**Department of Computer Science and Engineering,**

**University of Engineering Mardan**

****

**“ MULTI CHAT SYESTEM PROJECT”**

**Course Code: (CS-302L)**

**Course Name:**

**OPERATING SYETM LAB**

**PRESENTED TO:**

**PROF DR.INAYAT SIR**

**SECTION:**

**B**

**Team Members:**

|  |  |
| --- | --- |
| **Name :**  **MUHAMMAD MUSTAFA**  **REG NO : 21MDBCS121**  **Name :**  **MUHAMMAD ADIL KHAN**  **REG NO : 21MDBCS129** | **Name:**  **USAMA SARDAR**  **REG NO : 21MDBCS126** |

**PROJECT DESCRIPTION:**

**Overview:**

Design and implement a multichat system using Java socket programming. The goal is

to create a server that facilitates communication between multiple clients in a chat-like

environment.

**Project Components:**

**Server Implementation:**

Develop a server application that listens for incoming client connections.

Implement a mechanism to handle multiple clients concurrently.

Maintain a set of connected clients and ensure proper synchronization to handle

simultaneous messages.

**Client Implementation:**

Create a client application that connects to the server.

Implement a thread to listen for messages from the server and display them to the user.

Allow the user to input messages to be sent to the server and subsequently broadcasted

to all connected clients.

**Communication Protocol:**

Define a simple communication protocol between the server and clients. Specify how

messages should be formatted and transmitted.

**Error Handling:**

Implement robust error handling mechanisms on both the server and client sides.

Consider scenarios such as connection failures, unexpected disconnections, or invalid

messages.

**User Interface:**

If you want to take the project further, consider adding a basic user interface for the client

application to enhance the user experience.

**Documentation:**

Provide comprehensive documentation for your project. Include clear instructions on how

to compile, run, and test the server and client applications.

PROJECT:

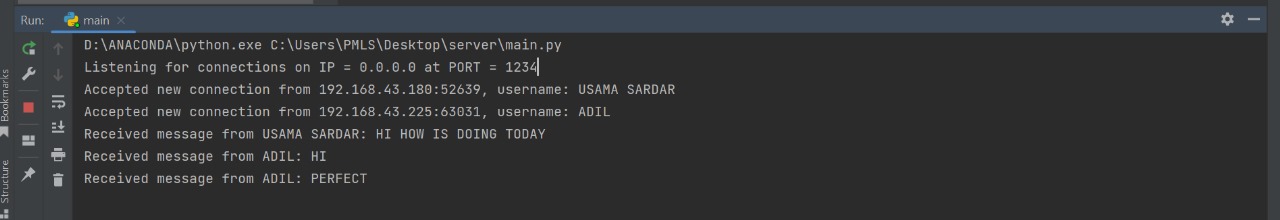
**SERVER SIDE CODE:**

# importing the necessary modules.  
import socket  
import select  
  
# defining the header length.  
HEADER\_LENGTH = 10  
  
# defining the IP address and Port Number.  
IP = "0.0.0.0"  
PORT = 1234  
  
"""  
Creating a server socket and providing the address family (socket.AF\_INET) and type of connection (socket.SOCK\_STREAM), i.e. using TCP connection.  
"""  
server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
  
"""  
Modifying the socket to allow us to reuse the address. We have to provide the socket option level and set the REUSEADDR (as a socket option) to 1 so that address is reused.  
"""  
server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  
  
# Binding the socket with the IP address and Port Number.  
server\_socket.bind((IP, PORT))  
  
# Making the server listen to new connections.  
server\_socket.listen()  
  
# List of sockets for select.select()  
sockets\_list = [server\_socket]  
  
# A set to contain the connected clients.  
clients = {}  
  
print(f'Listening for connections on IP = {IP} at PORT = {PORT}')  
  
  
# A function for handling the received message.  
def receive\_message(client\_socket):  
 try:  
 """  
 The received message header contains the message length, its size is defined, and the constant.  
 """  
 message\_header = client\_socket.recv(HEADER\_LENGTH)  
  
 """  
 If the received data has no length then it means that the client has closed the connection. Hence, we will return False as no message was received.  
 """  
 if not len(message\_header):  
 return False  
  
 # Convert header to an int value  
 message\_length = int(message\_header.decode('utf-8').strip())  
  
 # Returning an object of the message header and the data of the message data.  
 return {'header': message\_header, 'data': client\_socket.recv(message\_length)}  
  
 except:  
 return False  
  
  
# running an infinite loop to accept continuous client requests.  
while True:  
 # Read the data using a select module from the socketLists.  
 read\_sockets, \_, exception\_sockets = select.select(  
 sockets\_list, [], sockets\_list)  
  
 # Iterating over the notified sockets.  
 for notified\_socket in read\_sockets:  
 """  
 If the notified socket is a server socket then we have a new connection, so add it using the accept() method.  
 """  
 if notified\_socket == server\_socket:  
 client\_socket, client\_address = server\_socket.accept()  
  
 # Else the client has disconnected before sending his/her name.  
 user = receive\_message(client\_socket)  
  
 # If False - client disconnected before he sent his name  
 if user is False:  
 continue  
  
 # Adding the accepted socket to select.select() list.  
 sockets\_list.append(client\_socket)  
  
 # Also adding the username and username header.  
 clients[client\_socket] = user  
  
 print('Accepted new connection from {}:{}, username: {}'.format(  
 \*client\_address, user['data'].decode('utf-8')))  
  
 # Else the existing socket is sending a message so handling the existing client.  
 else:  
 # Receiving the message.  
 message = receive\_message(notified\_socket)  
  
