CS440 Computer Networks
2024-2025 Spring Semester
Socket Programming Project
“A Multi-user Chat Application”

Introduction

This report provides a detailed breakdown of a chat application built using Python's **Streamlit**, **Socket Programming**, and **Threading**. The application is designed to allow users to connect to a server, authenticate via login/registration, and engage in real-time messaging with support for general, private, and group chats. Additionally, it features message queue handling, client-server communication, and session management.

Technologies Used

1. **Streamlit**: For building the user interface (UI) of the chat application. Streamlit is a Python library for creating interactive web apps with minimal code.

2. **Socket Programming**: For communication between the client and server. The socket allows the exchange of data between the chat client and the server over a network.

3. **Threading**: For handling the reception of messages in the background without blocking the user interface.

4. **Queue**: To manage and store messages received from the server, ensuring they are processed and displayed efficiently in the UI.

5. **Datetime**: For timestamping messages and managing session-related timestamps.

6. **JSON**: For encoding and decoding messages sent between the client and server.

7. **Streamlit AutoRefresh**: To automatically refresh the UI periodically, allowing real-time updates of chat messages.

Key Components and Their Functionality

1. Session State Management

Streamlit's **session state** is used to persist information across interactions within the app. The following session state variables are defined:

- **username**: Stores the username of the currently authenticated user.
- **client_socket**: Holds the socket object used for communication with the server.
- **messages**: A list of chat messages that are displayed to the user.
- **connected**: A boolean flag indicating whether the client is connected to the server.
- **auth_status**: Tracks the authentication status (whether the login or registration was successful).
- **message_queue**: A queue that stores incoming messages for processing.
- **receive_thread**: A thread responsible for receiving messages from the server.
- **last_refresh**: Stores the timestamp of the last UI refresh.

These session variables are initialized to default values when the app starts and are updated as the user interacts with the application.

2. Server Connection

The `connect_to_server()` function attempts to establish a connection to the server by creating a socket and connecting to a predefined IP and port (`localhost:5555`). If the connection is successful, the client enters a blocking mode, and a confirmation message is awaited from the server.

Key steps include:

- **Socket Creation**: A TCP socket is created using `socket.socket()`.
- **Timeout Management**: A timeout of 5 seconds is set for the initial connection attempt.

- **Connection Confirmation**: The server sends a `CONNECTED` message upon successful connection.

If an error occurs during the connection process, an appropriate error message is shown in the UI.

3. Authentication Process

The `handle_authentication()` function handles both login and registration based on whether the user has an account. The following process occurs during authentication:

- **Authentication Type**: The user is asked if they have an existing account (`Yes` or `No`). Based on this, an authentication type (`yes` or `no`) is sent to the server.

- **Username & Password Prompts**: The client waits for the server's prompts for username and password. These are sent to the server after receiving the prompts.

- **Authentication Result**: After the username and password are sent, the server responds with one of several possible outcomes:

    - `LOGIN_SUCCESS`: The user is logged in successfully.

    - `REGISTER_SUCCESS`: The user is registered successfully.

    - `USERNAME_EXISTS`: The chosen username already exists.

    - `INVALID_CREDENTIALS`: The provided credentials are incorrect.

Upon successful authentication (login or registration), the `receive_messages` thread is started, allowing the client to begin receiving messages from the server.

4. Receiving Messages

The `receive_messages()` function is designed to run in a separate thread to continuously receive messages from the server. It uses the socket's

`recv()` function to read data from the server and then processes the received message. Control messages such as "USERNAME", "PASSWORD", etc., are ignored, while other messages are added to the message queue.

The `message_queue` is used to store incoming messages, and these messages are displayed in the chat UI in real-time.

5. Message Sending

Users can send different types of messages through the application:

- **General Chat**: A regular message sent to all users in the chat.
- **Private Message**: A message sent to a specific user.
- **Group Message**: A message sent to a specific group of users.

For each type of message, the following steps occur:

1. **Message Type Selection**: The user selects the type of message they wish to send.
2. **Private/Group Message Details**: For private or group messages, the recipient or group name is provided.
3. **Message Sending**: Upon clicking the "Send" button, the message is sent to the server using the socket in JSON format.

In the case of private or group messages, the message includes specific instructions (e.g., `/msg <recipient>` for private messages or `/group_msg <group_name> <message>` for group messages).

6. UI Layout

The application is designed with the following sections:

- **Authentication**: The user is prompted to log in or register if not already authenticated. A simple form collects the username and password.

- **Chat Interface**: Once authenticated, the user is taken to the chat interface, where:
    - A list of received messages is displayed.
    - A form is provided for sending messages.
    - Users can select the message type and provide additional information (e.g., recipient or group name).
- **Message Display**: Messages are displayed with different styles based on their type:
    - **Private Messages**: Displayed in purple.
    - **Group Messages**: Displayed in blue.
    - **General Messages**: Displayed in green.

Threading and Auto Refresh

To ensure that the chat application remains responsive, **threading** is used to receive messages from the server in the background. This allows the UI to stay interactive while continuously fetching new messages.

Additionally, **Streamlit AutoRefresh** is employed to periodically refresh the UI every 2 seconds (`interval=2000`). This ensures that the messages are updated in real-time without requiring the user to manually refresh the page.

Error Handling

Several error conditions are handled throughout the application:

- **Connection Errors**: If the connection to the server fails or times out, the user is informed via error messages.
- **Authentication Errors**: If the authentication fails (e.g., incorrect credentials or existing username), the user is notified.
- **Message Sending Failures**: Errors during the message-sending process are handled gracefully.

Conclusion

This chat application provides a simple yet powerful real-time messaging platform using **Streamlit**, **Socket Programming**, and **Threading**. It supports various messaging features, including general chat, private messages, and group chat, along with robust session management and authentication. The use of threading ensures that the user interface remains responsive while continuously receiving messages from the server. Additionally, real-time message updates are facilitated through the **Streamlit AutoRefresh** feature. This application is a solid foundation for building more advanced chat systems with additional features such as multimedia messaging, user presence, and more.