ELECTRICAL & COMPUTER ENGINEERING
TEXAS A&M UNIVERSITY

SHASHANK KAMATH KALASA MOHANDAS
UIN: **627003580**

SRINIVAS PRAHALAD SUMUKHA
UIN: **627008254**

TEAM 6
ASSIGNMENT 1

Computer Communication and Networks
ECEN 602
FALL 2018

Role of SHASHANK KAMATH KALASA MOHANDAS

- server.cpp
- README file

Role of SRINIVAS PRAHALAD SUMUKHA

- client.cpp
- MAKEFILE

How to run:

1. Run the makefile in the terminal by typing the command 'make'
2. Two object files named client and server gets created. Execute the server program by typing './server *port_number*'. The server gets executed.
3. In a new terminal run the client program by typing './client *ip_address port_number*'. The client gets executed. Now type any message to communicate and the server echoes back the message.

   NOTE: ip_address for same machine communication (loop back address): "127.0.0.1"

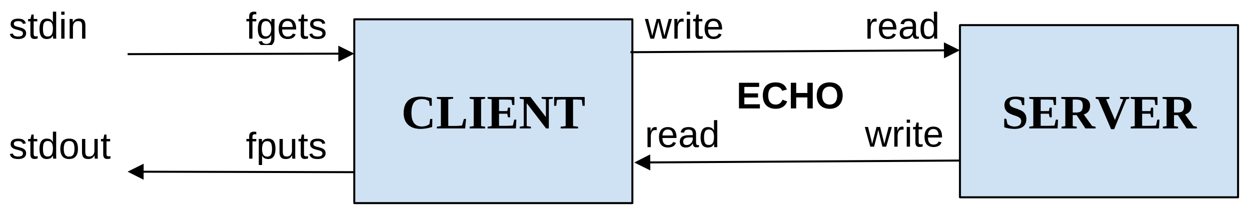
Files included in this assignment:

- server.cpp
- client.cpp
- makefile
- README

MAKEFILE:

```
# Owner: Srinivas Prahalad Sumukha\
UIN: 627008254\
Functionality: Makefile to complie client and server\

output: client server #make if there is any changes to client or server

A: client.cpp #compiles client.cpp
    gcc -c client.cpp
    #gcc -o client client.cpp

B: server.cpp #complies server.cpp
    gcc -c server.cpp
    #gcc -o server server.cpp

clean:
    rm client server server.o
```

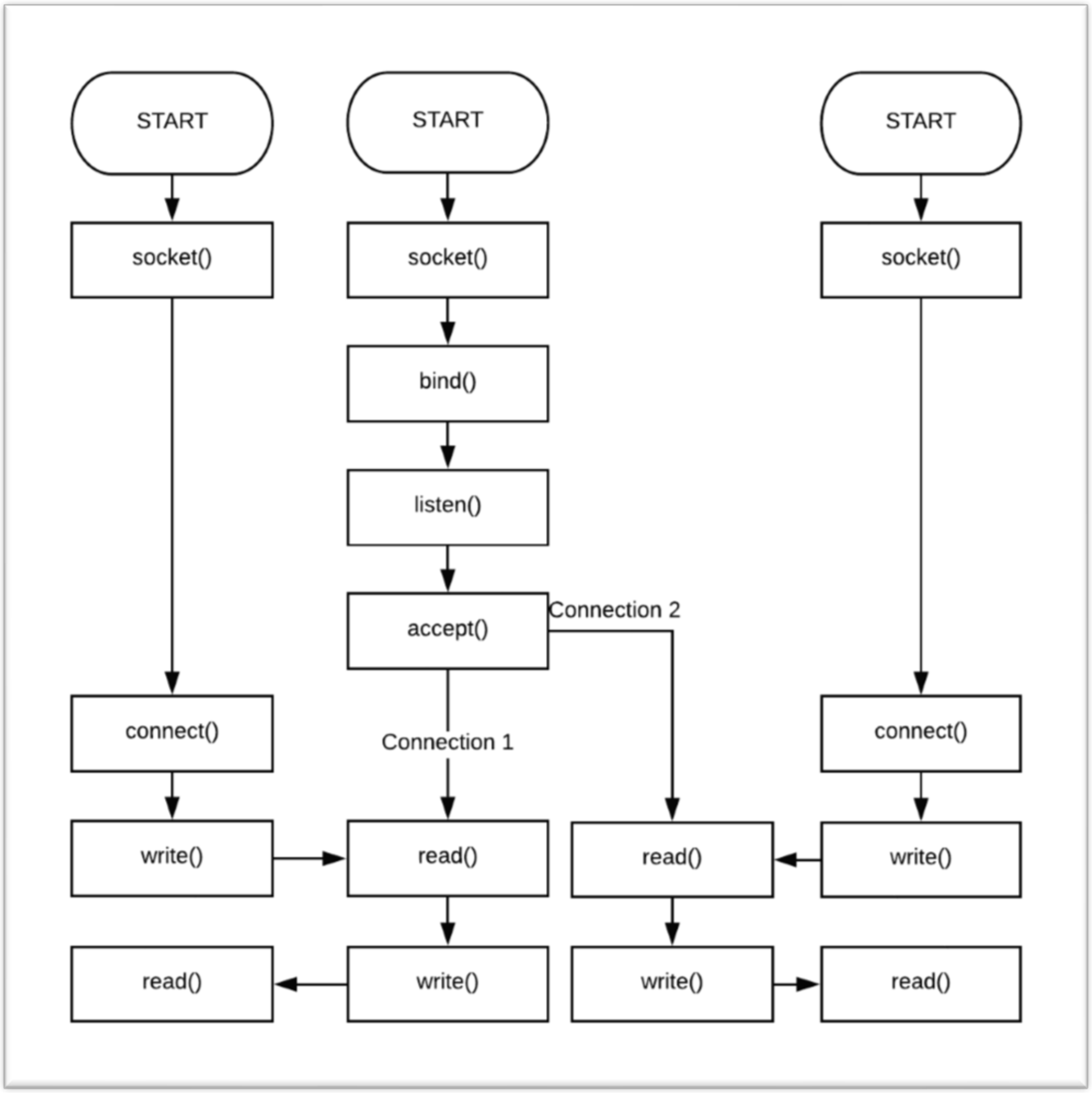
ARCHITECTURE



Our TCP Echo Client and Server follows the architecture as shown in the above figure. The client is responsible for taking the standard input from the user and to present the output on the standard output device (display monitor). The client uses the fgets() function to take the user input. The data got from the standard input is written onto the server via the connected socket. But before writing the data on the socket, an initial connection setup needs to happen between the server and client. Firstly, the server creates a socket dedicated for the connection and binds the socket. This socket keeps listening for incoming connections and accepts the connections. Upon the connection getting accepted, the client and server can communicate between each other via a new socket created. Once the server accepts the connection, it receives the data sent by the client. The main part of this assignment happens at this point i.e., the server sends back the received data to the client. Therefore, whatever data the client sends to the server, ultimately reaches back to the client. The client upon receiving back the data displays it on the monitor. The server is designed to handle multiple client connections using the fork() command. Therefore, the server can service multiple client programs at once.

FLOWCHART

This flowchart represents the important functions required for the functioning of the client and server programs. In the above flowchart there are two client programs connected to a single server. The server is capable of supporting multiple clients at the same time.



SERVER CODE:

server.cpp

```
/**************************************************************************
*******************************************
    Owner Name: SHASHANK KAMATH KALASA MOHANDAS
    UIN: 627003580
    Program functionality: SERVER program
```

```
****************************************************************************
*******************************************/

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

#define BACKLOG 10
#define MAXDATASIZE 1024
/***************************************************************************
*******************************************
The getaddress is a user defined function which compares the family to IPv4
or IPv6 and returns the IP address.
****************************************************************************
*******************************************/
void *getaddress(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET)
    {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }
        return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

/***************************************************************************
*******************************************
The getport is a user defined function which compares the family to IPv4 or
IPv6 and returns the port number.
****************************************************************************
*******************************************/
void *getport(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET)
    {
        return &(((struct sockaddr_in*)sa)->sin_port);
    }
        return &(((struct sockaddr_in6*)sa)->sin6_port);
}

int main(int argc, char *argv[])
{
    char *PORT;
    char s[INET6_ADDRSTRLEN];
    char message[MAXDATASIZE]={0}; //Creating a buffer to hold the data
received

    int sfd, newsfd, flag, numbytes, yes=1;

    struct addrinfo addressinfo, *servicelist, *loopvariable;
    struct sockaddr_storage str_addr;
    socklen_t addr_size;
    pid_t childpid; // Creating a variable to hold child process PID

    PORT = argv[1];

    memset(&addressinfo,0,sizeof (addressinfo)); // Making the addressinfo
struct zero
    addressinfo.ai_family = AF_UNSPEC;// Not defining whether the
connection is IPv4 or IPv6
    addressinfo.ai_socktype = SOCK_STREAM;
    addressinfo.ai_flags = AI_PASSIVE;

    if ((flag = getaddrinfo(NULL, PORT, &addressinfo, &servicelist)) != 0)
    {
        printf("GetAddrInfo Error");
        exit(1);
    }
    printf("Server: Done with getaddrinfo \n");
    // Traversing the linked list for creating the socket
    for(loopvariable = servicelist; loopvariable != NULL; loopvariable =
(loopvariable -> ai_next ))
    {
        if((sfd = socket(loopvariable -> ai_family, loopvariable ->
ai_socktype, loopvariable -> ai_protocol)) ==  -1 )
        {
            printf("Server: Socket Created.\n");
            continue;
        }
        if (setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) ==
-1)
        {
            perror("Server: SetSockOpt.\n");
            exit(1);
        }
        //Binding the socket
        if (bind(sfd, loopvariable->ai_addr, loopvariable->ai_addrlen) ==
-1)
        {
            close(sfd);
            perror("Server: Bind Error.\n");
            continue;
        }
        break;
    }
    // Freeing the linked  list
    freeaddrinfo(servicelist);

    if (loopvariable == NULL)
    {
        printf("Server: Failed to bind.\n");
        exit(1);
    }

    // Listen to the connection
    if (listen(sfd, BACKLOG) ==  -1)
    {
        perror("Server: Listen Error");
        exit(0);
    }
    printf("Server: Listening in progress \n");

    //Accept the connection
    while(1)
    {
        addr_size = sizeof str_addr;
        newsfd = accept(sfd, (struct sockaddr *)&str_addr, &addr_size);
            if (newsfd == -1)
            {
                perror("Server: Accept Error");
                exit(1);
            }
        printf("Server: Connection Accepted \n");

        inet_ntop(str_addr.ss_family,  getaddress((struct sockaddr
*)&str_addr), s, sizeof s);
        printf("Server: Connection from %s\n", s);
        //printf("Server: Connection port number: %d \n",
ntohs(getport((struct sockaddr *)&str_addr)));
        //Creating a child process for multiple connections
        if((childpid = fork()) == 0)
        {
            printf("Server: Child Process created with PID: %d \n",
getpid());
            close(sfd); // Closing the old socket
            while(1)
            {
                if ((numbytes = recv(newsfd, message, MAXDATASIZE-1, 0)) ==
-1) // Receving the data from client
                {
                    printf("Server: Receiving Error");
                    exit(1);
                }
                    message[numbytes] = '\0';
                    printf("Server: Received : '%s'  \n", message);
                    if(strcmp(message,"exit") == 0) // if message is "exit"
then close the new socket
                    {
                        close(newsfd);
                        printf("Server: Client Disconnected");
                        break;
                    }

                    if(send(newsfd, message, strlen(message), 0) ==  -1) //
Sending the data back to client
                    {
                        perror("Server: Send Error");
                        exit(1);
                    }
                    printf("Server: Echoing:  '%s' \n", message);
                    bzero(message, sizeof(message));
            }
        }
    }
    close(newsfd);  // Closing the new socket
    return 0;
}
```

