AIM

To create a chat application using socket programming.

THEORY

SOCKETS

Socket is the point where a local application process attaches to the network and that allows processes to communicate with each other over a network. They provide a way for programs running on different machines to exchange data, regardless of the underlying hardware or operating system. The socket interface defines operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket, and closing the socket.

TCP

TCP is a reliable, connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket.

When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket. The client has the job of initiating contact with the server. In order for the server to be able to react to the client's initial contact, the server has to be ready.

SOCKETS PROGRAMMING USING PYTHON

Sockets programming is the process of communicating between two endpoints over a network using sockets. A socket is an endpoint that is used to send or receive data over a network. Python provides a built-in module called **socket** for sockets programming.

SOURCE CODE

SERVER

```
from socket import *
import threading

clients = {}

def client_handler(cSocket, address):
    client_name = cSocket.recv(1024).decode()
    clients[client_name] = cSocket
    print(f"[NEW CONNECTION] {client_name} connected from {address}.")

    message = f"{client_name} has joined the chat room."
    for name, socket in clients.items():
        if socket != cSocket:
            socket.send(message.encode())

    while True:
        data = cSocket.recv(1024)
        if not data:
            break

        message = data.decode()
        if message == 'q:quit':
            del clients[client_name]
            cSocket.close()
            print(f"[CONNECTION CLOSED] {client_name} disconnected.")
            cSocket.send(message.encode())
            message = f"{client_name} has left the chat room."
            for name, socket in clients.items():
                if socket != cSocket:
                    socket.send(message.encode())
            return

        sender, msg = message.split(':', 1)
```

```
        print(f"[{client_name}] {msg}")

        for name, socket in clients.items():
            if socket != cSocket:
                socket.send(message.encode())

sPort = 4000
sSocket = socket(AF_INET, SOCK_STREAM)
ADDR = ('localhost', sPort)
sSocket.bind(ADDR)

sSocket.listen(4)
print("The chat server is listening at ", ADDR)

while True:
    cSocket, address = sSocket.accept()
    print("Server connected with ", address)

    cThread = threading.Thread(target=client_handler, args=(cSocket, address))
    cThread.start()
```

EXPLANATION

from socket import *
import threading

The socket module provides the basic networking functions needed for the server to listen for incoming connections and for the clients to connect to the server. The threading module is used to handle multiple clients concurrently.

def client_handler(cSocket, address):

The client_handler function is called in a separate thread for each connected client. It receives the client's socket object cSocket and the client's address address.

```
client_name = cSocket.recv(1024).decode()
clients[client_name] = cSocket
```

When a client first connects, it sends its name as a message to the server, which is received using the recv method which is then added to clients dictionary.

```
message = f"{client_name} has joined the chat room."
for name, socket in clients.items():
    if socket != cSocket:
        socket.send(message.encode())
```

A message is then sent to all connected clients (except the new one) indicating that the client has joined the chat room.

```
if message == 'q:quit':
    del clients[client_name]
    cSocket.close()
    print(f"[CONNECTION CLOSED] {client_name} disconnected.")
    cSocket.send(message.encode())
    message = f"{client_name} has left the chat room."
    for name, socket in clients.items():
        if socket != cSocket:
            socket.send(message.encode())
    return
```

The server then enters a loop to receive and send messages from/to the client. If the client sends a message starting with "q:quit", it is interpreted as a request to disconnect.

```
for name, socket in clients.items():
    if socket != cSocket:
        socket.send(message.encode())
```

Otherwise, the message is sent to all connected clients (except the sender).

```
while True:
    cSocket, address = sSocket.accept()
    print("Server connected with ", address)

    cThread = threading.Thread(target=client_handler, args=(cSocket, address))
    cThread.start()
```

Finally, the main loop of the server listens for incoming connections and spawns a new thread to handle each one.

CLIENT

```
from socket import *
import threading
import sys
import time

def receive_messages(cSocket):
    while True:
        message = cSocket.recv(1024).decode()
        if not message:
            break
        if message == 'q:quit':
            break
        if ':' not in message:
            print(f"\r\033[K{message}\n{name}: ", end="")
            if name in message:
                print("\r\033[K", end="")
        else:
            sender, msg = message.split(':', 1)
            print(f"\r\033[K<{sender}>:{msg}\n{name}: ", end="")
        sys.stdout.flush()

sHost = 'localhost'
sPort = 4000

cSocket = socket(AF_INET, SOCK_STREAM)
cSocket.connect((sHost, sPort))

name = input("Enter your name: ")
cSocket.send(name.encode())
print("[Press 'q' to exit]")

receive_thread = threading.Thread(target=receive_messages, args=(cSocket,))
receive_thread.start()

while True:
    message = input(f"{name}: ")
```

```
    if message == 'q':
        cSocket.send(f"q:quit".encode())
        break
    else:
        cSocket.send(f"{name}: {message}".encode())

time.sleep(0.01)
cSocket.close()
```

EXPLANATION

```
from socket import *
import threading
import sys
import time
```

