B.M.S College of Engineering
P.O. Box No.: 1908 Bull Temple Road,

Bangalore-560019

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

Course – Unix System Programming
Course Code – 19IS4PWUSP
AY 2019-20

Report on Unix System Programming Project

CHAT APPLICATION USING TCP/IP SOCKET PROGRAMMING IN C LANGUAGE

Submitted by –

G PRAMOD SAI – 1BM18IS070

HIRAL HONNOORALI - 1BM20IS406

Submitted to -

Sandeep Varma N

Assistant Professor

B.M.S College of Engineering

P.O. Box No.: 1908 Bull Temple Road,

Bangalore-560 019

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

CERTIFICATE

Certified that the Project has been successfully presented at **B.M.S College of Engineering** by **G Pramod Sai and Hiral Honnoorali ,** bearing **USN: 1BM18IS070 and 1BM20IS406,** in partial fulfilment of the requirements for the IV Semester degree in **Bachelor of Engineering in Information Science & Engineering** of **Visvesvaraya Technological University, Belgaum** as a part of the course **UNIX System Programming (19IS4PWUSP)** during academic year 2020-2021.

Faculty Name: Sandeep Varma N

Designation: Assistant Professor

Department of ISE, BMSCE

TABLE OF CONTENTS

SNO	CONTENT	PAGE NO
1	ABSTRACT	5
2	INTRODUCTION	5
3	PROBLEM STATEMENT	5
4	API USED IN PROJECT	6
5	EXPLANATION ABOUT THE API	6
6	IMPLEMENTATION	13
8	SNAPSHOT	17
9	REFERENCES	18

ABSTRACT

Several network systems are built to communicate with one another as well as made available through service-oriented architectures. In this project, the client server architecture is used to develop a full duplex server client chat application. Firstly, a chat application is created for both Client and Server which is based on Transmission Control Protocol (TCP) where TCP is connection oriented protocol and is a reliable connection protocol in C language.

INTRODUCTION

The Internet and WWW have emerged as global ubiquitous media for communication and changed the way we conduct science, engineering, and commerce. They are also changing the way we learn, live, enjoy, communicate, interact, engage, etc. Modern life activities are getting completely centered around or driven by the Internet.

To take advantage of opportunities presented by the Internet, businesses are continuously seeking new and innovative ways and means for offering their services via the Internet. This created a huge demand for software designers and engineers with skills in creating new Internet enabled applications or porting existing/legacy applications to the internet platform. The key elements for developing Internet-enabled applications are a good understanding of the issues involved in implementing distributed applications and sound knowledge of the fundamental network programming models.

PROBLEM STATEMENT

To build a chat application using TCP/IP socket programming in C Language with the help of Full Duplex(Communication on both sides simultaneously).

API USED IN PROJECT

1. Client/Server Communication

2. TCP (Transmission control protocol)

3. Transmission Modes in Computer Networks

4. Socket Programming in C

5. I/O Multiplexing

EXPLANATION ABOUT THE API

Client/Server Communication

At a basic level, network-based systems consist of a server client and a media for communication; a computer running a program that makes a request for services is called a client machine. A computer running a program that offers requested services from one or more clients is called a server machine. The media for communication can be wired or wireless.

Generally, programs running on client machines make requests to a program (often called a server program) running on a server machine. They involve networking services provided by the transport layer, which is part of the Internet software stack, often called TCP/IP (Transport Control Protocol/ Internet Protocol) stack, shown in Fig. 13.2. The transport layer comprises two types of protocols, TCP (Transport Control Protocol) and UDP (User Datagram Protocol). The most widely used programming interfaces for these protocols are sockets.

TCP is a connection-oriented protocol that provides a reliable flow of data between two computers. Example applications that use such services are HTTP, FTP, and Telnet.

UDP is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival and sequencing. Example applications that use such services include Clock server and Ping.

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer. Port is represented by a positive (16-bit) integer value.

TCP (Transmission control protocol)

A TCP (transmission control protocol) is a connection-oriented communication. It is an intermediate layer of the application layer and internet protocol layer in the OSI model. TCP is designed to send the data packets over the network. It ensures that data is delivered to the correct destination.

TCP creates a connection between the source and destination node before transmitting the data and keeps the connection alive until the communication is active.

In TCP before sending the data it breaks the large data into smaller packets and cares about the integrity of the data at the time of reassembling at the destination node. Major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP.

TCP also offers the facility of retransmission, when a TCP client sends data to the server, it requires an acknowledgment in return. If an acknowledgment is not received, after a certain amount of time transmitted data will be lost and TCP automatically retransmits the data.

The communication over the network in TCP/IP model takes place in the form of a client server architecture. i.e., the client begins the communication and establishes a connection with a server.

Transmission Modes in Computer Networks

Transmission mode means transferring of data between two devices. It is also known as communication mode. Buses and networks are designed to allow communication to occur between individual devices that are interconnected.