CLIENT CODE:

client.cpp

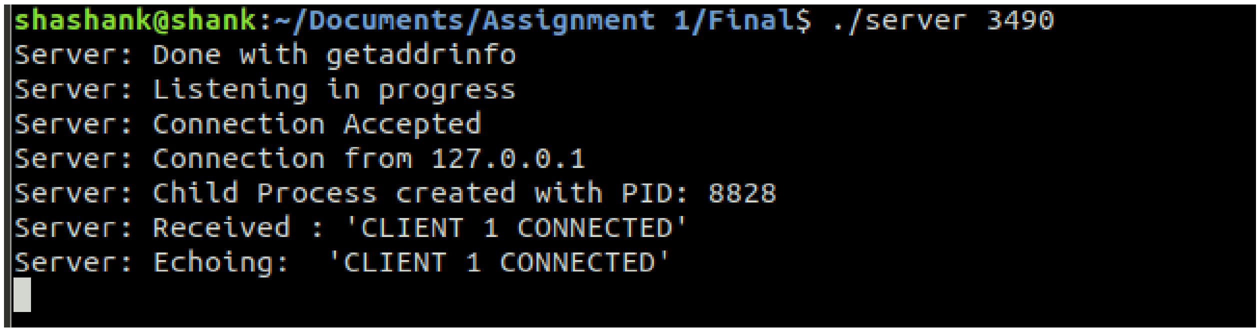
```
/****************************************************************************
*******************************
  Owner Name: Srinivas Prahalad Sumukha
  UIN: 627008254
  Program functionality: client program

****************************************************************************
*****************************/


#include<unistd.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<stdio.h>
#include<netdb.h>
#include<string.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
//#include<unp.h>

#define debug 0
#define String_size 1024
/****************************************************************************
*******************************

  The writen is a user defined function which uses built in function such as
send to send the input message
  to the server. If there is any error while sending the message, function
send returns a negitive value
  which implies it an error. If there isn't any error then the function send
returns a value 0. The function
  writen returns 0 or 1 based on the above explained condition.


****************************************************************************
*******************************/
int writen(int socketfd, char *msg, int str_length, int flag) {
#if debug
    //msg = "Hello world";
    //int string_len= strlen("I am srinivas");
    //int sendto_ack = sendto(socketfd, msg, string_len, 0, server_ptr-
>ai_addr, server_ptr->ai_addrlen);
#endif
    int sendto_ack = send(socketfd, msg, str_length, flag); //checking for
any error while sending
    printf("Client Sending: %s \n",msg);
    return sendto_ack;
    //return 0;
}
/****************************************************************************
********************************

  readline is a user defined function which is used to read the data sent by
the server. readline uses the
  built in function recv to read the data. The function returns 0 if read is
completed otherwise it will
  send negitive value indicating error


****************************************************************************
********************************/
int readline(int socketfd, char buffer[], int receiving_size, int flag) {
    int receive_from_ack = recv(socketfd, buffer, receiving_size, flag);
    if(receive_from_ack < 0)
    {
#if debug
        printf("receiving error");
#endif
        perror("error while receiving");
    }
    buffer[receive_from_ack] = '\0';
#if debug
    printf("\nClient Received: %s \n",buffer);
    printf("\n at the end\n");
#endif
    printf("\nClient Received:");
    fputs(buffer,stdout);
    return receive_from_ack;
}

int main(int argc, char *argv[]) {
    int status,flag=0;
    int string_len;
    char buffer[String_size];
    int receive_from_ack;
    int receiving_size = sizeof (buffer);
    socklen_t *fromlength;
    addrinfo server, *server_ptr;
    void *address_ptr_void;
    char *address,*port;
    char ipstr[INET6_ADDRSTRLEN];
    char input_string[String_size];
    char *msg;
    int socketfd;
    if (argc != 3)
    {

#if debug
        printf("\nEnter Ip and port\n");

#endif
        exit(0);
    }
    address = argv[1];
    port = argv[2];
    server.ai_family = AF_UNSPEC; //can be AF_INET
    server.ai_socktype = SOCK_STREAM;
    server.ai_flags = AI_PASSIVE;
    status = getaddrinfo(address, port, &server, &server_ptr);
    //printf("status = %d \n",status);
    if(status !=0) {
#if debug
        printf("\nunable to connect to ip address %s\n",address);
#endif
        exit(0);
    }
    addrinfo *ptr_copy;
    ptr_copy = server_ptr;

    for(ptr_copy = server_ptr; ptr_copy != NULL; ptr_copy= ptr_copy-
>ai_next) {    //going through the list
        struct sockaddr_in *ptr_with_address = (struct sockaddr_in *)
ptr_copy->ai_addr;
        address_ptr_void = &(ptr_with_address->sin_addr);
    }
    char *ipver = "IPv4";
    inet_ntop(server.ai_family, address_ptr_void, ipstr,sizeof ipstr);
    //printf("\n address is %s",ipstr);

    for(ptr_copy = server_ptr; ptr_copy != NULL; ptr_copy= ptr_copy-
>ai_next) {
        struct sockaddr_in *ptr_with_address = (struct sockaddr_in *)
ptr_copy->ai_addr;
        address_ptr_void = &(ptr_with_address->sin_addr);
    }
    socketfd = socket(server_ptr->ai_family, server_ptr->ai_socktype,
server_ptr->ai_protocol);
    if (socketfd == -1) {
#if debug
        printf("\nError at the socket call\n");
#endif
        perror("Error at socket");
        exit(0);
    }
    int connect_ack = connect(socketfd,server_ptr->ai_addr, server_ptr-
>ai_addrlen); //returns negitive value on failure to connect
    if(connect_ack == -1) {
#if debug
        printf("\ncould not connect\n");
#endif
        perror("Problem with connect");
        exit(0);
    }
    while(1)
    {
        printf("\nEnter the string\n");
        char * check_end = fgets(input_string,100,stdin);
        if(check_end == NULL) //send exit if ctrl D is pressed
        {
            msg = "exit";
            int sending_data = writen(socketfd, msg, string_len, flag);
            exit(0);
        }
        msg = &input_string[0]; // user input to message pointer
#if debug
        //strtok(msg, "\n");
#endif
        strtok(msg, "\n");
        string_len= strlen(input_string);
        int sending_data = writen(socketfd, msg, string_len, flag); //
function call to write
        if(sending_data == -1)
        {
            printf("\nerror sending\n");
        }
        int receiving_data = readline(socketfd,buffer,receiving_size, flag);
// function call to read
        if(receiving_data == -1) {
            printf("\nerror sending\n");
        }
#if debug
        else {
            printf("sending data...\n");
        }
#endif
    }
    close(socketfd);
    freeaddrinfo(server_ptr);
    return 0;
}
```

TEST CASES:

All the test cases are done using a loopback IP address “127.0.0.1” and the port number of “3490”.

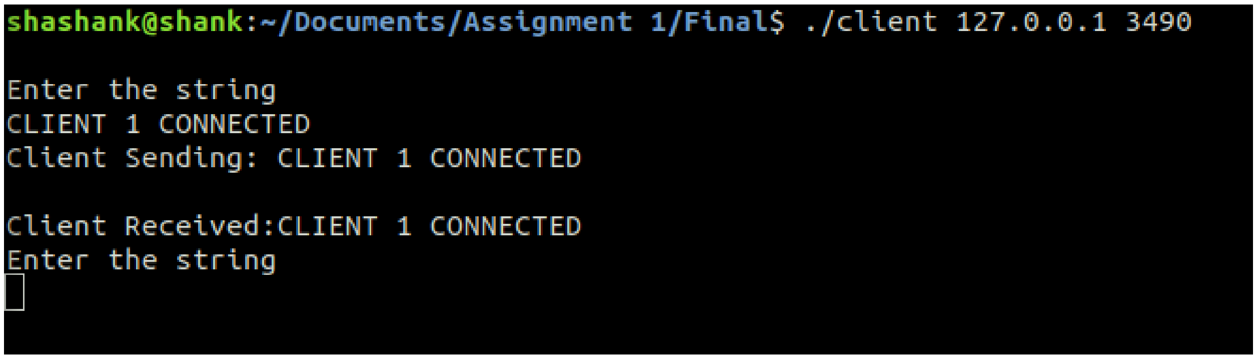
1. **Client connected to the server:**

Server Terminal:



```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8828
Server: Received : 'CLIENT 1 CONNECTED'
Server: Echoing:  'CLIENT 1 CONNECTED'
```

Client Terminal:

