

# **1. INTRODUCTION**



## **1.1 About the Project**

This Project provide user to communicate over network. Users can broadcast their message to all other currently logged in. System provides tool for conferencing and share the ideas with each other. This will be extremely helpful in the companies or any organization.

## **1.2 Software requirement specification:**

During the time of requirement analysis, the following requirements were noted:

- ❖ Software developed should be portable to any PC running under LINUX platform.
- ❖ It is application over the LAN.
- ❖ Should support future up gradation.
- ❖ Should have downward compatibility.
- ❖ A server program which a server program is running at the backend.
- ❖ The server should be able to send and receive messages between the users.
- ❖ Client Program used by different user to connect to the server and send messages.

Platform: Red Hat LINUX 7.1

Language: C programming under gcc compiler



## **2. ABSTRACT**

Computer Networks is an interconnected collection of autonomous computers. Here prominence is given to exchange of information and that has been the sole reason for its survival.

The various uses of computer networks have been:

- Access to remote information.
- Person-to-person communication.
- Interactive entertainment.

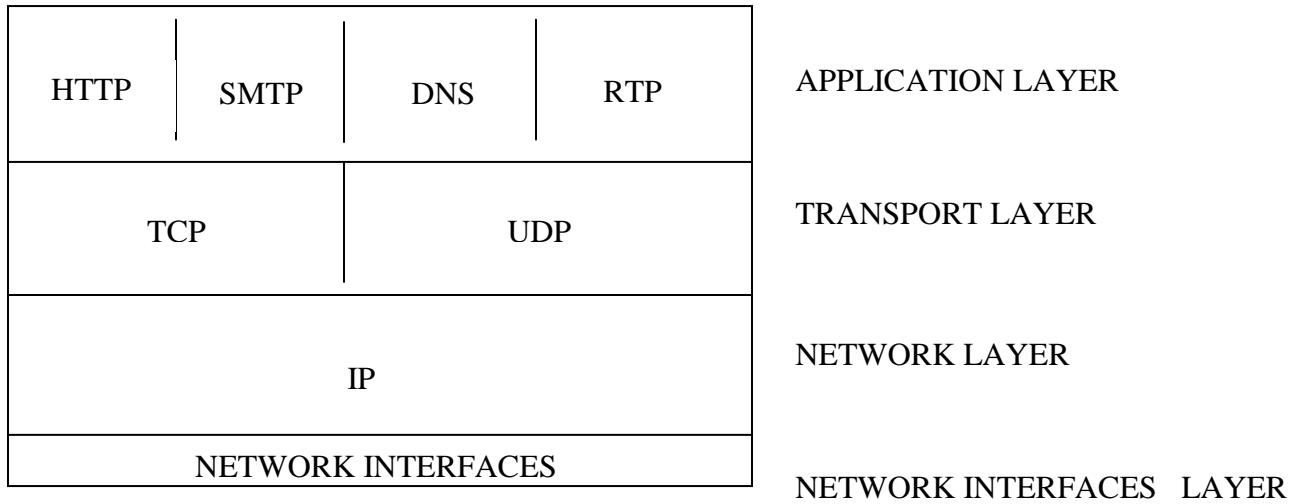
These are some of the important aspects of networks.

### **2.1 Protocol Stacks**

Very Early in the history of computer network development the concept of separating the problem into multiple levels was adapted. With a multilevel architecture each layer can handle a different aspect of networking and provide that functionality to the above layer. TCP/IP is a specific implementation of multilevel network architecture.



## TCP/IP PROTOCOL STACK



### 2.2 The Application layer

The Application layer contains all the higher-level protocols. The early ones included virtual terminal (TELNET), FTP, SMTP. The virtual terminal protocol allows a user on one machine to log into a distant machine and work there. The FTP provides a way to move data efficiently from one machine to another. HTTP protocol is used for fetching pages on the World Wide Web.



## **2.3 The Transport layer**

This layer provides two types of services

### **2.3.1 TCP**

TCP (the “Transmission Control Protocol “) has the responsibility for breaking up the message into data grams, reassembling them at the other end, resending anything that gets lost, and putting things back in the right order. It may seem that TCP is doing all the work. And in small network it is true. With TCP, there is no maximum message length. When a message is passed to the TCP protocol, if it is too large to be sent in one piece, the message is broken up into chunks or packets and sent one at a time to the destination address. The TCP packet contains the addressing information. The TCP message also contains a packet number and total number of packets. Because of the nature of the TCP/IP protocol, the packet may travel different paths and may arrive in a different order than sent. TCP reassemble the packets in the proper order and requests the retransmission of any missing or corrupted packets. TCP enables you to create and maintain a connection to a remote computer. By using the connection, both computers can stream data between each other.

The second protocol in this layer, UDP (user datagram protocol), is an unreliable, connectionless protocol for applications that do not want TCP’S sequencing or flow control and wish to provide their own. It is also widely used for one-shot, client-server type request-reply queries and applications in which prompt delivery is more important than accurate delivery, such as transmitting speech or video. The relation of IP, TCP, and UDP is shown in fig. Since the model was developed, IP has been implemented on many other networks.



### **2.3.2 IP**

As the number of computers networked become larger, a system becomes necessary to give remote computers the capability to recognize other remote computers; thus the IP addressing method was born.

Therefore, simply an IP address uniquely identifies any computer connected to a network. This address is made up of 32 bits divided into 4 four bytes. But since the number of connected computers is too large and since it is difficult to remember all their IP addresses, the Domain Name Service (DNS) was designed. It has the job of transforming the unique computer names (host name) into an IP address. Therefore, whenever in our project we run the client application and enter the host name, this means that we are writing the IP address of the remote computer we want to connect to indirectly. In general, TCP/IP is a set of protocols developed to allow cooperating computers to share resources across the network.

