CalPolyPomona
College of Science
Computer Science

Messaging Application using Self-Developed Application Layer Protocol

Fireflare-Flare & LITP

Noah King (ncking@cpp.edu)
Myo Aung (myoaung@cpp.edu)
Sungeun Kim (sungeunkim@cpp.edu)
Tony Alhusari (tsalhusari@cpp.edu)
Daniel Appel (dvappel@cpp.edu)

https://github.com/Maung945/socket-programming-chat-app

Abdelfattah Amamra
CS 3800
Spring 2024

Department of Computer Science

June 19, 2024

Abstract

This paper details the specifications pertaining to the proposed “Lit Internet Transmission Protocol”, hereby referred to in short as “LitP”. The Lit Internet Transmission Protocol is a messaging protocol intended to facilitate secure, reliable and simple multimedia communication (supporting text and image transmission). This particular general details of the LitP protocol (packet structure, and server-client handshake) and it’s utilization of the Kyber Post-Quantum Key-Encapsulation Mechanism alongside AES encryption in CFB mode.

Furthermore, this paper highlights the implementation details of a reference client and server utilizing the LitP protocol “Firefly-Flare”. There are different section pertaining to UI/UX, data organization, program architecture, and control flow structure.

Contents

1 Message Format **1**

1.1 Packet Structure (Header + Payload) 1

1.2 Contents of "FLAG" Field 2

1.3 LitP Packet Example Class 2

2 Data Processing Control Flow **3**

2.1 Key-Exchange Implementation 3

2.2 Sending a Message to Connected Clients 4

3 Encryption **5**

3.1 Hybrid Cryptosystem 5

3.2 Key-Encapsulation Mechanism 5

3.3 Symmetric Encryption 5

4 User Interface and Experience **6**

4.1 GUI with Tkinter 6

4.2 Harmonizing Functionality 7

4.3 Emoji Implementation 7

4.4 Duplicate Username Handling 9

4.5 Git and GitHub Collaboration 10

5 Other Features **11**

5.1 Chat record saving, Reading about app 11

5.2 Enhanced Chat Efficiency: Introducing Shortcut Integration for a Simplified Experience 11

6 Future Implementations **12**

6.1 File Sharing Capabilities 12

6.2 Password to Enter Chatroom 12

6.3 MAC Verification 12

6.4 Individual Chat (One-to-One Chat Capability) 12

6.5 Implementation Roadmap 13

1 Message Format

1.1 Packet Structure (Header + Payload)

Figure 1 below details the width of each component of a complete LitP packet (header and payload). A LitP packet has a maximum size of 2^{72} [B], which is around 4.3 [GB].

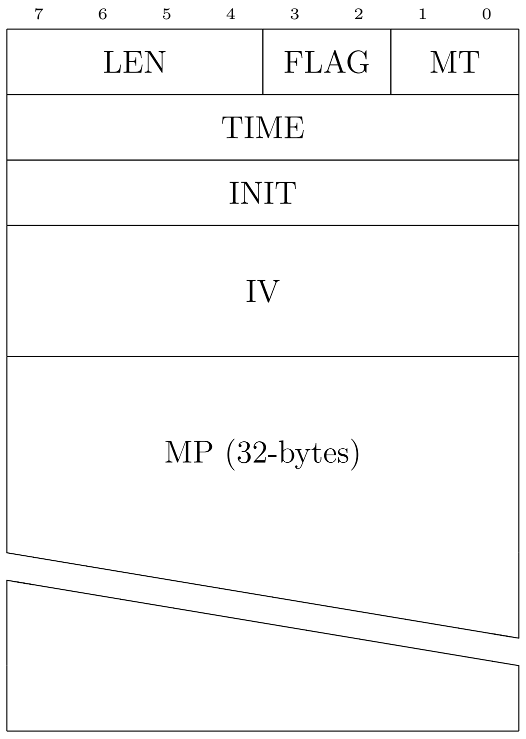


Figure 1: LitP Packet Structure

The initial element of the header, denoted as MT, specifies the category of the message (e.g., text, image, audio) utilizing two bytes. Following this, the second element, labeled FLAG, encompasses binary flags pertaining to different messaging options, options are detailed in the next section. The element TIME captures the timestamp of the message dispatch, represented by an 8-byte Unix timestamp. The Connection Exchange Number, labeled INIT, is a number used to signal the purpose of a message (whether it is involved in a key-exchange handshake or not). The identifier IV represents an initialization vector that can be used for AES encryption in CFB mode. Lastly, MP denotes the message payload, encapsulating the actual 32-byte content of the message.

The allocation of two bytes for MT and FLAG is is to accommodate future enhancements and functionalities not presently existing within the standard's subsequent iterations, this allows for flexibility in the creation of new message types.

Full Name of Packet Term	Abbreviation	Length in Bytes
Message Type	MT	2
Options Flags	FLAG	2
Packet Length in Bytes	LEN	4
UNIX Style Timestamp	TIME	8
Connection Exchange Number	INIT	8
Initialization Vector	IV	16

