PROJECT REPORT

“Creating a Cloud Chat-room Using Socket Programming and AWS EC2 ”

Computer Communication Network

Department of ECE

PES University

Bangalore

Faculty in charge

PROF. PRAJEESHA

Submitted by :

Koushik P R [PES1UG20EC098]
Govind A S [PES1UG20EC073]
Harsh Verma [PES1UG20CE078]

Sem : V SEMESTER

Sec : B SECTION

CONTENTS:

OBJECTIVE/ PROBLEM STATEMENT

To create a Chatroom between different Cloud Servers using socket programming.

INTRODUCTION

CLOUD COMPUTING

With the increase in data management and storage, Cloud Computing and storage is the trending way to programming and implementing a Cloud Infrastructure. Cloud computing allows us to provide 4 types of Services.

1. CaaS: Container as a Service
We will be implementing an Elastic Cloud Container (AWS EC2 Instance)
2. PaaS: Platform as a Service
We will be implementing in Amazon Web Services (Cloud Provider)
3. IaaS: Infrastructure as a Service
We will be implementing the code in Amazon Linux Operating System
4. SaaS: Software as a Service
We will be using socket programming using the Python interface.

SOCKET PROGRAMMING

A *socket* is a communications connection point (endpoint) that you can name and address in a network. Socket programming shows how to use socket APIs to establish communication links between remote and local processes.

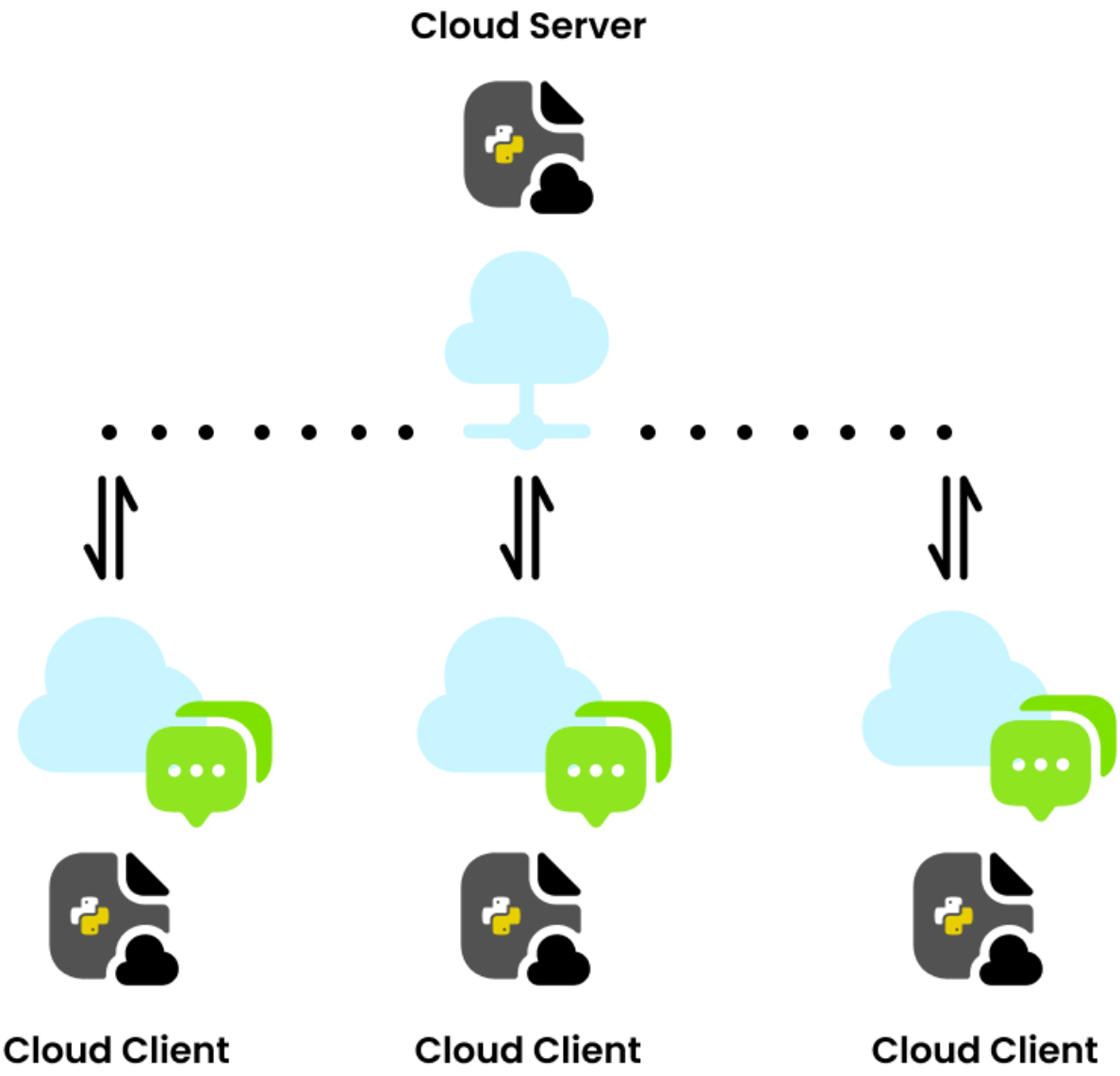
The type of sockets that are most suitable are **TCP sockets** as the TransmissionControl Protocol (TCP):

- **Is reliable:** packets dropped in the network are detected and retransmitted by the sender.
- **Has in-order data delivery:** data is read by your application in the order it was written by the sender.