### **2.4 Network layer**

The network layer is concerned with controlling the operation of the subnet. A key design issue is determining how packets are routed from source to destination. Routes can be based on static tables that are wired into the network and rarely changed.

If too many packets are present in the subnet at the same time, they will get in each other's way, forming bottlenecks. The control of such congestion also belongs to the network layer.

Finally, they can be highly dynamic, being determined a new for each packet, to reflect the current network load.



## **2.5 Network Interface Layer**

This layer is concerned with the network-specific aspects of the transfer of packets. As such, it must deal with part of the OSI network layer and data link layer. Various interfaces are available for connecting end computer systems to specific networks such as X.25, ATM, frame relay, Ethernet and token ring.

## **2.6 Conferencing Server**

The ideology behind this project is to help users communicate among themselves. To start the communication, firstly the server program has to be started. Each user can see other users who have logged into the server currently. Users can simultaneously communicate with many others.



### **3. DESIGN**



### **3.1 Client-Server Architecture**

When the program wishes to use TCP to exchange data, one of the programs should take the role of a client while the other must take the role of a server. The client application initiates what is called active open. It creates a socket and actively attempts to connect to server program. On the other hand, the server application creates a socket and passively listens for incoming connections from client, performing what is called passive open. When the client wants to connect a server, it sends a connection request. The server is notified that some process is trying to connect with it. By accepting the connection, the server completes what is called a virtual circuit, a logical communication pathway between the two programs. To review, there are five significant steps that a program that uses TCP must take to establish and complete a connection.

All communication protocols require certain Application Programming Interfaces (APIs). An API is an interface available to the programmer. The availability of an API depends on both the operating system being used and the programming language. The two most prevalent APIs for UNIX systems are:

1. Berkeley sockets
2. System V Transport Layer Interface (TLI).

Both these interface were developed for the C language.

The protocol being used to establish a connection between the client and the server can be either

1. Connection oriented, or
2. Connectionless.

The concept of Berkeley sockets is being used in this project to implement the Network File Search over an reliable, connection-oriented TCP.



### **3.2 Connection-oriented Service**

The connection-oriented service is done using the **Transmission Control Protocol (TCP)**.

To illustrate the role of TCP, it is instructive to follow a sample message between two machines. The processes are simplified at this stage, to be expanded on later today. The message originates from an application in an upper layer and is passed to TCP from the next higher layer in the architecture through some protocol (often referred to as an upper-layer protocol, or ULP, to indicate that it resides above TCP). The message is passed as a *stream*—a sequence of individual characters sent asynchronously. This is in contrast to most protocols, which use fixed blocks of data. This can pose some conversion problems with applications that handle only formally constructed blocks of data or insist on fixed-size messages.

TCP receives the stream of bytes and assembles them into TCP *segments*, or packets. In the process of assembling the segment, header information is attached at the front of the data. Each segment has a checksum calculated and embedded within the header, as well as a sequence number if there is more than one segment in the entire message. The length of the segment is usually determined by TCP or by a system value set by the system administrator. (The length of TCP segments has nothing to do with the IP datagram length, although there is sometimes a relationship between the two.)

If two-way communications are required (such as with Telnet or FTP), a connection (virtual circuit) between the sending and receiving machines is established prior to passing the segment to IP for routing. This process starts with the sending TCP software issuing a request for a TCP connection with the receiving machine. In the message is a unique number (called a socket number) that identifies the sending machine's connection. The receiving TCP software assigns its own unique socket number and sends it back to the original machine. The two unique numbers then define the connection between the two machines until the virtual circuit is terminated. (I look at sockets in a little more detail in a moment.)

After the virtual circuit is established, TCP sends the segment to the IP software, which then issues the message over the network as a datagram. IP can perform any of the changes to the segment that you saw in yesterday's material, such as fragmenting it and reassembling it at the destination machine. These steps are completely transparent to the TCP layers, however. After winding its way over the network, the receiving machine's IP passes the received segment up to the recipient machine's TCP layer, where it is processed and passed up to the applications above it using an upper-layer protocol.



If the message was more than one TCP segment long (not IP data grams), the receiving TCP software reassembles the message using the sequence numbers contained in each segment's header. If a segment is missing or corrupt (which can be determined from the checksum), TCP returns a message with the faulty sequence number in the body. The originating TCP software can then resend the bad segment.

If only one segment is used for the entire message, after comparing the segment's checksum with a newly calculated value, the receiving TCP software can generate either a positive acknowledgment (ACK) or a request to resend the segment and route the request back to the sending layer.

The receiving machine's TCP implementation can perform a simple flow control to prevent buffer overload. It does this by sending a buffer size called a window value to the sending machine, following which the sender can send only enough bytes to fill the window. After that, the sender must wait for another window value to be received. This provides a handshaking protocol between the two machines, although it slows down the transmission time and slightly increases network traffic.

As with most connection-based protocols, timers are an important aspect of TCP. The use of a timer ensures that an undue wait is not involved while waiting for an ACK or an error message. If the timers expire, an incomplete transmission is assumed. Usually an expiring timer before the sending of an acknowledgment message causes a retransmission of the datagram from the originating machine.

Timers can cause some problems with TCP. The specifications for TCP provide for the acknowledgment of only the highest datagram number that has been received without error, but this cannot properly handle fragmentary reception. If a message is composed of several data grams that arrive out of order, the specification states that TCP cannot acknowledge the reception of the message until all the data grams have been received. So even if all but one datagram in the middle of the sequence have been successfully received, a timer might expire and cause all the data grams to be resent. With large messages, this can cause an increase in network traffic.

