Department of Networking and Communications

COURSE PROJECT REPORT
ACADEMIC YEAR:2021-2022
ODD SEMESTER

PROJECT TITLE: **Simple Chat Room using Socket Programming in Python (1 central server, and n possible clients)**

Program(UG/PG): UG
Semester: V

Course Code: 18CSC302J
Course Title: COMPUTER NETWORKS
Student Name: Stuti Jain
Register Number: RA2011031010031
Faculty Name: Dr.S.Thenmalar
Branch with Specialization: CSE- INFORMATION TECHNOLOGY
Section: O1

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this course project report titled **"***Simple Chat Room using Socket Programming in Python (1 central server, and n possible clients***)"** is the bonafide work done by **"Shivendra Bharuka(RA2011031010020), Nandini (RA2011031010003), Stuti Jain (RA2011031010031`), Arghya Pahar (RA2011031010029)"** who carried out the course project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE
Dr.S.Thenmalar
Associate Professor
Course Handling Faculty

ABSTRACT:

The main objective of the Simple Chat Room project is to create a chat application which helps different users to communicate with each other through a server connected. This is a simple chat program with a server and can have many clients. The server needs to be started first and clients can be connected later. Simple Chat Room provides a bidirectional communication between client and server. It enables users to seamlessly communicate with each other. The user has an option to login to the chat room. The user should be able to give the ip address of the server and the port at which he is connecting. The user can chat using this chat application. The server shall poll for other users that are active in the chat room and make those users visible. If the user at the other end is active then they can start a chat session. The chat is recorded in the application. The user can save the chat transcript or clear it based on his requirement.

This Simple Chat Room application will enable the user to chat with the logged users in the chat room. The server should be active. The users shall connect to this server at the predefined port number. To start using this tool, the user has to register with this tool. Through the valid login, password details, the application shall allow the user to use the chat room. The user shall be able to connect to this server and can chat with other users. The logged in user can view other active users in the chat room. The user can initiate the conversation through the chat window. For the communication to happen, both the users should have connected to the server. This application is developed using socket programming in Python. The windows sockets are used in the application for communication between client and server. They provide a robust way of communication between nodes. The user also can clear the chat transcript if required. It also provides an option to save the chat transcript. Users can view the status of other users in the chat room like if the user is currently online or offline or busy.

This chat application, as described above, can lead to error free, secure, reliable and a fast management system. It can assist the user to concentrate on their other activities rather than concentrating on the record keeping. Thus it will help the organization in better utilization of the resources.

INTRODUCTION:

This application has been developed to override the problems prevailing in the practicing manual system. This software is supported to eliminate and in some cases reduce the hardships faced by this existing system. Moreover, this system is designed for the particular need of the company to carry out operations in a smooth and efficient manner.

The application is reduced as much as possible to avoid errors while entering the data. It also provides an error message while entering invalid data. No formal knowledge is needed for the user to use this system. Thus by this all it proves it is user-friendly chat application, as described above, can lead to error-free, secure, reliable and fast management system. This is designed to assist in strategic planning, and will help you ensure that your organization is equipped with the right level of information and details for your future goals.

PROBLEM STATEMENT:

Chat rooms have become a popular way to support a forum for n-way conversation or discussion among a set of people with interest in a common topic. Chat applications range from simple, text-based ones to entire virtual worlds with exotic graphics. In this project you are required to implement a simple text-based chat client/server application.

Email, newsgroup and messaging applications provide means for communication among people but these are one-way mechanisms and they do not provide an easy way to carry on a real-time conversation or discussion with people involved. Chat room extends the one-way messaging concept to accommodate multi-way communication among a set of people.

LITERATURE SURVEY:

Earlier there was no mode of online communication between users. In big or small organizations communication between users posed a challenge. There was a requirement to record these communications and store the data for further evaluation. The idea is to automate the existing Simple Chat Room system and make the users utilize the software so that their valuable information is stored digitally and can be retrieved for further management purposes. There was no online method of communicating with different users. There were many different interfaces available in the market but this method of

using windows sockets to communicate between nodes would be fast and reliable. The main objective of our Simple Chat Room project is to create a chat application that helps different users to communicate with each other through a server connected. This is a simple chat program with a server and can have many clients. The server needs to be started first and clients can be connected later. Simple Chat Room provides bidirectional communication between client and server. It enables users to seamlessly communicate with each other. The user can chat using this chat application. If the user at the other end is active then they can start a chat session. The chat is recorded in this application.

DESIGN AND IMPLEMENTATION:

Server Implementation:

1. We will need to import two libraries, namely socket and threading. The first one will be used for the network connection and the second one is necessary for performing various tasks at the same time.

2. The next task is to define our connection data and to initialize our socket. We will need an IP-address for the host and a free port number for our server. The port is actually irrelevant but we have to make sure that the port we are using is free and not reserved. If we are running this script on an actual server, we need to specify the IP-address of the server as the host.