SECURITY

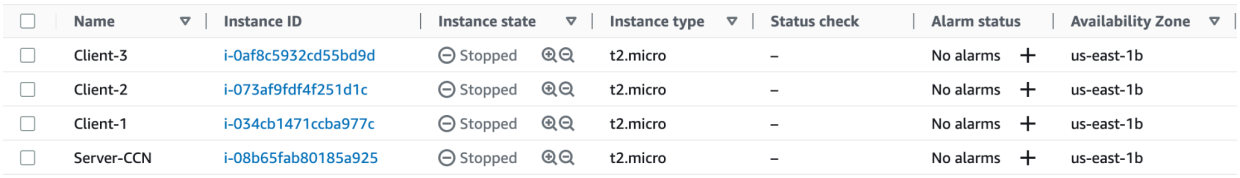
When working with Cloud Containers Security is a necessity. EC2 Instances comes with a default Security group provided by AWS which creates an access between our computer and the Cloud server. In our program we will be using the server as a Hub for generating a service. So hence all messages are received by the server and sent to all clients. This is a better approach while working on cloud platforms exposing only 1 port connecting to the server in security groups of EC2 Instances. The only drawback of this implementation is that the API is not end-end encrypted since the server acts as a third party object.

FLOW CHART



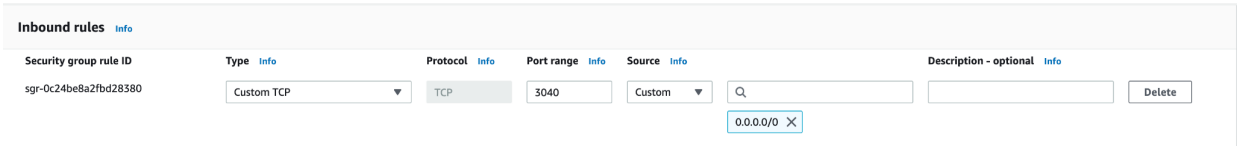
PROCEDURE:

1. Connect to EC2 instances via SSH command using the terminal.



Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
Client-3	i-0af8c5932cd55bd9d	Stopped	t2.micro	–	No alarms	us-east-1b
Client-2	i-073af9fdf4f251d1c	Stopped	t2.micro	–	No alarms	us-east-1b
Client-1	i-034cb1471ccba977c	Stopped	t2.micro	–	No alarms	us-east-1b
Server-CCN	i-08b65fab80185a925	Stopped	t2.micro	–	No alarms	us-east-1b

