System and code structure:

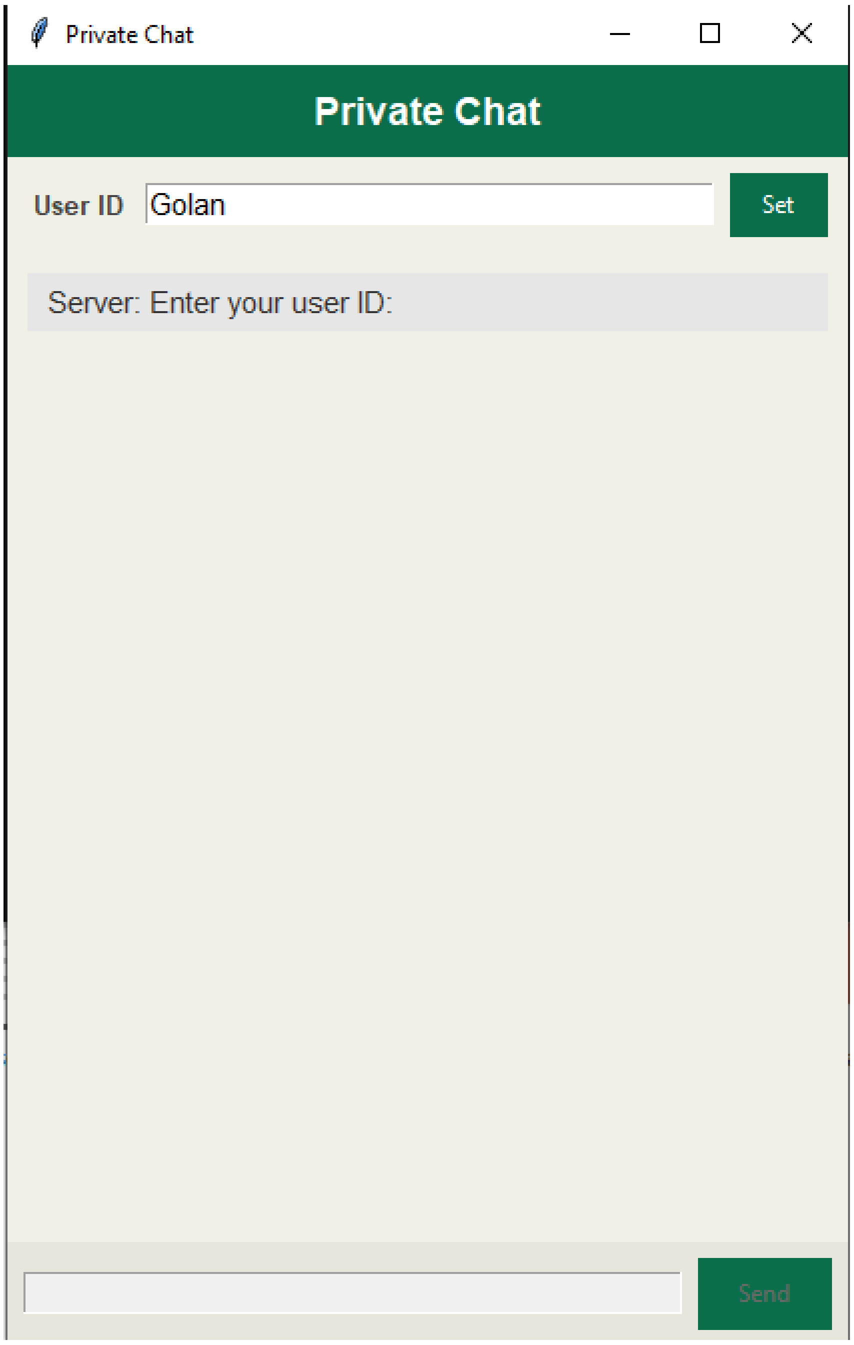
The server uses threads to handle multiple clients at the same time.
Each client has one thread that reads its messages and reacts based on the current mode (menu, requesting, confirming, or chatting). The server keeps track of all users and who they are chatting with. When two users accept a chat, the server forwards messages between them until one ends the chat. The client program simply sends user input to the server and prints whatever comes back.

Overall, the system is a simple multi-user text chat built on top of Python sockets, using a basic state-machine to manage interactions.