Table 1: LitP Packet Header Terms

1.2 Contents of "FLAG" Field

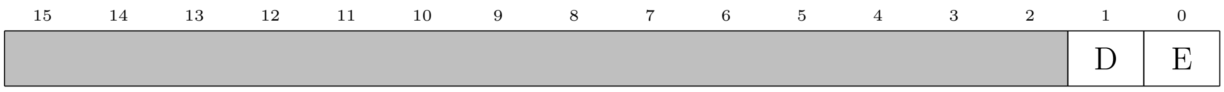
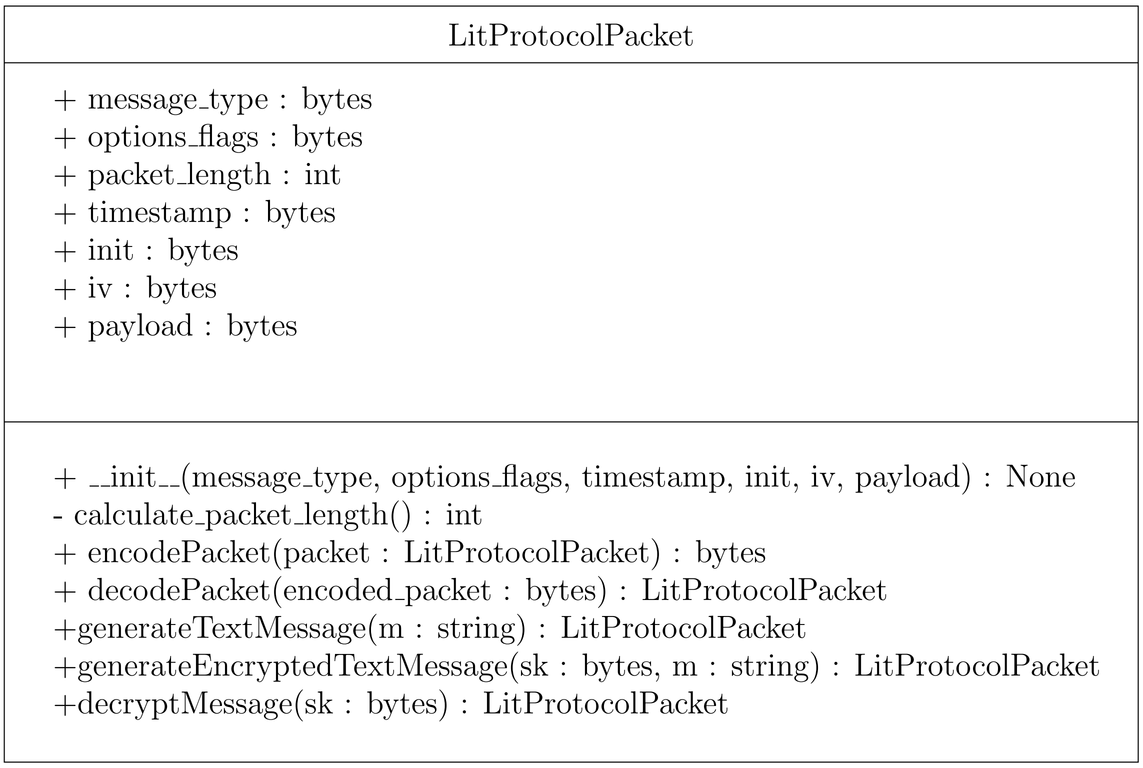


Figure 2: Contents of "FLAG" Field in LitP Packet Header

Figure 2 above depicts the contents of the 16-bit "FLAG" field shown in the previous section. As of now, most features are not in use, however the remaining 14 bits allows for flexibility in the protocol for the addition of different features. The first bit (LSB) "E", when set to 1 indicates an encrypted message, likewise, when it is 0, the message is not encrypted. The D flag indicates, that the user attempting to connect to the server has selected a username that the server has already reserved to another user. This scheme allows us to maintain end-to-end encryption while still allowing users to send commands to the server, enabling features such as moderation.

1.3 LitP Packet Example Class

An LitP packet can be represented as the class shown by the UML Class diagram shown below in Figure 3. This class is created with the intention of streamlining the organization of data prior to serialization for TCP transmission. Encryption/Decryption and Serialization/Deserialization functionality is included inside this class in the functions shown in Figure 3 below.



LitProtocolPacket

+ message_type : bytes
+ options_flags : bytes
+ packet_length : int
+ timestamp : bytes
+ init : bytes
+ iv : bytes
+ payload : bytes

+ __init__(message_type, options_flags, timestamp, init, iv, payload) : None
- calculate_packet_length() : int
+ encodePacket(packet : LitProtocolPacket) : bytes
+ decodePacket(encoded_packet : bytes) : LitProtocolPacket
+generateTextMessage(m : string) : LitProtocolPacket
+generateEncryptedTextMessage(sk : bytes, m : string) : LitProtocolPacket
+decryptMessage(sk : bytes) : LitProtocolPacket

Figure 3: LitP Packet UML Diagram (Python)

2 Data Processing Control Flow

2.1 Key-Exchange Implementation

To keep track of what stage of the key-exchange process occurs, and ensure the correct data is sent in the correct order, the packet header field "INIT" is incremented after each step of the process. The client and server can then use this metadata to indicate what computation should be performed for that particular stage of the key-exchange process. This exchange occurs after the initial TCP connection is established after the client sends a connection request to the server. After this process occurs, all the messages sent to the server will be encrypted and decrypted using symmetric AES encryption using the new shared keys. The server knows to process using symmetric encryption because INIT = 4.