There are three types of transmission mode: -

1. **Simplex Mode –**
   In Simplex mode, the communication is unidirectional, as on a one-way street. Only one of the two devices on a link can transmit, the other can only receive. The simplex mode can use the entire capacity of the channel to send data in one direction.
   Example: Keyboard and traditional monitors. The keyboard can only introduce input, the monitor can only give the output.
2. **Half-Duplex Mode –**
   In half-duplex mode, each station can both transmit and receive, but not at the same time. When one device is sending, the other can only receive, and vice versa. The half-duplex mode is used in cases where there is no need for communication in both directions at the same time. The entire capacity of the channel can be utilized for each direction.
   Example: Walkie- talkie in which message is sent one at a time and messages are sent in both directions.
3. **Full-Duplex Mode –**

   In full-duplex mode, both stations can transmit and receive simultaneously. In full duplex mode, signals going in one direction share the capacity of the link with signals going in other direction, this sharing can occur in two ways:

- Either the link must contain two physically separate transmission paths, one for sending and other for receiving.
- Or the capacity is divided between signals travelling in both directions.

Full-duplex mode is used when communication in both directions is required all the time.The capacity of the channel, however, must be divided between the two directions. Example: Telephone Network in which there is communication between two persons by a telephone line, through which both can talk and listen at the same time.

Socket Programming in C

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while another socket reaches out to the other to form a connection. Server forms the listener socket while the client reaches out to the server.

Stages for server

Socket creation: int sockfd = socket(domain, type, protocol)

sockfd: socket descriptor, an integer (like a file-handle)
domain: integer, communication domain e.g., AF_INET (IPv4 protocol) ,
AF_INET6 (IPv6 protocol) **type:** communication type
SOCK_STREAM: TCP(reliable, connection oriented)
SOCK_DGRAM: UDP(unreliable, connectionless)
protocol: Protocol value for Internet Protocol(IP), which is 0.

- **Setsockopt:** int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);

  Setsockopt helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port.

  Prevents errors such as: “address already in use”.

- **Bind:** int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

  After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure).

- **Listen:** int listen(int sockfd, int backlog);

  listen puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

- **Accept:**

  It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

Stages for Client

Socket connection:

Exactly same as that of server's socket creation ·

Connect: The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

I/O Multiplexing

When the TCP client is handling two inputs at the same time: standard input and a TCP socket, we encountered a problem when the client was blocked in a call to fgets (on standard input) and the server process was killed. The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input, it never saw the EOF until it read from the socket (possibly much later).

We want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output). This capability is called **I/O multiplexing** and is provided by the select and poll functions, as well as a newer POSIX variation of the former, called pselect.

I/O multiplexing is typically used in networking applications in the following scenarios:

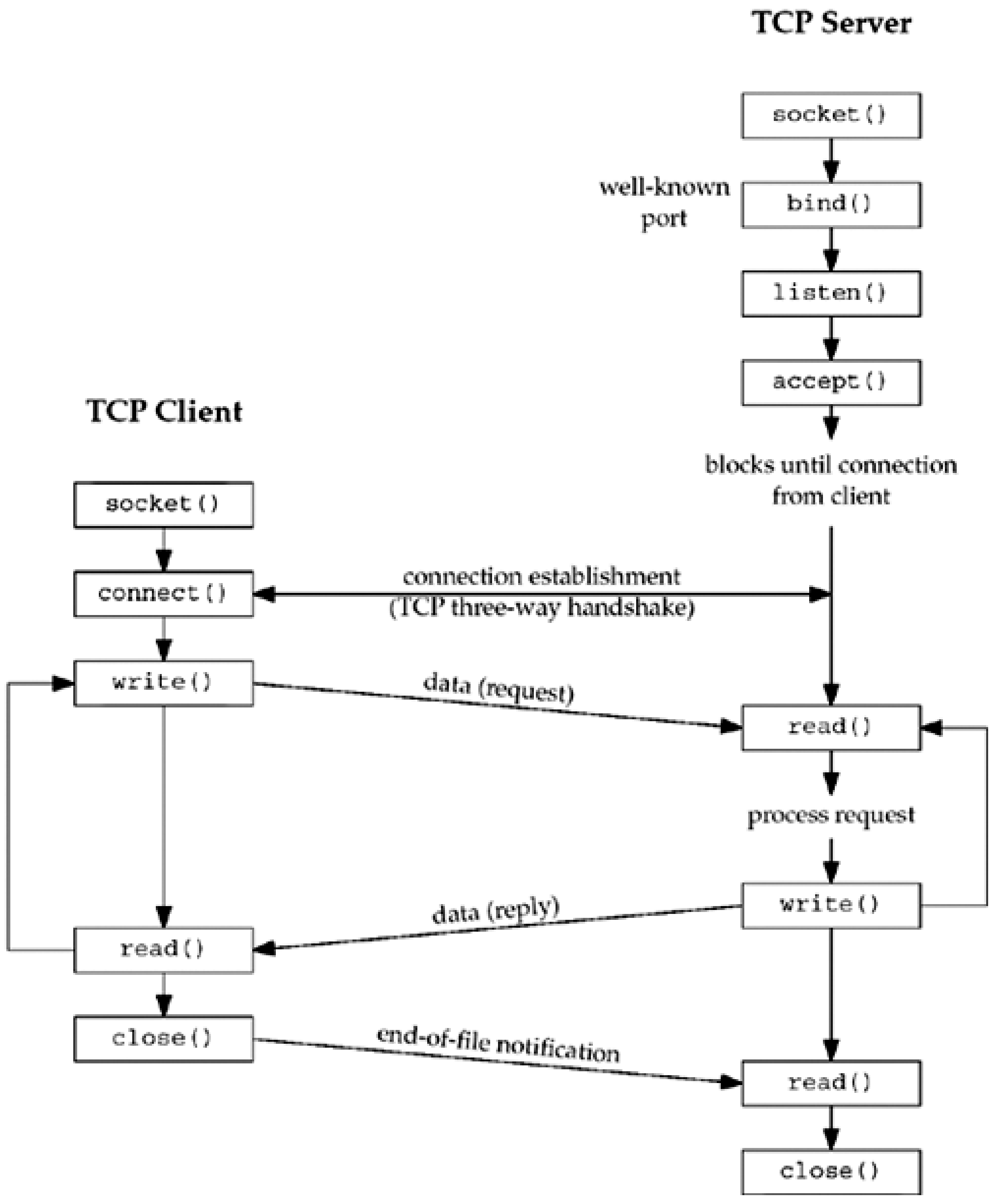
- When a client is handling multiple descriptors
- When a client to handle multiple sockets at the same time
- If a TCP server handles both a listening socket and its connected sockets
- If a server handles both TCP and UDP
- If a server handles multiple services and perhaps multiple protocols