2. Go to the security group of the all 4 instances and Edit Inbound Rules and select the following

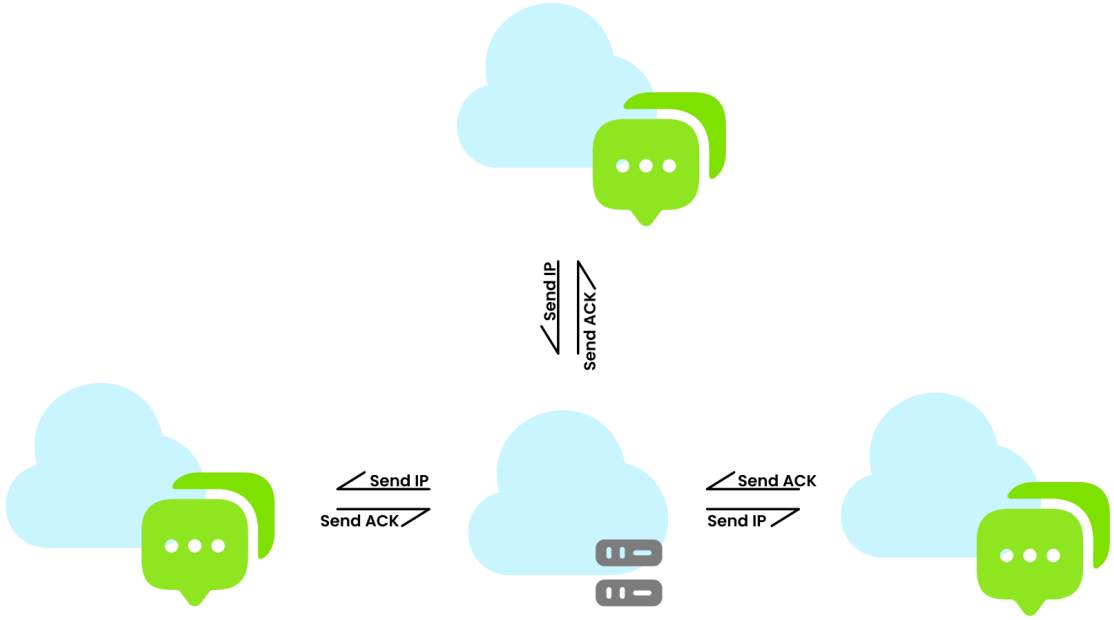


3. Copy files to the instances using the following commands
    a. Cloud Server instance (Server-CCN)
        - cd directory of server.py
        - cat server.py > | ssh -i ec2-user@<server-ip>
    b. Cloud Client Instances (Clients-1,2,3)
        - cd directory of client.py
        - cat client.py > | ssh -i ec2-user@<server-ip>
4. Ensure that Client and Server are exposed on Same TCP port as Inbound Rule.
5. There are 4 instances to be configured
    a. Cloud Server Instance: Run the following commands
        - sudo su - # Become the root user for exposing port
        - python server.py #Run the server python file
        - Obtain the IP address from the output
        - Enter name: “server” #Write down the Canonical name of the server.

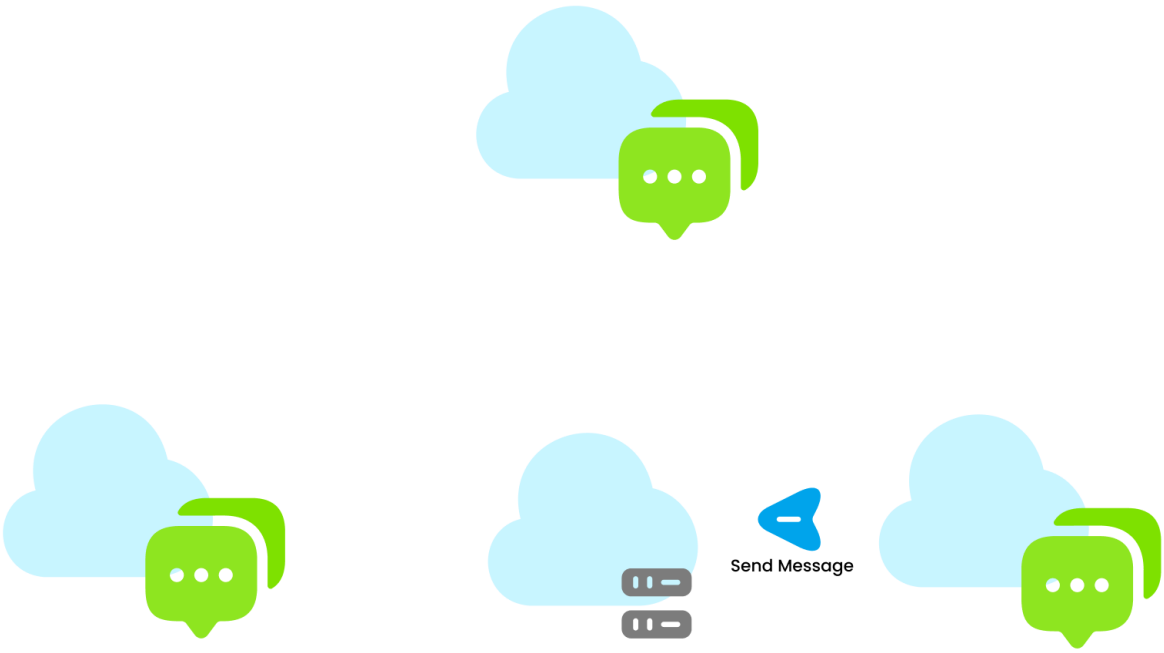
    b. Cloud Client Instance: Run the following commands
        - python client.py # Run the client python File
        - Enter Server IP Address: #Write down the IP of Cloud Server
        - Enter name: #Write down the user name for chat room.
        - Type messages to send in the chatroom
        - Type messages starting with“@<Username>” to send Private messages to respective username.

