DISYS Assignment-4

Source code: https://github.com/JaakkoLipp/Distributed-systems-chatapp/

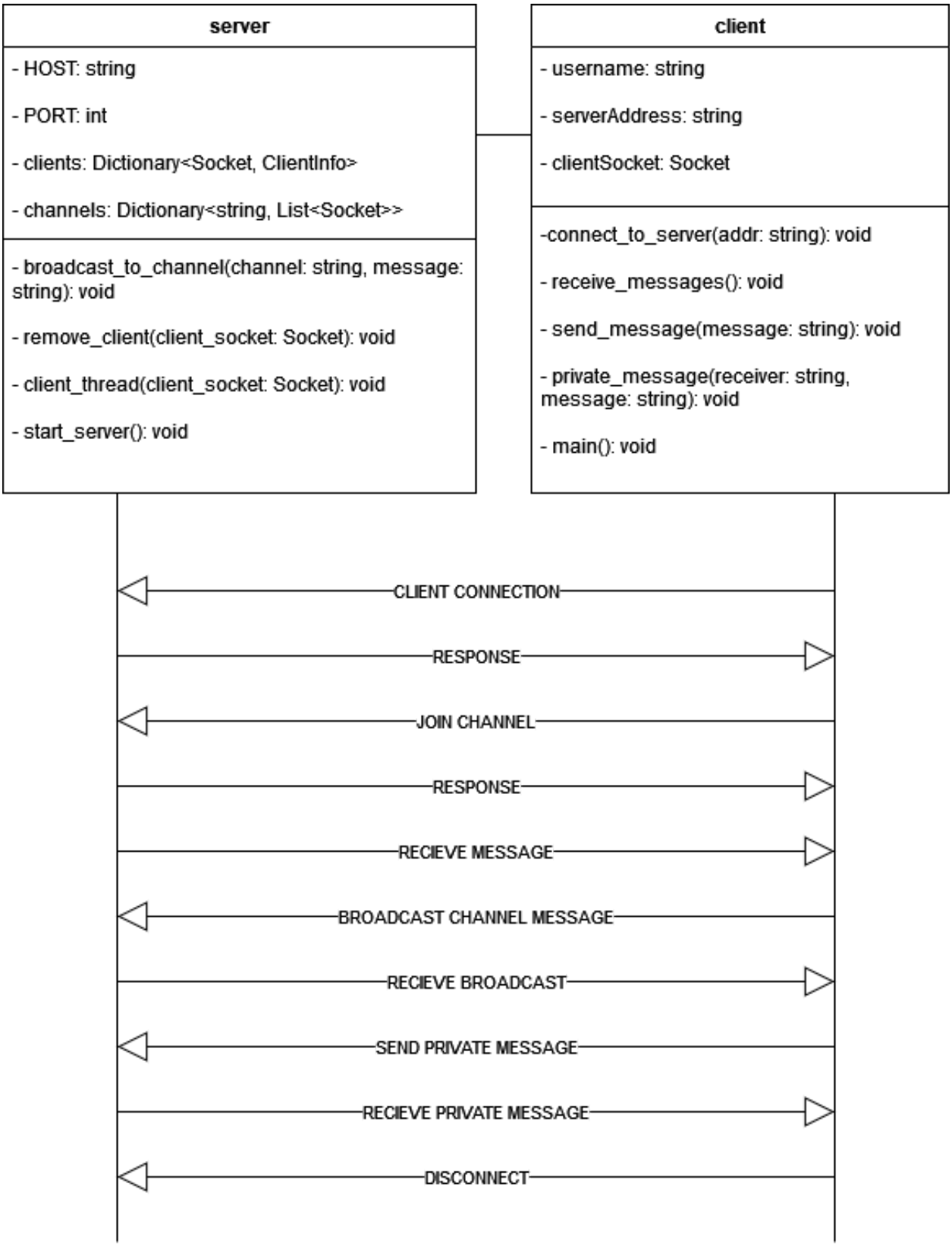
Architectural details of the system

The multi-user chat system is structured around a client-server model with TCP communication.

client-server model allows the system to manage communications between multiple clients by centralizing the message routing and connection handling within a remote server. Client CLI-applications can connect to the server using its IP address at port 8080, set up their unique nicknames, and engage in text-based conversations either publicly in channels or privately. This architecture ensures that the system can support multiple concurrent users, each interacting within the system independently, enhancing the user experience by providing a seamless and efficient communication platform.

The server side is responsible for maintaining a directory of active connections, available channels, and managing the flow of messages across the network. The use of threads allows the server to handle multiple client connections simultaneously without significant delays. This architecture is used because it is simple to implement and it has relatively good scalability, reliability, and fault tolerance.

Distributed chat system UML + Sequence diagram:


server
- HOST: string
- PORT: int
- clients: Dictionary<Socket, ClientInfo>
- channels: Dictionary<string, List<Socket>>
- broadcast_to_channel(channel: string, message: string): void
- remove_client(client_socket: Socket): void
- client_thread(client_socket: Socket): void
- start_server(): void
client
- username: string
- serverAddress: string
- clientSocket: Socket
-connect_to_server(addr: string): void
- receive_messages(): void
- send_message(message: string): void
- private_message(receiver: string, message: string): void
- main(): void
CLIENT CONNECTION
RESPONSE
JOIN CHANNEL
RESPONSE
RECIEVE MESSAGE
BROADCAST CHANNEL MESSAGE
RECIEVE BROADCAST
SEND PRIVATE MESSAGE
RECIEVE PRIVATE MESSAGE
DISCONNECT

▼ Sequence chart:

Client connection:

1. Client initiates connect_to_server with server address.
2. Server accepts connection and spawns a client_thread.
3. Client sends username.
4. Server acknowledges and welcomes the client.

Joining a channel:

1. Client sends join channel command.
2. Server updates the channel list and notifies the channel members (including current user).

Sending a channel message:

1. Client sends a message after joining a channel.
2. Server receives the message and broadcasts it to the appropriate channel members

Receiving a channel message:

1. Server broadcasts the message.
2. Client receives and displays the message.

Sending a private message:

1. Client sends message in form "/msg:{receiver}:{message}".
2. Server receives the message and forwards the message to 1 user with matching username in clients dictionary.

Receiving a private message:

1. Server forwards the message to 1 user.
2. Client receives and displays the message.
3. Reply possible if chat is open with other person.

Code is explained in the demonstration video.

Disclaimers:

Generative AI used as a teaching tool, generate examples, improving code and to improve documentation