With **I/O multiplexing**, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call

We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

select()- allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation (e.g., read(2), or a sufficiently small write(2)) without blocking.

FLOW CHART:



IMPLEMENTATION

server.c

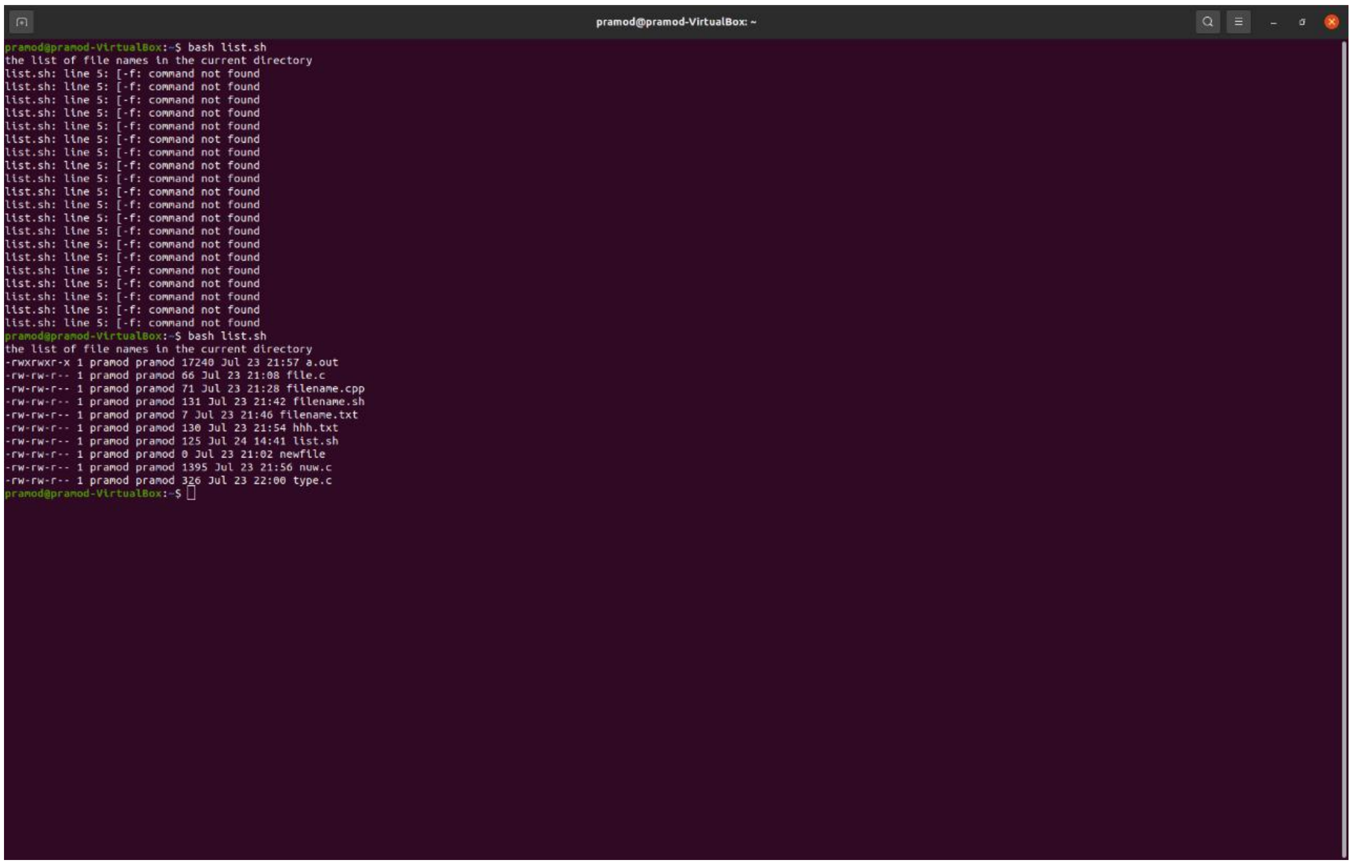
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
```