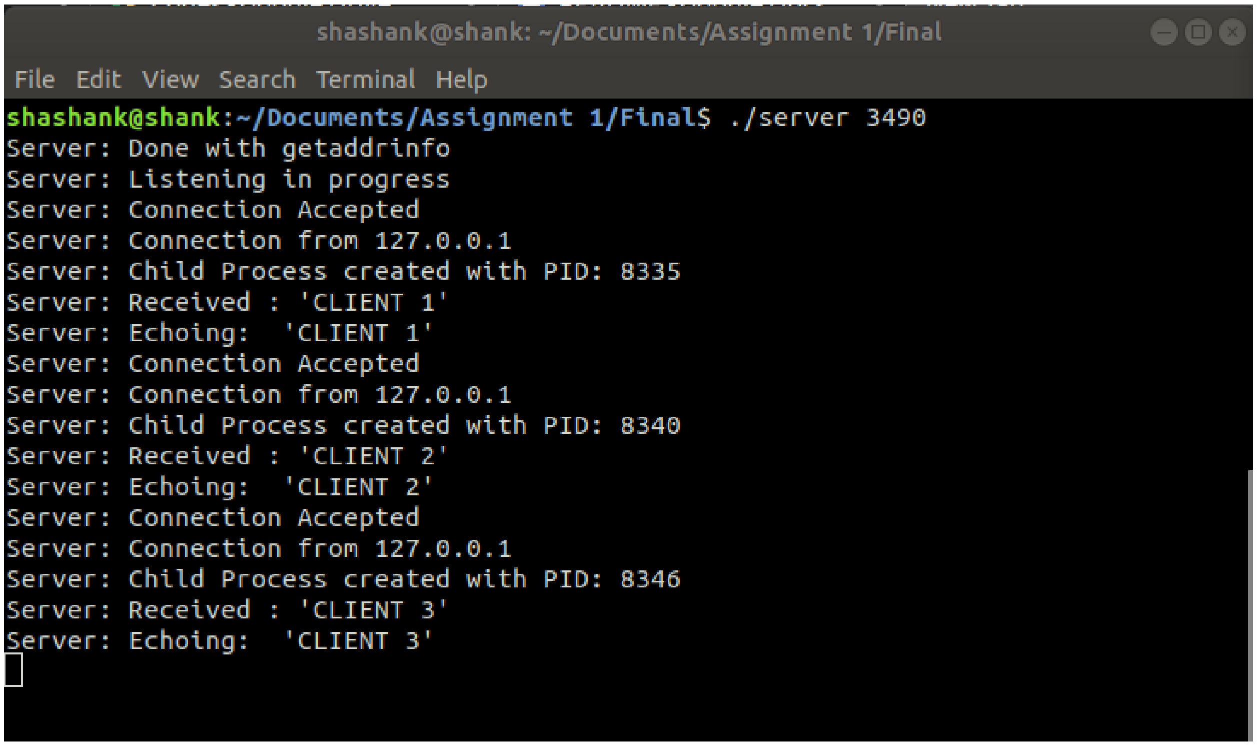


```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
CLIENT 1 CONNECTED
Client Sending: CLIENT 1 CONNECTED

Client Received:CLIENT 1 CONNECTED
Enter the string
```

In this, 1 client is connected to the server and “CLIENT 1” data is being exchanged.

2. **3 Clients connected to the server:**

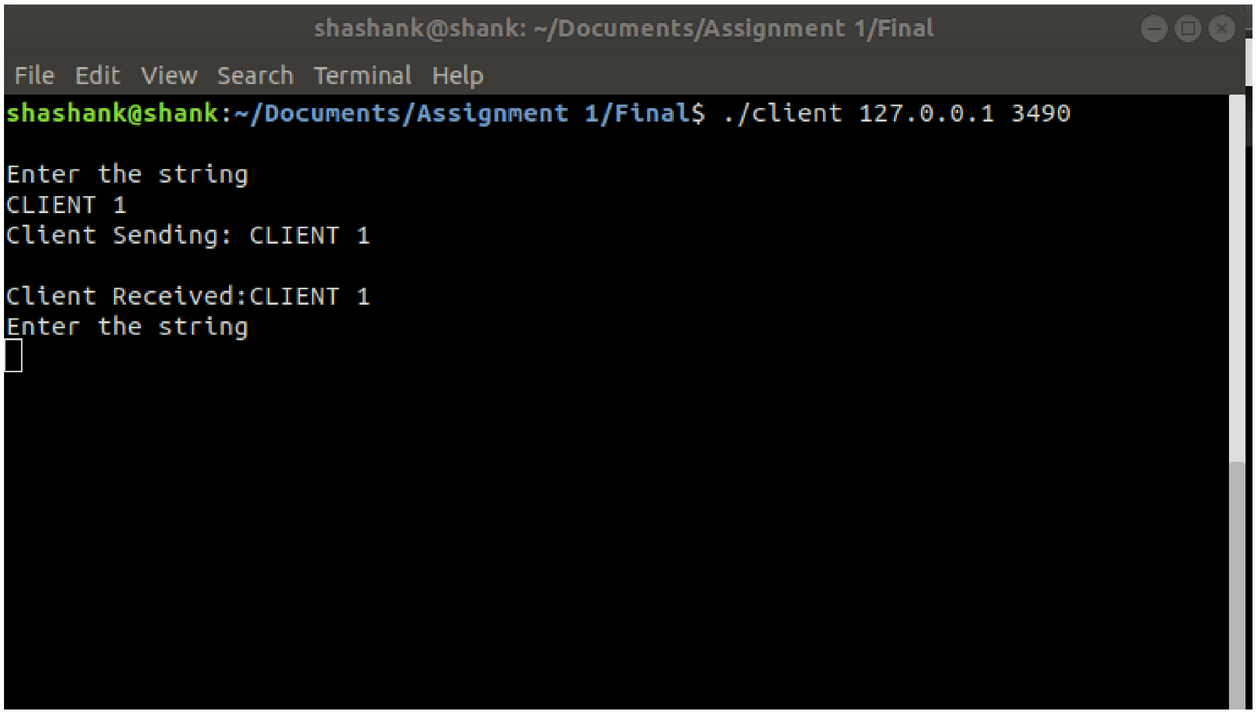
Server Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8335
Server: Received : 'CLIENT 1'
Server: Echoing:  'CLIENT 1'
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8340
Server: Received : 'CLIENT 2'
Server: Echoing:  'CLIENT 2'
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8346
Server: Received : 'CLIENT 3'
Server: Echoing:  'CLIENT 3'
```

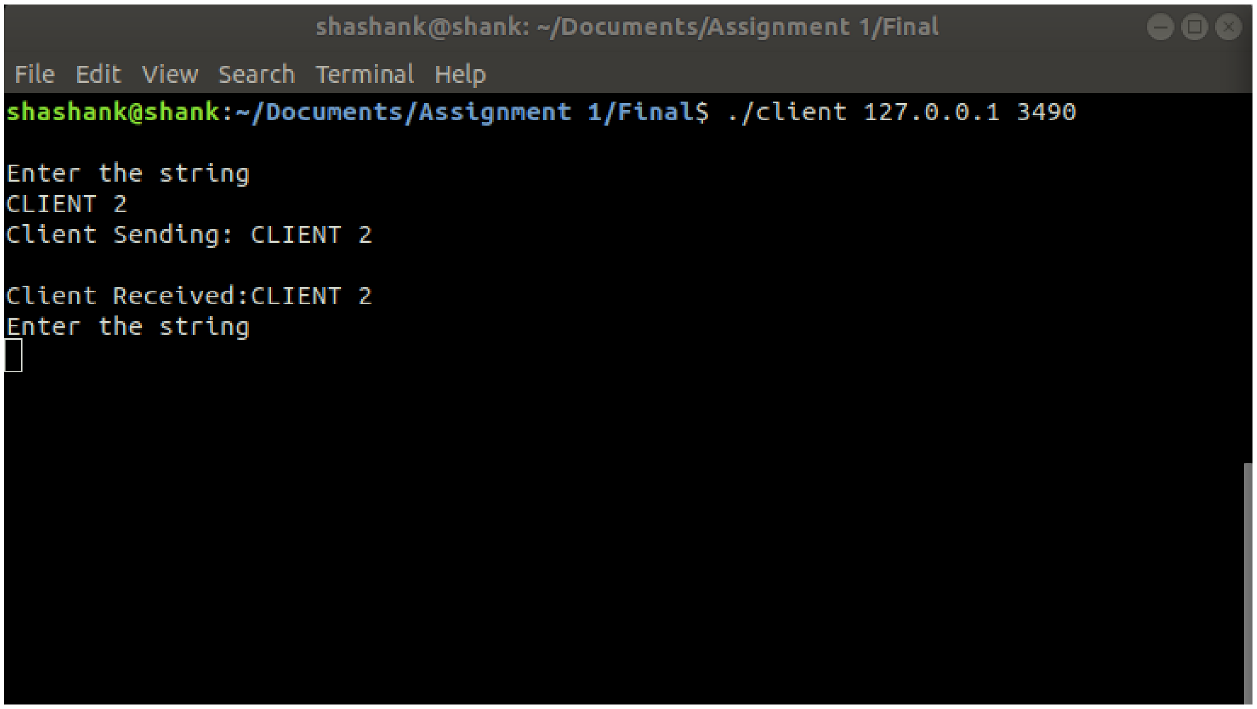
In this, 3 clients is connected to the server and correspondingly “CLIENT 1”, “CLIENT 2” and “CLIENT 3” data are being exchanged between the server and the clients. We can observe that for each client a new process is being created in the server and the process ID is being displayed along with the IP address of the connection.

Client Terminal 1:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

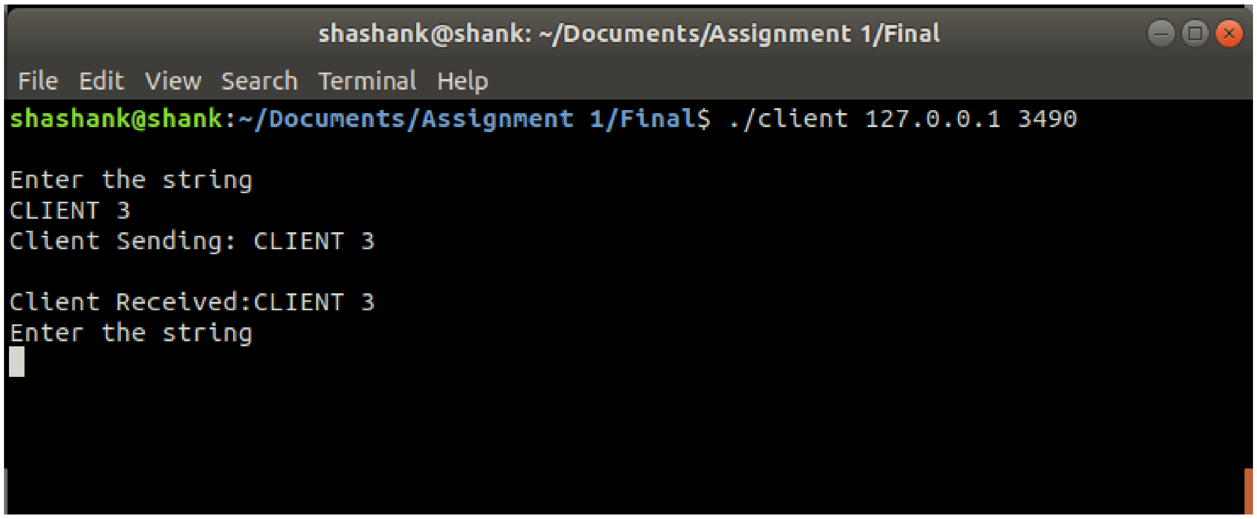
```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
CLIENT 1
Client Sending: CLIENT 1