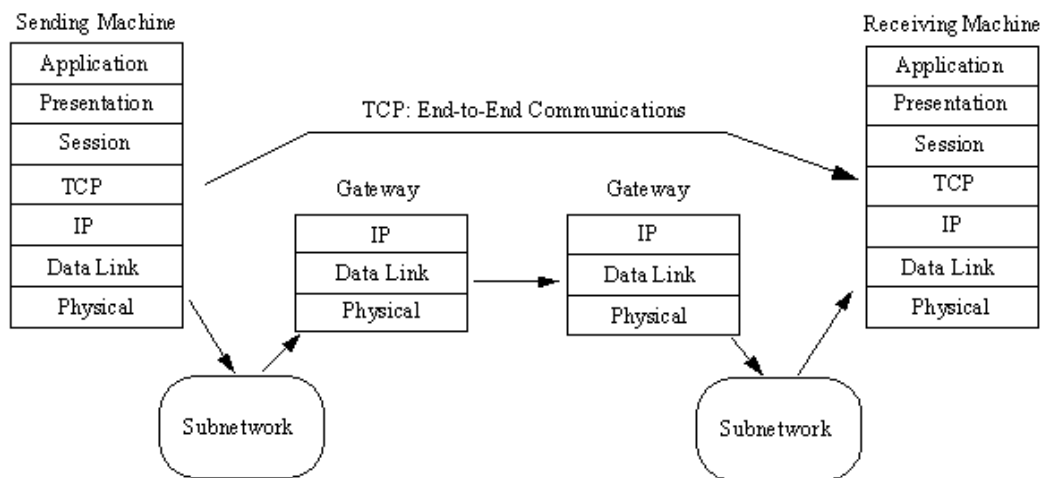
If the receiving TCP software receives duplicate data grams (as can occur with a retransmission after a timeout or due to a duplicate transmission from IP), the receiving version of TCP discards any duplicate



data grams, without bothering with an error message. After all, the sending system cares only that the message was received—not how many copies were received.

TCP does not have a negative acknowledgment (NAK) function; it relies on a timer to indicate lack of acknowledgment. If the timer has expired after sending the datagram without receiving an acknowledgment of receipt, the datagram is assumed to have been lost and is retransmitted. The sending TCP software keeps copies of all unacknowledged data grams in a buffer until they have been properly acknowledged. When this happens, the retransmission timer is stopped, and the datagram is removed from the buffer.

TCP supports a push function from the upper-layer protocols. A push is used when an application wants to send data immediately and confirm that a message passed to TCP has been successfully transmitted. To do this, a push flag is set in the ULP connection, instructing TCP to forward any buffered information from the application to the destination as soon as possible (as opposed to holding it in the buffer until it is ready to transmit it).



*Figure 3.1 The TCP Model*



### 3.3 Socket Addresses

Many of the BSD networking system calls require a pointer to a socket address as an argument. The definition of this structure is in `<sys/socket.h>`:

```
struct sockaddr
{
    u_short sa_family; /* Address family : AF_XXX value */
    char sa_data[14]; /* Up to 14 bytes of protocol-specific
        addresses */
};
```

The contents of the 14 bytes of protocol specific addresses are interpreted according to the type of address. For the Internet family, the following structures are defined in `<netinet/in.h>`:

```
struct in_addr
{
    u_long s_addr; /* 32-bit netid/hostid */
                /* network byte ordered */
};

struct sockaddr_in
{
    short sin_family; /* AF_INET */
    u_short sin_port; /* 16 bit port number */
    struct in_addr sin_addr; /* 32 bit netid/hostid */
                /* network byte ordered */
    char sin_zero[8]; /* unused */
};
```

The header file `<sys/types.h>` provides C definitions and datatype definitions (typedefs) that are used throughout the system. We will mainly use the names defined for the four unsigned integer datatypes, as shown below.



### **3.4 Socket System Calls Used**

In this section, we describe the elementary system calls required to perform network programming.

#### **1. *socket* System Call:**

To do network I/O, the first thing a process must do is call the socket system call, specifying the type of communication protocol desired (Internet TCP, Internet UDP, etc.)

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

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

family - Specifies the protocol to be used

Type - Specifies the socket type to be used.

Protocol - Specifies a particular protocol.

#### **2. *bind* System Call:**

The *bind* system call assigns a name to an unnamed socket.

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

```
int bind (int sockfd , struct sockaddr *myaddr , int addrlen);
```

sockfd - is the socket file descriptor

myaddr - is a pointer a protocol-specific address

addrlen - is the size of the address structure.

Uses of *bind* system call:

1. Servers register their well known address with the system. It tells the system “this is my address and any messages received for this address are to be given to me.” Both connection oriented and



connectionless servers need to do this before accepting client requests.

2. A client can register a specific address for itself.
3. A connectionless client needs to assume that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to. This corresponds to making certain an envelope has a valid return address, if we expect to get a reply from the person we send the letter to.

The *bind* system call fills in the *local-addr* and local-process elements of the association 5-tuple.

### 3. *send* System Call:

The *send* system call is used to make a request or send a reply over a connected socket.

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

```
int send (int sockfd , char *buff , int nbytes , int flags);
```

sockfd - is the socket file descriptor  
buff - is a temporary buffer  
nbytes - is th length of path of the filename  
flags - is formed by ORing one of the following constants

### 4. *recv* System Call:

The *recv* system call is used to receive data from a particular sender over a connected socket.