```c
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <time.h>
#define STDIN 0

int main()
{
system("clear");
int serverfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
if(serverfd==-1)
{
perror("Socket creation failed try again\n");
exit(0);
}

struct sockaddr_in server,client;
server.sin_family=AF_INET;
server.sin_port=htons(1110);
server.sin_addr.s_addr=INADDR_ANY;
int opt=1;
setsockopt(serverfd,SOL_SOCKET,SO_REUSEADDR,&opt,sizeof(opt));
int b=bind(serverfd,(struct sockaddr*)&server,sizeof(server));

if(b==-1)
{
perror("Error Bind failure\n");
exit(0);
}

listen(serverfd,5);
fd_set readfd;
FD_ZERO(&readfd);
int size = sizeof(server);
int clientfd=accept(serverfd,(struct sockaddr*)&client,&size);
if(clientfd==-1)
{
perror("Accept failed try again!\n");
exit(0);
}

printf("I am sever %s \n",inet_ntoa(client.sin_addr));
struct timeval tv;
char snd[50],rcv[50];
while(1)
{
FD_SET(STDIN,&readfd);
FD_SET(clientfd,&readfd);
tv.tv_sec=15;
tv.tv_usec=5000000;
int s=select(clientfd+1,&readfd,NULL,NULL,&tv);
if(s==-1)
perror("select error\n");
else if(s==0)
printf("Timeout occured!!\n");
else
{
if(FD_ISSET(STDIN,&readfd))
{
scanf("%s",snd);
send(clientfd,snd,strlen(snd),0);
}
if(FD_ISSET(clientfd,&readfd))
{
int r=recv(clientfd,rcv,sizeof(rcv),0);
rcv[r]='\0';
if(r==0)
{
printf("Client is now closed\n");
break;
}
time_t t;
int h,m,s;
time(&t);
struct tm *local = localtime(&t);
h=local->tm_hour;
m=local->tm_min;
s=local->tm_sec;
