Chat Application Development Report

Riabichenko Maksym, Rubchev Dmytro

May 14, 2025

1 Introduction to the Idea

The goal of this project was to develop a secure, real-time chat application capable of supporting both one-on-one and group messaging, with additional functionality for file sharing. The system comprises a client-side graphical user interface (GUI) built with the Iced framework and a server-side component using Tokio for asynchronous networking. Messages and files are encrypted using the AES-256-GCM algorithm to ensure privacy, and a SQLite database is used for persistent storage of users, groups, messages, and files. The application supports user registration, login, group creation, and management, making it a versatile tool for secure communication.

2 Requirements

The chat application was designed to meet the following high-level requirements:

- **User Authentication**: Users must register and log in using a username and public key for secure identification.
- **Real-Time Messaging**: Support for sending and receiving text messages in real-time for both individual and group chats.
- **File Sharing**: Users can send and receive files securely, with files stored on the server and downloaded to a local directory.
- **Group Management**: Users can create groups, add or remove members, and delete groups.
- **Data Security**: All messages and files are encrypted before transmission and decrypted upon receipt.
- **Persistent Storage**: User data, messages, and files are stored persistently using a lightweight database.
- **User Interface**: A responsive GUI allows users to interact with the application seamlessly.

These requirements aimed to balance functionality, security, and usability while keeping the system lightweight and scalable.

3 Design Diagram

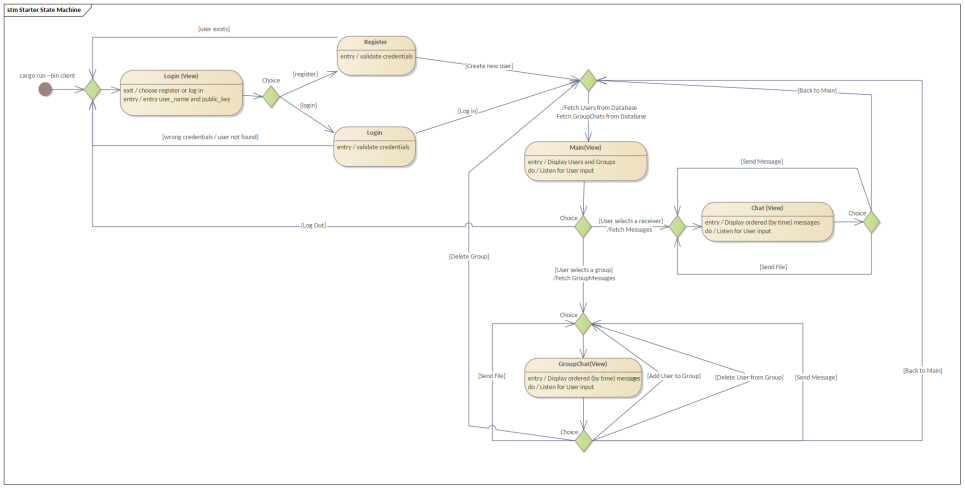

Figure 1: High-level design diagram of the chat application

4 Design Choices

Several key design choices were made during development:

- **Tokio for Networking**: Tokio was used for asynchronous TCP communication due to its robust async runtime and compatibility with Rust. Actix was an alternative, but Tokio's flexibility and lower-level control were preferred for custom networking logic.
- **AES-256-GCM for Encryption**: This algorithm was chosen for its strong security and performance. Alternatives like ChaCha20-Poly1305 were considered, but AES-256-GCM's widespread adoption and hardware acceleration support made it the preferred choice.
- **SQLite for Storage**: SQLite was selected for its simplicity and lightweight nature, suitable for a small-scale application. PostgreSQL was an alternative, but its setup complexity was unnecessary for the project's scope.
- **Iced for GUI**: Iced was selected for its simplicity and native Rust integration, providing a reactive GUI framework.

These choices prioritized performance, security, and ease of development while maintaining scalability.

5 Dependencies and Their Usage

The project relies on several Rust crates:

- **iced**: Provides the GUI framework for the client, handling widgets, state management, and rendering.
- **tokio**: Enables asynchronous networking for the server, managing TCP connections and message passing.

- **serde**: Facilitates JSON serialization and deserialization for client-server communication.
- **uuid**: Generates unique identifiers for users, groups, messages, and files.
- **ring**: Implements AES-256-GCM encryption for secure message and file transmission.
- **rusqlite**: Interfaces with SQLite for persistent storage of application data.
- **anyhow**: Simplifies error handling across the application.
- **base64**: Encodes and decodes binary data for transmission.
- **chrono**: Handles timestamp formatting for messages and files.

Each dependency was chosen to address specific functional needs while maintaining compatibility with Rust's ecosystem.

6 Evaluation

6.1 What Went Well

- **Security**: AES-256-GCM encryption ensured secure communication, and SQLite provided reliable data persistence.
- **Performance**: Tokio's async runtime handled multiple client connections efficiently, and Iced delivered a responsive GUI.

6.2 What Went Not So Well

- **Limited GUI Features**: Iced lacks some advanced features, limiting UI customization.
- **Error Handling**: Integrating multiple crates required complex error handling, as each had its own error types, leading to verbose code in some areas.