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

```
int recvfrom ( int sockfd , char *buff , int nbytes , int flags );
```

sockfd - is the socket file descriptor  
buff - is a temporary buffer  
nbytes - is th length of path of the filename  
flags - is formed by ORing one of the following constants



## **4. CODING AND TESTING**



## **4.1 Source Coding**

```
//Client Program

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netinet/in.h>
#include <errno.h>
#include <stdlib.h>

//----- #define's

#define PORT 3033
#define IO_BUFFER 255

//----- function prototypes
void doIntro(void);
void openChat(void);
void doParentTasks(void);
void doChildTasks(void);
pid_t Fork(void);
//----- global variables
```



## Online Chat Server

18

```
char CONNECTION_REQUEST[5] = "+EST";    // client requests connection
char ACCEPT_CONNECTION[5] = "+CON";     // server accepts connection
char NOT_ACCEPTED[5] = "-NOA";         // server is full and
                                        // did not accept connection
char DISCONNECTING[5] = "-DIS";        // the client is going to disconnect

char send_data[100];
char prompt[32];
char username[32];

int sockfd;
struct sockaddr_in address;
char input[IO_BUFFER], output[IO_BUFFER];

char *argv_1;        // export the argv so that we can use it else where.

char temp[20];

//----- int main(int,char**)

int main(int argc, char **argv)
{
    pid_t cpid;

    if(argc != 2){                // check for correct command line args.
        printf("usage: %s IPaddress\n",argv[0]);
        return -1;
    }
}
```



```
argv_1 = argv[1];           // export argv for use in doParentTasks.

doIntro();                  // get username

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{   // create the socket object.
    printf("Error, could not allocate socket.\n");
    return(-1);
}

memset(&address, 0, sizeof(address)); // set the socket object to 0.

address.sin_family = AF_INET;      // say the socket is an Inet socket.
address.sin_port = htons(PORT);    // specify the socket port.

if(inet_pton(AF_INET, argv[1], &address.sin_addr) <= 0)
{   // is it a valid IP address?
    printf("Error, domain %s invalid.", argv[1]);
    return(-1);
}

// can we connect?
if(connect(sockfd, (struct sockaddr *)&address, sizeof(address)) < 0)
{
    printf("Error, could not establish connection. (errno %d).\n",errno);
    return (-1);
}

openChat();                 // can we login to the server?
```



```
// now we are going to fork so that we may wait on two inputs, one from
// the server and one from the user at stdin.

if( (cpid = Fork()) == 0)
{ // now we are a child process
  doChildTasks();
  return 0; // so we never get to the parent loop.
}

doParentTasks(); // now the parent goes to work.

sprintf(temp, "%s %d", "kill -9 ", cpid);
printf("Killing child process with a system call: %s\n", temp);
system(temp);

close(sockfd); // now we are all done.
return 0;
}

//----- void doIntro(void)

void doIntro(void)
{
  printf("\n*****\n");
  printf("* The Ultimate Chat Client.\n");
  printf("*\n");
  printf("* Please enter your name: ");
}
```



```
scanf("%s", username);

printf("*\n");
printf("* Welcome %s !\n", username);

strcat(prompt, "Send > ");

printf("*\n");
printf("* Your prompt has been set to: %s\n", prompt);
printf("* \n");
printf("*****\n");
}

//----- void openChat(void)

void openChat(void)
{

write(sockfd, &CONNECTION_REQUEST, IO_BUFFER); // send a connection request
read(sockfd, &input, IO_BUFFER);

if( strcmp(input, ACCEPT_CONNECTION, 4) )
{
printf("Server is too busy, try back later. %s\n", input);
close(sockfd);
exit(0);
}
else
{
```



```
    printf("Connected to Chat server!\n");
}
}

//----- void doParentTasks(void)

void doParentTasks(void)
{

    char temp[IO_BUFFER];
    int tsockfd;           // temporary socket stuff.
    struct sockaddr_in address;    // more temp socket stuff.
    char quit[6] = "/quit";

    printf("%s", prompt);
    getchar();

    for(;;)
    {
        fgets(temp, IO_BUFFER, stdin);

        printf("%s", prompt);

        strcpy(output, username);
        strcat(output, " - ");
        strcat(output, temp);

        // Client has something to send so we will open up a
        // socket to send it to Server.
```



```
if((tsockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    printf("Error, could not allocate socket.\n");
    exit(-1);
}

memset(&taddress, 0, sizeof(taddress));

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

if(inet_pton(AF_INET, argv_1, &taddress.sin_addr) <= 0)
{
    printf("Error, domain %s invalid.", argv_1);
    exit(-1);
}

if(connect(tsockfd, (struct sockaddr *)&taddress, sizeof(taddress))<0)
{
    printf("Error, could not establish connection (errno %d).\n",errno);
    exit (-1);
}

// done creating a socket.

if( !strncmp(temp, quit, 5) )
{
    printf("Preparing to exit ...\n");
}
```



## Online Chat Server

24

```
    sprintf(temp, "%s %d", DISCONNECTING, sockfd);
    printf("Sending \"%s\" to the server...\n", temp);
    write(tsockfd, &temp, IO_BUFFER);
    break;
}/* endif */
```

```
write(tsockfd, &output, IO_BUFFER);
```

```
}
```

```
}
```

```
//----- void doChildTasks(void)
```

```
void doChildTasks(void)
```

```
{
```

```
    for(;;)
```

```
    {
```

```
        read(sockfd, &input, IO_BUFFER);
```

```
//        write(stdout, &input, IO_BUFFER);
```

```
        printf("%s\n", input);
```

```
    }
```

```
}
```

```
//----- pid_t Fork(void)
```

```
pid_t Fork(void)
```

```
{
```



```
pid_t pid;

if( (pid = fork()) == -1)
{
    printf("Error forking, your are dead!\n");
    exit(-1);
}

return pid;
}
```

```
// Server Program
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#include <stdio.h>
```