system("notify-send Unread \"Message from client\"");
printf("CILENT MESSAGE: %02d:%02d:%02d : %s\a\n",h,m,s,rcv);
}
}
}
}
```

client.c

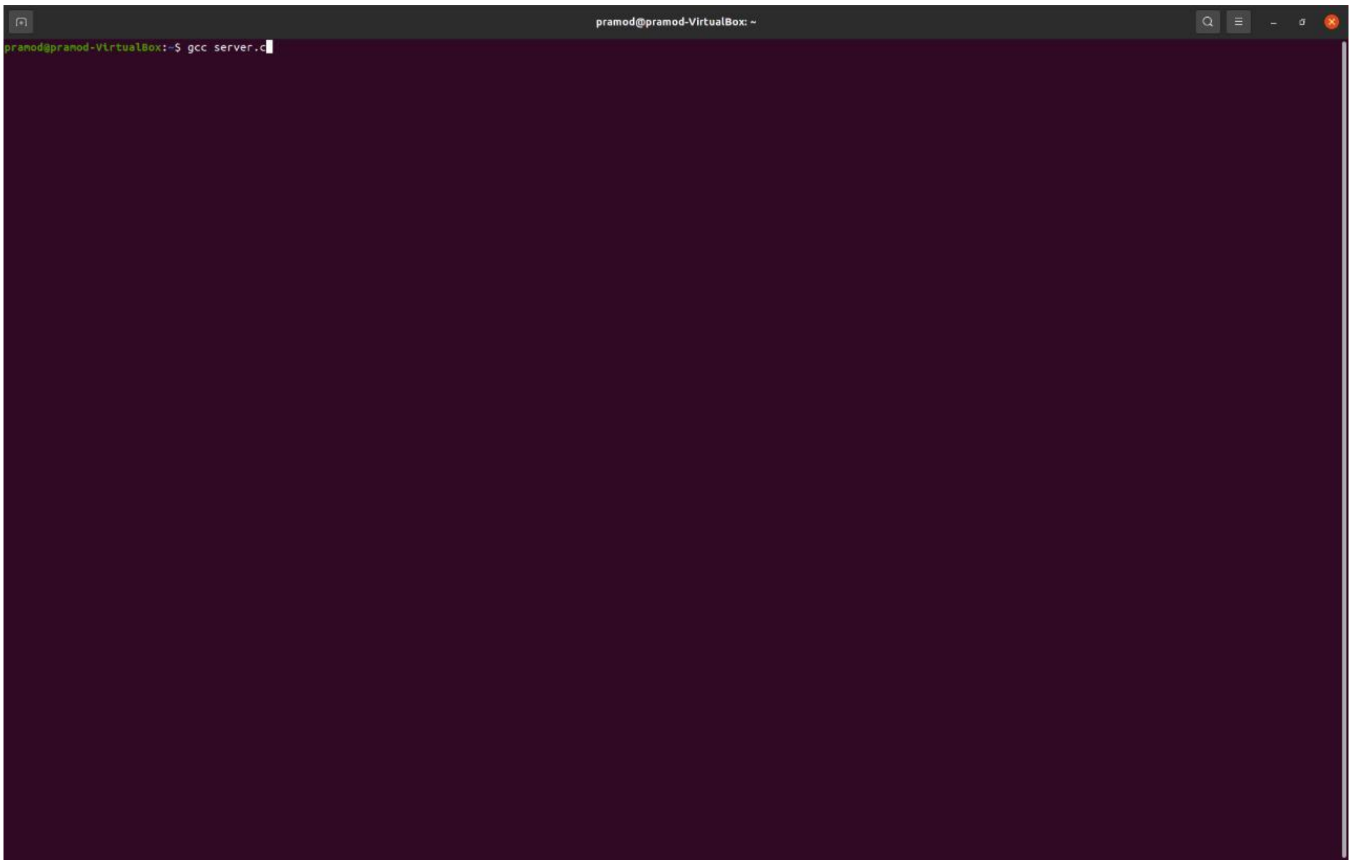
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <time.h>
#define STDIN 0

int main()
{
system("clear");
int clientfd = socket (AF_INET, SOCK_STREAM,IPPROTO_TCP);
if(clientfd==-1)
{
perror("Socket creation failed\n");
exit(0);
}

struct sockaddr_in server;
server.sin_family=AF_INET;
server.sin_port=htons(5000);
server.sin_addr.s_addr=INADDR_ANY;

int c=connect(clientfd,(struct sockaddr*)&server,sizeof(server));

if(c==-1)
{
perror("connection failure try again :(\n");
exit(0);
}

printf("Connectd to server\n");
fd_set readfd;
FD_ZERO(&readfd);
struct timeval tv;
char snd[50],rcv[50];
while(1)
{
FD_SET(STDIN,&readfd);
FD_SET(clientfd,&readfd);
tv.tv_sec=15;
tv.tv_usec=5000000;
int s=select(clientfd+1,&readfd,NULL,NULL,&tv);
if(s==-1)
perror("select error\n");
else if(s==0)
printf("Timeout occured\n");
else
{
if(FD_ISSET(STDIN,&readfd))
{
scanf("%s",snd);
send(clientfd,snd,strlen(snd),0);
}
if(FD_ISSET(clientfd,&readfd))
{
int r=recv(clientfd,rcv,sizeof(rcv),0);
rcv[r]='\0';
if(r==0)
{
printf("server is now closed\n");
break;
}
time_t t;
int h,m,s;
time(&t);
struct tm *local = localtime(&t);
h=local->tm_hour;
m=local->tm_min;
s=local->tm_sec;
system("notify-send Unread \"Message from server\"");
printf(" SERVER MESSAGE: %02d:%02d:%02d : %s\a\n",h,m,s,rcv);
}
}
}
}
```

