Real-Time Chatroom Application with GUI, File Transfer, Emojis, and Encryption

Phan Nhat Khoa (220133) - Nguyen Chinh Quan (220066)

Course Name: Computer Networks

Date of Submission: 30/11/2025

Github Repository Link: https://github.com/BroKisD/Chat_room

Tables of Contents

Objective & Scope

Objective

The primary objective of this project is to apply and reinforce the knowledge gained in the Computer Networks course, particularly focusing on the use of WebSockets, the TCP protocol, and how packets are transmitted between clients and servers in a real-time chat application. This project will allow the exploration of key networking concepts such as socket communication, concurrency, and message encryption, which are essential for building secure and efficient communication systems.
Moreover, the project provides an opportunity to gain hands-on experience with the Go programming language, a high-performance language that is ideal for building scalable and concurrent applications. By developing the chatroom application, students will not only gain practical experience in software development but also learn about the structure and architecture of chat applications, including the challenges of managing user interactions, real-time communication, and ensuring data privacy.

Scope

The scope of this project includes the development of a fully functional real-time chat application that supports a variety of core chat features. These features include the ability to send and receive public and private messages, share emojis, upload and download files, and display system messages. The application will allow multiple users to connect and interact simultaneously, with the server handling each user’s connection concurrently through thread management, ensuring that no user experiences delays or interruptions in communication.

The user interface will be designed with simplicity and ease of use in mind, ensuring that even those with limited technical knowledge can easily access and navigate the app. The minimalist design will focus on providing essential features without overwhelming the user with complex controls, while still offering a seamless and intuitive experience.
The chat application will be optimized for high performance using the Go programming language. Go's lightweight concurrency model makes it an ideal choice for handling multiple simultaneous users efficiently, ensuring smooth, lag-free communication even as the number of connected users grows. The system will incorporate robust error handling, data validation, and connection management features to ensure reliability and stability.
The scope also includes the integration of secure communication protocols, such as message encryption, to safeguard user data during transmission, ensuring confidentiality and privacy for all messages exchanged in the system. The system will handle potential disruptions like connection drops, client disconnections, and other edge cases to ensure the user experience remains uninterrupted.

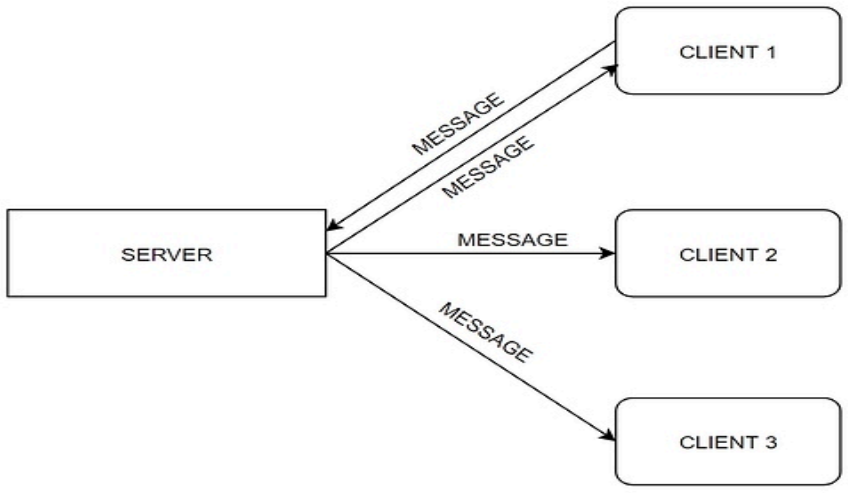
System Architecture

Technology of Choice

1. **Go:** Go is chosen as the primary programming language due to its lightweight nature, performance optimizations, and its powerful concurrency model via goroutines. Goroutines allow for efficient and easy management of multiple concurrent connections, which is crucial for handling a large number of users in real-time applications like chatrooms. Go's simplicity, coupled with its speed and efficiency, makes it the perfect choice for building a scalable and performant server-client system.

2. **Fyne for GUI:** Fyne is a lightweight graphical user interface (GUI) framework that integrates seamlessly with Go. It is designed to be simple and highly performant, making it ideal for building cross-platform desktop applications. Fyne's ability to provide a native look and feel while keeping the application lightweight aligns well with the goals of this project, where a minimalist, intuitive user interface is crucial for a smooth user experience.

3. **Go's Built-in TCP Protocol:** The chat application will use Go's built-in TCP protocol for network communication. TCP is fast, reliable, and ensures the ordered delivery of packets, which is essential for a real-time chat application. By leveraging Go's optimized TCP implementation, the system will benefit from reduced latency and increased throughput.



A visual representation of the system architecture would include the following components:

1. **Client:** The client is responsible for sending and receiving messages, managing the user interface (GUI), and interacting with the server. It connects to the server using

the TCP protocol and exchanges data with other clients.

2. **Server:** The server is the central hub that manages all connections, processes messages, and routes packets between clients. It handles the packet forwarding: The server receives packets from the client and forwards them to other connected clients, based on message type (public or private).

3. **Message Encryption:** The system employs different encryption techniques based on the message type:

    - **Public Messages:** Public messages are encrypted and decrypted on the client side using the AES (Advanced Encryption Standard) algorithm with a shared room key.

    - **Private Messages:** For private messages, the system uses a hybrid encryption method: RSA encryption for exchanging the symmetric AES key and AES encryption for encrypting the actual message content. The server's role is only to forward the message, without decrypting or accessing its content.

    - **File Transfer:** Files are encrypted similarly to messages (AES encryption) and sent in chunks from the client to the server, which forwards them to the recipient.

Core Components

The chat application consists of several core components that work together to provide real-time communication, secure messaging, file transfers, and user management. Each component plays a vital role in ensuring the system's performance, security, and usability.
Server Core
The server is the heart of the system, responsible for:

- **Accepting TCP Connections:** The server listens for incoming client connections and handles the connection lifecycle.

- **Managing Users:** The server authenticates users and maintains user sessions, including managing user connections, usernames, and public keys.

- **Broadcasting Messages:** The server is responsible for broadcasting public messages to all users and forwarding private messages to the intended recipients.

- **Persisting Room State:** The server ensures that the room state, such as the room key, is persisted on disk and reloaded when the server restarts, allowing clients to reconnect seamlessly.

Per-Connection Handler
Each connection to the server is managed by a dedicated handler:

- **Authentication:** When a client connects, the server first authenticates the user via the TypeAuth message. This involves verifying the user's credentials and sending an authentication response.

- **Spawning Reader Goroutines:** A reader goroutine is created for each connection to read incoming messages from the client.

- **Dispatching Messages:** The handler dispatches incoming messages to the appropriate handler function (e.g., for private/public messages, file transfers, or user list updates).

Users Manager

The Users Manager is responsible for managing the list of active users:

- **User Objects:** Stores user information, including their connection, username, and public key.

- **Thread-Safe Accessors:** Exposes methods such as GetByUsername, GetAll, SetPublicKey, and Remove for safe and concurrent access to user data.

File Transfer Backend

The file transfer backend manages the upload and download of files:

- **Public Files:** Files uploaded by clients are stored in the server's uploads directory. Clients can request files, and the server will send them in raw byte format.