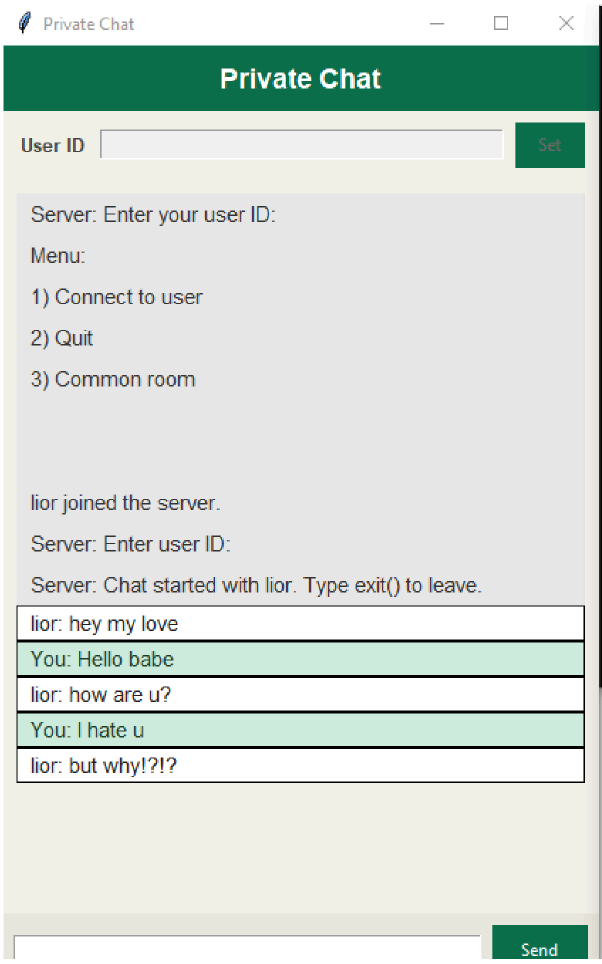
Installation and running instructions:

1) Make sure you have python installed.
2) Run server.py.
3) Run multiple instances of client.py (up to 5).
4) Each client is required to give an id
5) Each client is presented with a menu. “1” for requesting connection, “2” for exit and “3” for the common room.
6) When requesting connection, enter the id of the desired client to converse with.
7) When entering the desired client, and the client is free, the client will receive a notification for the request and he can accept with “y” or refuse with “n”.
8) If the client accepts, a chat will commence and each message from either client will be sent to the other client.
9) To close the chat and return to the menu, one of the clients needs to type “exit()”.
10) In the common room, more than 2 clients can enter, and every message will be broadcasted to all the clients in the common room.

I/O examples:

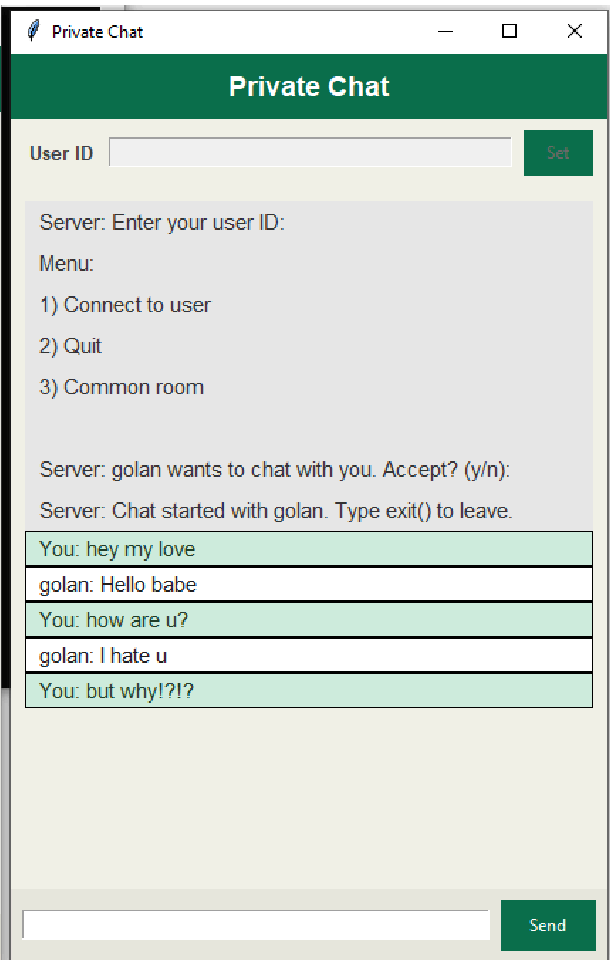


Server: Enter your user ID:



```
Server: Enter your user ID:
Menu:
1) Connect to user
2) Quit
3) Common room


lior joined the server.
Server: Enter user ID:
Server: Chat started with lior. Type exit() to leave.
lior: hey my love
You: Hello babe
lior: how are u?
You: I hate u
lior: but why!?!?
```



```
Server: Enter your user ID:
Menu:
1) Connect to user
2) Quit
3) Common room

Server: golan wants to chat with you. Accept? (y/n):
Server: Chat started with golan. Type exit() to leave.
You: hey my love
golan: Hello babe
You: how are u?
golan: I hate u
You: but why!?!?
```

Traffic analysis:

1. Application Layer Behavior

At the application layer, the communication consists of simple text messages exchanged between the chat server and clients. These include:

- User ID submissions
- Menu options (e.g., "1" or "2")
- Chat requests ("y" / "n")
- Actual chat messages
- Exit commands like "exit()"

Since the program uses Python sockets directly, no higher-level application protocol (like HTTP or FTP) is present.
Instead, the program defines its own custom text-based protocol where each line represents a command or response.