SNAPSHOT



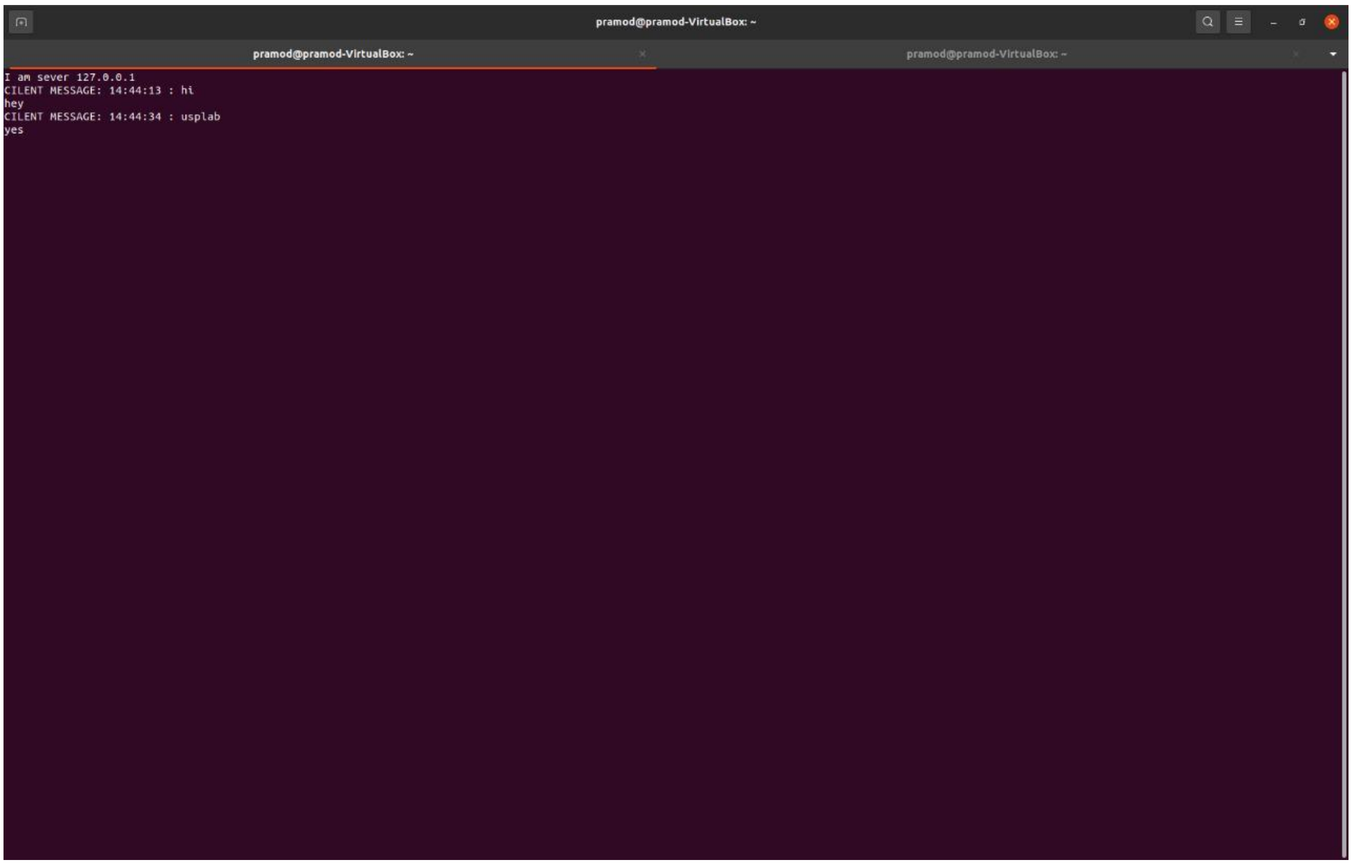
```
pramod@pramod-VirtualBox:~$ bash list.sh
the list of file names in the current directory
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
pramod@pramod-VirtualBox:~$ bash list.sh
the list of file names in the current directory
-rwxrwxr-x 1 pramod pramod 17240 Jul 23 21:57 a.out
-rw-rw-r-- 1 pramod pramod 66 Jul 23 21:08 file.c
-rw-rw-r-- 1 pramod pramod 71 Jul 23 21:28 filename.cpp
-rw-rw-r-- 1 pramod pramod 131 Jul 23 21:42 filename.sh
-rw-rw-r-- 1 pramod pramod 7 Jul 23 21:46 filename.txt
-rw-rw-r-- 1 pramod pramod 130 Jul 23 21:54 hhh.txt
-rw-rw-r-- 1 pramod pramod 125 Jul 24 14:41 list.sh
-rw-rw-r-- 1 pramod pramod 0 Jul 23 21:02 newfile
-rw-rw-r-- 1 pramod pramod 1395 Jul 23 21:56 nuw.c
-rw-rw-r-- 1 pramod pramod 326 Jul 23 22:00 type.c
pramod@pramod-VirtualBox:~$ 
```