- **Private Files:** Files sent privately between users are stored as JSON-wrapped encrypted payloads. The file is named according to a pattern that includes the sender and recipient, such as private_{from}to{to}_{name}.enc.

Shared Protocol

The system uses a standardized message protocol for communication:

- **Newline-Delimited JSON Framed Messages:** All messages are sent as newline-delimited JSON objects with the structure defined by shared.Message.

- **Message Types:** Messages are classified by their MessageType (e.g., public messages, private messages, file transfer requests).

Data & Control Flow

Connection Lifecycle

1. **Accept TCP Connection:** The server listens for incoming TCP connections and spawns a new goroutine to handle the connection.

2. **Authentication:** The server reads the initial TypeAuth message and calls AuthenticateUser. If authentication is successful, an AuthResponse message is sent back to the client.

3. **Broadcast Join:** After successful authentication, the server broadcasts the user's join message and updates the user list for all clients.

4. **Reader Goroutine:** Each connection has a dedicated reader goroutine that reads messages from the client (with a read deadline) and pushes them to msgChan.

5. **Message Handling:** The main server loop spawns a goroutine for each message received, processes it using the handleMessage function, and waits for cleanup via a WaitGroup.

Message Dispatch

- **Message Routing:** The handleMessage function routes each message based on its type (msg.Type) to the appropriate handler. Handlers include:
    - Public and private messages
    - File transfers
    - Public key handling
    - Reconnection requests
    - User list operations
- **Broadcasting:** Public messages are broadcast through s.broadcast, which enqueues messages to broadcastCh for fan-out to all connected clients.

Persistence & Room Key Lifecycle

- **Room Key:** A 32-byte roomKey is persisted on disk and reloaded on server startup. This ensures that clients can reconnect after restarts or crashes without requiring a new key exchange.

- **Key Encryption:** The server encrypts the shared roomKey using the client's RSA public key and sends it as a TypeRoomKey message, allowing the client to decrypt it locally.

Security & Cryptography

Asymmetric Encryption

- **RSA Keys:** RSA keys are used for user identity and encrypting AES keys. RSA-OAEP with SHA-256 is used for secure key exchange.

Symmetric Encryption

- **AES-GCM:** AES-GCM is used to encrypt message and file payloads. The room key is used for encrypting public messages.

Hybrid Encryption for Private Messages

- **Private Messages/Files:** For private messages and files, a hybrid encryption method is used. The AES key used for encryption is encrypted with the recipient's RSA key, ensuring that only the intended recipient can decrypt the AES key.

Public Key Exchange

- **PEM Format:** Clients send their public keys in PEM format to the server. The server uses the public keys to encrypt the room key and forward public key responses securely.

Concurrency Model

Per-Connection Goroutines

- **Reader Goroutine:** Each client connection has a dedicated reader goroutine that listens for incoming messages and pushes them to the message channel (msgChan).

- **Handler Goroutine:** A new goroutine is spawned for each message to process it using handleMessage, and the server waits for cleanup with a WaitGroup.

Server-Level Goroutines

- **Broadcast Goroutines:** The server spawns goroutines that consume messages from s.broadcastCh and fan them out to all connected clients.

Synchronization

- **Channels & WaitGroups:** Synchronization is handled using channels, WaitGroup, and context cancellation to ensure thread safety and proper message dispatching.

Error Handling & Robustness

Timeout Handling

- **Read Deadlines:** Each connection has a read deadline to prevent hanging connections due to network issues.

- **Error Messages:** Errors are sent to clients as TypeError messages. The server sends error messages to a specific client via the sendError or sendErrorToConn functions.

Cleanup

- **Disconnect Handling:** When a user disconnects or cancels the connection, the server removes the user from the list, broadcasts a leave message, and waits for in-flight handlers to finish before cleanup.

Reconnect Behavior

- **Reconnect Requests:** Clients can send a TypeReconnect message to request to reconnect to the chatroom. The server responds by resending the room key and user list, enabling seamless reconnection.

- **Persistent Room Key:** The persistent room key ensures that clients can reconnect without requiring a new key exchange.

Implementation of Core Features

GUI Design and Features

The chat application's GUI is designed to be minimalist, functional, and user-friendly, inspired by the aesthetic of Yahoo Messenger. The interface focuses on providing essential features without unnecessary clutter, ensuring a smooth and intuitive experience for users.

Key Features:

- **Minimalist and Functional Aesthetic**
    - The GUI follows a clean and simple design that emphasizes usability.
    - Inspired by Yahoo Messenger aesthetics, it features a familiar layout with an easy-to-navigate chat window, message input box, and user list.
    - The interface includes only essential elements, such as:
        - A scrollable message window to display chat history.
        - A text input area for composing messages.
        - A list of connected users with their online status.

- **Built with Fyne (Cross-Platform and Lightweight)**
    - The GUI is built using the Fyne framework, which is a cross-platform UI toolkit that integrates well with Go.
    - Lightweight and responsive, Fyne ensures that the chat application runs efficiently across different platforms (Windows, macOS, Linux) with minimal resource consumption.
    - The design is simple and straightforward, without sacrificing the essential features needed for a functional chat application.

- **Real-Time Message Updates via Goroutine Coordination**
    - The GUI is integrated with the client backend, allowing for real-time message updates.
    - Messages are handled and processed in the background goroutines, ensuring that the user interface remains responsive even during heavy usage.
    - When a new message arrives (either public or private), the backend notifies the GUI, which updates the message display in real time.

GUI Layout and Components

The GUI follows a simple layout that is easy to understand and use:

- **Message Window:**
    - Displays chat history with public and private messages, system messages.
    - Users can scroll to view older messages.
- **User List:**
    - Displays a list of active users.
- **Message Input Area:**
    - The input box where users type their messages.
    - Supports commands and plain text input.

Example Workflow: Sending a Message

1. **User Types a Message:** A user types a message in the message input box.
2. **Backend Processing:** The client backend encrypts the message (for private messages) and handles the message sending via the server.
3. **Real-Time GUI Update:** As soon as the backend processes the message, the GUI is updated to display the new message in the message window.

4. **Broadcast:** For public messages, the server sends the message to all connected users, and the GUI updates in real time for all users.

Emoji Support

Emoji Map Structure

The application supports emoji usage through a text-to-emoji mapping system that is organized into six categories:

- Smileys (e.g., 🙂 → 😊)
- Gestures (e.g., 🤞 → 🤞)
- Animals (e.g., 🐱 → 🐱)
- Food (e.g., 🍕 → 🍕)
- Weather (e.g., 🌦️ → 🌦️)
- Sports (e.g., ⚽ → ⚽)

The mapping of text shortcodes to emoji Unicode characters is stored in a static map within emojis.go, meaning there are no external dependencies or APIs required for emoji support. The mapping includes 60+ emojis across all categories.

- **Text-to-Emoji Mapping:** Shortcodes like :), <3, and :dog: will be mapped to their corresponding emojis (e.g., :) → 😊, <3 → ❤️, :dog: → 🐶).