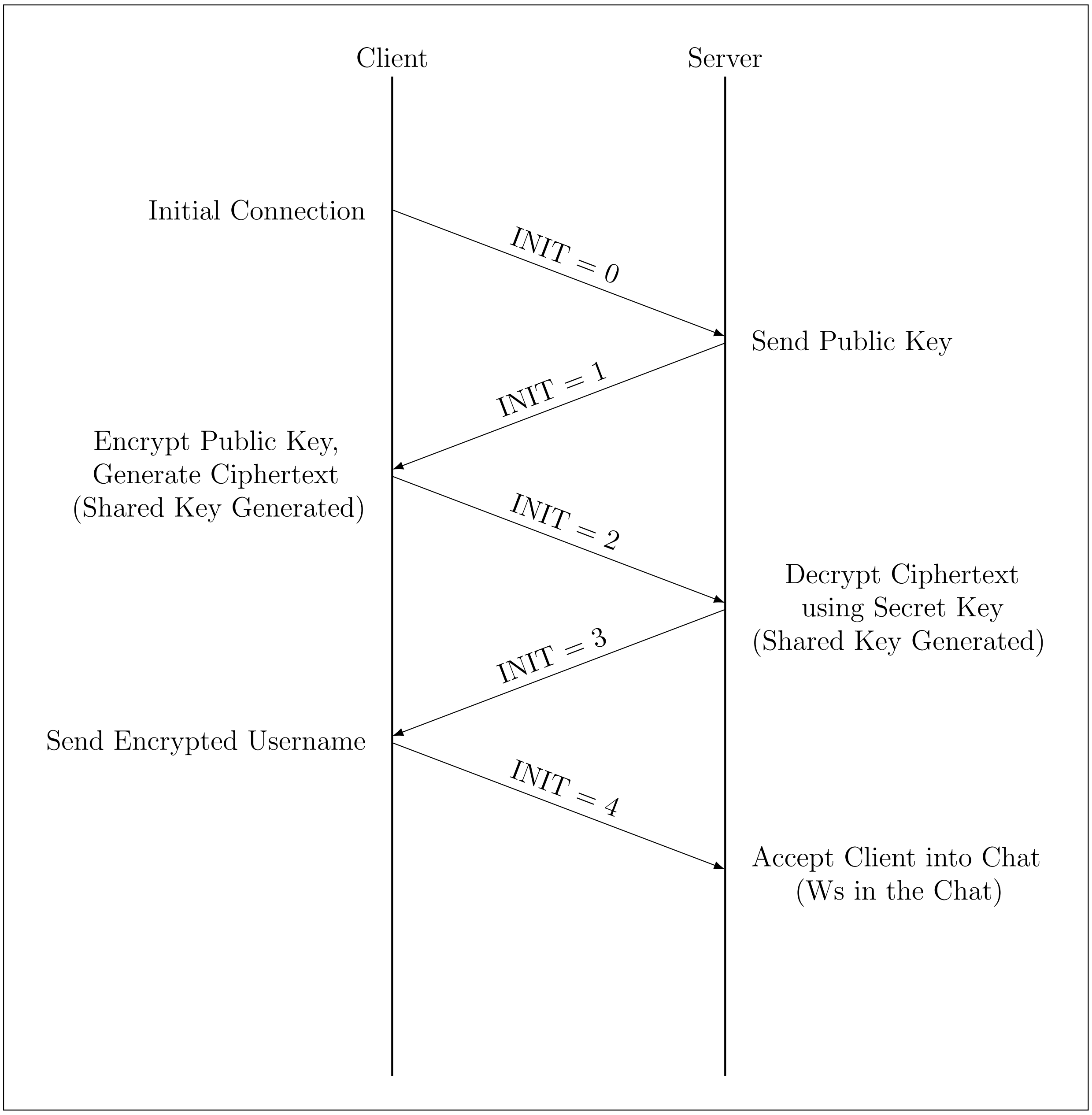

Client
Server
Initial Connection
INIT = 0
Send Public Key
INIT = 1
Encrypt Public Key, Generate Ciphertext (Shared Key Generated)
INIT = 2
Decrypt Ciphertext using Secret Key (Shared Key Generated)
INIT = 3
Send Encrypted Username
INIT = 4
Accept Client into Chat (Ws in the Chat)

Figure 4: Key-Exchange Stages

2.2 Sending a Message to Connected Clients

After key-exchange occurs, the transmission of messages happens according to Figure 5 and Figure 6 below. Figure 5 depicts the order in which an LitP packet should be processed when the header fields FLAG = 0x00_01 and INIT = 0x...04. The message m_p (a LitP packet) will first be encoded into a binary string prior to encryption. Following encryption, the serialized binary data will then be sent to the server via a transportation layer protocol such as TCP. The server will decrypt the message using the shared secret between the client sending the message (SK_0) and the IV of the message included in the LITP packet header, then it send the message to all other clients connected (SK_1, ... ,SK_n), first re-encrypting the data using the shared secret of all other server-client connections, then re-serialize the data for TCP transmission. This data is then decoded and decrypted by the recipient clients. Depending on the number encoded in the MT field in the packet header, the data could be processed by the client as a image, text, audio file, etc. after decryption.

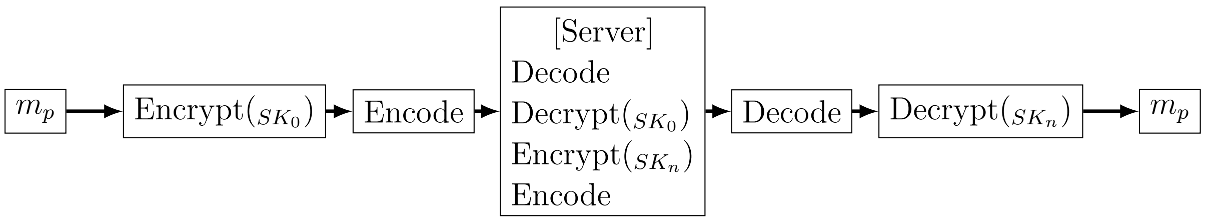


Figure 5: Data Processing Stages (FLAG = 0x00_01, INIT = 0x...04)

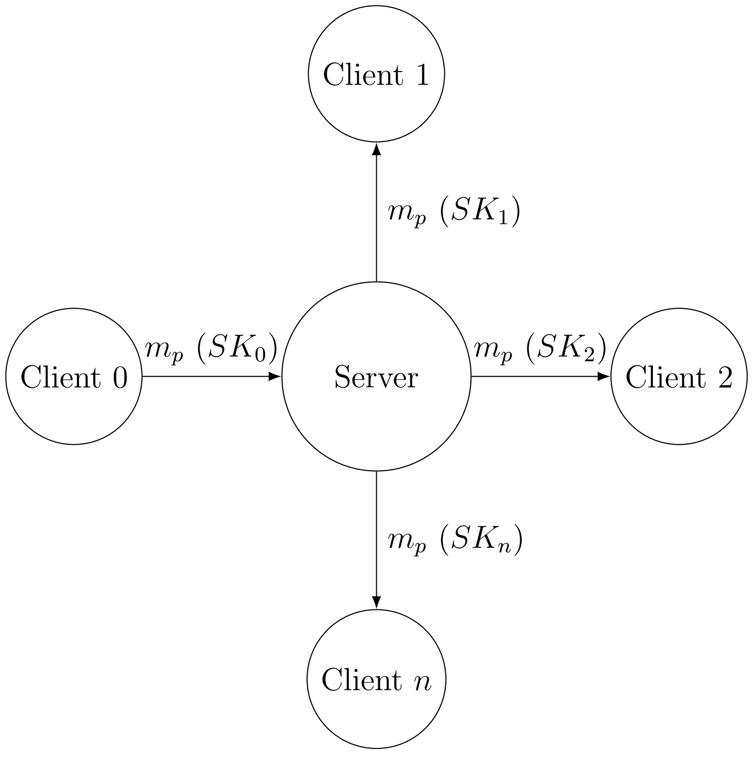


Figure 6: Server Relays (m_p) & Encrypts/Decrypts using Shared Keys per Connection