 # If no message is accepted then finish the connection.  
 if message is False:  
 print('Closed connection from: {}'.format(  
 clients[notified\_socket]['data'].decode('utf-8')))  
  
 # Removing the socket from the list of the socket.socket()  
 sockets\_list.remove(notified\_socket)  
  
 # Removing the user from the list of users.  
 del clients[notified\_socket]  
  
 continue  
  
 # Getting the user by using the notified socket, so that the user can be identified.  
 user = clients[notified\_socket]  
  
 print(  
 f'Received message from {user["data"].decode("utf-8")}: {message["data"].decode("utf-8")}')  
  
 # Iterating over the connected clients and broadcasting the message.  
 for client\_socket in clients:  
 # Sending to all except the sender.  
 if client\_socket != notified\_socket:  
 """  
 Reusing the message header sent by the sender, and saving the username header sent by the user when he/she connected.  
 """  
 client\_socket.send(  
 user['header'] + user['data'] + message['header'] + message['data'])

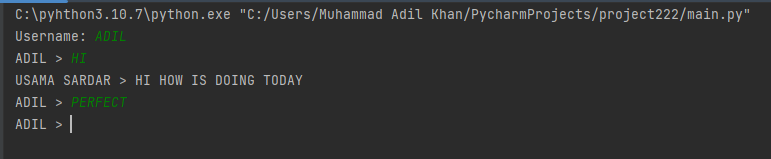
**CLIENT SIDE CODE:**

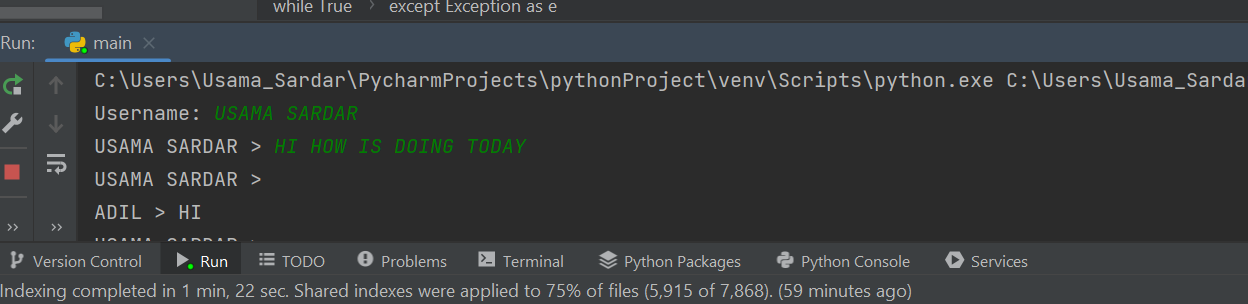
# importing the necessary modules.  
import socket  
import sys  
import errno  
  
# defining the header length.  
HEADER\_LENGTH = 10  
  
# defining the IP address and Port Number.  
IP = "192.168.43.209"  
PORT = 1234  
  
# Getting the name of the client.  
my\_username = input("Username: ")  
  
"""  
Creating a client socket and providing the address family (socket.AF\_INET) and type of connection (socket.SOCK\_STREAM), i.e. using TCP connection.  
"""  
client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
  
# Connecting the socket with the IP address and Port Number.  
client\_socket.connect((IP, PORT))  
  
"""  
Setting the connection to a non-blocking state so that the recv() function call will not get blocked. It will return some exceptions only.  
"""  
client\_socket.setblocking(False)  
  
# Setting the username and header.  
username = my\_username.encode('utf-8')  
username\_header = f"{len(username):<{HEADER\_LENGTH}}".encode('utf-8')  
  
"""  
Here, we have encoded the username into bytes, counted the number of bytes, and then prepared a header of fixed size, that we have encoded to bytes as well.  
"""  
  
# sending the data.  
client\_socket.send(username\_header + username)  
  
  
  
  
# running an infinite loop to send continuous client requests.  
while True:  
 # Getting user input.  
 message = input(f'{my\_username} > ')  
  
 # Sending the non-empty message.  
 if message:  
 """  
 encode the message into bytes, counted the number of bytes, and then prepared a header of fixed size, that we have encoded to bytes as well.  
 """  
 message = message.encode('utf-8')  
 message\_header = f"{len(message):<{HEADER\_LENGTH}}".encode('utf-8')  
 # sending the message.  
 client\_socket.send(message\_header + message)  
  
 try:  
 # looping over the received messages and printing them.  
 while True:  
 # getting the header.  
 username\_header = client\_socket.recv(HEADER\_LENGTH)  
  
 # If no header is accepted then finish the connection.  
 if not len(username\_header):  
 print('Connection closed by the server')  
 sys.exit()  
  
 # Converting the header to an int value.  
 username\_length = int(username\_header.decode('utf-8').strip())  
  
 # Decoding the received username.  
 username = client\_socket.recv(username\_length).decode('utf-8')  
  
 # Decoding the received message.  
 message\_header = client\_socket.recv(HEADER\_LENGTH)  
 message\_length = int(message\_header.decode('utf-8').strip())  
 message = client\_socket.recv(message\_length).decode('utf-8')  
  
 # Printing the message.  
 print(f'{username} > {message}')  
  
 except IOError as e:  
 # handling the normal error on nonblocking connections.  
 if e.errno != errno.EAGAIN and e.errno != errno.EWOULDBLOCK:  
 print('Reading error: {}'.format(str(e)))  
 sys.exit()  
  
 # If we did not receive anything, then continue.  
 continue  
  
 except Exception as e:  
 print('Reading error: '.format(str(e)))  
 sys.exit()

**SERVER SIDE OUTPUT:**



**CLIENT SIDE OUTPUT:**

****



**((((((((((((((((END)))))))))))))))**