Client Received:CLIENT 1
Enter the string
```

Client Terminal 2:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

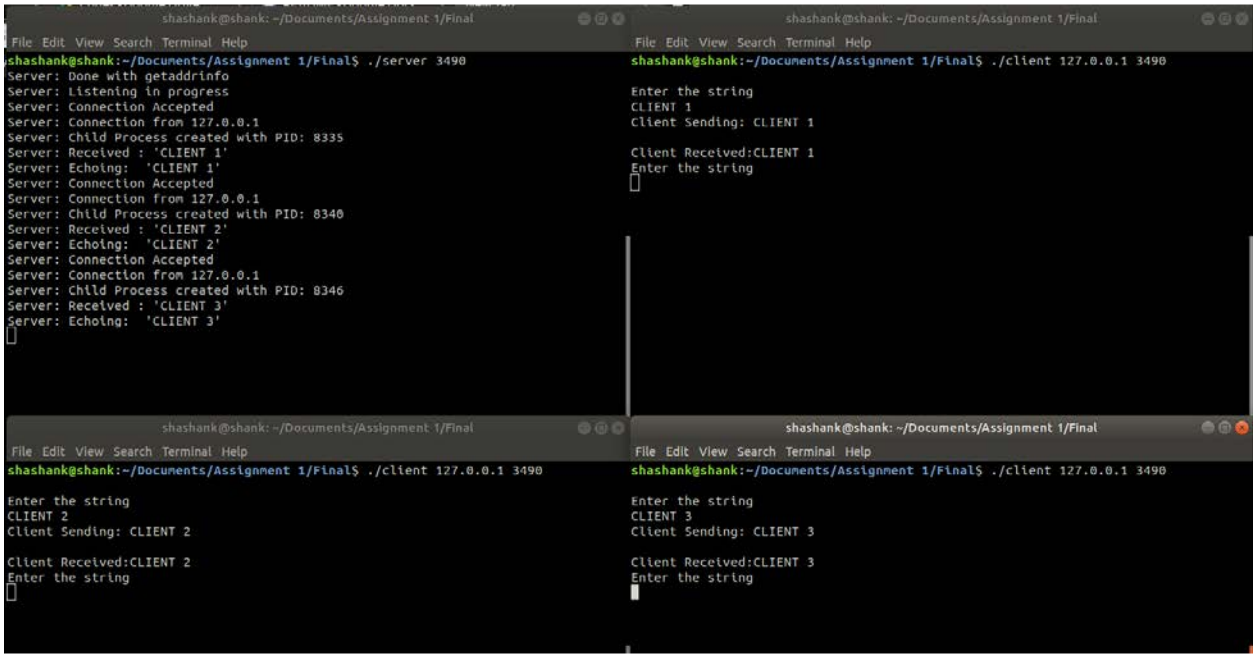
```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
CLIENT 2
Client Sending: CLIENT 2

Client Received:CLIENT 2
Enter the string
```

Client Terminal 3:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
CLIENT 3
Client Sending: CLIENT 3

Client Received:CLIENT 3
Enter the string
```

All terminals together.


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8335
Server: Received : 'CLIENT 1'
Server: Echoing:  'CLIENT 1'
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8340
Server: Received : 'CLIENT 2'
Server: Echoing:  'CLIENT 2'
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8346
Server: Received : 'CLIENT 3'
Server: Echoing:  'CLIENT 3'
```

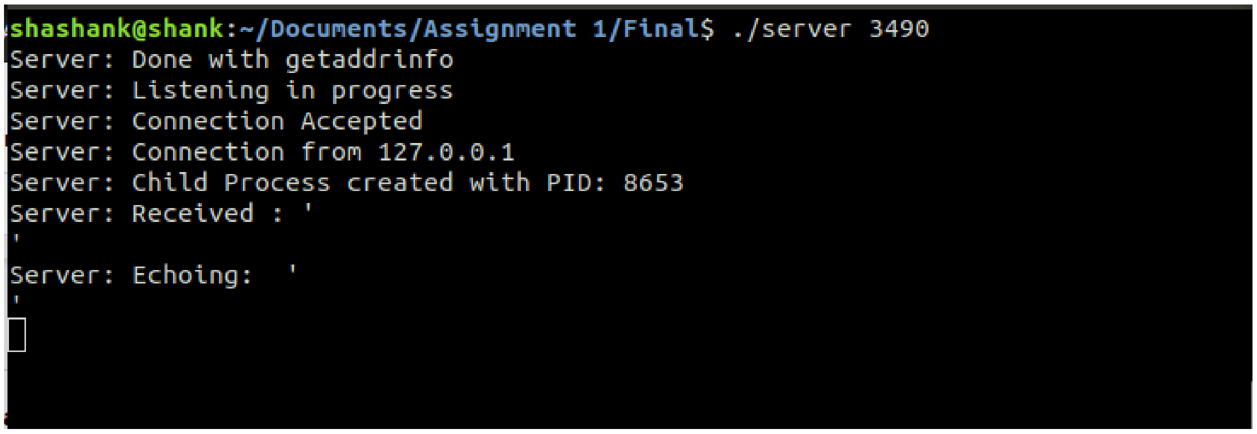
```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
CLIENT 1
Client Sending: CLIENT 1

Client Received:CLIENT 1
Enter the string
```

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
CLIENT 2
Client Sending: CLIENT 2

Client Received:CLIENT 2
Enter the string
```

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
CLIENT 3
Client Sending: CLIENT 3

Client Received:CLIENT 3
Enter the string
```

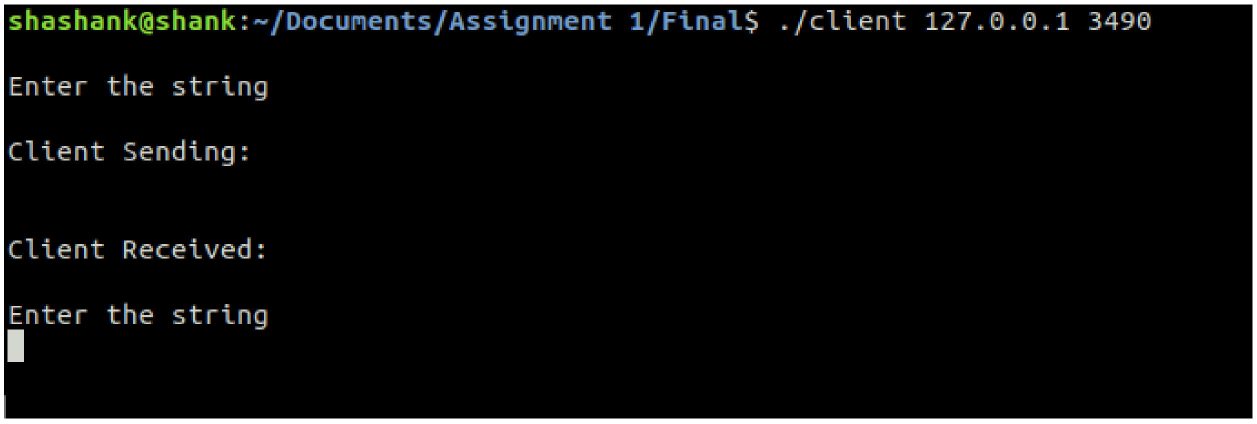
3. **Line with newline character**

Server Terminal:



```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8653
Server: Received : '
'
Server: Echoing:  '
'
```

Client Terminal:

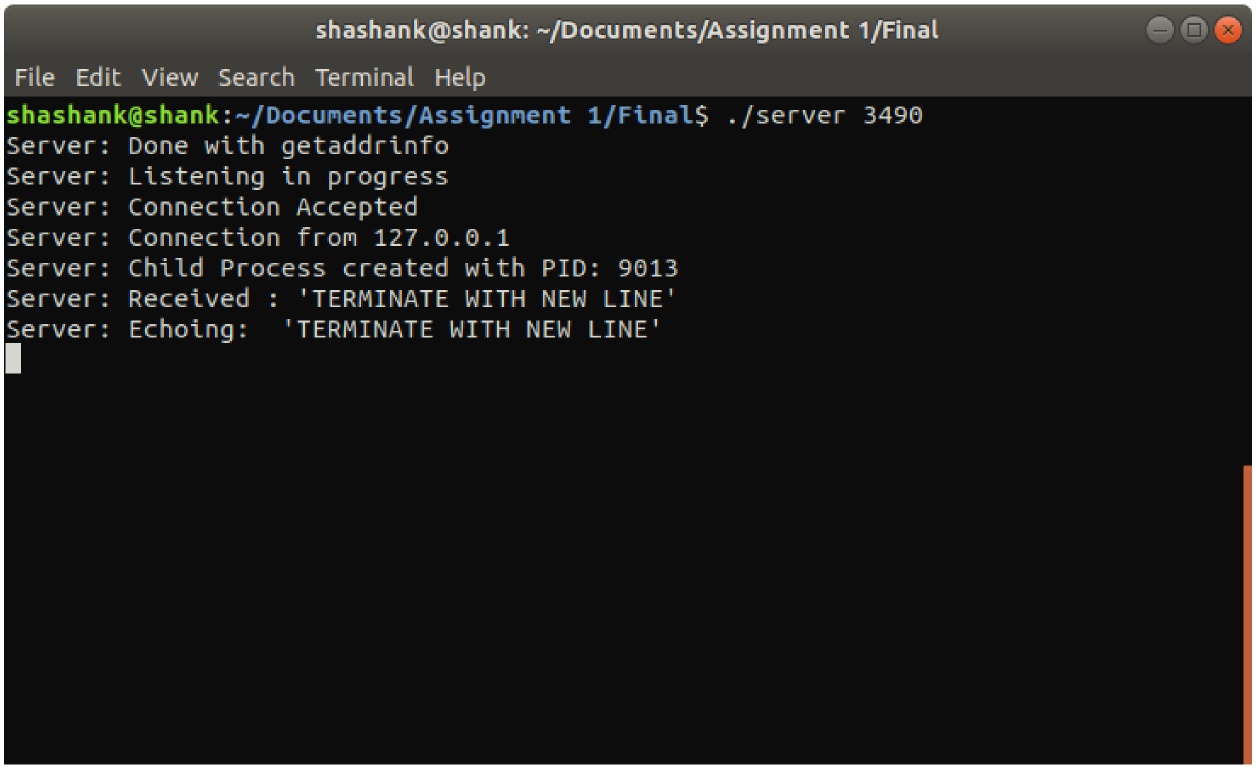


```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string

Client Sending:


Client Received:

