REPORT ASSIGNMENT

IMPLEMENTATION CLIENT SERVER WITH SOCKETS

Name : Adam Bastian Chaniago

NIM : 20240040129

Course : System Parallel and Distribution

1. Implementation & Code

The goal of this program is to create a simple chat application between a single client and a server, handled sequentially, using Python.

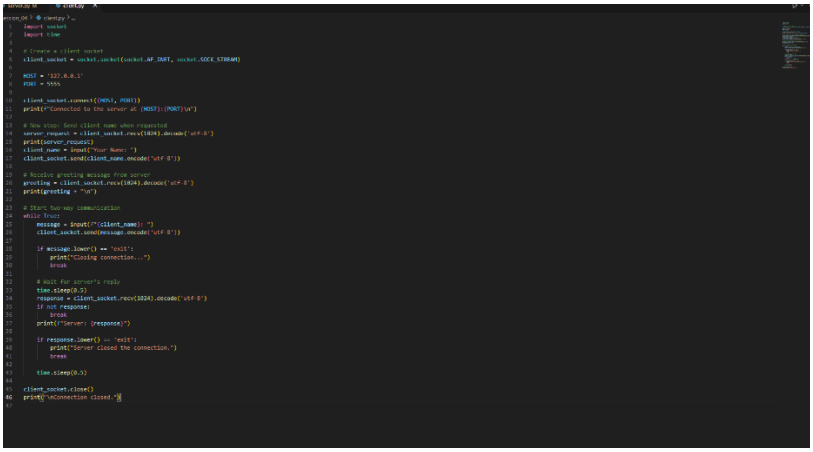
a. Server Code (server.py)

Core Logic: The server creates a socket, binds it to the localhost address (127.0.0.1) on port 5555, and enters listening mode. The server will wait to accept a connection from a client. After connecting, the server will enter a loop to recv (receive) messages and send replies until the connection is closed.

```python
import socket
import time

# Create a server socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = '127.0.0.1'
PORT = 5555

server_socket.bind((HOST, PORT))
server_socket.listen()

print("Server is running on (HOST):(PORT)")
print("Waiting for a client to connect...\n")

while True:
    try:
        # Accept a new client connection
        client_socket, client_address = server_socket.accept()
        print("Connected to client: {client_address}\n")

        # Ask for client's name
        client_socket.send("Please enter your name: ".encode('utf-8'))
        client_name = client_socket.recv(1024).decode('utf-8')
        print("New client joined with name: {client_name}")
        client_socket.send(f"Hello {client_name}! Welcome to the server !\".encode('utf-8'))

        # Two-way communication with the client
        while True:
            message = client_socket.recv(1024).decode('utf-8')
            if not message:
                print(f"Connection with {client_name} was lost.\n")
                break

            print(f"{client_name}: {message}")

            if message.lower() == 'exit':
                print(f"{client_name} has left the server.\n")
                break

            # Server's reply
            time.sleep(0.5)
            response = input("Server: ")
            client_socket.send(response.encode('utf-8'))

            if response.lower() == 'exit':
                print("Server has closed this chat session.\n")
                break

        # Close connection with the client after the session ends
        client_socket.close()
        print("Waiting for the next client...\n")

    except KeyboardInterrupt:
        print("Server manually stopped.")
        break
    except Exception as e:
        print(f"An error occurred: {e}")
        continue

# Close the server after exiting the loop
server_socket.close()
print("Server has been completely shut down.")
```

b. Client Code (client.py)

Core Logic: The client creates a socket and actively attempts to connect to the server's IP address and port. After connecting, the client enters a loop to send messages from user input and wait for replies from the server (recv).



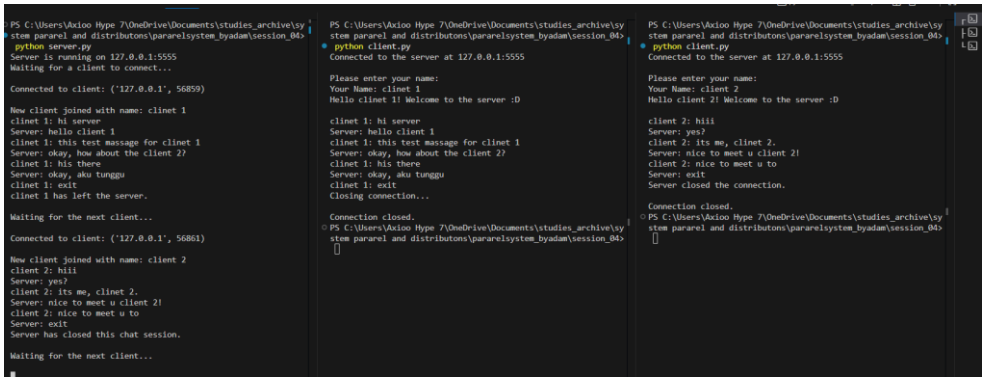
```python
import socket
import time

# Create a client socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = '127.0.0.1'
PORT = 5555

client_socket.connect((HOST, PORT))
print(f"Connected to the server at (HOST):(PORT)\n")

# Now step: Send client name when requested
server_request = client_socket.recv(1024).decode('utf-8')
print(server_request)
client_name = input("Your Name: ")
client_socket.send(client_name.encode('utf-8'))

# Receive greeting message from server
greeting = client_socket.recv(1024).decode('utf-8')
print(greeting + "\n")

# Start two-way communication
while True:
    message = input(f"{client_name}: ")
    client_socket.send(message.encode('utf-8'))

    if message.lower() == 'exit':
        print("Closing connection...")
        break

    # Wait for server's reply
    time.sleep(0.2)
    response = client_socket.recv(1024).decode('utf-8')
    if not response:
        break
    print(f"Server: {response}")

    if response.lower() == 'exit':
        print("Server closed the connection.")
        break

    time.sleep(0.5)

client_socket.close()
print("\nConnection closed.")
```

2. Testing and Result

The test was conducted by running server.py first, followed by client.py in two separate terminals

- Run python server.py in **Terminal 1**. The server will display a message that it is waiting for a connection.
- Run python client.py in **Terminal 2**. The client will connect to the server.
- Run python client.py in **Terminal 3** to add a second client. The client will connect to the server.
- In the client's terminal, **enter a name** when prompted.
- **Engage in a conversation**. The client sends a message, and then the server replies.

3. Sample Session Results:



The test results show that the program runs according to the scenario. The server successfully accepted connections, exchanged messages, and handled the connection closure before being ready for a new client. Additional clients can also connect sequentially and were successful in connecting and communicating according to the designed flow.