Readme:

Multiple Client-Server Chat system using Socket Programming

The system allows multiple clients to connect to a central server and exchange messages in a **group chat** or engage in **private conversations**.

File Description:

Client.py: This file contains the implementation of the client-side application. It provides a graphical user interface (GUI) using **Tkinter** for users to **login**, view the **chat interface**, send messages to all users, and initiate **private conversations** with specific users.

Server.py: This file contains the implementation of the server-side application. It listens for incoming connections from clients, maintains a list of connected clients, and facilitates **message broadcasting** among clients. Additionally, it provides functionality to **send** updated **user lists** to all clients.

Features:

- **Login System**: Users can enter their names to log in to the chat system.
- **Group Chat**: Users can send messages that are broadcasted to all connected clients.

- **Private Messaging**: Users can initiate private conversations with specific users by double-clicking on their names in the user list.
- **Real-time Updates**: The user list is dynamically updated to reflect the current online users.
- **Error Handling**: The system provides error handling for failed connections and communication errors.

<u>Working</u>:

Client Side:

1. Initialization:
    - The client initializes a socket object (self.client) using the socket module to establish a connection with the server.
    - It also initializes variables like name to store the user's name, connected to track the connection status, and private_chat_windows to manage private chat windows.

2. GUI Setup:
    - The setup_login_gui() method creates a login window using Tkinter, where users can enter their name and click the login button.
    - Upon successful login, the setup_chat_gui() method creates the main chat window, including chat history, message entry field, user list, etc.

3. Connection to Server:

- The connect_to_server() method attempts to connect the client to the specified server address (SERVER_HOST and SERVER_PORT).
- If the connection is successful, the client sends the user's name to the server for authentication and starts a separate thread (receive_messages()) to handle incoming messages asynchronously.

4. Sending and Receiving Messages:
    - The client sends messages to the server using the send_message() method, which encodes the message into bytes and sends it over the socket.
    - The receive_messages() method continuously listens for incoming messages from the server. It decodes received messages, updates the GUI accordingly, and handles different types of messages (public messages, user list updates, private messages).

5. Private Chat:
    - Users can initiate private chat sessions by double-clicking on a user's name in the user list.
    - The client maintains a dictionary (private_chat_windows) to manage private chat windows and associated components (chat history, message entry).
    - Private messages are prefixed with a special tag ([PRIVATE MESSAGE]) before being sent to the server for proper routing.

Server-side:

1.Initialization:

- The server initializes a socket object (server_socket) to listen for incoming connections from clients.
- It maintains data structures to store connected clients, their usernames, and manages client connections.

2. Connection Handling:

- Upon accepting a new client connection, the server creates a separate thread (client_thread) to handle communication with that client.
- The client's username is received and validated by the server.
- User authentication ensures that each client has a unique username.

3. Broadcasting Messages:

- The server receives messages from clients and broadcasts them to all connected clients, ensuring that each message is prefixed with the sender's username.
- User lists are periodically updated and broadcasted to all clients to reflect changes in online users.

4. Private Messaging:

- Private messages are routed through the server, which ensures that they are only delivered to the intended recipient.
- The server facilitates private communication by identifying the recipient and delivering the message accordingly.

5. Error Handling:

- The server handles various errors, such as client disconnection, connection timeouts, and exceptions during message handling, to ensure robustness and reliability.

6. Multithreading:
    - Multithreading is employed to handle multiple client connections concurrently, allowing the server to serve multiple clients simultaneously without blocking.

7. Scalability and Performance:
    - The server architecture is designed to be scalable and performant, capable of handling a large number of concurrent connections and efficiently managing communication between clients.

Conclusion:

- The client and server components work together to enable real-time text-based communication between multiple users.
- The client provides a user-friendly interface for interacting with the chat system, while the server manages the underlying communication infrastructure and facilitates message routing between clients.

For the Demo, refer to the Report.
FortheDemovideo:
https://drive.google.com/file/d/1dKGGwPipN3cnnVGk64uOzRo0rNplaQN-/view?usp=drive_link

Contributor:

Dhruva Kumar Kaushal (B22AI017)
Saurav Soni (B22AI035)