Task diagram

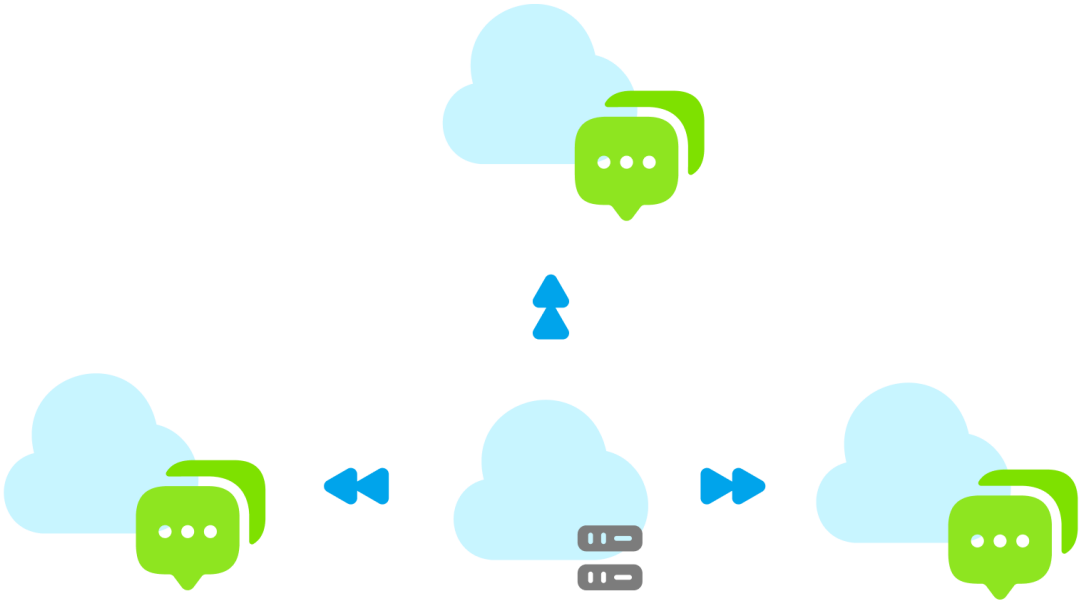
Phase 1: Cloud Server Broadcasts Connect in port 3040



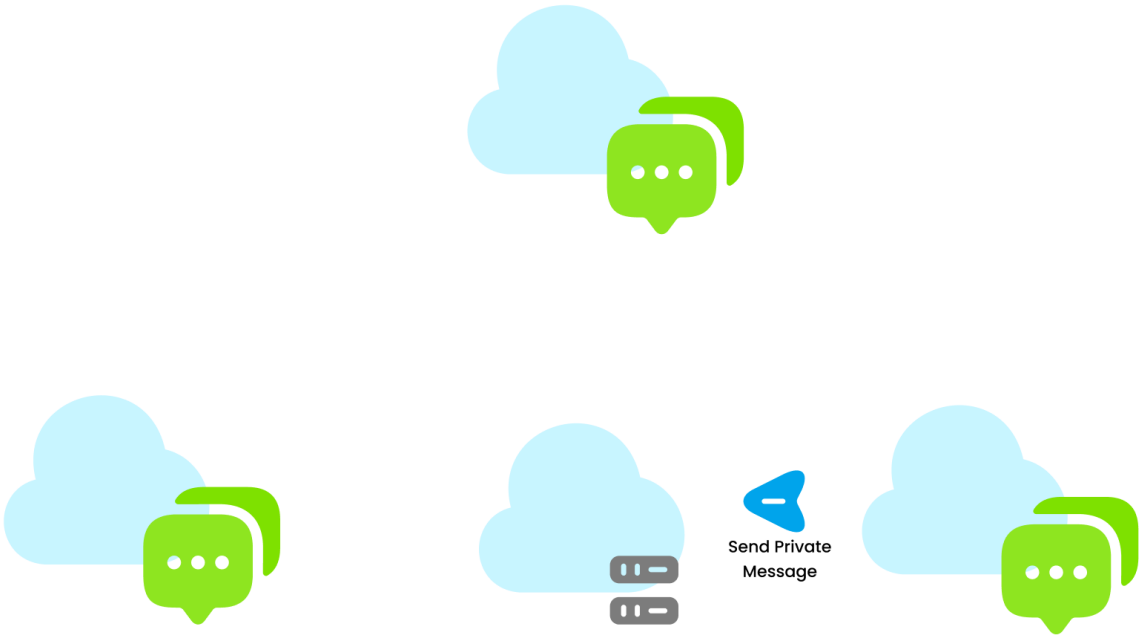
Phase 2: Cloud Clients With Access to Server IP Request Connection



