accept()* is blocking, it waits for connection before returning and dequeues the

next connection on the queue for socket (sockid).

6. Exchanging data with stream socket

Application running in server and client(s) can transfer data using *send()* and *receive()system call.*

Syntax:

int count = send(sockid, msg, msgLen, flags);

- *Sockid* is the new socket descriptor created by accept in server side and

  socket in client side, depending on where it is used.
- *Msg* is an array holding message to be transmitted.
- *msgLen* holds length of message (in bytes) to transmit.
- flags are integer, special options, usually set 0
- Return value *count* has number of bytes transmitted and is set to -1 on error Syntax:
  int count = recv(sockid, recvBuf, bufLen, flags);
- *recvBuf* stores received message
- *bufLen* holds number if bytes

7. Closing the socket using *close()*

When finished using a socket, the socket should be closed.

Syntax:

int status= close(sockid);

- *sockid*: the file descriptor (socket being closed)
- status: 0 if successful, -1 if error

Closing a socket closes a connection (for stream socket) and frees up the port used by the socket.

Program:

// Client side C program to demonstrate Socket

```
// programming
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 8080

int main(int argc, char const* argv[])
{
    int status, valread, client_fd;
    struct sockaddr_in serv_addr;
    char* hello = "Hello from client";
    char buffer[1024] = { 0 };
    if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary
    // form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)
        <= 0) {
        printf(
            "\nInvalid address/ Address not supported \n");
        return -1;
    }

    if ((status
         = connect(client_fd, (struct sockaddr*)&serv_addr,
                   sizeof(serv_addr)))
        < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
    send(client_fd, hello, strlen(hello), 0);
    printf("Hello message sent\n");
    valread = read(client_fd, buffer,
                   1024 - 1); // subtract 1 for the null
                              // terminator at the end
    printf("%s\n", buffer);

    // closing the connected socket
    close(client_fd);
    return 0;
}
```

// **Server side C program to demonstrate Socket**

```
// programming
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 8080
int main(int argc, char const* argv[])
{
    int server_fd, new_socket;
    ssize_t valread;
    struct sockaddr_in address;
    int opt = 1;
    socklen_t addrlen = sizeof(address);
    char buffer[1024] = { 0 };
    char* hello = "Hello from server";

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET,
                   SO_REUSEADDR | SO_REUSEPORT, &opt,
                   sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Forcefully attaching socket to the port 8080
    if (bind(server_fd, (struct sockaddr*)&address,
             sizeof(address))
        < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket
         = accept(server_fd, (struct sockaddr*)&address,
                  &addrlen))
```

```
    < 0) {
    perror("accept");
    exit(EXIT_FAILURE);
  }
  valread = read(new_socket, buffer,
            1024 - 1); // subtract 1 for the null
                  // terminator at the end
  printf("%s\n", buffer);
  send(new_socket, hello, strlen(hello), 0);
  printf("Hello message sent\n");

  // closing the connected socket
  close(new_socket);
  // closing the listening socket
  close(server_fd);
  return 0;
}
```

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 filename –o executablefileName //renaming the a.out file***

   b. $./ executablefileName

4. Compile and run the client using the same instructions as listed in 3a & 3b.