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

```
#include <arpa/inet.h>
```

```
#include <unistd.h>
```

```
#include <errno.h>
```

```
//----- #define's
```

```
#define PORT 3033
```

```
#define LISTEN_QUEUE_SIZE 5
```

```
#define MAX_CONNECTIONS 5 //50
```

```
#define IO_BUFFER 255
```

```
#define CLIENTFD_INIT -1
```



```
//----- global vars

char CONNECTION_REQUEST[5] = "+EST";    // client requests connection
char ACCEPT_CONNECTION[5] = "+CON";    // server accepts connecton
char NOT_ACCEPTED[5] = "-NOA";        // server is full and
                                        // did not accept connection
char DISCONNECTING[5] = "-DIS";        // client requesting to disconnect

int clientfd[MAX_CONNECTIONS];        // array of connected clients' sockfd's
int current_connections = 0;          // number of clients currently connected

int listenfd;                          // fd of the listening socket.
int sockfd;                             // fd of the synchronous socket.
struct sockaddr_in address;            // sockfd's struct object.

char input[IO_BUFFER], output[IO_BUFFER]; // input and output buffers.
char *temp;
char tempbuf[6];
int i;

//----- int main()

int main()
{
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = htonl(INADDR_ANY);
```



```
address.sin_port = htons(PORT);
bind(listenfd, (struct sockaddr *)&address, sizeof(address));
listen(listenfd, LISTEN_QUEUE_SIZE);
for(i = 0; i <= MAX_CONNECTIONS-1; i++)
{
    clientfd[i] = CLIENTFD_INIT;
}/* endfor */
for(;;)
{
    sockfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
    read(sockfd, &input, IO_BUFFER);
    printf("\n\nServer receives: %s\n",input);
    if( !strncmp(input, CONNECTION_REQUEST, 4) )
    { // if it is a connection request.

        printf("Server has a connection request.\n");

        if( current_connections < MAX_CONNECTIONS )
        {
            clientfd[current_connections] = sockfd;
            current_connections++;
            if(current_connections < MAX_CONNECTIONS)
                printf("There is room for another connection.\n");
            else
                printf("The Chat room is full. No more connections.\n");
            write(sockfd, &ACCEPT_CONNECTION, IO_BUFFER);
        }
    }
    else
    {
```



```
printf("Server is full!\n");
    write(sockfd, &NOT_ACCEPTED, IO_BUFFER);
}
}

/* Disconnect request from client */
else if( !strcmp(input, DISCONNECTING, 4) )
{
    printf("Server received a disconnect request with sockfd == ");
    sockfd = -1;
    sscanf( input, "%5c%d", tempbuf, &sockfd);
    printf( "%d\n", sockfd);

    if(sockfd == -1)
    {
        printf("Error while reading -DIS...\n");
        printf("This data will be ignored.\n");
    }/* endif */

    else
    {
        printf("Successful -DIS reading\n");

        while( clientfd[i] != sockfd && i <= MAX_CONNECTIONS-1 )
        {
            i++;
        }
        clientfd[i] = CLIENTFD_INIT;
        current_connections--;
    }
}
```



```
        }/* endelse */
    }/* endif */

    /* Input data will be sent to all connected users */
    else
    {
        printf("Printing this out to all clients: \n");
        printf("\t%s\n", input);

        /* writing input to all active sockets */
        for(i = 0; i <= MAX_CONNECTIONS-1; i++)
        {
            sockfd = clientfd[i];
            if( sockfd != CLIENTFD_INIT)
            {
                write(sockfd, &input, IO_BUFFER);
            }/* endif */
        }/* endfor */
    }/* endelse */
}
return 0;
}
```



## 4.2 TESTING



Testing forms an important part of the project work.

We implemented module testing. As soon as a module was developed, we subjected it to testing to see that it worked flawlessly to satisfy its intended requirements.

The project is found working efficiently by thorough testing like Unit testing, Integration testing and System Testing.



## **5. USER MANUAL**

**User can run the module as follows:**

### **Step 1: First Compile & Start the Chat Server**

**Open the Console at the folder where cserver.c exists & Pass the following Commands.**

-----  
**gcc cserver.c**

**./a.out**  
-----

**Now Server is Started & actively waiting for Clint connections**

**The Clint may be n number.**



**step 2 : compile & Executed the Chat Clint**

**Open the Console at the folder where cserver.c exists & Pass the following Commands.**

-----  
**gcc cclient.c**

**./a.out <IP address of the Server Machine where the chat server is executing >**

**Example : ./a.out 127.0.0.1**  
-----

**run the server (cserver.c) in any machine connected to the clients**

**run the client in any other machine or in different login of the same machine in which server is running.**

**run as many as clients you want, but we have limited it to 50.**



## 6. Output of the Projects



**Output 1 : Server Side out put where Server accepted 2 connection and ready to connect other clients.**

```
[swarna@localhost surv]$ ./a.out  
  
Server receives: +EST  
Server has a connection request.  
There is room for another connection.  
  
Server receives: +EST  
Server has a connection request.  
There is room for another connection.
```