Enter the string
```

In this, the client sends a newline character to the server and the server echoes it back.

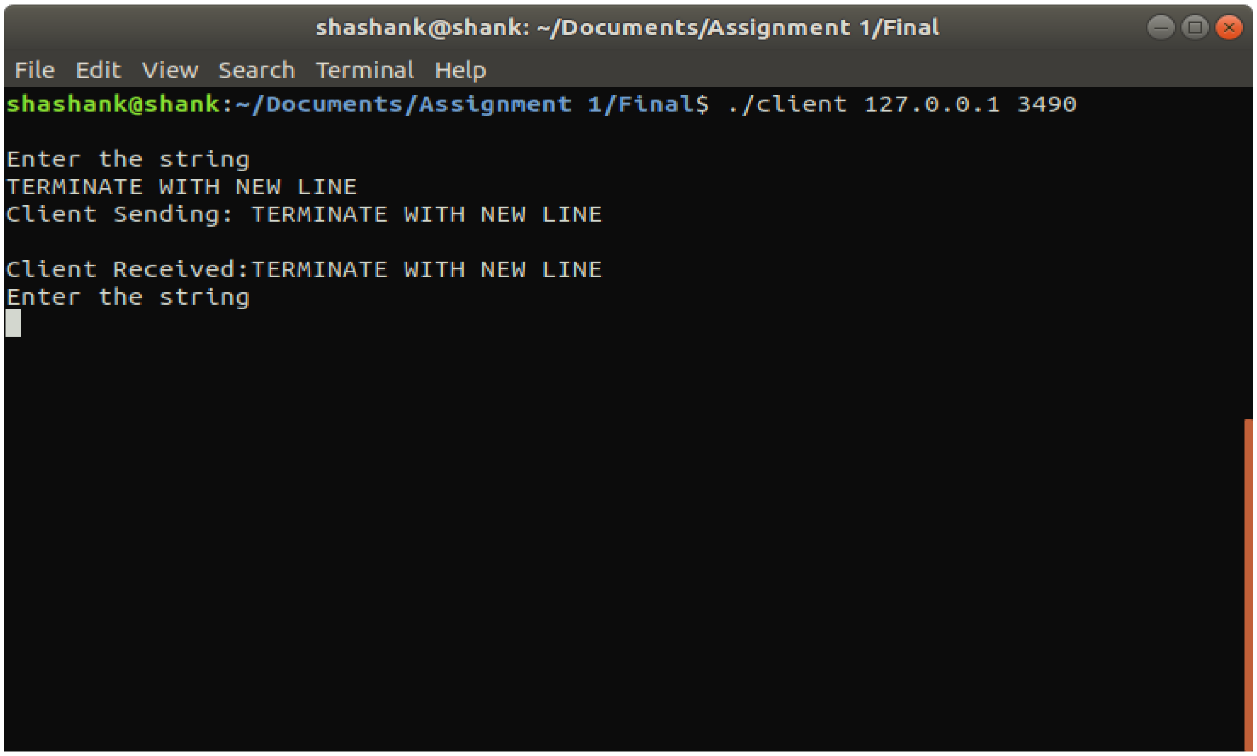
4. Line of text terminated by a newline

Server Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 9013
Server: Received : 'TERMINATE WITH NEW LINE'
Server: Echoing:  'TERMINATE WITH NEW LINE'
```

Client Terminal:

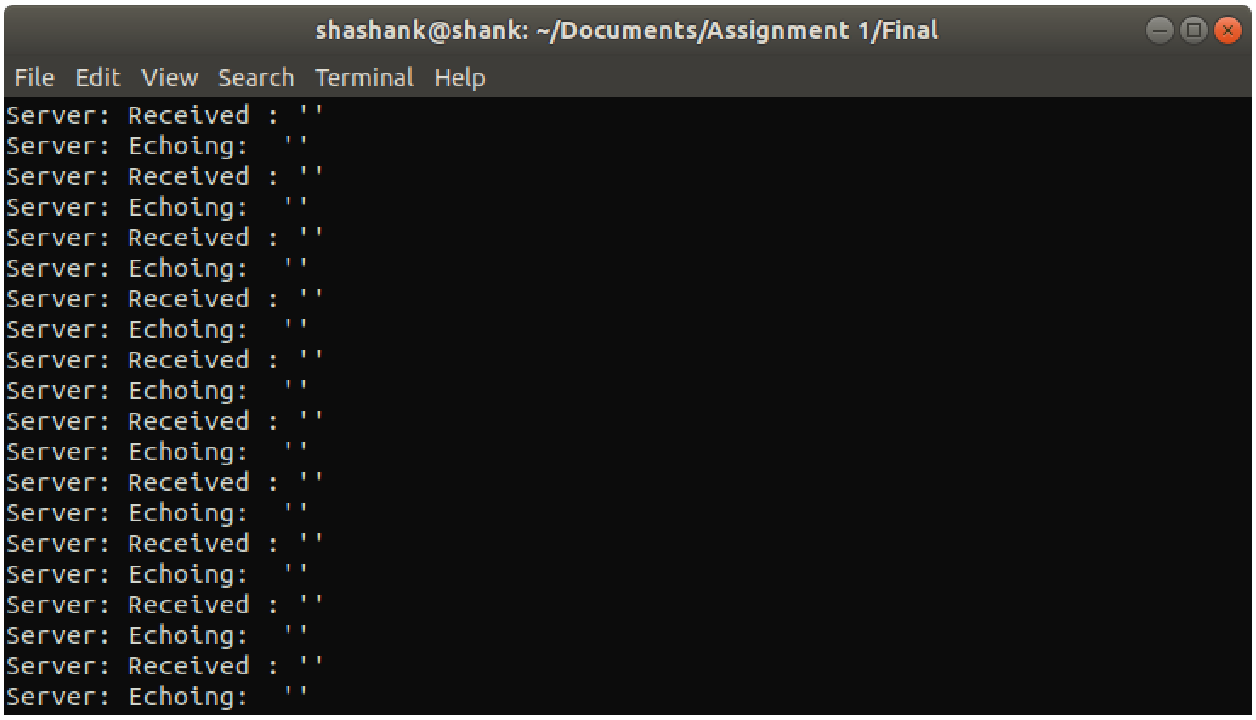

shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
TERMINATE WITH NEW LINE
Client Sending: TERMINATE WITH NEW LINE

Client Received:TERMINATE WITH NEW LINE
Enter the string
```

In this, the client sends a message ending with a newline character and the server echoes it back

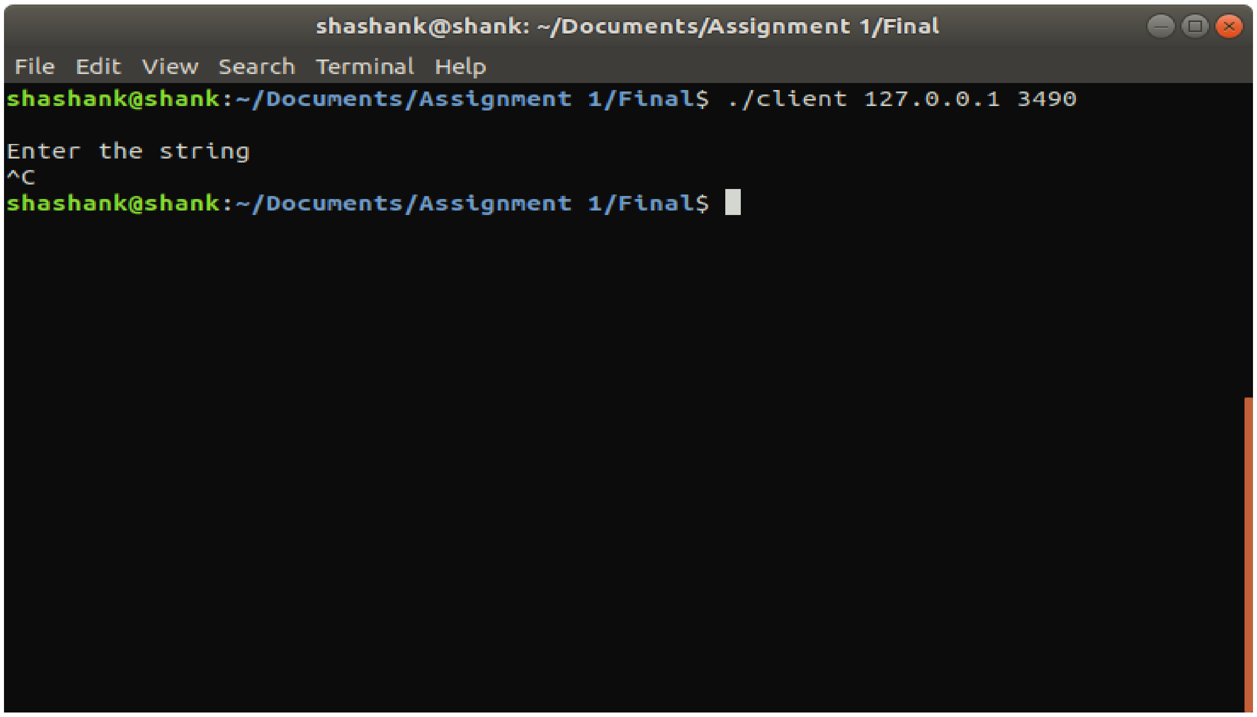
5. Pressing ‘CTRL+C’ in client terminal:

Server Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
Server: Received : ''
Server: Echoing:  ''
```