3. When we define our socket, we need to pass two parameters. These define the type of socket we want to use. The first one (AF_INET) indicates that we are using an internet socket rather than an unix socket. The second parameter stands for the protocol we want to use. SOCK_STREAM indicates that we are using TCP and not UDP.

4. After defining the socket, we bind it to our host and the specified port by passing a tuple that contains both values. We then put our server into listening mode, so that it waits for clients to connect. At the end we create two empty lists, which we will use to store the connected clients and their nicknames later on.

5. Next we define a function that is going to help us in broadcasting messages and makes the code more readable. It sends a message to each client that is connected and therefore in the clients list. We will be using this method in the other

methods. Now we will start with the implementation of the first major function. This function will be responsible for handling messages from the clients.

6. This function will run a while-loop. It won't stop unless there is an exception because of something that went wrong. The function accepts a client as a parameter. Everytime a client connects to our server we run this function for it and it starts an endless loop.

7. This function is receiving the message from the client (if he sends any) and broadcasting it to all connected clients. So when one client sends a message, everyone else can see this message. Now if for some reason there is an error with the connection to this client, we remove it and its nickname, close the connection and broadcast that this client has left the chat. After that we break the loop and this thread comes to an end.

8. When we are ready to run our server, we will execute this receive function. It also starts an endless while-loop which constantly accepts new connections from clients. Once a client is connected it sends the string to it, which will tell the client that its nickname is requested. After that it waits for a response (which hopefully contains the nickname) and appends the client with the respective nickname to the lists. After that, we print and broadcast this information. Finally, we start a new thread that runs the previously implemented handling function for this particular client. Now we can just run this function and our server implementation is done.

9. In the server implementation, we are always encoding and decoding the messages here. The reason for this is that we can only send bytes and not strings. So we always need to encode messages (for example using ASCII), when we send them and decode them, when we receive them.

Code:

```
#Now let's start by implementing the server first. For this we will need
to import two libraries, namely socket and threading. The first one will
be used for the network connection and the second one is necessary for
performing various tasks at the same time.

import socket
import threading
```

```
#The next task is to define our connection data and to initialize our
socket. We will need an IP-address for the host and a free port number for
our server. In this example, we will use the localhost address (127.0.0.1)
and the port 55555. The port is actually irrelevant but you have to make
sure that the port you are using is free and not reserved. If you are
running this script on an actual server, specify the IP-address of the
server as the host. Check out this list of reserved port numbers for more
information.

# Connection Data
host = '127.0.0.1'
port = 55555

# Starting Server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((host, port))
server.listen()

# Lists For Clients and Their Nicknames
clients = []
nicknames = []
#When we define our socket, we need to pass two parameters. These define
the type of socket we want to use. The first one (AF_INET) indicates that
we are using an internet socket rather than an unix socket. The second
parameter stands for the protocol we want to use. SOCK_STREAM indicates
that we are using TCP and not UDP.

#After defining the socket, we bind it to our host and the specified port
by passing a tuple that contains both values. We then put our server into
listening mode, so that it waits for clients to connect. At the end we
create two empty lists, which we will use to store the connected clients
and their nicknames later on.

# Sending Messages To All Connected Clients
def broadcast(message):
    for client in clients:
        client.send(message)
#Here we define a little function that is going to help us broadcast
messages and make the code more readable. What it does is just send a
```

```
message to each client that is connected and therefore in the clients
list. We will use this method in the other methods.

#Now we will start with the implementation of the first major function.
This function will be responsible for handling messages from the clients.

# Handling Messages From Clients
def handle(client):
    while True:
        try:
            # Broadcasting Messages
            message = client.recv(1024)
            broadcast(message)
        except:
            # Removing And Closing Clients
            index = clients.index(client)
            clients.remove(client)
            client.close()
            nickname = nicknames[index]
            broadcast('{} left!'.format(nickname).encode('ascii'))
            nicknames.remove(nickname)
            break
#As you can see, this function is running in a while-loop. It won't stop
unless there is an exception because of something that went wrong. The
function accepts a client as a parameter. Everytime a client connects to
our server we run this function for it and it starts an endless loop.

#What it then does is receives the message from the client (if he sends
any) and broadcasts it to all connected clients. So when one client sends
a message, everyone else can see this message. Now if for some reason
there is an error with the connection to this client, we remove it and its
nickname, close the connection and broadcast that this client has left the
chat. After that we break the loop and this thread comes to an end. Quite
simple. We are almost done with the server but we need one final function.

# Receiving / Listening Function
def receive():
    while True:
        # Accept Connection
        client, address = server.accept()
```

```
        print("Connected with {}".format(str(address)))

        # Request And Store Nickname
        client.send('NICK'.encode('ascii'))
        nickname = client.recv(1024).decode('ascii')
        nicknames.append(nickname)
        clients.append(client)

        # Print And Broadcast Nickname
        print("Nickname is {}".format(nickname))
        broadcast("{} joined!".format(nickname).encode('ascii'))
        client.send('Connected to server!'.encode('ascii'))

        # Start Handling Thread For Client
        thread = threading.Thread(target=handle, args=(client,))
        thread.start()
#When we are ready to run our server, we will execute this receive
function. It also starts an endless while-loop which constantly accepts
new connections from clients. Once a client is connected it sends the
string 'NICK' to it, which will tell the client that its nickname is
requested. After that it waits for a response (which hopefully contains
the nickname) and appends the client with the respective nickname to the
lists. After that, we print and broadcast this information. Finally, we
start a new thread that runs the previously implemented handling function
for this particular client. Now we can just run this function and our
server is done.

#Notice that we are always encoding and decoding the messages here. The
reason for this is that we can only send bytes and not strings. So we
always need to encode messages (for example using ASCII), when we send
them and decode them, when we receive them.

receive()
```