**Output 2: 1<sup>st</sup> Clint side output from first server where Clint is logged in server**

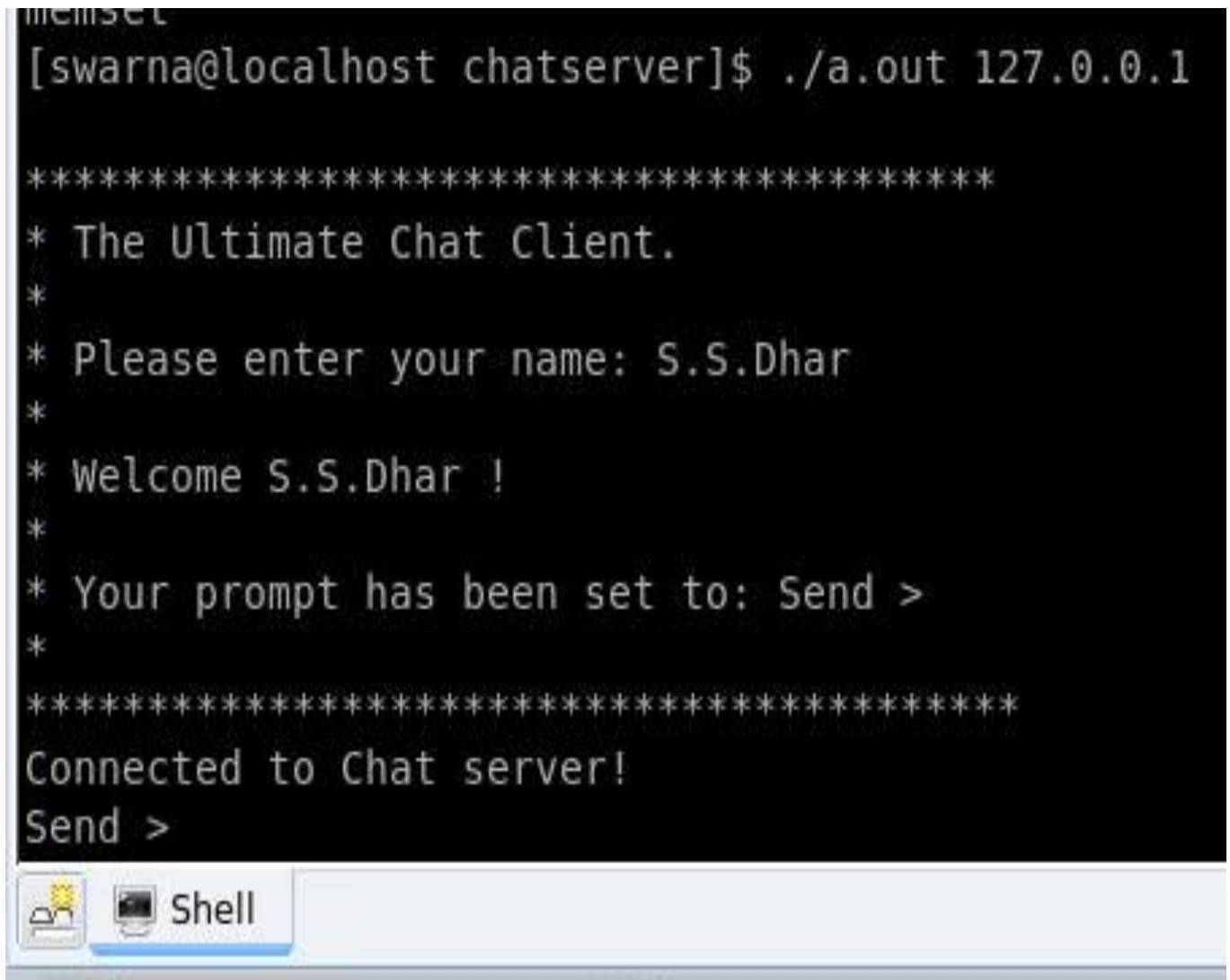
```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: Baburao
*
* Welcome Baburao !
*
* Your prompt has been set to: Send >
*
*****
Connected to Chat server!
Send > █
```



Output 3: 2<sup>nd</sup> out put of 2<sup>nd</sup> Clint Logged on to the server

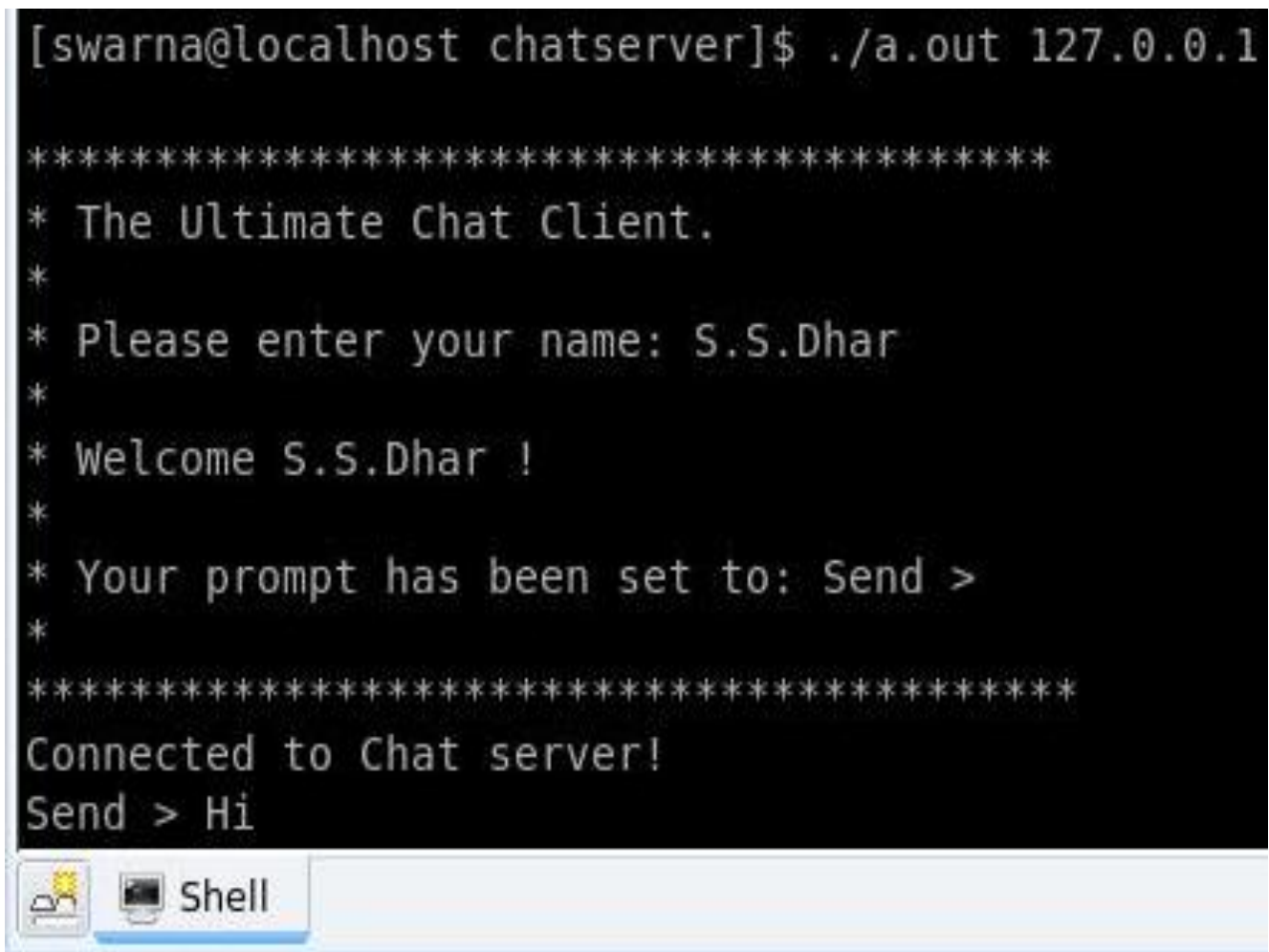
```
memisc
[swarna@localhost chatserver]$ ./a.out 127.0.0.1

*****
* The Ultimate Chat Client.
*
* Please enter your name: S.S.Dhar
*
* Welcome S.S.Dhar !
*
* Your prompt has been set to: Send >
*
*****
Connected to Chat server!
Send >
```