3 Encryption

3.1 Hybrid Cryptosystem

To achieve secure end-to-end encrypted conversation, the implementation of a hybrid cryptosystem is needed. A hybrid cryptosystem has two essential components: a key encapsulation mechanism (asymmetric cryptography) and a data encapsulation scheme(symmetric cryptography). Generally, symmetric encryption costs less computing resources for large amounts of data compare to asymmetric encryption. However, since both parties need to use the same key for symmetric encryption, asymmetric cryptography is needed to securely share the one key.

3.2 Key-Encapsulation Mechanism

For our project, we implemented Kyber, which is built to resist quantum attacks and is considered much more efficient than RSA which is troubled by larger key sizes. To implement Kyber, we referenced Giacomo's project "kyber-py". First, the client that wants to connect to the server asks the server for the server's public key. After receiving the server's public key, the client encrypts it using Kyber's encryption algorithms which has a random component to it. Then it will generate a cipher text and a new key (shared key for symmetric encryption). The client sends cipher text back to the server. The server can decrypt the ciphertext with its own private key to get the shared key that is identical to the one the client has. Then, the shared key will be used for symmetric encryption which will be discussed next.

3.3 Symmetric Encryption

For symmetric encryption, we implemented AES CFB mode using the Python Crypto library. CFB mode was chosen since the stream-like encryption features of CFB allow encryption of plaintext in streams of arbitrary length. CFB does not require padding which will lead to less error in coding compared to CBC which needs to deal with block size and overhead.

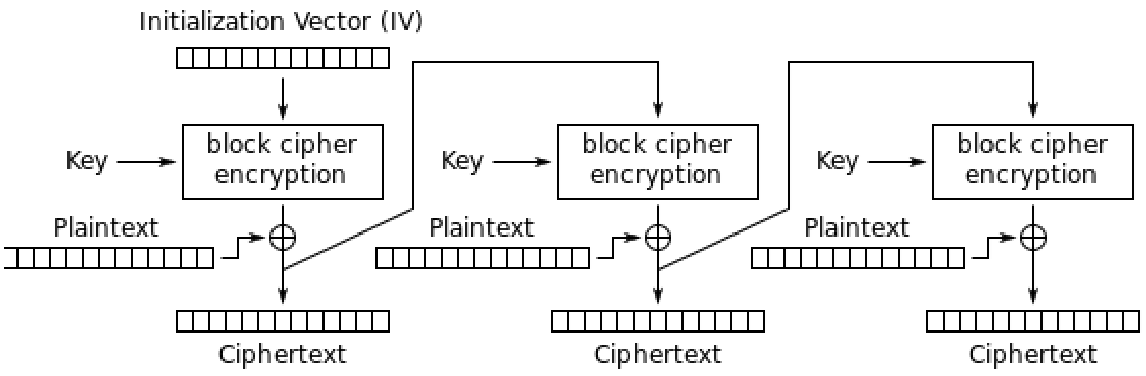


Figure 7: AES in Cipher Feedback Mode

4 User Interface and Experience

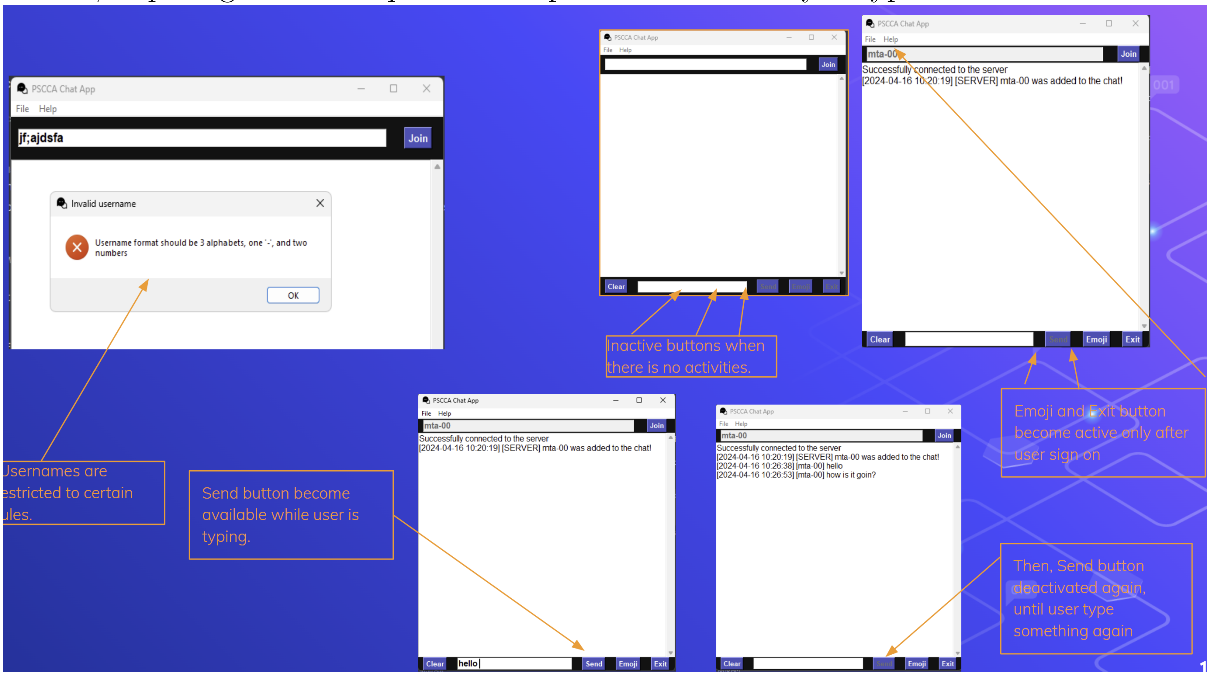
4.1 GUI with Tkinter

We took on the responsibility of crafting a visually appealing and intuitive graphical user interface (GUI) using Tkinter for our chat application. This involved careful consideration of component allocation to ensure a smooth user experience. The message text box was strategically positioned as the focal point, enabling users to engage in conversations effortlessly. Alongside, the “join,” “send,” and “exit” buttons which were placed conveniently, each serving a distinct function within the application’s workflow. An overview of UI/UX features is shown below.