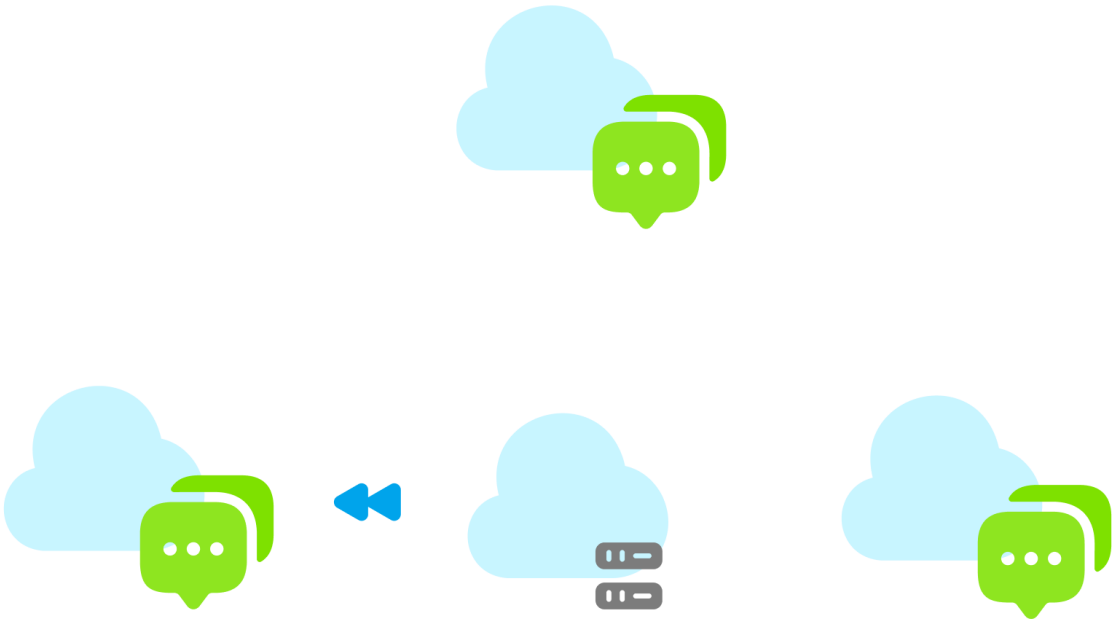
Phase 3: Cloud Clients Send Messages to Chatbox Via Server



Phase 4: Cloud Server Sends Messages to all Clients using Multicast



Phase 5: Cloud Client Sends Private Message to Specific User



Phase 6: Cloud Server Sends Message to Specific User using Unicast

Socket programming

1. Cloud Server python File : Server.py

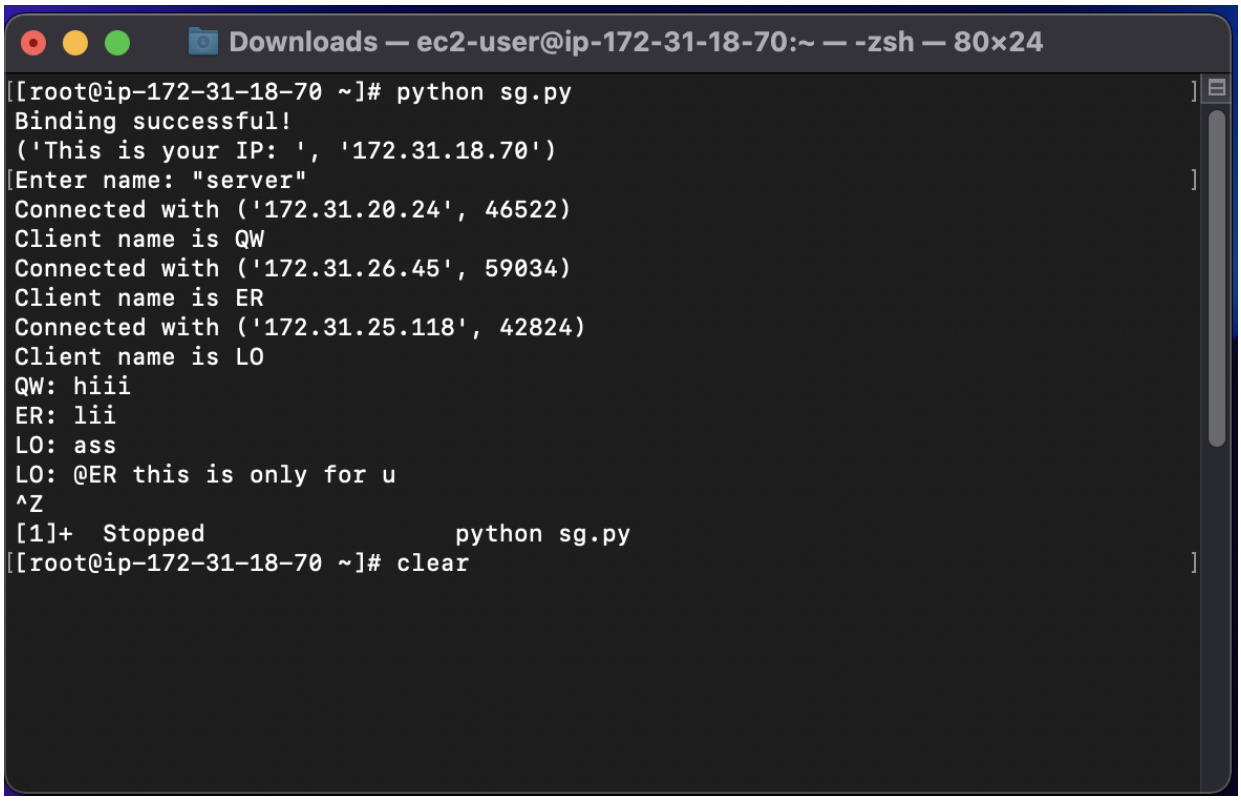
```
import time, socket, sys,threading
server = socket.socket()
host_name = socket.gethostname()
s_ip = socket.gethostbyname(host_name)
port = 3040
server.bind((host_name, port))
print("Binding successful!")
print("This is your IP: ", s_ip)
name = input('Enter name: ')
server.listen(1)
clients = []
nicknames = []
pp = {}

def private(message):          #UniCast Function Declaration
    x = message
    neww = str(x.decode("ascii"))
    index = neww.index("@") + 1
    neww = neww[index:]
    for x in nicknames:
        if neww.startswith(str(x)):
            client = pp[x]
            client.send(message)

def broadcast(message):        #MultiCast function declaration
    for client in clients:
        client.send(message)

def handle(client):
    while True:
        try:                   #recieving valid messages from client
            message = client.recv(1024)
            x = message
            test = x.decode("ascii")
            print(test)
            if "@" in test:
                private(message)
            else:
                broadcast(message)
        except:                 #removing clients
            index = clients.index(client)
            clients.remove(client)
            client.close()
            nickname = nicknames[index]
            broadcast('{} left!'.format(nickname).encode('ascii'))
            nicknames.remove(nickname)
            break

def receive():                 #accepting multiple clients
    while True:
        client, address = server.accept()
        print("Connected with {}".format(str(address)))
        client.send('NICKNAME'.encode('ascii'))
        nickname = client.recv(1024).decode('ascii')
        nicknames.append(nickname)
        clients.append(client)
        pp[nickname] = client
        print("Client name is {}".format(nickname))
        broadcast("{} joined!".format(nickname).encode('ascii'))
        client.send('Connected to server!'.encode('ascii'))
        thread = threading.Thread(target=handle, args=(client,))
        thread.start()
```