Emoji Picker UI

The emoji picker provides a user-friendly interface for selecting emojis:

- **Modal Dialog:** The emoji picker is displayed as a modal dialog with a tabbed interface.
    - Each tab corresponds to one emoji category (Smileys, Gestures, Animals, etc.).
- **Grid Layout:** The emojis are displayed in a grid with 8 columns, making it easy for users to select an emoji visually.
- **Emoji Selection:** When an emoji is clicked, it is appended to the text input field, enabling users to quickly insert emojis without needing to memorize shortcodes.

Emoji Conversion (ConvertEmojis function)

- **Before Sending Messages:** The ConvertEmojis function is called to convert any text-based shortcodes (e.g., :), <3) to their corresponding Unicode emoji characters.
    - **Example:** A message like “hello :) world” will be converted to “hello 😊 world” before it is sent to the server.
- **Bidirectional Conversion:**
    - **Client-to-Server:** The client sends emojis as Unicode characters, which are stored and transmitted in the message content.
    - **Server-to-Client:** When the server sends the message back to the client, it includes emojis as Unicode characters, and the client renders them as their visual glyphs.
- **No Sanitization Issues:** Emojis are valid Unicode characters and pose no issues when transmitted over the JSON wire protocol, as they are natively supported.

Message Display

Message Types & Color-Coding

Each message type is assigned a distinct color for easier identification and visual distinction:

- **System Messages:** Displayed in red (e.g., system notifications or server messages).
- **Global/Public Messages:** Displayed in blue (e.g., public messages that are broadcast to all users).
- **Private Messages:** Displayed in purple (e.g., direct messages between users).
- **User's Own Messages:** Displayed in dark green with the suffix (you) to indicate the message was sent by the current user.
- **Regular Text:** Default text is displayed in black.

Message Rendering Pipeline

When a message is received from the client backend, it follows a series of steps to render and display on the GUI:

1. **Raw Message Received:** The message is received from the client backend in raw format.

2. **Prefix Parsing:** The message type is determined based on its prefix (e.g., (System), (Global), (Private)), and the appropriate color is assigned.

3. **Text Rendering:** A canvas text object is created with the message text and assigned color (canvas.NewText(displayMsg, msgColor)).

4. **Message Container:** The text object is added to the message container (a.messageList.Add(msgText)).

5. **Auto-Scrolling:** The scrollable message window automatically scrolls to the bottom (a.messagesScroll.ScrollToBottom()), ensuring that the latest messages are always visible.

Message Container

- **Layout**: The message container uses a vertical box (VBox) layout to stack messages.

- **Scrollable Area**: The container is placed inside a scrollable area (container.NewVScroll()), ensuring that the chat history is scrollable and users can view older messages.

- **History**: The application keeps a memory-based message history with no persistence (history is cleared when the app is closed).

    - Implicit Limit: The number of messages displayed is limited by available RAM, typically displaying up to 1000+ messages without performance issues.

File Messages

Public File Transfers

- **Format**: Public file transfer messages are parsed from the [FILE] prefix.

    - Example: [FILE] from: filename — This indicates that a file is being shared by another user.

- **Display:**

    - The message is displayed in a custom message box with a blue header (e.g., "from: filename is sharing a file").
    - A Download button is provided for users to download the file.

Private File Transfers

- **Format**: Private file transfer messages are parsed from the [PRIVATE FILE] prefix.
    - Example: [PRIVATE FILE] sender sent you: filename — This indicates a private file transfer from the sender to the current user.
- **Display**:
    - The message is displayed in a custom message box with a purple header (e.g., "sender sent you: filename").
    - An "Accept and Download" button is shown, allowing the recipient to download the file securely.

File Message Boxes

- **Custom Layout**: File messages have their own distinct layout with:
    - Text (e.g., file sender, file name).
    - A button for downloading the file.
- The design ensures that file messages are visually separated from regular chat messages, making it easy for users to identify and interact with them.

User Authentication

The user authentication flow ensures that clients are verified and registered before participating in the chatroom. The server validates the username, checks for conflicts, and establishes a secure session for each user.

Authentication Flow

1. **Initial Connection**:

- The client sends a TypeAuth message as the first message over the established TCP connection. This message includes the requested username for authentication.

2. **Server Authentication**:
    - The server reads the incoming TypeAuth message and extracts the requested username from the payload.
    - The server then sanitizes and validates the username based on the following criteria:
        - **Trim** any leading/trailing whitespace.
        - **Check length**: The username must be between 3 and 32 characters long.
        - **Validate allowed characters**: Only letters, numbers, periods, underscores, and hyphens are allowed.
        - **Check for reserved names**: Usernames like server, admin, and others are reserved and cannot be used.
3. **Uniqueness Check**:
    - The server checks if the requested username already exists among currently connected users. If it exists, the server returns an error.
4. **User Registration**:
    - If the username is valid and unique, the server creates a new User entry in the **Users Manager**. This entry includes:
        - Connection: The client's TCP connection.
        - Public Key: A placeholder for the client's public key (to be used for encryption).
        - Created At: Timestamp of when the user was registered.
5. **Authentication Response**:
    - The server responds to the client with a TypeAuthResponse message:

        - If authentication is successful, the response indicates success (success=true).
        - If authentication fails, the response contains an error message (success=false, with the reason, such as "username already in use").

6. **Post-Authentication**:

   After a successful authentication, the server broadcasts the user's **join message** to all connected clients and sends the **user list**. The server also proceeds with other handshake messages, such as sending the room key or other security protocols.

7. **Reconnect Feature**:
    - The server implements the TypeReconnect message. This allows clients to re-associate with their previous username only if:
        - The previous connection is no longer active.
        - A short grace period has expired after disconnection.

Expected TypeAuth JSON

The **TypeAuth** message, sent by the client during the authentication process, would be expected to look like the following minimal JSON structure:

```
{
   "type": "TypeAuth",
   "username": "desired_username"
}
```

Where:

- "type": Identifies the type of message (TypeAuth).
- "username": The username that the client wishes to use for authentication.

Public Message

The public messaging functionality ensures that messages can be broadcasted to all users in a chatroom while maintaining security through end-to-end encryption. Here's how the public message flow works:

Client-side (Sending a Public Message)

The client encrypts the plaintext message using **AES-GCM** and the shared roomKey. This encryption process combines a nonce (unique number used once) with the ciphertext to ensure both security and integrity.

The result is base64-encoded and sent in the message payload. The structure of the message is as follows:

```
{
    "type": "public",
    "from": "<user>",
    "content": "<base64(nonce||ciphertext)>",
    "ts": "<timestamp>"
}
```

Where:

- **type**: Identifies the message type (public in this case).
- **from**: The username of the sender.
- **content**: The base64-encoded AES-GCM ciphertext (nonce and ciphertext concatenated).
- **ts**: The timestamp indicating when the message was sent.

Server-side

- The server receives the public message and **treats the content field as an opaque base64-encoded AES-GCM blob**.
- **The server does not decrypt the message** but validates the format and size of the received blob. The server only extracts and sets metadata for the message,

such as the From field (sender's username) and the timestamp (ts).

- The server **broadcasts the message to all connected clients** without decrypting the content, ensuring the integrity and confidentiality of the message.

Recipient Client (Receiving the Public Message)

- When a client receives the broadcast message, it base64-decodes the content field, which reveals the nonce||ciphertext (the encrypted message).

- The client then **decrypts** the message using the shared roomKey and **AES-GCM** decryption, which recovers the plaintext.

