

Name: Adwait Purao

UID: 2021300101

Batch: B2

Experiment no.: 9

Aim: To design a chat application using client server model

Theory:

Socket programming

Sockets can be thought of as endpoints in a communication channel that is bi-directional and establishes communication between a server and one or more clients. Here, we set up a socket on each end and allow a client to interact with other clients via the server. The socket on the server side associates itself with some hardware port on the server-side. Any client that has a socket associated with the same port can communicate with the server socket.

Multi-Threading

A thread is a sub-process that runs a set of commands individually of any other thread. So, every time a user connects to the server, a separate thread is created for that user, and communication from the server to the client takes place along individual threads based on socket objects created for the sake of the identity of each client.

We will require two scripts to establish this chat room. One to keep the serving running, and another that every client should run in order to connect to the server.

Server Side Script

The server-side script will attempt to establish a socket and bind it to an IP address and port specified by the user (windows users might have to make an exception for the specified port number in their firewall settings, or can rather use a port that is already open). The script will then stay open and receive connection requests and will append respective socket objects to a list to keep track of active connections. Every time a user connects,

a separate thread will be created for that user. In each thread, the server awaits a message and sends that message to other users currently on the chat. If the server encounters an error while trying to receive a message from a particular thread, it will exit that thread.

Usage

This server can be set up on a local area network by choosing any on the computer to be a server node, and using that computer's private IP address as the server IP address.

For example, if a local area network has a set of private IP addresses assigned ranging from 192.168.1.2 to 192.168.1.100, then any computer from these 99 nodes can act as a server, and the remaining nodes may connect to the server node by using the server's private IP address. Care must be taken to choose a port that is currently not in usage. For example, port 22 is the default for ssh, and port 80 is the default for HTTP protocols. So these two ports preferably, shouldn't be used or reconfigured to make them free for usage.

However, if the server is meant to be accessible beyond a local network, the public IP address would be required for usage. This would require port forwarding in cases where a node from a local network (node that isn't the router) wishes to host the server. In this case, we would require any requests that come to the public IP addresses to be re-routed towards our private IP address in our local network, and would hence require port forwarding.

Client-Side Script

The client-side script will simply attempt to access the server socket created at the specified IP address and port. Once it connects, it will continuously check as to whether the input comes from the server or from the client, and accordingly redirects output. If the input is from the server, it displays the message on the terminal. If the input is from the user, it sends the message that the user enters to the server for it to be broadcasted to other users.

Code:

Server

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

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

```
#include <unistd.h>

#include <string.h>

#include <fcntl.h>

#include <pthread.h>

#include <netinet/in.h>

#include <sys/socket.h>

#include <sys/types.h>

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

```
int clientCount = 0;
```

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
struct client
```

```
{
```

```
    int index;
```

```
    int sockID;
```

```
    struct sockaddr_in clientAddr;
```

```
    int len;
```

```
};
```

```
struct client Client[1024];
```

```
pthread_t thread[1024];
```

```
void *doNetworking(void *ClientDetail)
```

```
{
```

```
    struct client *clientDetail = (struct client *)ClientDetail;
```

```
    int index = clientDetail->index;
```

```
    int clientSocket = clientDetail->sockID;
```

```
    printf("Client %d connected.\n", index + 1);
```

```
    while (1)
```

```
{
```

```
    char data[1024];
```

```
    int read = recv(clientSocket, data, 1024, 0);
```

```
    data[read] = '\0';
```

```
    char output[1024];
```

```
    if (strcmp(data, "LIST") == 0)
```

```
{
```

```
        int l = 0;
```

```

for (int i = 0; i < clientCount; i++)
{

    if (i != index)

        l += snprintf(output + l, 1024, "Client %d is at socket %d.\n", i + 1,
Client[i].sockID);

}

    send(clientSocket, output, 1024, 0);
    continue;
}
if (strcmp(data, "SEND") == 0)
{

    read = recv(clientSocket, data, 1024, 0);
    data[read] = '\0';

    int id = atoi(data) - 1;

    read = recv(clientSocket, data, 1024, 0);
    data[read] = '\0';

    send(Client[id].sockID, data, 1024, 0);
}

```

```

    }

    return NULL;
}

int main()
{

    int serverSocket = socket(PF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddr;

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr))
    == -1)
        return 0;

    if (listen(serverSocket, 1024) == -1)
        return 0;

    printf("Server started listening on port 8080 .....\\n");

```

```
while (1)
{

    Client[clientCount].sockID = accept(serverSocket, (struct sockaddr
*)&Client[clientCount].clientAddr, &Client[clientCount].len);

    Client[clientCount].index = clientCount;

    pthread_create(&thread[clientCount], NULL, doNetworking, (void
*)&Client[clientCount]);

    clientCount++;

}

for (int i = 0; i < clientCount; i++)
    pthread_join(thread[i], NULL);
}
```

Client

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <fcntl.h>

#include <pthread.h>

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

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

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

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

```
void *doReceiving(void *sockID)
```

```
{
```

```
    int clientSocket = *((int *)sockID);
```

```
    while (1)
```

```
    {
```

```
        char data[1024];
```

```
        int read = recv(clientSocket, data, 1024, 0);
```

```
        data[read] = '\0';
```

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

```
    }
```

```
}
```

```
int main()
```

```
{
```

```
    int clientSocket = socket(PF_INET, SOCK_STREAM, 0);
```



```

struct sockaddr_in serverAddr;

serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080);
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);


if (connect(clientSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr))
== -1)
    return 0;


printf("Connection established .....\\n");


pthread_t thread;
pthread_create(&thread, NULL, doReceiving, (void *)&clientSocket);


while (1)
{

    char input[1024];
    scanf("%s", input);

    if (strcmp(input, "LIST") == 0)
    {

        send(clientSocket, input, 1024, 0);
    }
}