2. Cloud Client python file: Client.py

```
import time, socket, sys,threading

server_host = socket.gethostname()
ip = socket.gethostbyname(server_host)
sport = 3040

print('This is your IP address: ',ip)
server_host = input('Enter server IP address:')
nickname = input("Choose your nickname: ")


client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)      #socket initialization
client.connect((server_host, sport))                            #connecting client to server


def receive():
    while True:                                                 #making valid connection
        try:
            message = client.recv(1024).decode('ascii')
            if message == 'NICKNAME':
                client.send(nickname.encode('ascii'))
            else:
                print(message)
        except:                                                 #case on wrong ip/port details
            print("An error occured!")
            client.close()
            break
def write():
    while True:                                                 #message layout
        print("You: ")
        x = input("")
        message = '{}: {}'.format(nickname, x)
        client.send(message.encode('ascii'))

write_thread = threading.Thread(target=write)                   #sending messages
write_thread.start()
receive_thread = threading.Thread(target=receive)               #receiving multiple messages
receive_thread.start()
```

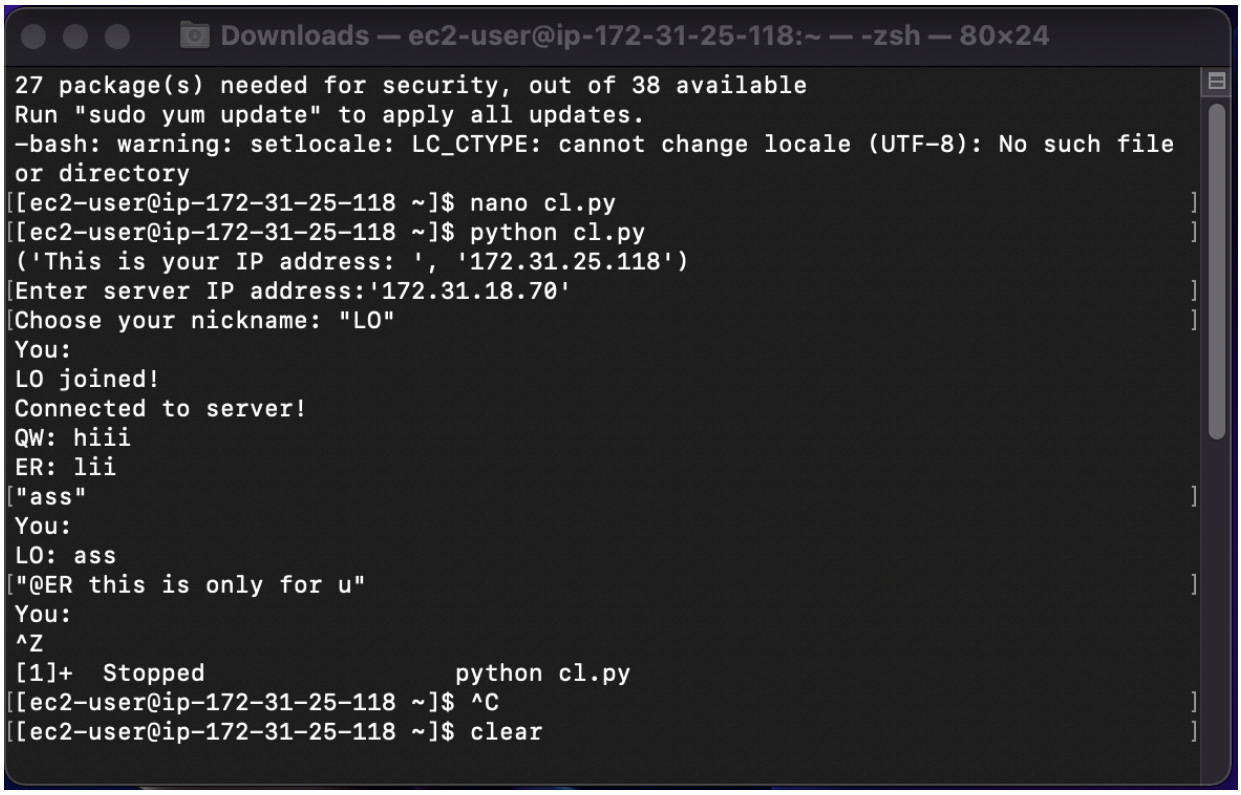
RESULTS

1. Cloud Server Instance that Acts as a Hub.

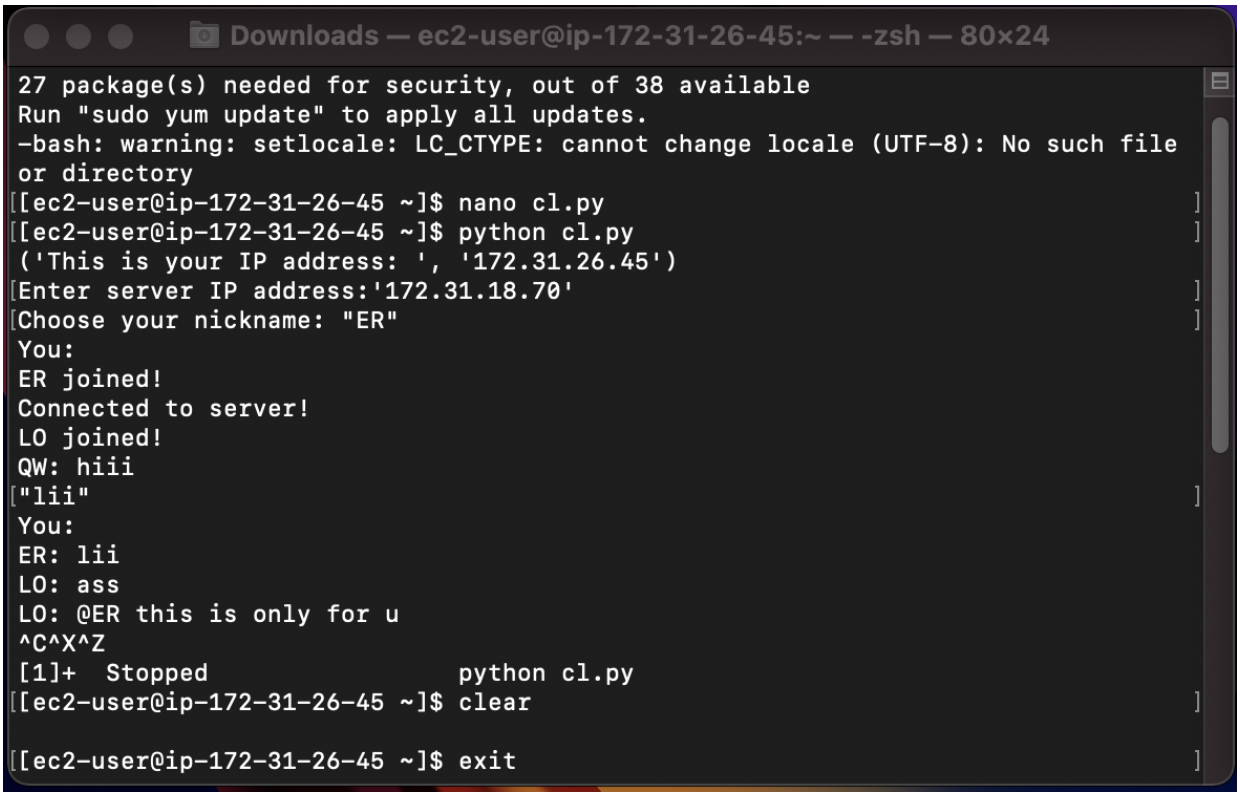


Inference: The Cloud Server is connected to 3 Clients (QW, ER and LO)