- GUI development using Tkinter
  - Ensuring smooth interaction between components
- GUI Component Allocation
  - Message text box placement
  - Positioning of “join,” “send,” and “exit” buttons
- GUI Info Pop-ups
  - Info messages pop-up when hovering over buttons to ensure clarity and proper usage
- Harmonizing Functionality
  - Activation of ”Send” button based on chatbox input
  - Username field deactivation after successful joining
  - Implementation of specific username format for consistency
- Username Authentication
  - Three alphabet, hyphen, two numbers format
  - Visual coherence and consistency in usernames
- Handling Username Duplication
  - Mechanism for detecting and handling duplicate usernames
  - Ensuring clarity and avoiding confusion within the chat environment
- Git and Github
  - Setting up a dedicated repository for collaboration
  - Learning branching and merging for parallel development
  - Resolving merge conflicts
  - Utilizing pull requests and code reviews for effective collaboration
  - Ensuring clarity and avoiding confusion within the chat environment

4.2 Harmonizing Functionality

To enhance the functionality of the GUI, we implemented various features to harmonize the interaction between different components. For instance, the activation of the ”Send” button was contingent upon user input in the chat-box, promoting meaningful engagement by preventing the transmission of empty messages. Moreover, to streamline the user onboarding process, we devised a mechanism where the username field became deactivated after the user successfully joined the chat, preventing inadvertent changes during active sessions. Additionally, to maintain consistency, we enforced a specific username format, requiring users to input three alphabets followed by a hyphen and two numbers.


PSCCA Chat App
File Help
jf;ajdsfa
Join
Invalid username
OK
Clear
Send
Emoji
Exit
mta-00
hello
Usernames are restricted to certain rules.
Send button become available while user is typing.
Inactive buttons when there is no activities.
Emoji and Exit button become active only after user sign on
Then, Send button deactivated again, until user type something again

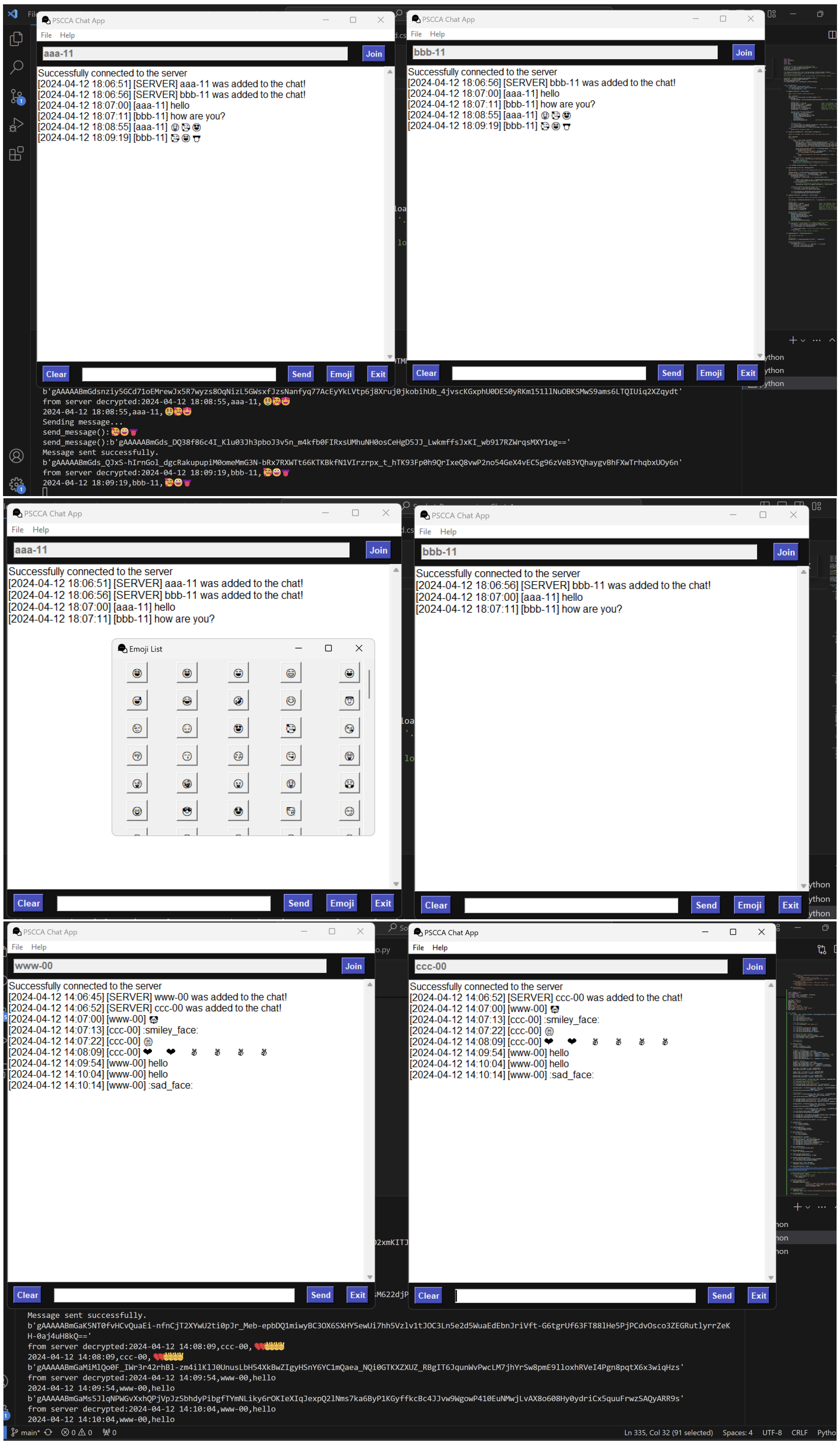
Username format should be 3 alphabets, one '-', and two numbers

```
Successfully connected to the server
[2024-04-16 10:20:19] [SERVER] mta-00 was added to the chat!
[2024-04-16 10:26:38] [mta-00] hello
[2024-04-16 10:26:53] [mta-00] how is it goin?
```