```

```

    }

    if (strcmp(input, "SEND") == 0)
    {

        send(clientSocket, input, 1024, 0);

        scanf("%s", input);

        send(clientSocket, input, 1024, 0);

        scanf("%[^\n]s", input);

        send(clientSocket, input, 1024, 0);

    }

}
}

```

Output:

Server

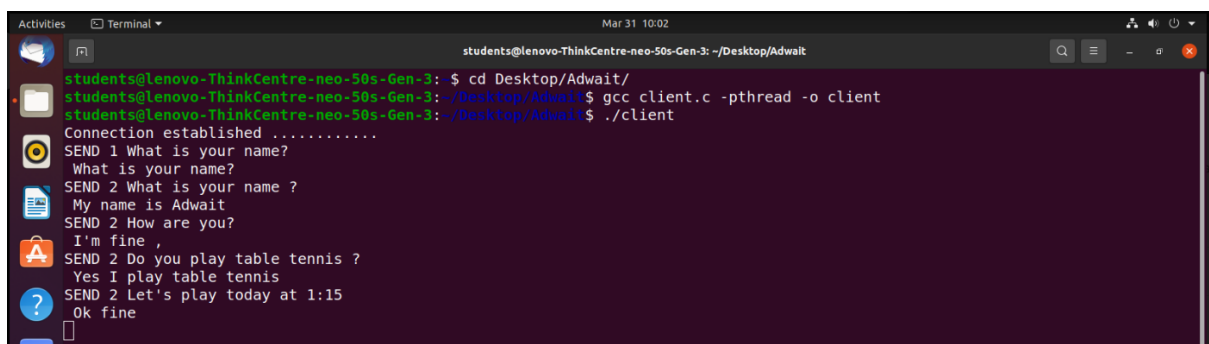


```

students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait
students@lenovo-ThinkCentre-neo-50s-Gen-3: $ cd Desktop/Adwait/
students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait $ gcc server.c -pthread -o server
students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait $ ./server
Server started listening on port 8080 .....
Client 1 connected.
Client 2 connected.

```

Client 1

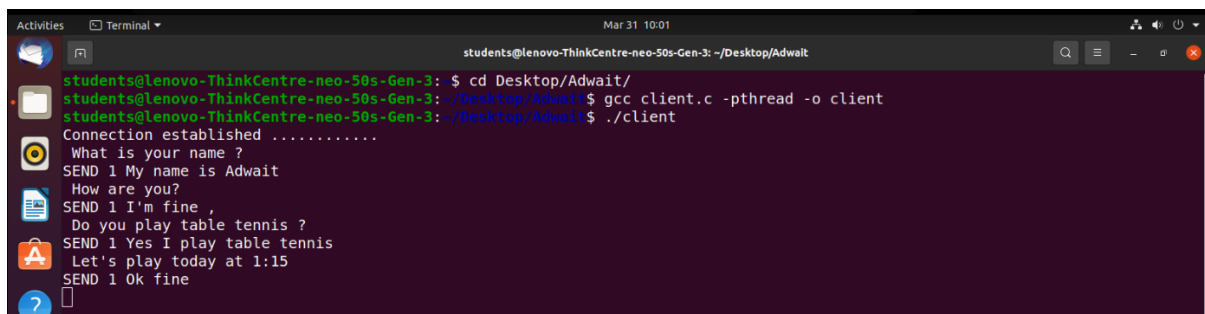


```

students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait
students@lenovo-ThinkCentre-neo-50s-Gen-3: $ cd Desktop/Adwait/
students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait $ gcc client.c -pthread -o client
students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait $ ./client
Connection established .....
SEND 1 What is your name?
What is your name?
SEND 2 What is your name ?
My name is Adwait
SEND 2 How are you?
I'm fine ,
SEND 2 Do you play table tennis ?
Yes I play table tennis
SEND 2 Let's play today at 1:15
Ok fine

```

Client 2

A terminal window titled 'students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait' showing the execution of a C program. The user navigates to the Desktop/Adwait directory, compiles 'client.c' with 'gcc client.c -pthread -o client', and runs './client'. The program outputs 'Connection established', asks 'What is your name?', and receives 'SEND 1 My name is Adwait'. It then asks 'How are you?', receives 'SEND 1 I'm fine,', asks 'Do you play table tennis?', receives 'SEND 1 Yes I play table tennis', asks 'Let's play today at 1:15', and receives 'SEND 1 Ok fine'.

```
students@lenovo-ThinkCentre-neo-50s-Gen-3: ~/Desktop/Adwait
students@lenovo-ThinkCentre-neo-50s-Gen-3:~/Desktop/Adwait$ cd Desktop/Adwait/
students@lenovo-ThinkCentre-neo-50s-Gen-3:~/Desktop/Adwait$ gcc client.c -pthread -o client
students@lenovo-ThinkCentre-neo-50s-Gen-3:~/Desktop/Adwait$ ./client
Connection established .....
What is your name ?
SEND 1 My name is Adwait
How are you?
SEND 1 I'm fine ,
Do you play table tennis ?
SEND 1 Yes I play table tennis
Let's play today at 1:15
SEND 1 Ok fine
```

Conclusion:

In conclusion, using a client-server model for a chat application provides a reliable and scalable way for users to communicate with each other. The server acts as a central hub, allowing multiple clients to connect and exchange messages in real-time. By separating the application logic between the client and server, the system becomes more modular, allowing for easier maintenance and updates. Additionally, the server can enforce security measures and handle data storage, making the application more secure and efficient. Overall, a client-server model is a solid choice for building a chat application that can accommodate multiple users and provide a seamless experience.