Fig 1, Bash list.h



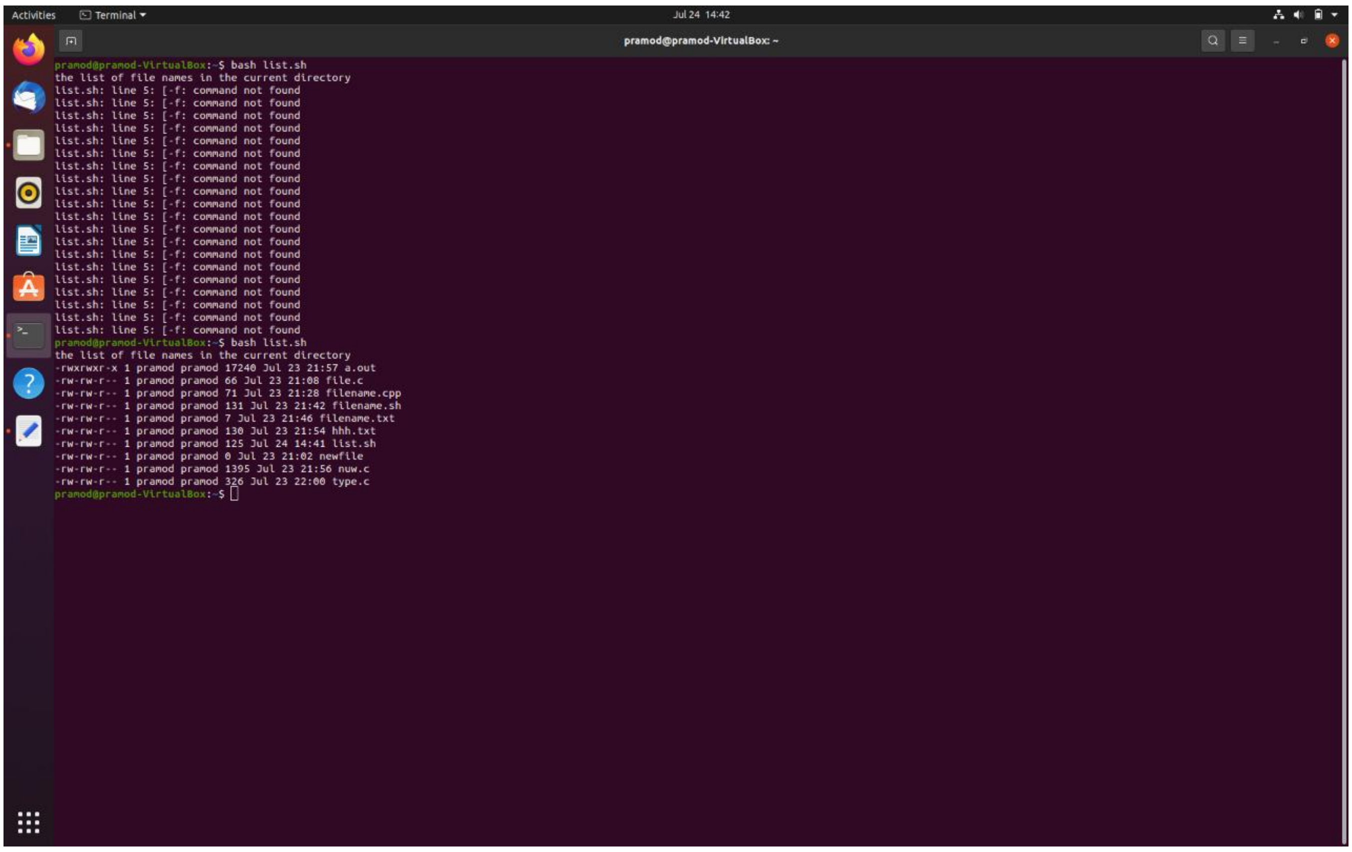
```
pramod@pramod-VirtualBox:~$ gcc server.c
```

Fig 2, Server.c


pramod@pramod-VirtualBox: ~
pramod@pramod-VirtualBox: ~
pramod@pramod-VirtualBox: ~