A terminal window titled "Shell" with a taskbar icon of a computer monitor. The terminal output shows the execution of a chat client program. The user enters the IP address 127.0.0.1. The program displays a series of asterisks, then a welcome message for the user "S.S.Dhar", and finally "Connected to Chat server!". The prompt is set to "Send >".

**Output 4: Clint 1 sending message to others**

```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: S.S.Dhar
*
* Welcome S.S.Dhar !
*
* Your prompt has been set to: Send >
*
*****
Connected to Chat server!
Send > Hi
```

A screenshot of a terminal window titled "Shell" with a taskbar icon. The terminal output shows the execution of a chat client program. It prompts for a name, which is "S.S.Dhar", and then displays a welcome message. The prompt is set to "Send >". The user enters "Hi", and the terminal shows "Connected to Chat server!" and "Send > Hi".

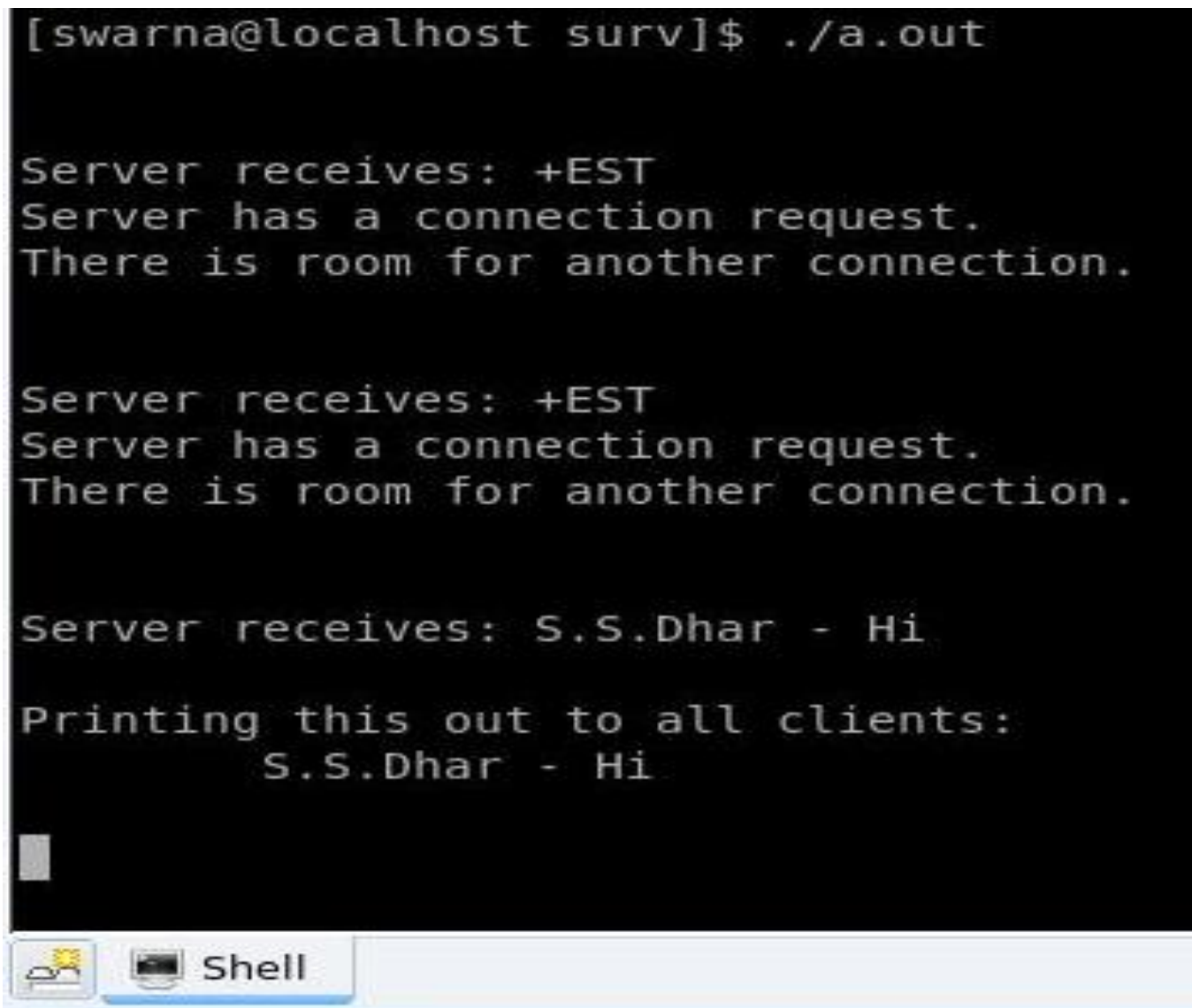
**Output 5: Server Side Activity on Clint side message sending**

```
[swarna@localhost surv]$ ./a.out

Server receives: +EST
Server has a connection request.
There is room for another connection.

Server receives: +EST
Server has a connection request.
There is room for another connection.

Server receives: S.S.Dhar - Hi
Printing this out to all clients:
    S.S.Dhar - Hi
```

A terminal window titled "Shell" with a light blue title bar. The terminal output shows the execution of a program named 'a.out'. The program receives two '+EST' connection requests and then receives a message 'S.S.Dhar - Hi'. It then prints this message to all clients. The terminal background is black with white text. At the bottom left of the terminal window, there are three icons: a yellow sun, a computer monitor, and a shell icon.