Figure 8: Username Rules and GUI Harmonizing

4.3 Emoji Implementation

In addition to our responsibilities in GUI setup with Tkinter, we were also entrusted with implementing Emoji functionality to enrich user interactions. To achieve this, we utilized the “emoji” library in conjunction with Tkinter. First, we integrated the “emoji” library into our project environment, ensuring compatibility and seamless integration with Tkinter. Next, we designed a feature that allowed users to launch an emoji window from within the chat application. This window presented users with a selection of emojis to choose from, displayed in a user-friendly interface. Upon selecting an emoji, we programmed the application to insert the chosen emoji into the message text box, enabling users to effortlessly express themselves with emoticons during conversations. To streamline the user experience, we ensured that the emoji window closed automatically upon selecting an emoji, minimizing disruption to the chat flow. By implementing Emoji functionality with the “emoji” library in Tkinter, we enhanced the richness of user interactions within our chat application, fostering a more engaging and expressive communication environment.



aaa-11

```
Successfully connected to the server
[2024-04-12 18:06:51] [SERVER] aaa-11 was added to the chat!
[2024-04-12 18:06:56] [SERVER] bbb-11 was added to the chat!
[2024-04-12 18:07:00] [aaa-11] hello
[2024-04-12 18:07:11] [bbb-11] how are you?
[2024-04-12 18:08:55] [aaa-11] 🤑🥰🤩
[2024-04-12 18:09:19] [bbb-11] 🥰😁👅
```

bbb-11

```
Successfully connected to the server
[2024-04-12 18:06:56] [SERVER] bbb-11 was added to the chat!
[2024-04-12 18:07:00] [aaa-11] hello
[2024-04-12 18:07:11] [bbb-11] how are you?
[2024-04-12 18:08:55] [aaa-11] 🤑🥰🤩
[2024-04-12 18:09:19] [bbb-11] 🥰😁👅
```

```
b'gAAAAABmGdsnziy5GCd71oEMrewJx5R7wyzs8OqNizL5GWsxfJzsNanfyq77AcEyYkLVtp6j8Xruj0jkobihUb_4jvscKGxphU0DES0yRKm151llNuOBKSMwS9ams6LTQIUiq2XZqydt'
from server decrypted:2024-04-12 18:08:55,aaa-11,🤑🥰🤩
2024-04-12 18:08:55,aaa-11,🤑🥰🤩
Sending message...
send_message():🥰😁👅
send_message():b'gAAAAABmGds_DQ38f86c4I_Klu03Jh3pboJ3v5n_m4kfb0FIRxsUMhuNH0osCeHgD5JJ_LwkmffsJxKI_wb917RZWrqsMXY1og=='
Message sent successfully.
b'gAAAAABmGds_QJxS-hIrnGol_dgcRakupupiM0omeMmG3N-bRx7RXWTt66KTKBkfN1VIrzrpx_t_hTK93Fp0h9QrIxeQ8vwP2no54GeX4vEC5g96zVeB3YQhaygvBhFXwTrhqbxUOy6n'
from server decrypted:2024-04-12 18:09:19,bbb-11,🥰😁👅
2024-04-12 18:09:19,bbb-11,🥰😁👅
```

aaa-11

```
Successfully connected to the server
[2024-04-12 18:06:51] [SERVER] aaa-11 was added to the chat!
[2024-04-12 18:06:56] [SERVER] bbb-11 was added to the chat!
[2024-04-12 18:07:00] [aaa-11] hello
[2024-04-12 18:07:11] [bbb-11] how are you?
```

bbb-11

```
Successfully connected to the server
[2024-04-12 18:06:56] [SERVER] bbb-11 was added to the chat!
[2024-04-12 18:07:00] [aaa-11] hello
[2024-04-12 18:07:11] [bbb-11] how are you?
```

www-00

```
Successfully connected to the server
[2024-04-12 14:06:45] [SERVER] www-00 was added to the chat!
[2024-04-12 14:06:52] [SERVER] ccc-00 was added to the chat!
[2024-04-12 14:07:00] [www-00] 😭
[2024-04-12 14:07:13] [ccc-00] :smiley_face:
[2024-04-12 14:07:22] [ccc-00] 😑
[2024-04-12 14:08:09] [ccc-00] ❤ ❤ 👑 👑 👑 👑
[2024-04-12 14:09:54] [www-00] hello
[2024-04-12 14:10:04] [www-00] hello
[2024-04-12 14:10:14] [www-00] :sad_face:
```

ccc-00

```
Successfully connected to the server
[2024-04-12 14:06:52] [SERVER] ccc-00 was added to the chat!
[2024-04-12 14:07:00] [www-00] 😭
[2024-04-12 14:07:13] [ccc-00] :smiley_face:
[2024-04-12 14:07:22] [ccc-00] 😑
[2024-04-12 14:08:09] [ccc-00] ❤ ❤ 👑 👑 👑 👑
[2024-04-12 14:09:54] [www-00] hello
[2024-04-12 14:10:04] [www-00] hello
[2024-04-12 14:10:14] [www-00] :sad_face:
```