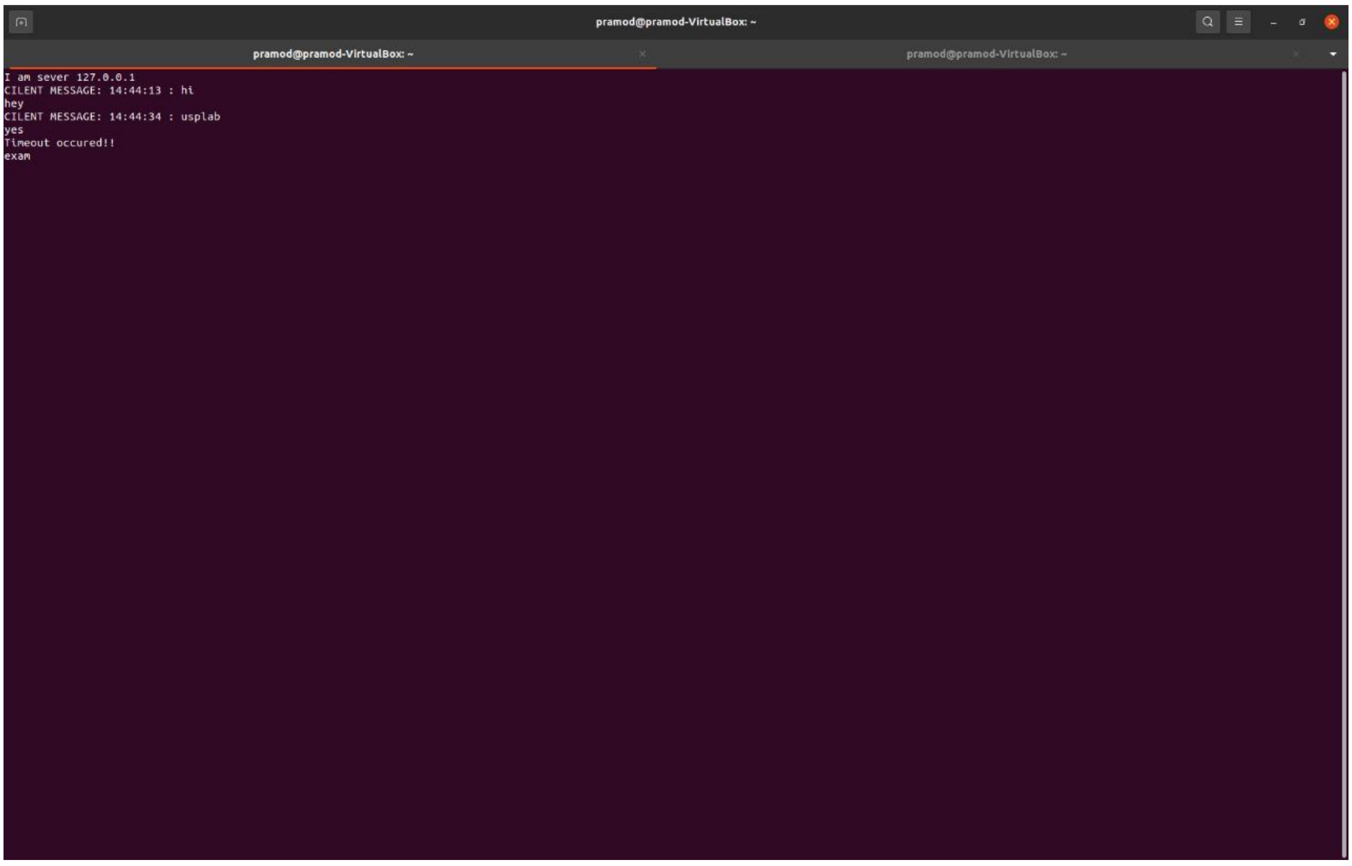
```
I am sever 127.0.0.1
CILENT MESSAGE: 14:44:13 : hi
hey
CILENT MESSAGE: 14:44:34 : usplab
yes
```

Fig 3,Server File



```
pramod@pramod-VirtualBox:~$ bash list.sh
the list of file names in the current directory
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
list.sh: line 5: [-f: command not found
pramod@pramod-VirtualBox:~$ bash list.sh
the list of file names in the current directory
-rwxrwxr-x 1 pramod pramod 17240 Jul 23 21:57 a.out
-rw-rw-r-- 1 pramod pramod 66 Jul 23 21:08 file.c
-rw-rw-r-- 1 pramod pramod 71 Jul 23 21:28 filename.cpp
-rw-rw-r-- 1 pramod pramod 131 Jul 23 21:42 filename.sh
-rw-rw-r-- 1 pramod pramod 7 Jul 23 21:46 filename.txt
-rw-rw-r-- 1 pramod pramod 130 Jul 23 21:54 hhh.txt
-rw-rw-r-- 1 pramod pramod 125 Jul 24 14:41 list.sh
-rw-rw-r-- 1 pramod pramod 0 Jul 23 21:02 newfile
-rw-rw-r-- 1 pramod pramod 1395 Jul 23 21:56 nuw.c
-rw-rw-r-- 1 pramod pramod 326 Jul 23 22:00 type.c
pramod@pramod-VirtualBox:~$ 
```

Fig 4, Client.c



pramod@pramod-VirtualBox: ~
pramod@pramod-VirtualBox: ~
pramod@pramod-VirtualBox: ~

```
I am sever 127.0.0.1
CILENT MESSAGE: 14:44:13 : hi
hey
CILENT MESSAGE: 14:44:34 : usplab
yes
Timeout occured!!
exam
```

Fig 5, Client file

REFERENCES

1. https://www.tutorialspoint.com/unix_sockets/index.htm
2. https://www.geeksforgeeks.org/socket-programming-cc/
3. https://youtu.be/dEHZb9JsmOU
4. http://www.scottklement.com/rpg/socktut/nonblocking.html
5. https://aticleworld.com/socket-programming-in-c-using-tcpip/
6. https://stackoverflow.com/questions/1543466/how-do-i-change-a-tcp-socket-tobe-non-blocking/22339017

7.https://www.ibm.com/docs/en/zvse/6.2?topic=SSB27H_6.2.0/fa2ti_what_is_socket_connection.html