- Finally, the plaintext message is displayed to the recipient.

Why Server Should Not Decrypt

The reason the server does not decrypt public messages is primarily for **security and privacy**. Here's why:

- **End-to-End Confidentiality**: By ensuring that only the clients can decrypt the message using the shared roomKey, the server cannot access the plaintext of any public message. This preserves confidentiality between users, meaning even if the server is compromised, the actual message content remains secure.

- **Simplified Server Security Posture**: If the server were to decrypt messages, it would need to store and potentially expose plaintext messages. By keeping messages encrypted, the server avoids the complexity of securing plaintext data and reduces the potential attack surface.

- **Server Cannot Leak Plaintext**: If the server does not decrypt the messages, there is no risk of the server unintentionally leaking plaintext messages (e.g., through logs, bugs, or compromised server access).

- **Moderation and Indexing**: If the server needs to index, search, or moderate public messages, a separate pipeline can be implemented. In such cases, clients could opt-in to send plaintext messages (for moderation purposes) or rely on a secure moderation service where messages are decrypted in a secure, isolated environment.

Private Message

Private messages are relayed as hybrid-encrypted envelopes: the client encrypts the message payload with a one-time AES key (AES-GCM), encrypts the AES key with the recipient's RSA public key (RSA-OAEP), and sends an opaque envelope containing both the encrypted AES key and the encrypted message content to the server. The server simply validates and forwards the envelope without decrypting it.

Private Message Workflow

1. **Client A (Sender) Composes a Private Message to Client B**:
    - **Obtain Client B's RSA Public Key**: Client A either retrieves Client B's public key through a prior public key exchange or makes a direct request to the server for the key.
    - **Generate a Random AES Key**: Client A generates a random 256-bit AES key, which will be used to encrypt the actual message content.
    - **Encrypt the Message**: Client A encrypts the plaintext message using **AES-GCM** and the generated AES key. This produces a nonce||ciphertext.
    - **Encrypt the AES Key**: The generated AES key is then encrypted with Client B's **RSA public key** using the **RSA-OAEP** encryption scheme.
    - **Send the Encrypted Message**: The client sends the encrypted payload as part of a TypePrivate message. The message consists of:
        - **EncryptedKey**: The AES key, encrypted using Client B’s RSA public key.
        - **Content**: The encrypted message (nonce||ciphertext), base64-encoded.
2. **Server Receives the Private Message**:
    - **Validate Sender Authentication**: The server ensures that Client A is authenticated before processing the message.
    - **Check Recipient Existence**: The server verifies that Client B exists and is currently online.
    - **Validate Encrypted Data**: The server checks that both the EncryptedKey and Content fields are non-empty, base64-encoded, and within the acceptable size

limits.

- **Metadata Handling**: The server sets the From (sender's username) and Timestamp of the message for proper metadata handling.

- **Forward the Message**: The server forwards the opaque envelope (the encrypted AES key and encrypted message) to Client B without decrypting it.

3. **Client B (Recipient) Receives the Private Message**:

   - **Base64 Decode**: Client B receives the TypePrivate message and base64-decodes both the EncryptedKey and Content.

   - **Decrypt the AES Key**: Client B uses their own **RSA private key** to decrypt the EncryptedKey and recover the AES key that was used to encrypt the message.

   - **Decrypt the Content**: Client B decrypts the Content (nonce||ciphertext) using the recovered AES key and the **AES-GCM** decryption method to recover the plaintext message.

   - **Display the Message**: Finally, Client B displays the decrypted plaintext message.

Private Message Example (JSON)

Here's a sample **private message** that follows the structure mentioned:

```
{
  "type": "private",
  "from": "alice",
  "to": "bob",
  "content": "BASE64(nonce||ciphertext)",
  "encryptedKey": "BASE64(encrypted AES key)",
  "ts": "2025-11-28T12:34:56Z"
}
```

Where:

- "type": "private": Indicates that this is a private message.
- "from": "alice": The sender's username.
- "to": "bob": The recipient's username (could be omitted or inferred in some cases).
- "content": "BASE64(nonce||ciphertext)": The base64-encoded message content, encrypted with AES.
- "encryptedKey": "BASE64(encrypted AES key)": The base64-encoded AES key, encrypted with Client B's RSA public key.
- "ts": "2025-11-28T12:34:56Z": The timestamp when the message was sent.

Sending Private Messages via /w Command

Private messages can be sent using the /w command followed by the username of the recipient and the message content. The format is:

/w <username> <message content>

For example, if Alice wants to send a private message to Bob:

/w bob Hello Bob, how are you?

This will trigger the client to encrypt the message and send it as a **private message** to Bob.

This approach combines the performance benefits of AES encryption with the secure key exchange provided by RSA, making it suitable for secure, private communication in real-time applications.

Concurrent Connection Model

This project uses a per-connection goroutine architecture combined with server-level broadcast goroutines to efficiently handle concurrent connections. By leveraging Go's concurrency model (goroutines and channels), the system ensures that each connection operates independently, processing messages, reading data, and handling cleanup in parallel while maintaining thread-safe access to shared state.

1. Server-Level Goroutines

Accept Loop

- **Single Goroutine**: The server runs a single goroutine to listen on the specified TCP port (e.g., :8080).

- **Incoming Connections**: For each new incoming connection, the accept loop spawns a new handleConnection goroutine to handle that specific connection.

- **Non-blocking**: The accept loop does not block message processing and continues accepting new connections while other goroutines handle existing ones.

- **Why**: This allows the server to handle unlimited concurrent connections since each connection is processed independently in its own goroutine, preventing blocking of other connections.

Broadcast Goroutine

- **Dedicated Goroutine**: A single broadcaster goroutine consumes messages from the broadcastCh channel.

- **Message Delivery**: When a message (e.g., a public message or user join notification) arrives in broadcastCh, the broadcaster:

    - Iterates over all active users (using a thread-safe read lock on s.users.GetAll()).

    - Sends the message to each user's connection, either sequentially or by spawning mini-goroutines for each write.

    - Waits for all writes to complete (using WaitGroup or a similar synchronization mechanism) before accepting the next broadcast.

- **Why**: Decouples the message reception from message delivery, ensuring that the server can continue to process incoming messages while avoiding blocking due to slow clients.

2. Per-Connection Goroutines

Each authenticated TCP connection gets its own set of goroutines to handle reading, message processing, and cleanup independently.

Reader Goroutine

- **One Goroutine Per Connection**: Each connection has a dedicated reader goroutine that calls shared.ReadMessage(reader) in a loop.
- **Read Deadline**: The reader blocks on socket read with a read deadline (e.g., 5 minutes).
- **Message Processing**: For each message received, the reader:
    - Parses the JSON message (newline-delimited).
    - Sends the parsed message into a buffered msgChan channel (with a capacity of 100).
- **Why**: Ensures that a slow or hung message does not block subsequent messages, allowing the reader loop to continue independently.

  Error Handling:
    - If the read times out or encounters EOF, the reader sends an error to errChan and exits.
    - The main connection loop detects this error and triggers the cleanup procedure.