---

2. Transport Layer – TCP

All communication in the capture uses TCP, which provides a reliable byte stream.

2.1 TCP Connection Establishment

The capture begins with the classical three-way handshake:

1. Client → Server: SYN
2. Server → Client: SYN, ACK
3. Client → Server: ACK

Example from the capture (summarized):

Client port 53278 → Server port 65432: SYN
Server port 65432 → Client port 53278: SYN, ACK
Client port 53278 → Server port 65432: ACK

This establishes a reliable channel before any data is exchanged.

2.2 Data Transfer Phase

The bulk of the capture consists of TCP segments carrying the chat messages.
These usually appear as:

[PSH, ACK] Len=X

Where:

- PSH tells the receiver to push data to the application immediately
- ACK confirms previous data
- Len=X shows payload length (text message length)

The program sends many small payloads, since every user input or server prompt is transmitted in its own segment.

2.3 A Concrete Example Packet

Below is a representative packet from your capture:

65432 → 53278 [PSH, ACK] Len=20

This corresponds to:

- Src port: 65432 (server)
- Dst port: 53278 (client)

- Flags: PSH + ACK
- Payload: 20 bytes (likely a menu or text prompt)

This is a server-generated message sent directly after the handshake. It represents the server asking the client for input (for example, “Enter your user ID” or “Menu: 1) Connect 2) Quit”).

This packet is a typical example of the program’s application traffic, where each prompt is transmitted as a small TCP segment.

2.4 TCP Acknowledgments

Between data segments, many packets contain only:

```
[ACK] Len=0
```

These are acknowledgement packets confirming successful receipt of segments. This is expected in a conversational text-based interaction.

2.5 Connection Termination

Near the end of the capture you can see:

```
[RST, ACK]
```

This indicates that either the client or the server closed the socket abruptly (common when a Python script ends without calling `close()`), causing TCP to send a reset.

---

3. Network Layer – IPv4

Even though all communication happens on localhost (127.0.0.1), the OS still constructs full IPv4 packets.

Characteristics:

- Source IP: 127.0.0.1

- Destination IP: 127.0.0.1
- Protocol field: 6 (TCP)
- TTL: Typically 64
- Total length: Header + TCP header + payload

Even though these packets never leave the machine, Wireshark shows them exactly as any other IPv4 packets, only traveling through the loopback interface.

---

4. Summary

The captured traffic belongs to a TCP-based chat application built with Python sockets. The application layer uses simple text messages without a formal protocol. TCP handles connection setup, reliable delivery, and acknowledgments. IPv4 encapsulates all packets locally on the loopback interface.

A representative packet (server → client, PSH/ACK, length 20) illustrates how text messages are carried in TCP segments throughout the session.

Description of AI Usage

1. Purpose of Using AI

AI was used to support the development and debugging of a TCP client–server communication system in Python.
The main goals were:

- Understanding how to correctly implement multithreaded handling of multiple clients.
- Fixing issues related to thread concurrency and message routing.
- Reviewing and improving the overall structure of the server and client code.
- Receiving examples of simple TCP client–server implementations for reference.
- Getting an explanation of socket programming concepts and best practices.

---

2. Example Prompts (in English)

Below are invented prompts representing the type of interaction we used:

- **“Give me a simple TCP server and client in Python.”**
- **“Make the server support multiple users using threads.”**
- **“We have a threading issue in our chat server. The receiving user stays stuck in the menu thread. How can we fix this?”**
- **“Show a clean server–client structure with safe threading and correct message handling.”**
- **“Explain how to use sockets properly in Python for TCP communication.”**

These prompts helped refine the program logic and improve concurrency control.

Data Structure Selection and Justification

To solve the problem, the server uses dictionary-based data structures (hash maps) to manage connected clients and their states.
Specifically, the following dictionaries are used:

- A dictionary that maps user IDs to socket connections.
- A dictionary that maps user IDs to their current state.
- A dictionary that maps user IDs to their current chat partner.

Why This Data Structure Was Chosen

Dictionaries were chosen because they allow fast and direct access to client-related information using the user ID as a key.
This is essential in a multi-client chat system where the server frequently needs to check user availability, retrieve sockets, and update user states efficiently.

Advantages of the Chosen Data Structure

- Efficient constant-time operations (O(1)) for lookups and updates.
- Clear and readable mapping between users and their data.
- Scalable design that supports multiple users.
- Flexible structure that can be easily extended.

Conclusion

Using dictionaries provides an efficient, readable, and scalable solution for managing multiple clients in a TCP-based chat server.