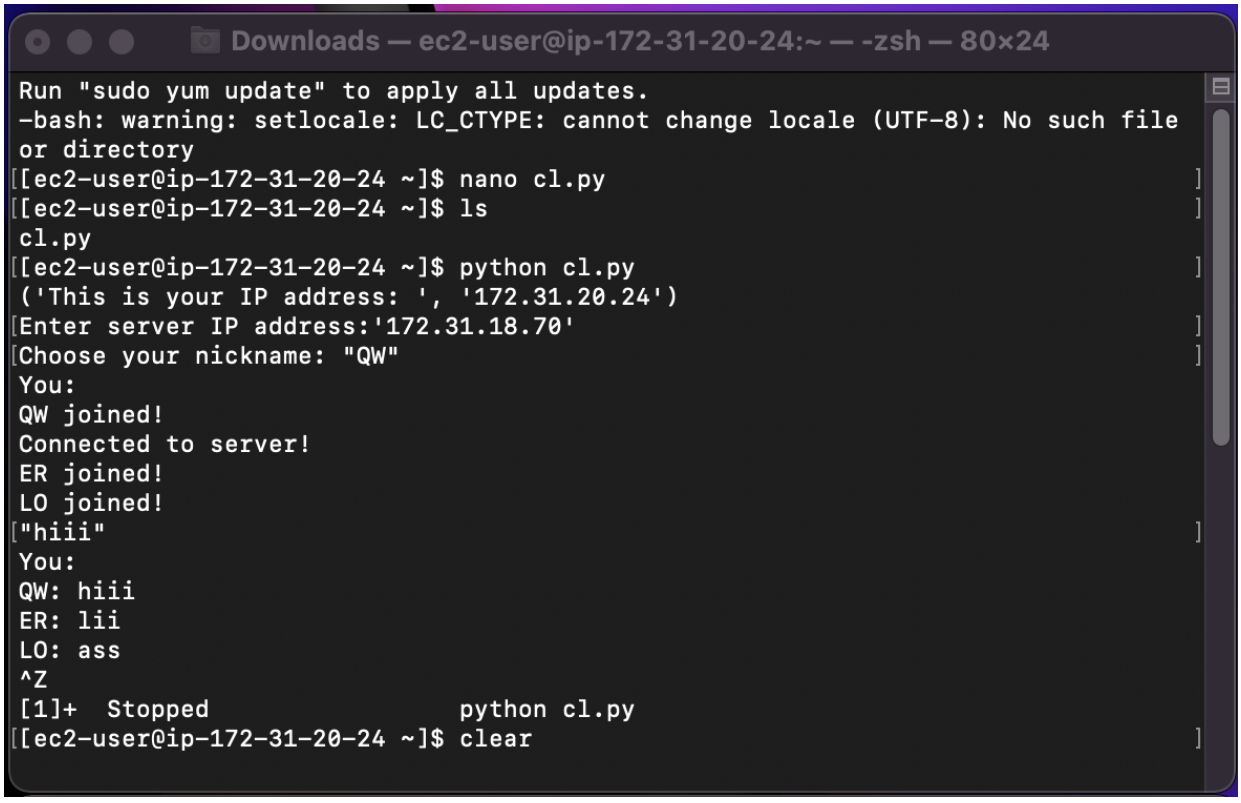
2. Cloud Clients Instance That sends Messages
    a. Client- 1 “LO” Sends Public Messages and Private Message to “ER”



b. Client -2 “ER” Sends Public Messages and Receives Private Message from “LO”



c. Client -3 “QW” Sends Public Messages and Does not Receive Private Messages



CONCLUSION

- Cloud Server acts as a **Hub**, allowing users to send data via passing through the server
- The connection is not **end-end encrypted** since the server can also receive the data from the clients and can be accessed unless encrypted
- The Cloud Server **broadcasts** the access to the exposed port.
- The Connection cannot be made with the Server Unless the **IP address of the server is granted** to the clients hence ensuring **Secure Connection**.
- The Client has the ability to send messages to A single user (Private message) using **"@<username>"** and multi-user (Public message).
- The Server by default sends the data to all users using **Multicast** hence users which are not Connected to the hub cannot access the data from outside.
- The Server checks the data for "@<username>" and sends a **Unicast** Message to the specific Client.
- The data is being processed in the Server and hence the Clients program utilizes less resources and hence much more feasible.
- The program runs on **TCP** and hence the **secure data transfer** is ensured.
- The Cloud provides a flexibility to select the **type of traffic** (In this case it is TCP) and expose our desired port (In this case it is 3040). So no other port can access the data.

FUTURE SCOPE

1. This implementation can be further extended to transfer various types of data between different Cloud Clients, which is way more feasible that usage of 3rd party means like Git Clone (Data Manipulation) , Boto Bucket (High interface demand),etc
2. Servers or Hubs can **handle offensive texts** and report data to the users.
3. Servers can grant access to specific users present in the database by creating **authentication** for all clients.
4. With the help of Cloud computing, creating replicas of same Cloud Server hubs can reduce traffic at a single server and allows use to implement **Load Balancing.**
5. Different Cloud Servers can be made to handle different types of data that carry different sizes of data by generating **Containers with higher Resources.**

Tools Used:

1. **Python:** For socket programming
2. **AWS EC2 Instance (Amazon Linux OS) :** For creating Cloud Server and Cloud Client
3. **Figma:** For creating Flowchart and Phase of Implementation of the API
4. **Bash Terminal:** Scripting Data
5. **Amazon Security Group:** For exposing ports on Cloud Servers and Clients