Message Handler Goroutines

- **New Goroutine Per Message**: Each message in msgChan triggers the spawning of a new goroutine to handle the message.
- **Types of Handlers**: Depending on the message type, the handler goroutine calls the appropriate handler function:
    - handlePrivateMessage() — Relays encrypted envelope to recipient.
    - handlePublicMessage() — Broadcasts to all users.

    - handlePublicKey() — Stores/encrypts the user's RSA public key.
    - handleFileTransfer() — Handles file uploads and downloads.
- **Tracked by messageWg**: The message handler goroutines are tracked by a WaitGroup (messageWg) to ensure that all handlers complete before cleanup.

Main Connection Loop

- **Single Goroutine**: The main connection loop runs in a single goroutine and selects on multiple channels:
    - msgChan: New messages are read and sent to handler goroutines.
    - errChan: If an error occurs, the loop initiates cleanup.
    - ctx.Done(): If the connection is canceled (e.g., manual disconnect), cleanup is triggered.
- **Cleanup Procedure**: The cleanup() function does the following:
    - Calls cancel() to signal the reader goroutine to stop.
    - Closes msgChan to prevent new handlers from being started.
    - Waits for all in-flight handlers to complete using messageWg.Wait().
    - Removes the user from the s.users registry and broadcasts a user leave notification.
    - Closes the TCP connection.

3. Thread-Safe Shared State

Users Manager

- **Data Structure**: Stores active users in a map: map[string]*User.
- **Concurrency Control**:
    - Read operations (GetByUsername, GetAll) acquire a **read lock** (RLock).

    - Write operations (Add, Remove, SetPublicKey) acquire a **write lock** (Lock).

- **Why**: This ensures that multiple goroutines (both readers and handlers) can access the user registry concurrently without causing data races.

Broadcast Channel

- **Buffered Channel**: A buffered channel with a capacity of ~10–100 messages for broadcasting.

- **Decoupling**: Any goroutine can send messages to broadcastCh (e.g., from handler goroutines when processing public messages or user join/leave notifications).

- **Broadcasting**: The broadcaster goroutine consumes messages from broadcastCh and delivers them to all connected clients in order.

- **Why**: This decouples message production from message delivery, preventing slow clients from blocking the message flow.

User Connection

- **Concurrency Control**: Each user has a TCP connection (net.Conn), but read and write operations on the connection happen in different goroutines:

    - The **reader goroutine** reads messages from the connection.

    - The **broadcast goroutine** or **handler goroutines** write messages to the connection.

- **Solution**: Writes to the connection are serialized by having only one goroutine (the broadcaster) handle writes to each connection, ensuring that concurrent writes are avoided.

4. Synchronization Mechanisms

WaitGroup

- **Tracks In-Flight Handlers**: messageWg tracks the number of in-flight message handlers. It is incremented when a handler is spawned and decremented when the handler completes.

- **Graceful Shutdown**: cleanup() waits for all handlers to finish before closing the connection using messageWg.Wait().

Context

- **Signals Reader Goroutine to Stop**: ctx.Done() is used to cancel the reader goroutine when the connection is closed or canceled.

Channels (chan)

- **msgChan**: Buffered channel (capacity ~100) queues messages for processing. It allows the reader to continue reading while the main loop processes the messages.
- **errChan**: Buffered channel (capacity 1) signals errors (EOF, timeout).
- **broadcastCh**: Buffered channel (capacity ~10) queues broadcast messages.

5. Message Flow & Concurrency Example

Scenario: User A sends a public message while User B joins simultaneously.

- **User A** sends a message: The reader goroutine processes the message, and the main loop spawns a handler that processes the message and sends it to broadcastCh.
- **User B** joins: User B's connection completes authentication, triggering a broadcast of their join notification to broadcastCh.
- The **server broadcaster** processes both messages in order, ensuring no blocking between message reads and writes.

File Transfer Architecture

The file transfer system in this chat application is designed to securely handle both **public** and **private** file transfers between users. Files are stored on the server's uploads directory, and each transfer employs **hybrid encryption**: **RSA-OAEP** is used for encrypting the AES key, while **AES-GCM** is used for encrypting the file content. Below is an overview of how both public and private file transfers work within the system.

1. Public File Transfer Workflow

Phase 1: Client Initiates Upload

1. **User Action**:
    - The user selects a file from their local machine and chooses "Everyone (Public)" from the options in the GUI to make the file accessible to all users.
2. **Client-Side Processing**:
    - **File Validation**:
        - Check if the file exists and is readable.
        - Validate file size (e.g., limit set to 100 MB).
        - Extract the filename from the file path (e.g., /path/to/document.pdf → document.pdf).
    - **File Encryption**:
        - The file is read into memory (fileBytes := io.ReadAll(file)).
        - The file is encrypted using **AES-256-GCM** with the shared roomKey. A random nonce (12 bytes) is generated, and the file content is encrypted.
        - The result is nonce || ciphertext, which is then **base64-encoded** for transmission.
    - **Message Composition**:
        - The client builds a TypeFileTransfer message, which includes:
            - **Type**: "file_transfer"
            - **From**: Username of the sender
            - **Filename**: document.pdf
            - **Content**: Base64-encoded nonce || ciphertext

        - **Timestamp**: Current timestamp (now())

3. **Sending to Server**:
    - The client sends the TypeFileTransfer message over the TCP connection using client.SendFile(filename, fileBytes) or similar.
    - GUI displays: >>> Uploading document.pdf...

Phase 2: Server Receives & Stores

1. **Server-Side Processing**:
    - **Validate Sender**:
        - Ensure the user is authenticated and their connection is active.
    - **Extract & Decode**:
        - Extract the filename (msg.Filename) and the encrypted file content (msg.Content).
        - **Base64-decode** the encrypted file content (encryptedBytes := base64.Decode(msg.Content)).
    - **Sanitize Filename**:
        - Use filepath.Base() to strip any directory components and prevent path traversal.
        - Validate that the filename length is reasonable (e.g., max 255 characters).
    - **Store File**:
        - Create the uploads directory if it doesn't exist.
        - Write the encrypted file to uploads/document.pdf (overwrite if it already exists).
        - Set file permissions to 0600 (read/write for owner only).

- **Broadcast Notification**:
  - The server creates a TypeFileAvailable message:
    - **Type**: "file_available"
    - **From**: "server"
    - **Content**: "alice is sharing a file: document.pdf"
    - **Filename**: "document.pdf"
  - The server broadcasts this notification to all connected clients via s.broadcast().
- **Send Success Response**: the server sends a success message to the uploader:

Phase 3: Other Clients Receive Notification

1. **Client-Side (Other Users)**:
   - **Receive TypeFileAvailable Message**:
     - Other clients receive the TypeFileAvailable message and parse it.
   - **Display Notification**:
     - The filename is extracted from msg.Filename.
     - A notification is added to the chat UI: [server]: alice is sharing a file: document.pdf [Download].
   - **Download Option**:
     - The message is rendered as a box with:
       - A **blue header**: *** alice is sharing a file: document.pdf
       - A **Download button**: [Download]
     - The user can click the **Download** button to trigger the download workflow (Phase 5).

Phase 4: Uploader Receives Confirmation

1. **Client-Side (Original Uploader)**:
    - **Receive TypeFileAvailable Broadcast**:
        - The uploader receives the broadcasted TypeFileAvailable message, recognizing it as their own file.
    - **Display Confirmation**:
        - The uploader's UI shows: [you]: File uploaded: document.pdf.
        - No user action is needed; this is a confirmation that the file has been uploaded successfully.