The code imports the necessary modules for creating a socket, threading, system interaction, and time.

```
def receive_messages(cSocket):
  while True:
    message = cSocket.recv(1024).decode()
    if not message:
      break
    if message == 'q:quit':
      break
    if ':' not in message:
      print(f"\r\033[K{message}\n{name}: ", end="")
      if name in message:
        print("\r\033[K", end="")
    else:
      sender, msg = message.split(':', 1)
      print(f"\r\033[K<{sender}>:{msg}\n{name}: ", end="")
    sys.stdout.flush()
```

The receive_messages() function is defined to handle incoming messages from the server. The received message is decoded and checked for a colon. If there is no colon in the message, it means it is a system message from the server (like a new user joined). If the message has a colon, it means it is a message from another client, so it is split into the sender and message content and printed accordingly. ys.stdout.flush() is called to make sure the printed message is displayed immediately.

```
sHost = 'localhost'
sPort = 4000

cSocket = socket(AF_INET, SOCK_STREAM)
cSocket.connect((sHost, sPort))

name = input("Enter your name: ")
cSocket.send(name.encode())
print("[Press 'q' to exit]")
```

A client socket is created and connected to the server with the specified host and port. The client's name is obtained through user input, encoded, and sent to the server using the cSocket.send() function.

```
receive_thread = threading.Thread(target=receive_messages, args=(cSocket,))
receive_thread.start()
```

A new thread is created to run the receive_messages() function in the background. This allows the client to receive and display messages while still being able to send messages in the main thread.

```
while True:
    message = input(f"{name}: ")
    if message == 'q':
        cSocket.send(f"q:quit".encode())
        break
    else:
        cSocket.send(f"{name}: {message}".encode())
```

The main thread runs in an infinite loop until the user inputs 'q' to exit. If the user inputs 'q', the client sends 'q:quit' to the server to indicate that the client is disconnecting.

```
time.sleep(0.01)
cSocket.close()
```

After exiting the loop, a small delay is introduced using time.sleep() to make sure the last message is received before closing the connection with cSocket.close().

RESULTS

Server when a connection is made and name is entered

```
PS D:\ECE\CN\CHAT> py server.py
The chat server is listening at  ('localhost', 4000)
Server connected with  ('127.0.0.1', 54757)
[NEW CONNECTION] A connected from ('127.0.0.1', 54757).
```

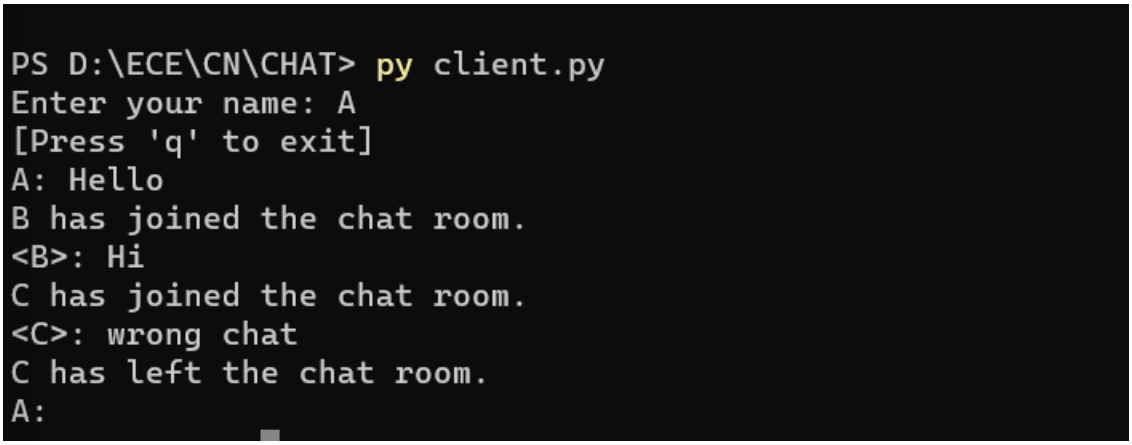
Exiting from the client application (using char 'q')

```
PS D:\ECE\CN\CHAT> py client.py
Enter your name: C
[Press 'q' to exit]
C: wrong chat
C: q
```

Server console after some events

```
PS D:\ECE\CN\CHAT> py server.py
The chat server is listening at  ('localhost', 4000)
Server connected with  ('127.0.0.1', 54757)
[NEW CONNECTION] A connected from ('127.0.0.1', 54757).
[A]  Hello
Server connected with  ('127.0.0.1', 54765)
[NEW CONNECTION] B connected from ('127.0.0.1', 54765).
[B]  Hi
Server connected with  ('127.0.0.1', 54766)
[NEW CONNECTION] C connected from ('127.0.0.1', 54766).
[C]  wrong chat
[CONNECTION CLOSED] C disconnected.
```

Client console for A after some messages



OBSERVATIONS

We can observe that

- The server binds a socket to a specific IP address and port number using the bind() function of the socket module and listens for incoming connections using the listen() function.
- Upon receiving a connection request from a client, the server accepts the request using the accept() function and creates a new thread to handle communication with the client.
- Threads are used to allow multiple tasks to run concurrently within a program, which can improve program performance, responsiveness, and efficiency.
- Each client thread continuously receives messages from its associated socket and broadcasts the message to all other connected clients by sending the message to their corresponding sockets.

CONCLUSION

Chat application is implemented using Socket programming and it is used to chat between clients.