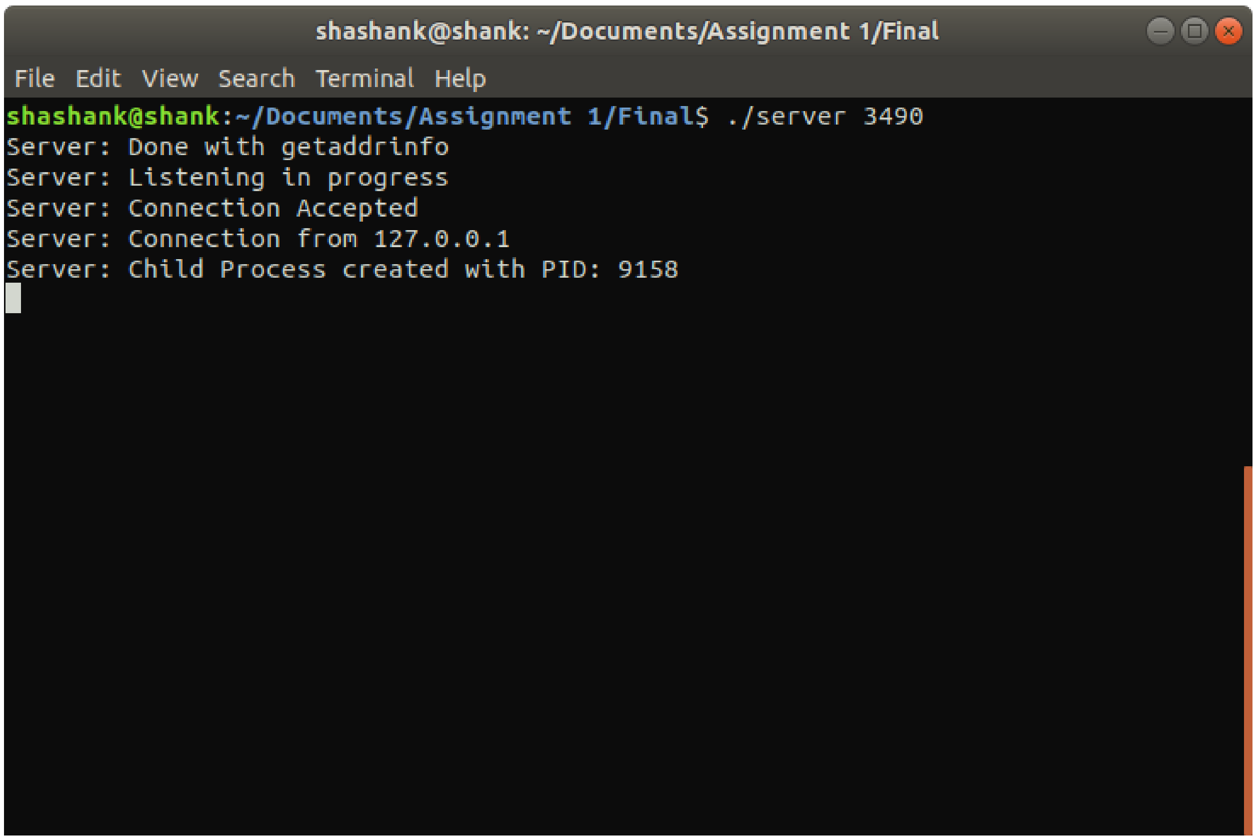
Client Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
^C
shashank@shank:~/Documents/Assignment 1/Final$ 
```

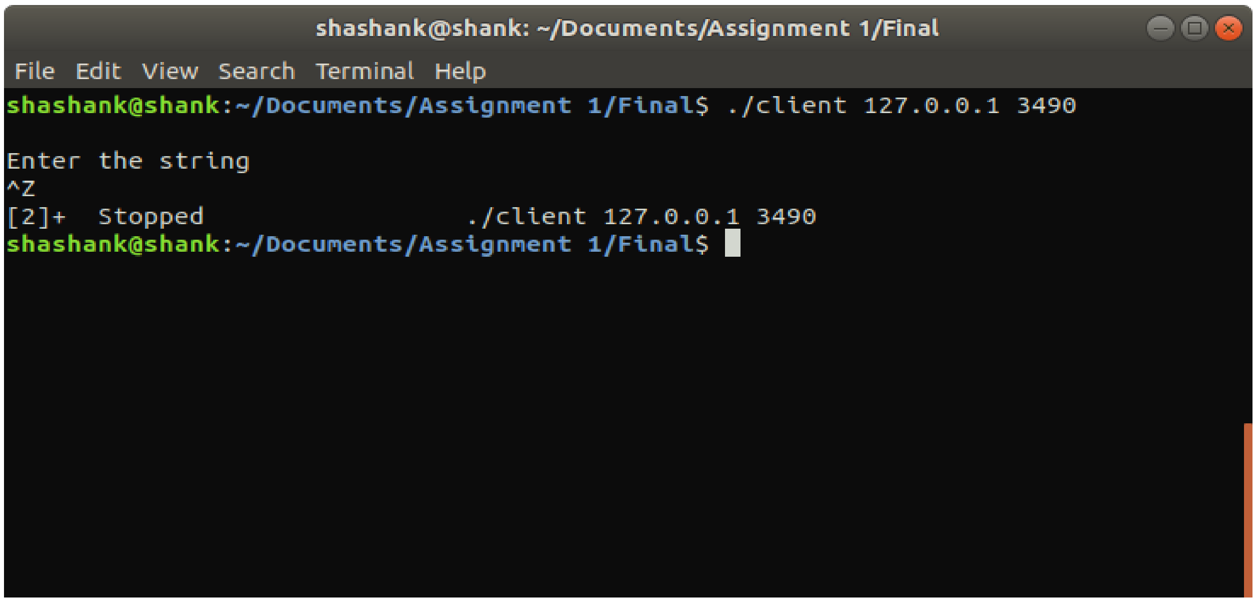
The server runs into a infinite loop of receiving NULL and sending NULL when the client is abruptly terminated using CTRL+C.

6. Pressing ‘CTRL+Z’ in client terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 9158
```

Client Terminal:

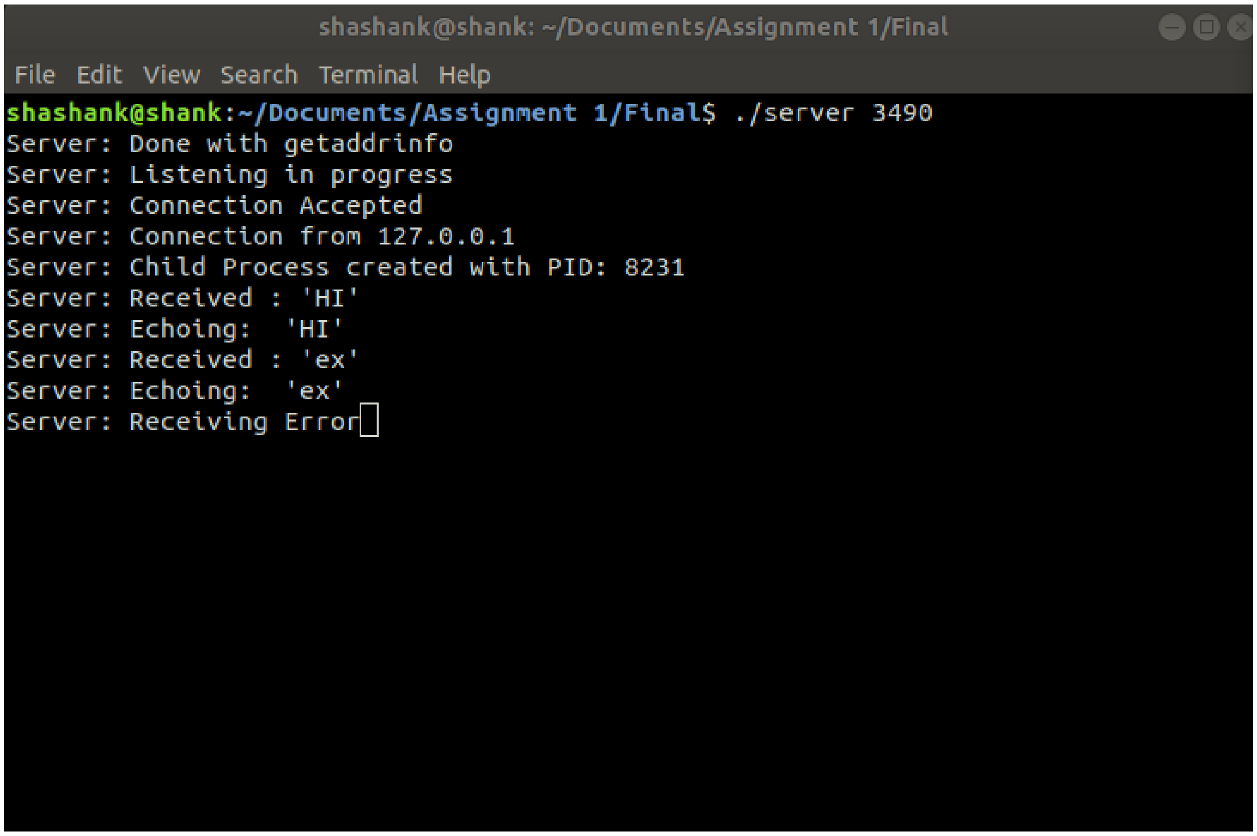

shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
^Z
[2]+  Stopped                 ./client 127.0.0.1 3490
shashank@shank:~/Documents/Assignment 1/Final$ 
```

When the client process is stopped using CTRL+Z, the server does not end the connection and keeps running.

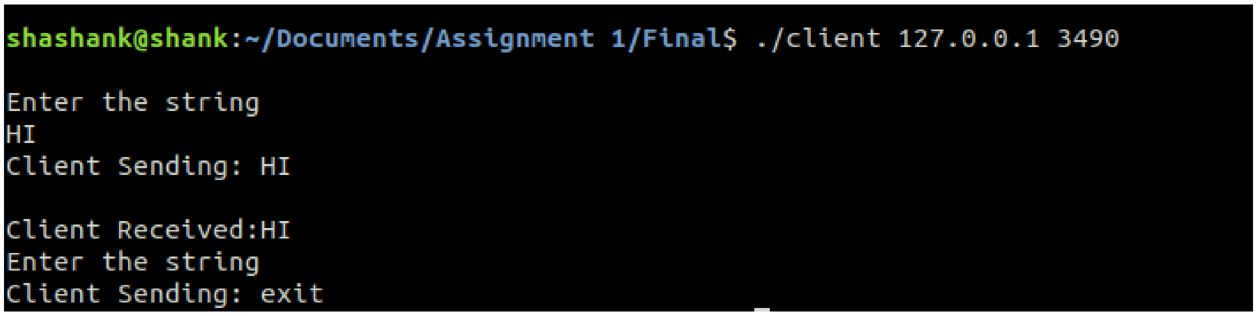
7. **Pressing 'CTRL+D' in client terminal:**

Server Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 8231
Server: Received : 'HI'
Server: Echoing:  'HI'
Server: Received : 'ex'
Server: Echoing:  'ex'
Server: Receiving Error
```

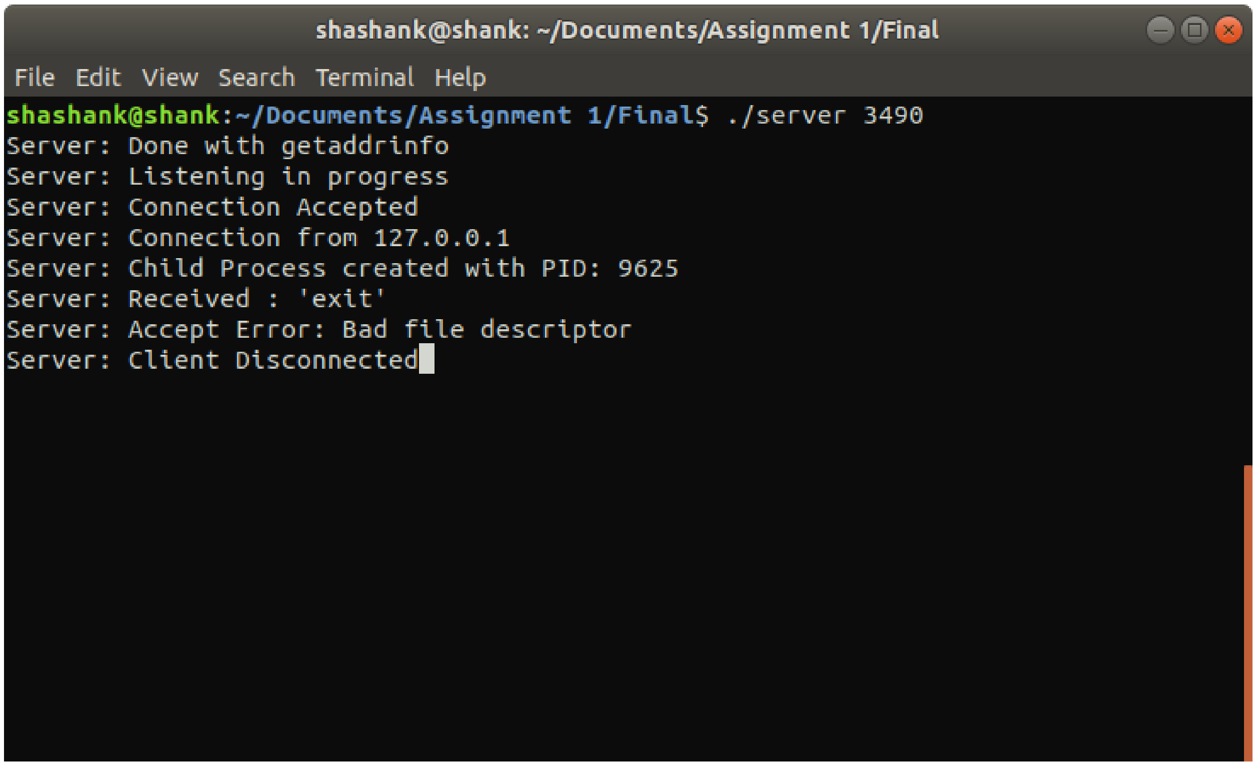
Client Terminal:



```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
HI
Client Sending: HI

Client Received:HI
Enter the string
Client Sending: exit
```