Phase 5: Client Downloads File

1. **User Action**:
    - The user clicks the **[Download]** button next to the file message in the GUI.
2. **Client-Side Processing**:
    - **Send Download Request**:
        - The client sends a TypeFileDownloadRequest message to the server, requesting the file.
        - The message includes:
            - **Type**: "file_download_request"
            - **From**: Username of the requester
            - **Filename**: document.pdf
    - **GUI Displays**: >>> Downloading document.pdf...
3. **Receive Encrypted File from Server**:

- The server responds with a TypeFileDownload message containing the base64-encoded encrypted file (nonce || ciphertext).

4. **Decrypt File**:
    - The client base64-decodes the content and decrypts the file using the shared roomKey with **AES-256-GCM**.
        - **Extract nonce** (first 12 bytes) and **ciphertext** (remainder).
        - The decrypted file bytes are obtained.
5. **Save to Disk**:
    - The client opens a "Save File" dialog and allows the user to choose a destination path.
    - The file is saved to disk using ioutil.WriteFile(destPath, fileBytes, 0644).
6. **GUI Displays Success**:
    - The UI shows: [INFO] File saved: ~/Downloads/document.pdf.
    - The chat shows: [you]: Downloaded document.pdf.

Phase 6: Server Handles Download Request

1. **Server-Side (in HandleFileRequest)**:
    - **Validate Requester**:
        - Ensure the requester is authenticated and authorized to download the file.
    - **Check File Existence**:
        - Sanitize the filename to prevent path traversal and check if the file exists in the uploads directory.
    - **Read Encrypted File**:

        - Read the encrypted file content from disk: `fileBytes := ioutil.ReadFile("uploads/document.pdf")`.

    - **Send Encrypted File**:

        - The server sends the encrypted file to the requester using `shared.WriteMessage(requesterConn, msg)`.

    - **Log**:

        - `[INFO] Public file sent to requester: alice requested document.pdf`

2. Private File Transfer Workflow

Phase 1: Client Initiates Upload (Private)

1. **User Action**:

    - The user selects a file and chooses "One Person (Private)", entering the recipient's username.

2. **Client-Side Processing**:

    - **Recipient Validation**:

        - Check if the recipient's public key is cached. If not, send a `TypePublicKeyRequest` to the server and wait for a response.

    - **File Encryption**:

        - Generate a random AES key and use it to encrypt the file with **AES-256-GCM**.

        - Encrypt the AES key using the recipient's **RSA public key** with **RSA-OAEP-SHA256**.

        - Base64-encode both the encrypted file content and the encrypted AES key.

    - **Message Composition**:

        - Build the `TypePrivateFileTransfer` message and send it to the server:

        - **Type**: "private_file_transfer"
        - **From**: Username
        - **To**: Recipient
        - **EncryptedKey**: Base64-encoded AES key
        - **Content**: Base64-encoded encrypted file bytes
    - GUI displays: >>> Uploading document.pdf to bob...

Phase 2: Server Receives & Stores (Private)

1. **Server-Side Processing**:
    - **Validate**:
        - Ensure sender and recipient are authenticated and valid.
    - **File Handling**:
        - Store the encrypted file with a special filename pattern (e.g., private_alice_bob_document.pdf.enc).
        - Optionally store metadata in a JSON format for easy retrieval.
    - **Notify Recipient**:
        - Build a TypePrivateFileTransferAvailable message and send it directly to the recipient (not broadcasted).
        - GUI displays: [server]: alice sent you a private file: document.pdf [Accept and Download].

Phase 3: Recipient Receives Notification

- **Recipient Action**:
    - The recipient clicks the **[Accept and Download]** button in the GUI to download the private file.

Phase 4: Recipient Downloads (Private)

1. **Download Request**:
    - The recipient sends a TypePrivateFileDownloadRequest to the server.
2. **Decrypt the File**:
    - The server sends the encrypted file and AES key to the recipient.
    - The recipient decrypts the file using **RSA** and **AES-GCM**.
3. **Save the File**:
    - The recipient saves the decrypted file to disk and receives a confirmation in the GUI.

Testing & Known Issues

1. Testing Process

We conducted manual testing throughout the development cycle to verify core functionality, usability, and stability of the chatroom application.

Functional Testing

- **User Login/Logout:** Confirmed that users can authenticate successfully and that invalid input triggers proper error messages.
- **Public Messaging:** Checked that messages sent from one user appear immediately on all connected clients.
- **Private Messaging:** Ensured that private messages are visible only to the intended recipient.
- **Emoji Support:** Verified that selecting an emoji inserts it into the message field and displays correctly in the chat window.
- **File Upload:** Tested sending images and documents, ensuring that files appear properly in both public and private chat contexts.

- **Active User List:** Confirmed that the user list updates when users join or leave the chatroom.

2. Error-Handling Tests

- Attempted to enter a duplicate username → system rejected it and displayed a pop-up: “Username already taken.”
- Tried sending a private message to a non-existent username → client displayed an error: “User not found.”
- Simulated unstable network conditions → System attempted reconnection as expected.
- Sent malformed emoji codes (e.g. :note-emoji:) → message delivered as plain text.
- Closed the client abruptly (force quit) → server detected unexpected disconnect and removed the user from the active list.
- Attempted to access GUI functions while disconnected (sending messages, uploading files) → interface rejected actions and displayed: “Connection inactive.”