Client Implementation:

1. A server is pretty useless without clients that connect to it. So now we are going to implement our client. For this, we will again need to import the same libraries in a second separate script.

2. The first steps of the client are to choose a nickname and to connect to our server. We will need to know the exact address and the port at which our server is running. We are using a different function here. Instead of binding the data and listening, we are connecting to an existing server.

3. Now, a client needs to have two threads that are running at the same time. The first one will constantly receive data from the server and the second one will send our own messages to the server. So we will need two functions here. Let's start with the receiving part first.

4. We have an endless while-loop here. It constantly tries to receive messages and to print them onto the screen. If the message is the same as the nickname however, it doesn't print it but it sends its nickname to the server. In case there is some error, we close the connection and break the loop. Now we just need a function for sending messages.

5. The writing function also runs in an endless loop which is always waiting for input from the user. Once it gets some, it combines it with the nickname and sends it to the server. The last thing we need to do is to start two threads that run these two functions.

6. Now we start the threads for listening and writing. We have a fully-functioning server and working clients that can connect to it and communicate with each other. The client implementation is now complete.

Code:

```
import socket
import threading
#The first steps of the client are to choose a nickname and to connect to
our server. We will need to know the exact address and the port at which
our server is running.

# Choosing Nickname
nickname = input("Choose your nickname: ")

# Connecting To Server
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 55555))
#As you can see, we are using a different function here. Instead of
binding the data and listening, we are connecting to an existing server.

#Now, a client needs to have two threads that are running at the same
time. The first one will constantly receive data from the server and the
second one will send our own messages to the server. So we will need two
functions here. Let’s start with the receiving part.

# Listening to Server and Sending Nickname
def receive():
    while True:
        try:
            # Receive Message From Server
            # If 'NICK' Send Nickname
            message = client.recv(1024).decode('ascii')
            if message == 'NICK':
                client.send(nickname.encode('ascii'))
            else:
                print(message)
        except:
            # Close Connection When Error
            print("An error occured!")
            client.close()
            break
#Again we have an endless while-loop here. It constantly tries to receive
messages and to print them onto the screen. If the message is ‘NICK’
however, it doesn’t print it but it sends its nickname to the server. In
```

```
case there is some error, we close the connection and break the loop. Now
we just need a function for sending messages and we are almost done.

# Sending Messages To Server
def write():
    while True:
        message = '{}: {}'.format(nickname, input(''))
        client.send(message.encode('ascii'))
#The writing function is quite a short one. It also runs in an endless
loop which is always waiting for an input from the user. Once it gets
some, it combines it with the nickname and sends it to the server. That's
it. The last thing we need to do is to start two threads that run these
two functions.

# Starting Threads For Listening And Writing
receive_thread = threading.Thread(target=receive)
receive_thread.start()

write_thread = threading.Thread(target=write)
write_thread.start()
```

PROTOCOL USED:

The protocol used in this chat application project is TCP because we need to establish a connection between the server and the multiple clients. We will also need a security layer in the future of the application, so implementing it under the TCP protocol will help in securing the chat application once it is deployed. Another two very important reasons for using the TCP protocol is that because of the transmission guarantee and advanced error-checking mechanisms. Since it is a n-client chat application, we need to ensure that the messages being sent from client A to client B is being transmitted successfully, else at least a transmission failure error must be raised to inform the sender client. The message being sent also must not be corrupted while in the transmission path, so error checking mechanisms at the receiver end is a must.

RESULT/CONCLUSION:

While going for a test run , we always need to start the server first because otherwise the clients can't connect to a non-existing host. We noticed that the server was running without any errors and multiple clients were able to connect to the server and successfully chat in the chat room that was created. We can modify and customize the chat as required. We can also have multiple chat rooms or maybe we want to have different roles like admin and moderator.

FUTURE WORK:

A GUI version of this chat can be implemented for a better user experience. It works quite differently, since it works more with events than with endless loops.

REFERENCES:

1. https://github.com/topics/tcp-chat
2. https://www.codeproject.com/Articles/23868/A-TCP-Chat-Application
3. https://forgetcode.com/c/1202-chat-app-tcp
4. https://osandnetworkingcslab.wordpress.com/client-server-chat-using-tcp/
5. https://www.neuralnine.com/tcp-chat-in-python/