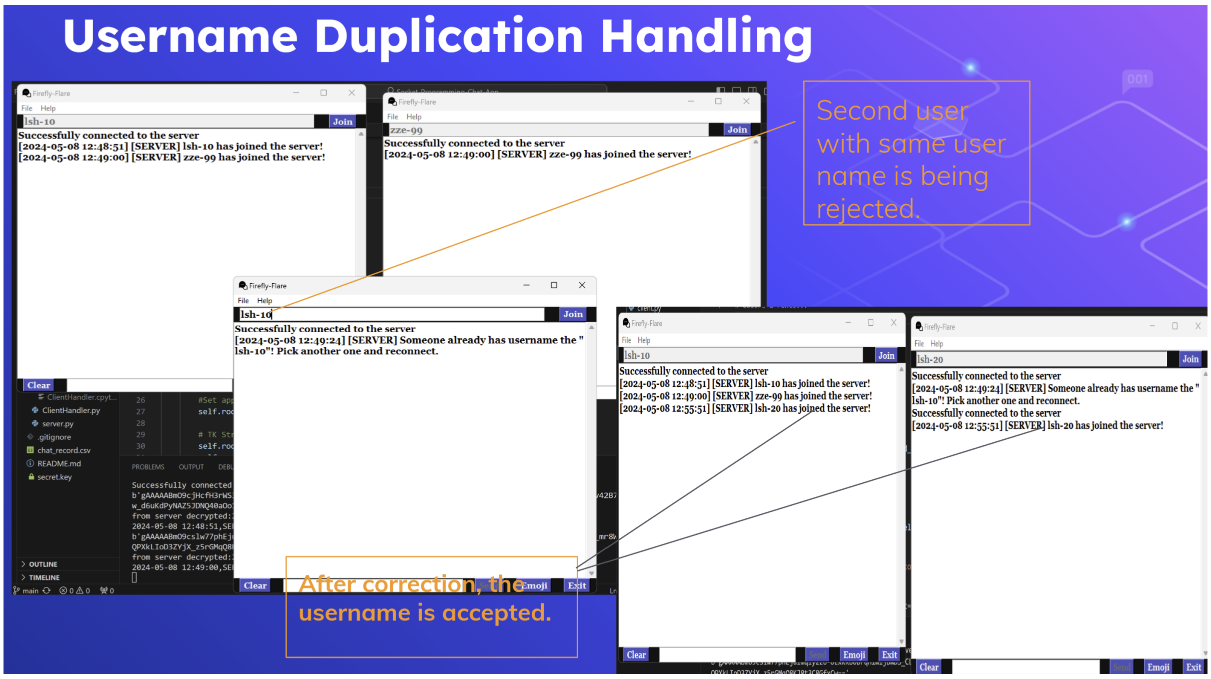
```
Message sent successfully.
b'gAAAAABmGaK5NT0fvHCvQuaEi-nfnCjT2XYwU2ti0pJr_Meb-epbDQ1miwyBC3OX6SXHY5ewUi7hh5Vzlv1tJOC3Ln5e2d5WuaEdEbnJriVft-G6tgrUf63FT88lHe5PjPCdvOsco3ZEGRutlyrrZeK
H-0aj4uH8kQ=='
from server decrypted:2024-04-12 14:08:09,ccc-00,❤❤👑👑👑👑
2024-04-12 14:08:09,ccc-00,❤❤👑👑👑👑
b'gAAAAABmGaMiMlQo0F_IWr3r42rhBl-zm4ilKlJ0UnusLbH54XkBwZIgyHSnY6YC1mQaea_NQi0GTKXZXUZ_RBgIT6JqunWvPwcLM7jhYrSw8pmE9lloxhRVeI4Pgn8pqtX6x3wiqHzs'
from server decrypted:2024-04-12 14:09:54,www-00,hello
2024-04-12 14:09:54,www-00,hello
b'gAAAAABmGaMs5JlqNPWGvXxhQPjVpJzSbhdyPibgfTYmNLiky6rOKIeXIqJexpQ2lNms7ka6ByP1KGyffkcBc4JJvw9WgowP410EuNMwjLvAX8o608Hy0ydriCx5quuFrwzSAQyARR9s'
from server decrypted:2024-04-12 14:10:04,www-00,hello
2024-04-12 14:10:04,www-00,hello
```

main* ⊗ 0 ⚠ 0 0   Ln 335, Col 32 (91 selected) Spaces: 4 UTF-8 CRLF Python

Figure 9: Emoji Implementation

4.4 Duplicate Username Handling

Additionally, we also managed to complete the implementation of a sophisticated mechanism to handle duplicated usernames within our chat application, ensuring clarity and cohesion in the chat environment. Leveraging robust algorithms, we programmed the application to promptly detect and prompt users about duplicated usernames, guiding them through an iterative process to select a unique username. Conditional access to the chat was granted only upon successful validation of a unique username, contributing to a seamless user experience. By prioritizing user-friendly feedback and continuous improvement, we ensure that our system remains effective in handling username conflicts, ultimately enhancing user satisfaction and facilitating smooth interactions within the chat environment. This creates 1,757,600 possible username combinations, resetting each time the server is relaunched.


Username Duplication Handling
Second user with same user name is being rejected.
After correction, the username is accepted.
Firefly-Flare
File
Help
Join
Clear
Emoji
Exit

lsh-10

```
Successfully connected to the server
[2024-05-08 12:48:51] [SERVER] lsh-10 has joined the server!
[2024-05-08 12:49:00] [SERVER] zze-99 has joined the server!
```

zze-99

```
Successfully connected to the server
[2024-05-08 12:49:00] [SERVER] zze-99 has joined the server!
```

lsh-10

```
Successfully connected to the server
[2024-05-08 12:49:24] [SERVER] Someone already has username the "
lsh-10"! Pick another one and reconnect.
```

lsh-10

```
Successfully connected to the server
[2024-05-08 12:48:51] [SERVER] lsh-10 has joined the server!
[2024-05-08 12:49:00] [SERVER] zze-99 has joined the server!
[2024-05-08 12:55:51] [SERVER] lsh-20 has joined the server!
```

lsh-20

```
Successfully connected to the server
[2024-05-08 12:49:24] [SERVER] Someone already has username the "
lsh-10"! Pick another one and reconnect.
Successfully connected to the server
[2024-05-08 12:55:51] [SERVER] lsh-20 has joined the server!
```