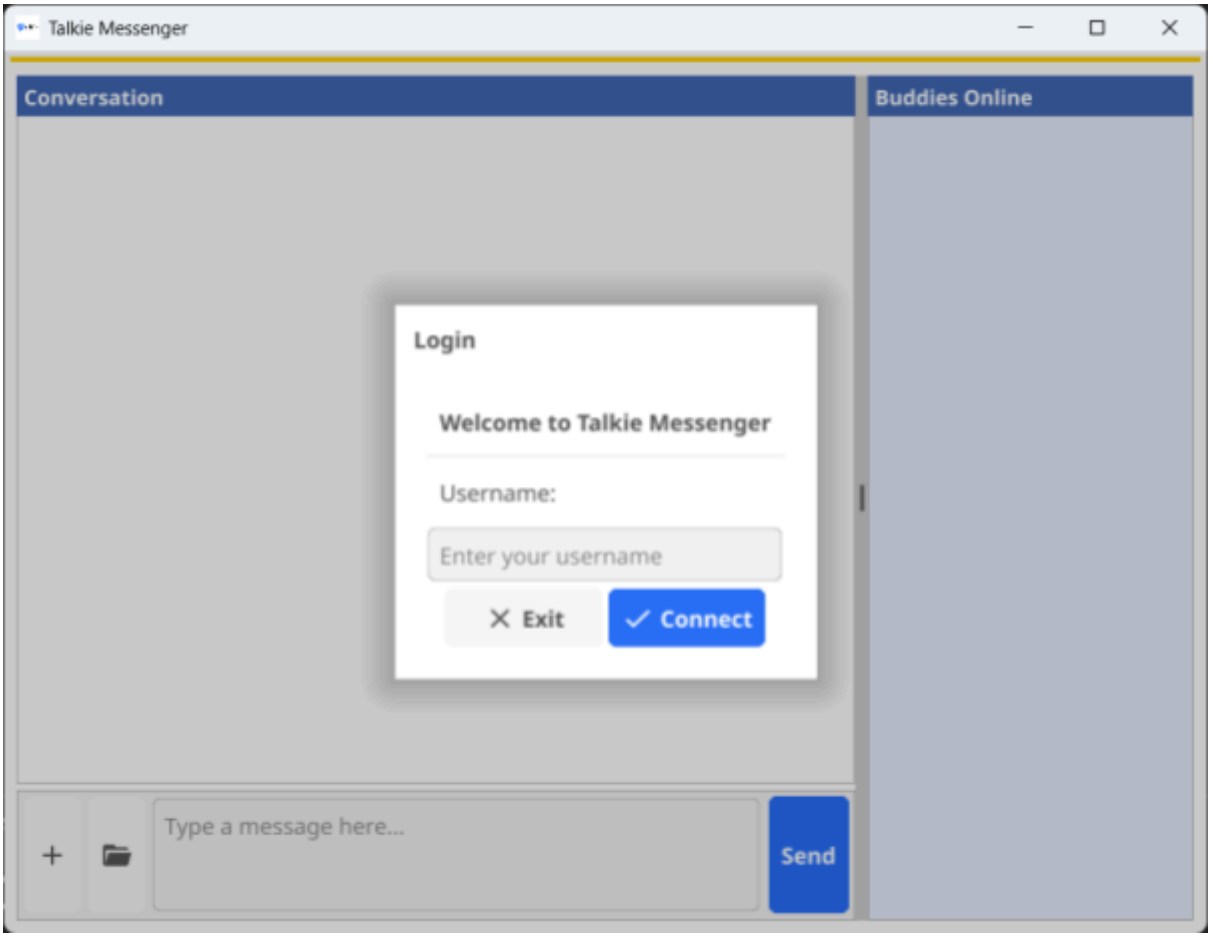
3. Results

All core functionalities performed correctly. A few minor UI refinement opportunities were identified, but they do not affect usability or system behavior.

Screenshots of GUI

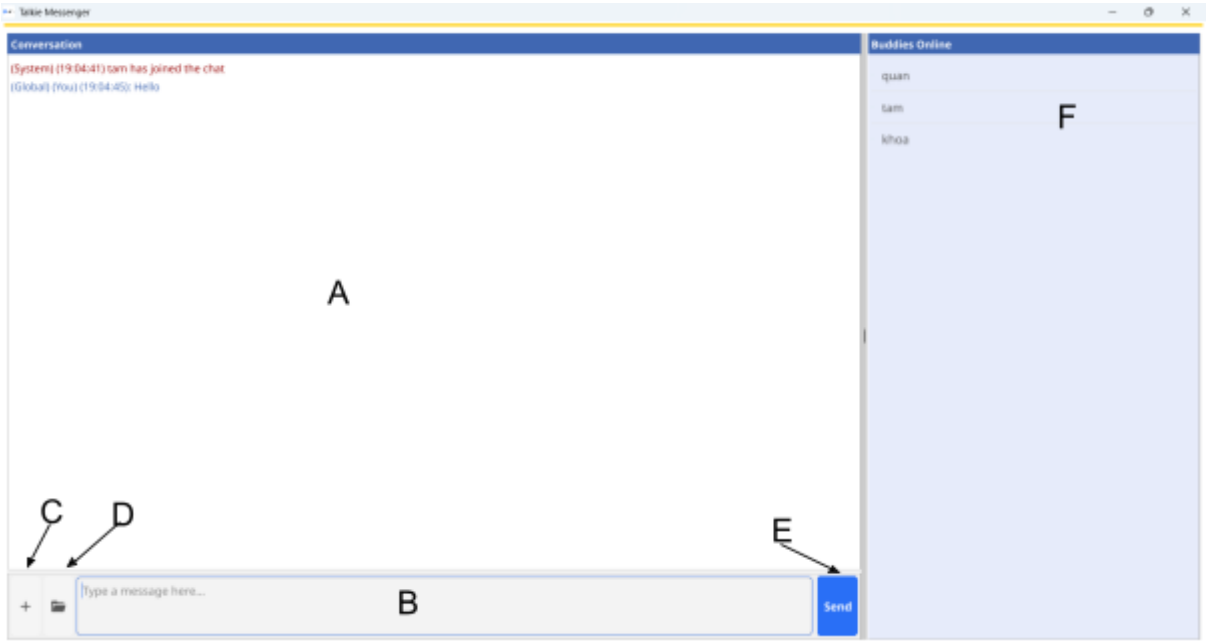
1. Login Screen

- Users are prompted to enter a username before joining the chatroom. The interface displays an error message if the username is empty or already taken.



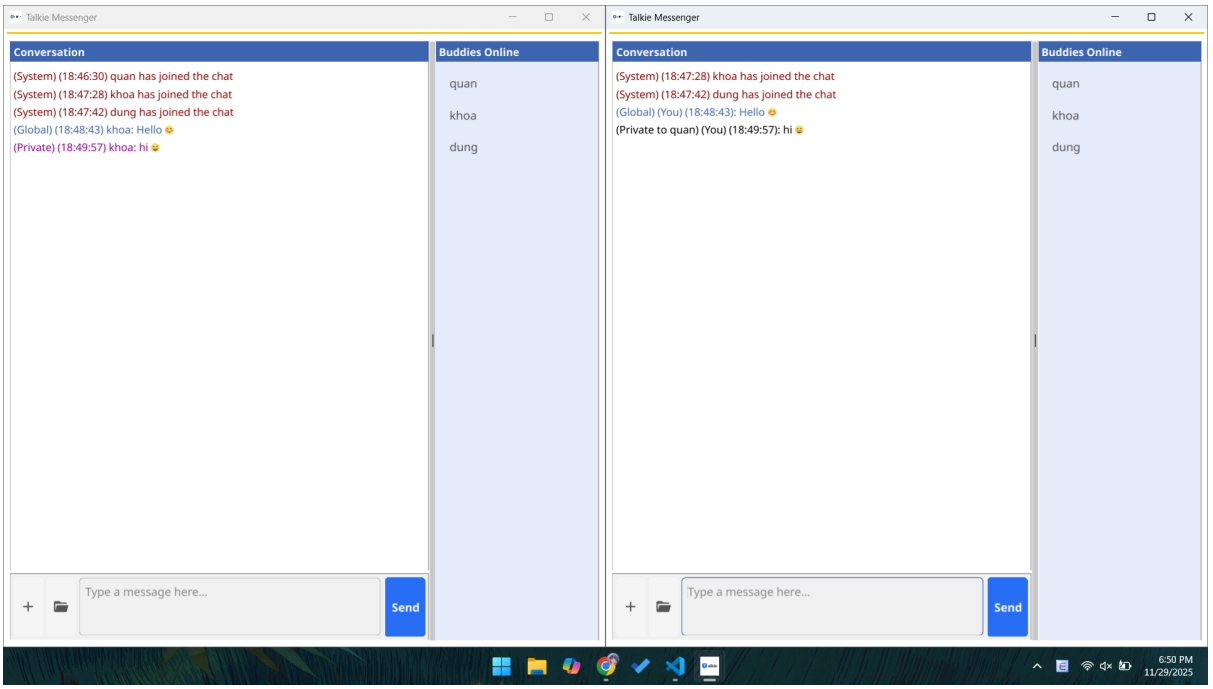
2. Main Chat Interface

- **A. Global Chat Window:** Displays all public messages exchanged among connected users.
- **B. Message Input Box:** Field for composing messages, including text, emojis, and attachments.
- **C. Emoji Picker:** A clickable icon that opens an emoji selection panel.
- **D. File Upload Button:** Allows users to attach images or documents.
- **E. Send Button:** Sends the current message to the server.
- **F. Active User List:** Shows the usernames of all currently connected participants.