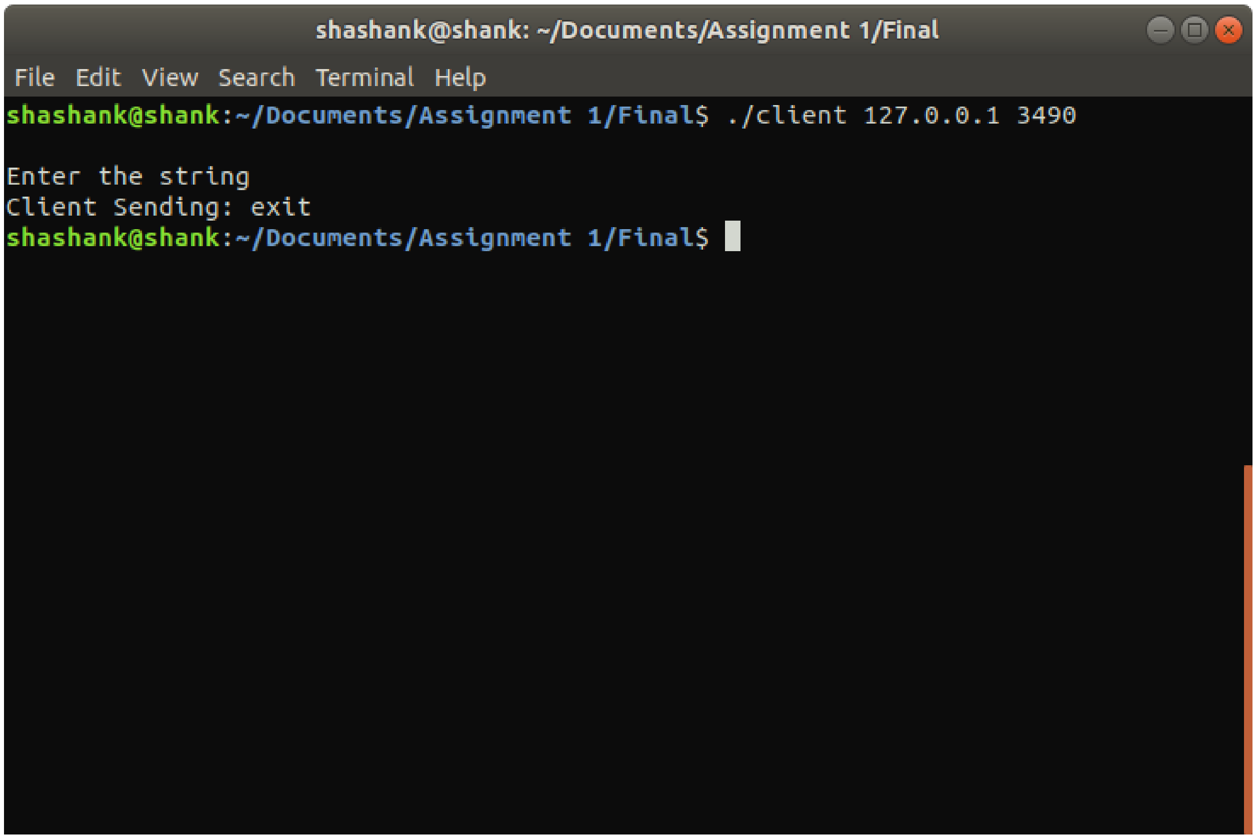
While the client process is running, on pressing CTRL+D, the client sends the message exit to the server and the server disconnects with the client.

Server Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 9625
Server: Received : 'exit'
Server: Accept Error: Bad file descriptor
Server: Client Disconnected
```

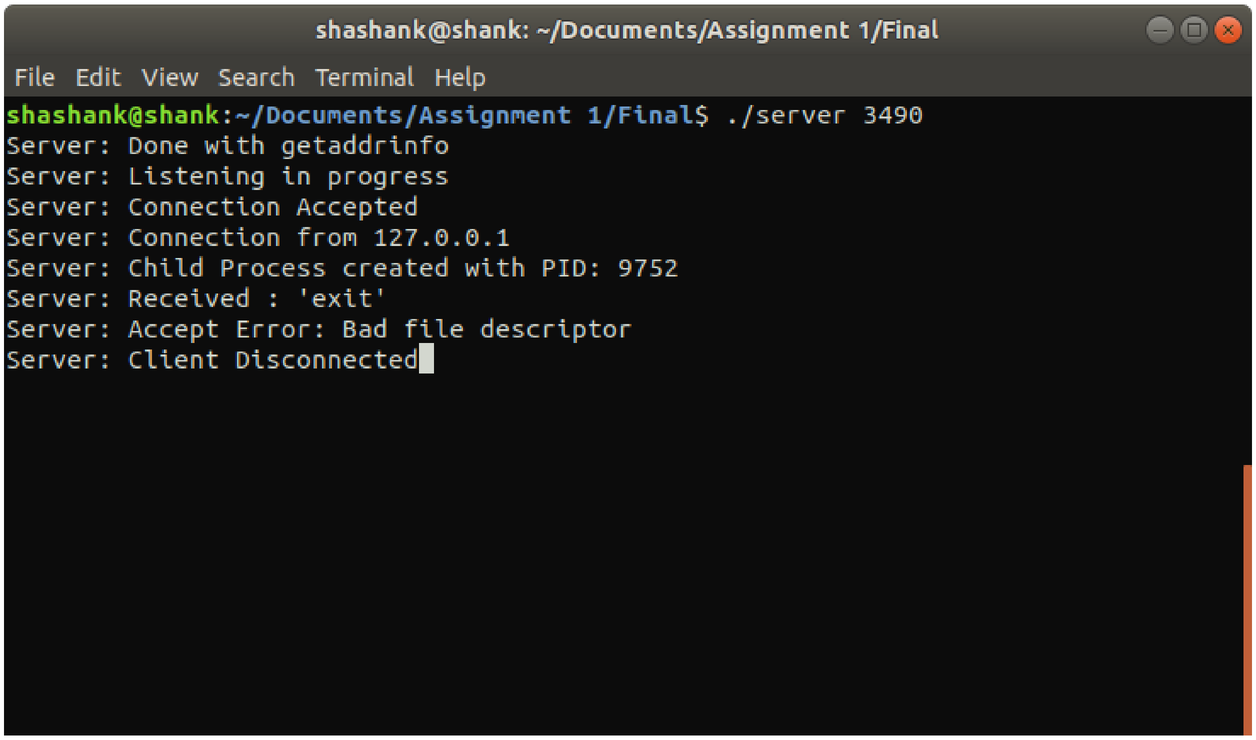
Client Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
Client Sending: exit
shashank@shank:~/Documents/Assignment 1/Final$
```

8. Client sending 'exit' to server:

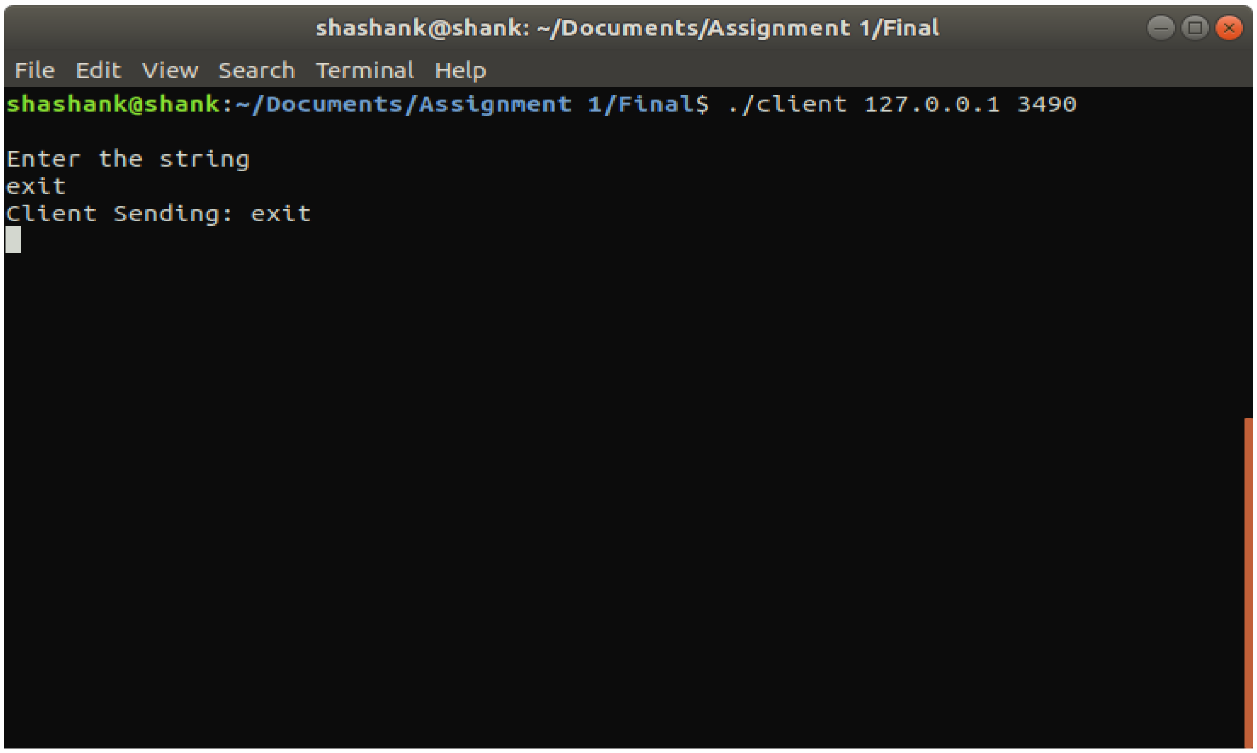
Server Terminal:



shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 9752
Server: Received : 'exit'
Server: Accept Error: Bad file descriptor
Server: Client Disconnected
```

Client Terminal:



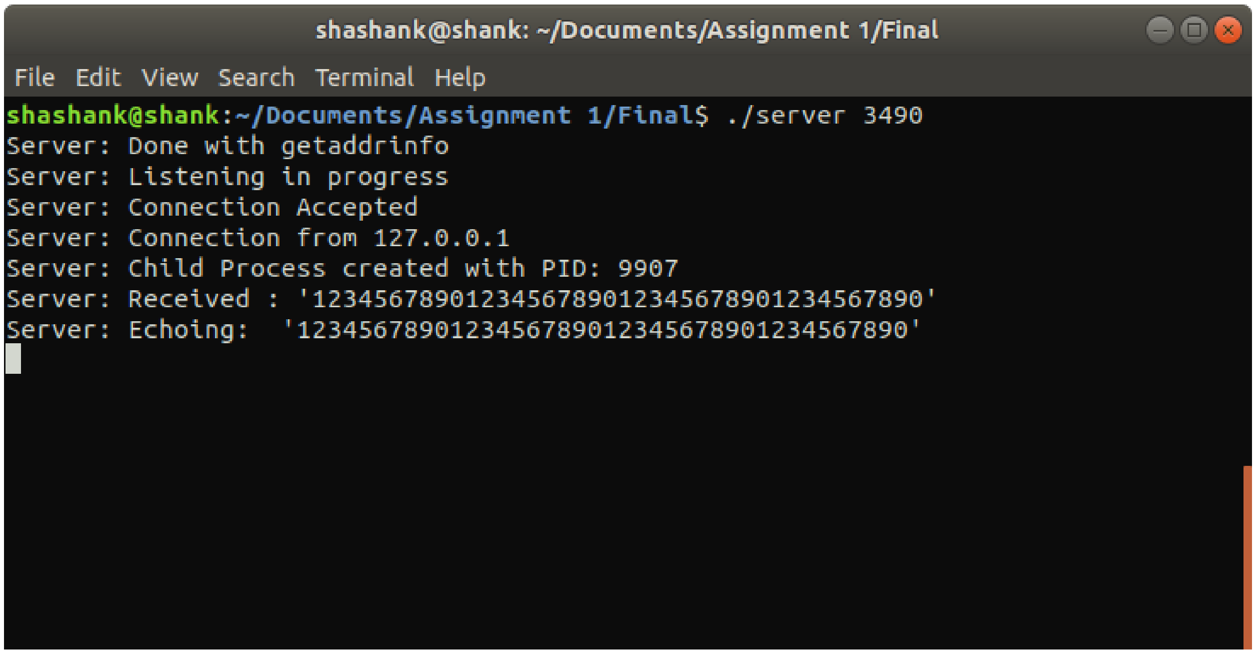
shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
exit
Client Sending: exit
```

When the client sends the message "exit" the server closes the connection and disconnects the client.

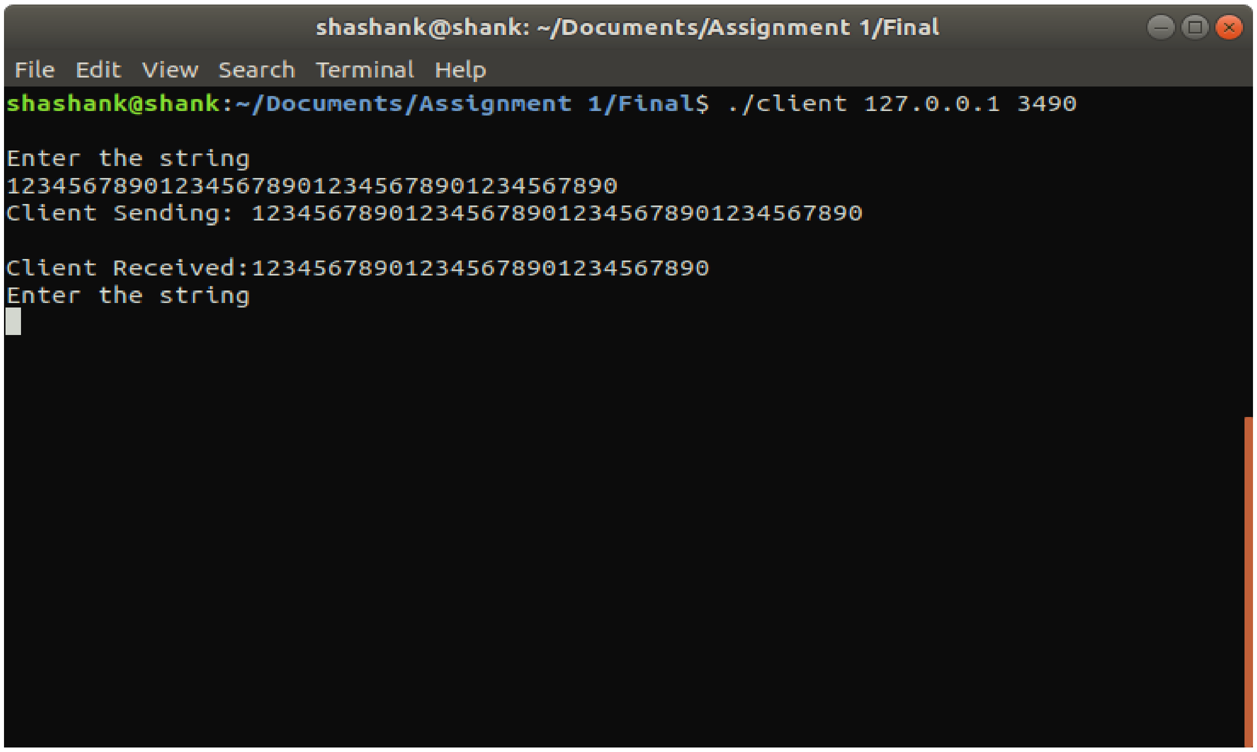
9. Max buffer size in client:

Server Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./server 3490
Server: Done with getaddrinfo
Server: Listening in progress
Server: Connection Accepted
Server: Connection from 127.0.0.1
Server: Child Process created with PID: 9907
Server: Received : '1234567890123456789012345678901234567890'
Server: Echoing:  '1234567890123456789012345678901234567890'
```

Client Terminal:

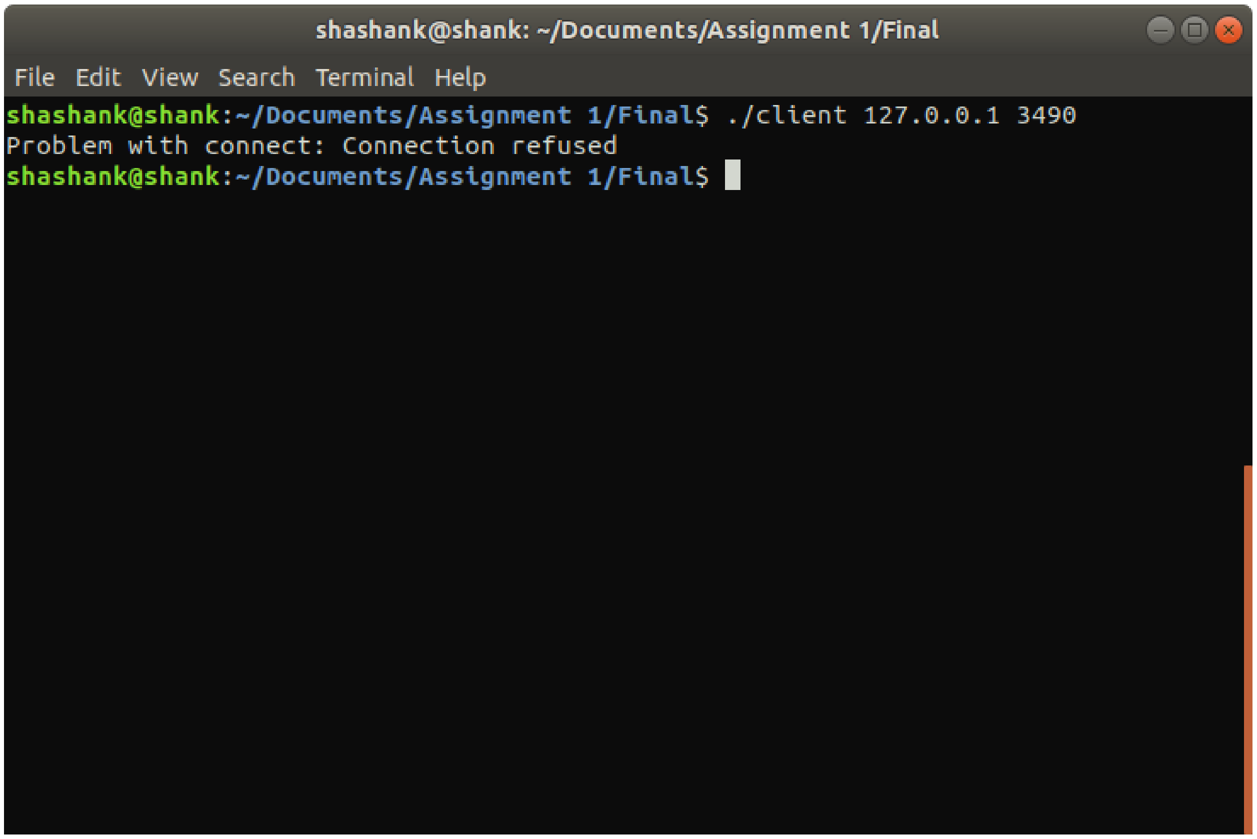

shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490

Enter the string
1234567890123456789012345678901234567890
Client Sending: 1234567890123456789012345678901234567890

Client Received:123456789012345678901234567890
Enter the string
```

In the above screenshot, the client sends 40 characters to the server. The server echoes back 40 characters of data to the client. But, the client buffer can handle on 30 characters at a time. Therefore, we can observe that the client has received only 30 characters out of the 40 characters sent by the server.

10.Connection Error:

Client Terminal:


shashank@shank: ~/Documents/Assignment 1/Final
File Edit View Search Terminal Help

```
shashank@shank:~/Documents/Assignment 1/Final$ ./client 127.0.0.1 3490
Problem with connect: Connection refused
shashank@shank:~/Documents/Assignment 1/Final$ 
```

The above screenshot is of connection error with the server.

Shortcoming of this assignment:

- Zombie processes have not been dealt with in this assignment.
- On the client side, the maximum buffer size that could be executed was 30 bits. If the buffer size exceeds greater than 30 bits then the client would not connect to the server.