**It Receive the Message & pass to all Clint's**

**Output 6: Sample Output for Passing Messages on server side Activity**

```
Server has a connection request.  
There is room for another connection.  
  
Server receives: S.S.Dhar - Hi  
Printing this out to all clients:  
    S.S.Dhar - Hi  
  
Server receives: Baburao - hellow ! How r U Dude  
Printing this out to all clients:  
    Baburao - hellow ! How r U Dude  
  
Server receives: S.S.Dhar - I am Fine what abpot u  
Printing this out to all clients:  
    S.S.Dhar - I am Fine what abpot u
```



**Output 7: Some Client Side Message passing activity**

```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: Baburao
*
* Welcome Baburao !
*
* Your prompt has been set to: Send >
*
*****
Connected to Chat server!
Send > S.S.Dhar - Hi

hellow ! How r U Dude
Send > Baburao - hellow ! How r U Dude

S.S.Dhar - I am Fine what abpot u

Same here
Send > Baburao - Same here

Ok bye
Send > Baburao - Ok bye
```

**Output 8:**

```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: S.S.Dhar
*
* Welcome S.S.Dhar !
*
* Your prompt has been set to: Send >
*
*****
Connected to Chat server!
Send > Hi
Send > S.S.Dhar - Hi

Baburao - hellow ! How r U Dude

I am Fine what abpot u
Send > S.S.Dhar - I am Fine what abpot u

Baburao - Same here

Baburao - Ok bye

ok tata
Send > S.S.Dhar - ok tata
```

**Output 9:**

```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: Baburao
*
* Welcome Baburao !
*
* Your prompt has been set to: Send >
*
*****
Connected to Chat server!
Send > S.S.Dhar - Hi

hellow ! How r U Dude
Send > Baburao - hellow ! How r U Dude

S.S.Dhar - I am Fine what abpot u

Same here
Send > Baburao - Same here

Ok bye
Send > Baburao - Ok bye

S.S.Dhar - ok tata

exit
Send > Baburao - exit

[1]+  Stopped                  ./a.out 127.0.0.1
[swarna@localhost chatserver]$
```

**Output 10 :**

```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: S.S.Dhar
*
* Welcome S.S.Dhar !
*
* Your prompt has been set to: Send >
*
*****
Connected to Chat server!
Send > Hi
Send > S.S.Dhar - Hi

Baburao - hellow ! How r U Dude

I am Fine what abpot u
Send > S.S.Dhar - I am Fine what abpot u

Baburao - Same here

Baburao - Ok bye

ok tata
Send > S.S.Dhar - ok tata

Baburao - exit
```



## Errors

### Error Output 1:



**Run time Error when Server is Disconnected or Terminated.**



**Error Output2:** It happens when Clint request for connection & Server is not running or given wrong IP address of Server.

```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: swarna
*
* Welcome swarna !
*
* Your prompt has been set to: Send >
*
*****
Error, could not establish connection. (errno 111).
[swarna@localhost chatserver]$ █
```

## Rejection of Client Request

When server is busy it rejects more client connections. Then it sends a refusal message to the client.

Server side activity:

```
[swarna@localhost surv]$ ./a.out

Server receives: +EST
Server has a connection request.
There is room for another connection.

Server receives: +EST
Server has a connection request.
There is room for another connection.

Server receives: +EST
Server has a connection request.
There is room for another connection.

Server receives: +EST
Server has a connection request.
There is room for another connection.

Server receives: +EST
Server has a connection request.
The Chat room is full. No more connections.

Server receives: +EST
Server has a connection request.
Server is full!
```



**Client side activity:**

```
[swarna@localhost chatserver]$ ./a.out 127.0.0.1
*****
* The Ultimate Chat Client.
*
* Please enter your name: other
*
* Welcome other !
*
* Your prompt has been set to: Send >
*
*****
Server is too busy, try back later. -NOA
[swarna@localhost chatserver]$
```

**\* In this Project the Server Client acceptance limit is 5 that can be increased or decreased by changing the code.**

## **7. CONCLUSION AND SCOPE FOR FURTHER ENHANCEMENT**



## **7.1 CONCLUSION**

On **Conclusion**, the project titled “Conferencing” was completed in accordance to the requirements stated in the problem definition.

## **7.2 SCOPE FOR FUTURE ENHANCEMENT**

**The project can be further enhanced by introducing the following features:**

- Transferring of sound files, video files, picture files etc can be made possible.
- Module can be made to include ready made messages like birthday wishes, wishes to congratulate etc. so that the user can just select and send the mails without spending much of the time.
- Permanent User ID that keeps User Chat history & User Information for Security purpose.
- Graphical user Interface for user friendly Environment.



## **8. BIBLIOGRAPHY**

No stones have been left unturned; rather no relevant books have been spared being read, for the successful completion of the project.

The following books were the main books that were referred to implement the project:

1. **UNIX NETWORK PROGRAMMING – Richard Stevens.**
2. **INTERNET PROGRAMMING –Kris Jamsa.**
3. **LINUX PROGRAMMING – PROGRAMMERS TO PROGRAMMERS SERIES.**
4. **The ‘C’ Odyssey –Gandhy Setti, Saha**

SWARNA SEKHAR DHAR



**T.John College**  
(AFFILIATED TO BANGALORE UNIVERSITY)  
Approved by AICTE