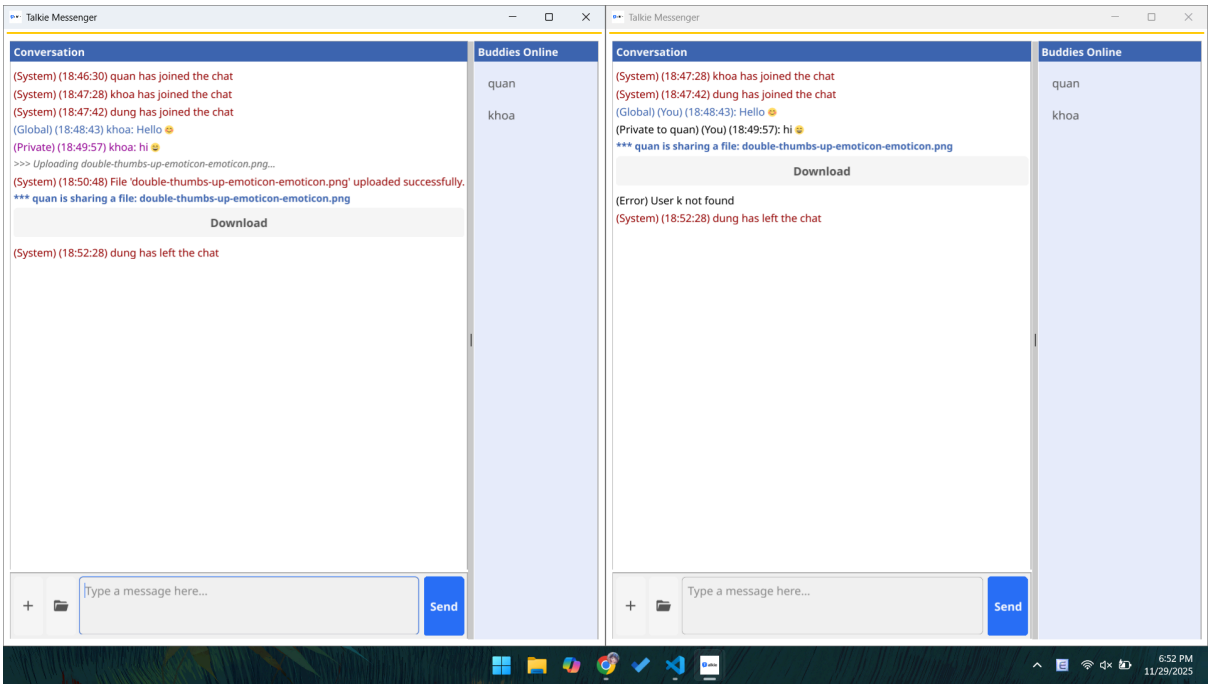
3. Private Messaging Panel

- Shows a one-to-one conversation between the current user and another selected user.
- Includes all features from the main chat (emoji, file upload, typing area).



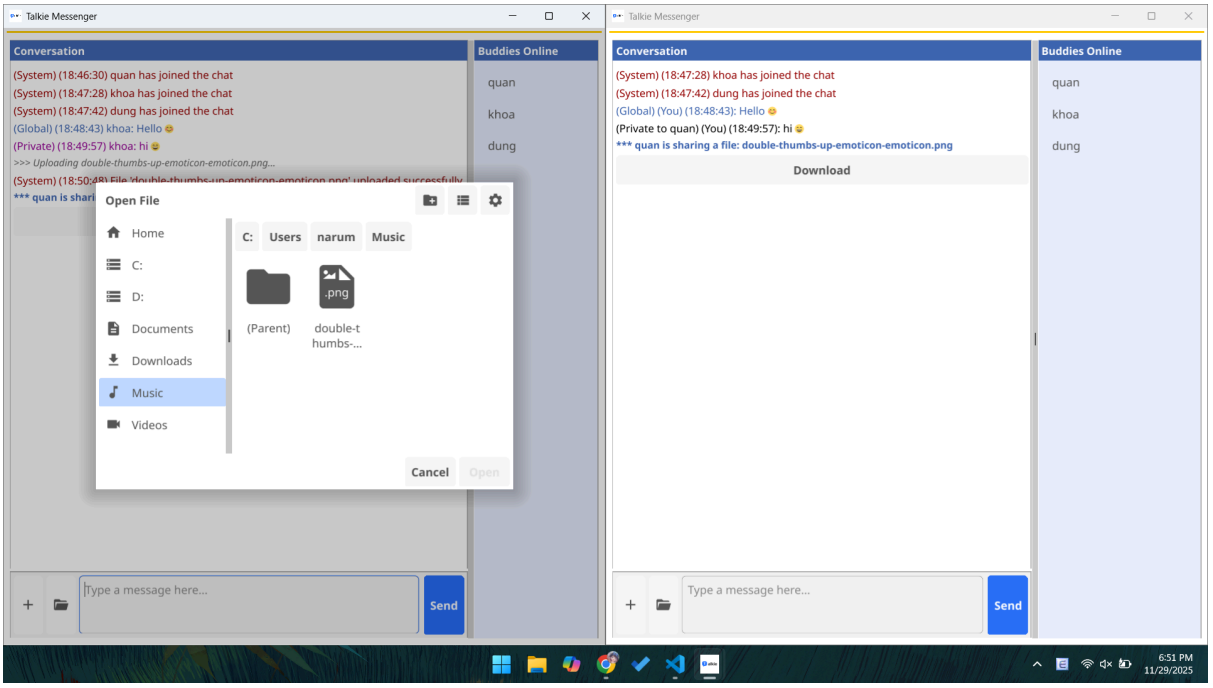
4. System Messages

- Displays join/leave notifications and system-level announcements.
- Helps users identify when others enter or exit the chatroom.



5. File & Media Preview

- Sent files appear inside the chat window with a preview (for images) or download link (for other file types).



Conclusion

This project successfully delivers a functional real-time chatroom application with the following key features:

- Secure user authentication
- Public and private messaging
- Emoji support for enhanced user interaction
- File sharing capabilities
- Dynamic active user list
- Real-time updates
- Clean and responsive graphical user interface

Future Enhancements

To further improve the chat application, the following extensions are recommended:

- **Chat history storage** using a database (MongoDB, PostgreSQL, etc.).
- **Typing indicators** show when a user is composing a message.
- **Dark mode** for better accessibility.
- **Admin tools** such as muting or removing users.
- **Improved file handling**, including upload progress bars and size validation.