VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

B.Tech. Winter Semester 2024-25
School Of Computer Science and Engineering (SCOPE)

Digital Assignment - I
Information Security

Apurva Mishra: 22BCE2791
Date: 26 March 2025

Contents

1 Chat App

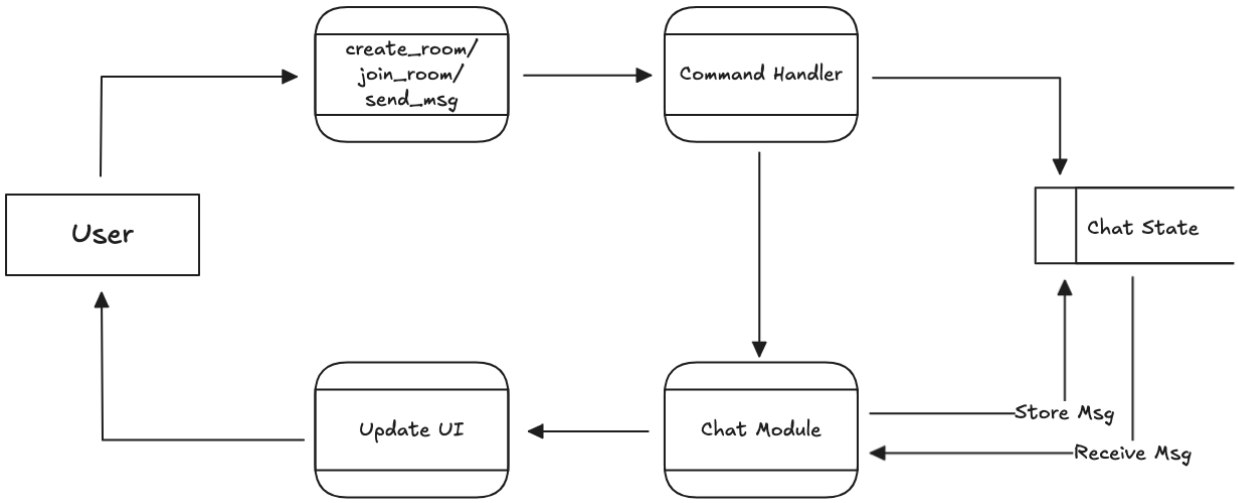
A chat application which is end to end encrypted and does not require login to ensure complete user privacy

Code: https://github.com/mav3ri3k/p2p-chat

1.1 Stack / Technical Coverage

- **Full Stack**: The project covers both frontend (UI, user interaction) and backend (networking, state management) development.
- **Frontend Stack**: React, TypeScript, Vite, Deno - a modern web technology stack.
- **Backend Stack**: Rust, Tokio (async runtime), iroh (P2P networking library based on QUIC).
- **Integration Framework**: Tauri is used effectively to bridge the Rust backend and the web frontend, handling inter-process communication and packaging the application for desktop use.
- **Networking Layer**: Utilizes QUIC via iroh and defines a simple application-layer protocol on top.

1.2 Data Flow Diagram



1.3 Module Contributions

Implementation of core backend logic and communication.

1. **Backend Core & Networking**:
    - Implemented the core P2P connection logic using the iroh Rust library.
    - Developed the node setup, binding, and connection establishment mechanisms for both creating (“hosting”) and joining chat rooms via NodeTickets.

- Managed the asynchronous tasks for accepting connections, handling handshakes, and continuously reading incoming messages from the peer.
- Implemented the backend state management (ChatState) to hold the active connection's send stream.

2. **Tauri Framework Integration**:
   - Configured the Tauri application structure, including build settings (tauri.conf.json) and Rust dependencies (Cargo.toml).
   - Defined and implemented the Tauri commands (create_chat_room, join_chat_room, send_message) to expose backend functionality to the frontend.
   - Utilized Tauri's state management (tauri::State) to share the ChatState across command handlers.
   - Implemented the event emission (app_handle.emit("new-message", …)) from the backend to notify the frontend of incoming messages.

1.4 Knowledge Gained

- **Rust & Asynchronous Programming**: The backend is written in Rust, utilizing Tokio for asynchronous operations, which is essential for handling network I/O non-blockingly
- **P2P Networking**: The project uses the iroh library for establishing peer-to-peer connections over QUIC.
- **Tauri Framework**: Integrating a Rust backend with a web frontend (React/TypeScript) using Tauri which allows for cross-platform applications.
- **Frontend Technologies**: Use of React with TypeScript, Vite for bundling, and managing dependencies, and Deno as javascript runtime.

Link to Github Commits

1.5 Algorithm Description

- Uses Iroh/QUIC for encrypted transport.
- Asynchronous tasks handle connections and messages.
- Event Driven Architecture

There are 4 core events:

1. Create Chat Room:
   1. Get/Generate Secret Key.
   2. Create Iroh Endpoint (with key, ALPN).
   3. Bind Endpoint.
   4. Get Node Address (from relay).
   5. Create NodeTicket.