Figure 10: Username Handling

4.5 Git and GitHub Collaboration

In parallel with GUI development, our front-end team spearheaded the establishment of a GitHub repository to facilitate seamless collaboration among team members. This involved learning the intricacies of branching and merging, allowing for simultaneous feature development without disrupting the main codebase. Furthermore, we actively participated in resolving merge conflicts, ensuring the integration of different branches proceeded smoothly. Leveraging GitHub's collaborative features such as pull requests and code reviews, we fostered effective communication and coordination within the team, thereby maximizing productivity and accelerating the project's progress. Overall, by meticulously designing the Tkinter GUI and harnessing the power of GitHub for collaborative development, we contributed to the creation of a user-friendly chat application with enhanced functionality and streamlined team collaboration.
Following is the GitHub Link:
https://github.com/Maung945/Socket-Programming-Chat-App.git

5 Other Features

5.1 Chat record saving, Reading about app

Furthermore, the chat storing feature not only preserves talks within the application but also enables users to export their stored chats to text files, extending its usefulness and adaptability. This feature offers customers enhanced versatility and authority in managing their stored conversations. Users can conveniently arrange and oversee their conversation history outside of the application context by preserving talks as text files. This allows users to include their chats into pre-existing workflows, such as documentation, analysis, or sharing with collaborators who may not be able to use the application. In addition, by saving chats as text files, it guarantees the ability to move and access data over time, as text files may be stored, moved, and retrieved on many systems and devices. In summary, the integration of conversation storing functionality within the application, along with the option to export chats to text files, provides users with a complete solution for keeping and utilizing their important communication exchanges.

5.2 Enhanced Chat Efficiency: Introducing Shortcut Integration for a Simplified Experience

The incorporation of shortcuts in this program greatly improves user experience and productivity. The focusing function on the username text field directs users to type their username automatically when they activate the application, making the login procedure more efficient. In addition, the incorporation of the Enter key to join a chat room avoids the necessity of manually clicking the join button, providing a prompt and instinctive method to enter chats. In addition, the capability to send messages by utilizing the Ctrl + Enter shortcut accelerates the chat process, eliminating the requirement to click the send button for every message. These shortcuts not only make it easier to navigate throughout the program but also enable users to participate in conversations more smoothly and efficiently, thereby improving overall usability and productivity.

6 Future Implementations

6.1 File Sharing Capabilities

We plan to enhance our chat app with robust file-sharing capabilities, starting with picture sharing. The user interface will include a dedicated button or icon for uploading and sharing images, conveniently placed in the message input area. Users will also have the ability to drag and drop pictures directly into the chat window. To ensure a smooth user experience, a preview feature will be implemented, allowing users to view images before sending them. We will set file size limits to maintain server performance and automatically compress and optimize images to facilitate faster uploads and downloads without significantly compromising quality. Our system will support common image formats such as JPEG, PNG, and GIF. On the backend, we will utilize cloud storage solutions like AWS S3 or Google Cloud Storage for efficient and scalable image storage. A database will store metadata about shared files, including URLs, senders, recipients, timestamps, and file types, while ensuring secure transmission and storage through encryption protocols.

6.2 Password to Enter Chatroom

To enhance security, we will implement a password layer for entering chatrooms. The user interface will feature a clean, user-friendly password prompt screen that appears before access is granted. To aid users who may forget their passwords, we will offer a password recovery option via email or phone number verification. Clear password requirements, such as minimum length and the inclusion of special characters, will be communicated to users to ensure robust security. Our backend will feature a secure authentication system that verifies passwords using strong hashing algorithms like bcrypt. We will also develop an access control system to manage user permissions and keep logs of login attempts to monitor and respond to suspicious activities, implementing measures to prevent brute force attacks, such as account lockouts after multiple failed attempts.

6.3 MAC Verification

In order to bolster the security capabilities of our application and improve its defense against potential man in the middle attacks such as delay attacks or replay attacks, we plan to implement some form of MAC verification in the future. The MAC integration would be included within the currently unused flag bits contained in the packet header of the payload and would be checked against the server to verify the integrity of the messages being sent and the proper user sending them.

6.4 Individual Chat (One-to-One Chat Capability)

Adding one-to-one chat capability is another key enhancement for our app. The user interface will include a contact list or friend list feature, where users can easily see their available contacts for direct messaging. By clicking on a contact's name, users will access a separate direct messaging interface distinct from group chatrooms. Real-time notifications for incoming messages will ensure users are promptly informed of new communications, with distinct notifications for one-to-one chats versus group messages. On the backend, we will employ WebSockets or a similar protocol to enable real-time messaging. A carefully designed database schema will support one-to-one messaging, storing essential details like sender and recipient IDs, timestamps, and message statuses (e.g., delivered, read). To maintain the privacy and security of conversations, end-to-end encryption will be implemented.

6.5 Implementation Roadmap

Our implementation will follow a phased approach. In Phase 1, we will focus on file sharing, designing the UI/UX for picture sharing and integrating cloud storage solutions. After deploying the frontend features and conducting user testing, we will optimize performance based on feedback. Phase 2 will address the password layer for chatroom access, developing the authentication system, and designing the password prompt UI. We will also implement password recovery mechanisms and test security measures comprehensively. Phase 3 will introduce individual chat capabilities, with UI/UX design for direct messaging, backend support for real-time messaging, and the integration of contact lists and messaging interfaces. A beta version will be launched to gather user feedback for further refinements. This structured approach ensures a progressive enhancement of our chat app, improving user experience, security, and overall functionality.

References

htpps://docs.python.org/3/library/tkinter.html

https://github.com/BerntA/SecureChatServer

https://www.youtube.com/watch?v=hBnOdIg0jAM

https://www.youtube.com/watch?v=mkXdvs8H7TA

https://pythonprogramming.net/server-chatroom-sockets-tutorial-python-3/

https://github.com/GiacomoPope/kyber-py