**EECE 350 PROJECT:**

**YALLACHAT**

**Report:**

*25%* Bassam Tarshishi **202302507**

*25%* Mahmoud Chaer **202302465**

*25%* Amir Basha **202205806**

*25%* Omar Chehab **202301375**

**System Architecture:**

**Client**: Initiates communication by sending requests to the server. It creates a socket and connects to the server by specifying the server's IP address and port number. The client sends data, waits for a response, and processes the received data.

**Server**: Waits for incoming connections from clients. It listens on a specified port, accepts incoming connections, and processes received requests. After processing, it sends a response back to the client.

**Protocol Used:**

**TCP (Transmission Control Protocol)** is commonly used between the communicating entities in a client-server architecture. It ensures reliable, ordered, and error-checked delivery of a stream of bytes between applications running on hosts communicating via an IP network.

**Mode of Communication**: The communication generally follows the client-server model where the server provides services that the client requests. A hybrid mode could involve peer-to-peer (P2P) elements, but typically in simple client-server applications, it's mainly the server responding to the client's requests.

This architecture ensures that data flow is managed smoothly and efficiently, with the server handling potentially multiple client requests simultaneously, depending on its implementation (e.g., multi-threaded server).

**PART 1:**

| Feature | Description | Status |
| --- | --- | --- |
| Client Connection Setup | Establishing connection to the server. | Implemented |
| Data Transmission | Sending and receiving data between client and server. | Implemented |
| Concurrency Handling | Server can handle multiple clients simultaneously. | Implemented |
| Data Integrity Checks | Ensuring data sent is the data received (no corruption). | Implemented |
| User Authentication | The server performs authentication by verifying the username and password | Implemented |
| Error Handling | Managing connection losses, timeouts, etc. | Implemented |
| Logging | Keeping records of server or client activity. | Implemented |
| Graphical User Interface (GUI) | Graphical interface for client interactions. | Implemented |
| Database Integration | Backend database for storing data. | Implemented |
| Session Management | Handling user sessions securely and efficiently. | Implemented |
| List Of Chats | Showing a list of chats the client made with friends | Implemented |
| Add Friends/View Friends Status | The user can add friends and view whether they are online or offline | Implemented |
| Start Chat | The user can initiate communication with a selected friend and exchange text messages through the server. | Implemented |
| Offline Receiver | If a user sends messages to an offline friend, the server keeps these messages in its repository and delivers them immediately once the friend goes online | Implemented |

**Description of the implementation of the different functionalities:**

1. **Login**

When the user enters their username and password, a login request is sent to the server. The server then proceeds to check if the provided input data is not empty and exists in the database. If the credentials are found in the database, the user successfully logs into the app and if not, an error message will be displayed.

1. **Signup**

The user is presented with a signup form where he is required to enter their details. These details are then sent to the server in order to be validated (check if empty, email in correct format…). After the server’s validation, the details are stored in the database, the user is now able to login using his new credentials.

1. **View Friends' Status**

The server maintains a list of user statuses that is updated with each login/logout with the help of a function named set keep alive(dictionary that keeps count of the online users). The client requests this list and stays updated displaying the current status of each friend.

1. **View Friend Requests**

The server stores and manages incoming and outgoing friend requests for each user. The client fetches this list from the server and displays it, allowing users to accept or reject requests.

1. **Send Friend Requests**

The client sends a request to the server with the target user's identifier. The server then updates its records to show a pending request between the two users.

1. **Chatting**

In the chatting part of the system, clients and servers communicate via socket-based connections, enabling real-time message exchange. The server listens for incoming messages and processes them according to the client's instructions, such as sending a direct message or retrieving chat history. Upon receiving a message, the server looks up the intended recipient in its list of online users and forwards the message to the correct connection. If the recipient is offline, the server stores the message in a SQLite database to deliver later. On the client side, separate listener threads maintain the connection to the server, receiving and displaying new messages in a PyQt5-based graphical interface. The system supports additional chat-related functions, such as sending group messages and handling file transfers (partially).

1. **GUI**

The GUI starts with a window. This is the container where all graphical elements like buttons, text fields, and images live. The window needs to handle user interactions such as resizing. Each widget in a GUI has event listeners that respond to these actions, making the GUI interactive.Layout Management involves the arrangement of widgets within a window. Layout managers help organize widgets, automatically adjusting their positions and sizes according to the overall window size.Rendering is the process of drawing the GUI on the screen. It involves computing how each widget should appear based on its properties and then displaying it. Rendering ensures that when you update a widget, its appearance changes immediately on the screen.Threading is used to keep the interface responsive. Long operations are run on separate threads, allowing the main GUI thread to remain responsive to user actions.

**PART 2:**

| Feature | Description | Status |
| --- | --- | --- |
| Groups | Users can create group of friends and communicate with them | Implemented |
| Sending Files | Did not fully work but provided the code that includes them. | 1/2 Implemented |
| Peer-to-Peer Architecture | Did not fully work but provided the code that includes them. | 1/2 Implemented |
| Chat History | Display all the chats between two users | Implemented |
| Searching In Chats | Searching efficiently for a specific word | Implemented |
| Notifications | If received a message while offline, a notification pops up once the user logs into his account | Implemented |

**Description of the implementation of the different functionalities:**

**1.Groups (option a)**

We created the groups by using tables in our database as follows :

Groups Table: This table is responsible for storing basic information about each group. Each group has a unique ID (id), a name (name), and a record of who created the group (created\_by). The id field is set to auto-increment, meaning it automatically assigns a new, unique number to each group when it’s created.

Group Members Table: This table links users to the groups they belong to. It records which user (username) is in which group (group\_id). The group\_id in this table is a reference to the id in the Groups table, establishing a relationship between a user and a group. The combination of group\_id and username is set to be unique, ensuring that each user is listed only once per group.

Group Messages Table: This table stores messages sent within each group. Each message has its own unique ID (id), is linked to a group via group\_id (which references id in the Groups table), and includes the sender’s username (sender), the message text (message), and a timestamp (timestamp) indicating when the message was sent. The delivered field tracks whether the message has been delivered (1) or not (0).

Together these tables allow for commands such as listing the groups for each user, adding users to group, and fetching undelivered messages and sending messages to each other. Please note that in order to fetch undelivered messages when you select a chat from the list of groups, it displays the chat history (sometimes needs multiple refreshes) . Also note that according to what the professor instructed us to do , I will be including separate client and server codes , Server\_groups and client\_groups . These codes are an older version with a working groups functionality.

**2. Chat searching**

We thought the user needed to search in his specific chat with the other user. So we made all chat histories available. We did not understand exactly why the task demanded that one user can see the chat history of another user with all his chatting peers.

Instead, we implemented a “search by topic” feature. This feature allows users to search for a specific word/topic within a chat and highlights the word. This would help users search for important information in extensive chat histories.

**Description of an interesting networking project:**

We think an interesting networking project would be an online chess game. We can also add ranks among friends, for example everyone starts with a base rating of 400 once they sign up and put something like for each game won +10, each game lost -10, maybe a forfeit is -12… So, the players connect in a client server manner or peer to peer manner both could be implemented.

We think this would be a really nice networking project, and if it was made properly, maybe we could also let our chess club use it at the university.