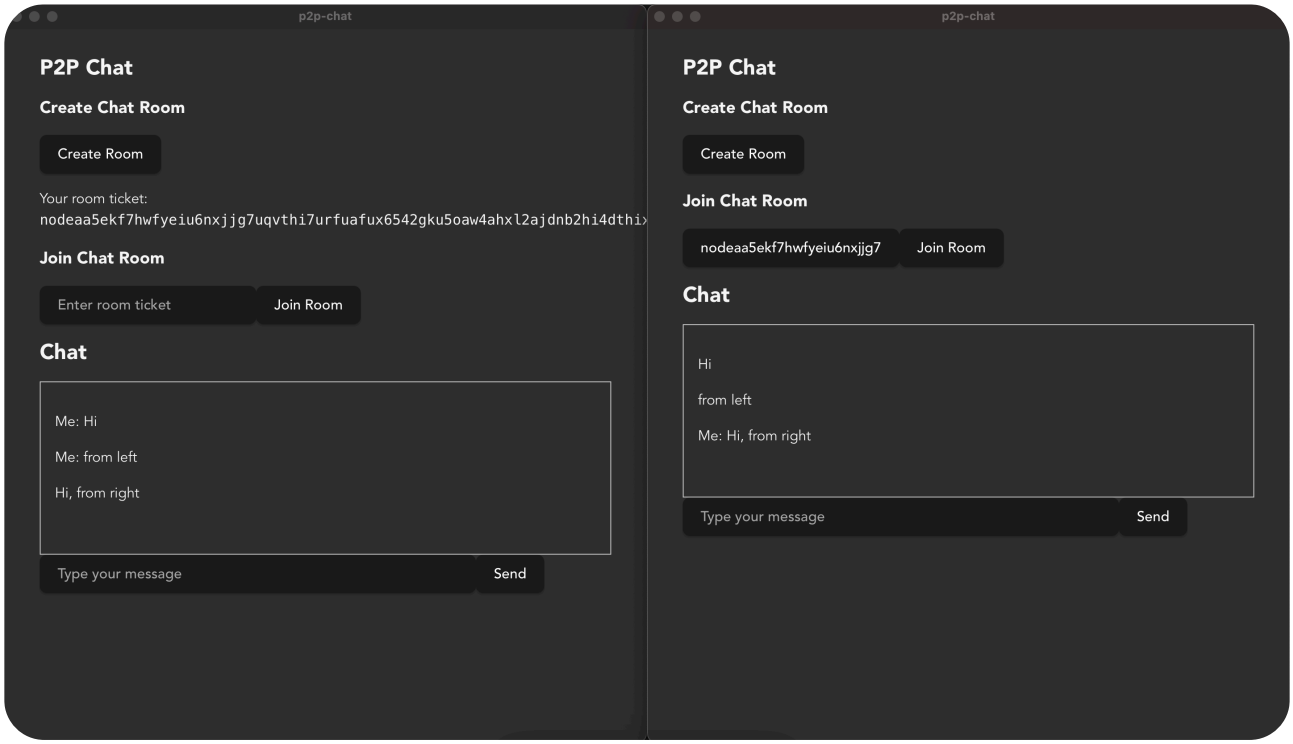
6. **Async Task:** Accept connection, bi-directional stream. Read/ Verify Handshake.
7. **Async Task:** Read messages, emit “new-message” event. Store SendStream in ChatState.

2. Join Chat Room:
   1. Parse NodeTicket.
   2. Get/Generate Secret Key.
   3. Create Iroh Endpoint
   4. Bind Endpoint.
   5. Connect to peer (using NodeID).
   6. Open bi-directional stream. Send Handshake.
   7. **Async Task:** Read messages, emit “new-message” event. Store SendStream in ChatState.

3. Send Message:
   1. Lock ChatState.
   2. Lock SendStream (from ChatSession).
   3. Write message to SendStream.
   4. Release Locks.

4. Receiving a Message:
   1. Read from RecvStream: Asynchronously read data from the `RecvStream` of the established bi-directional QUIC stream.
   2. Decode received bytes to to string.
   3. Emit Tauri Event for emit a “new-message” event.
   4. The frontend is subscribed to the “new-message” event. Upon receiving the event, the frontend updates the UI.
   5. Loop: Repeat steps 1-4 to continuously listen for new messages on the stream.

1.6 Output



1.7 Information Security

The implementation is fundamentally based on information security and the CIA Triad:

- **Confidentiality:** Use of TLS encryption, protecting the confidentiality of data in transit. The use of `SecretKey` and `PublicKey` pairs ensures that only the intended recipient can decrypt the data.
- **Integrity:** QUIC provides built-in integrity protection through its authenticated encryption.
- **Availability:** The combination of P2P architecture, QUIC's resilience to network changes, and the use of relay servers as a fallback mechanism enhances availability.
- **Authenticity:** QUIC provides strong **peer** authentication. The connecting node verifies the identity of the other node using its `PublicKey` (NodeId).
- **Accountability:** Requires user authentication as a prerequisite.
- **Privacy:** Encryption protects the content.