Real-Time Chat Application - Detailed Project Report

Author: Nikhil Rupala | Organization: Elevate Labs | Internship Project

Executive Summary

This document provides an in-depth technical report of the Real-Time Chat Application developed in Java during an internship at Elevate Labs. It covers detailed design decisions, system architecture, class responsibilities, networking protocol, encryption strategy, persistence, threading model, error handling, testing, deployment instructions, and potential future improvements. The objective is to give a comprehensive reference that can be used to maintain, extend, or hand over the project.

Table of Contents

1. Introduction and Goals

The Real-Time Chat Application provides instant messaging functionality between multiple clients using a centralized Java server. The project aims to be a learning and production-ready prototype demonstrating: socket programming, secure message transmission (AES), concurrency handling for multiple clients, persistence of chat history, and a responsive user interface. Primary goals include security, reliability, maintainability, and clarity of code for internship assessment and further development.

2. Functional and Non-Functional Requirements

Functional Requirements: • Multiple clients can connect to the server simultaneously. • Clients can send and receive messages in real-time. • Messages are delivered to intended recipients (broadcast or private). • Message history is persisted and retrievable. • Basic GUI to display messages, input text, and show connected users. Non-Functional Requirements: • Secure transmission using AES encryption to protect messages in transit. • Server must be able to handle concurrent client connections with minimal latency. • Graceful handling of client disconnects and reconnections. • Clear logging for debugging and monitoring. • Buildable with Gradle/Maven and runnable on standard Java runtimes.

3. High-Level Architecture

The application follows a modular client-server architecture. Key layers: - Network Layer: Socket-based TCP connections using java.net.Socket / ServerSocket. - Application Layer: Protocol for message framing, routing, and commands. - Security Layer: EncryptionUtil for AES encrypt/decrypt operations. - Persistence Layer: ChatDatabase for storing messages and metadata (file-based or SQLite). - Presentation Layer: ChatGui (Swing/JavaFX) for user interaction. At a high level: Clients <--> Server (accepts connections, authenticates, routes messages) <--> Database

4. Detailed Design

4.1 Components and Class Responsibilities

Server.java - Responsible for initializing ServerSocket, accepting new client connections, handing each to ClientHandler. - Holds server-side collections (e.g., Map) to track active clients. ClientHandler.java - Runs on a separate thread per client; listens for incoming messages, decrypts, validates and forwards them. - Handles client lifecycle events (connect, heartbeat, disconnect). ChatServer.java - Business logic for broadcasting messages, routing private messages, handling server commands (shutdown, list users). ChatClient.java - Client-side networking: connects to server socket, sends encrypted messages, and listens for incoming messages. ChatGui.java - Presents message view, input box, send button, user list. Acts as controller for user actions and delegates network calls. ChatDatabase.java - Interface/implementation for storing message history: append-only logs or SQLite tables with indexes on sender, timestamp. EncryptionUtil.java - AES encryption/decryption with a secure mode (CBC/GCM suggested). Handles IV generation and provides helpers to convert bytes<->base64. ServerConfig.java / ClientConstants.java - Centralized configuration: ports, host, buffer sizes, timeouts, encryption parameters (e.g., key length). MessageReceiver.java - Utility to parse incoming byte streams and reassemble framed messages reliably.

4.2 Data Models and Message Formats

Data Model (Message): - messageId: UUID - type: ENUM {TEXT, IMAGE, FILE, CONTROL} - senderId: string (username or unique id) - receiverId: string (username or 'ALL' for broadcast) - timestamp: ISO-8601 string or epoch milliseconds - payload: encrypted string (base64) - metadata: optional JSON (e.g., filename, size) Example JSON message (before encryption): { "messageId":"c2f1e5a4-...", "type":"TEXT", "senderId":"alice", "receiverId":"bob", "timestamp":169..., "payload":"Hello Bob!", "metadata":{} } Framing Protocol over TCP: - Each message is prefixed with a 4-byte length header (big-endian). - The receiver reads length, then reads the exact number of bytes to reconstruct the message. - This prevents partial reads causing message corruption.

4.3 Networking Protocol and Socket Handling

Connection Lifecycle: - Client connects to server TCP socket on configured port. - Optional: perform a handshake to exchange clientId and public metadata. - Symmetric AES key: can be pre-shared via ServerConfig for the prototype, or derived via an asymmetric key exchange for production (RSA/ECDH). - Once connected, client sends framed, encrypted messages to server. Server decrypts, checks routing, and sends to recipients. Socket Handling Best Practices Used: - Use BufferedInputStream and BufferedOutputStream for efficient IO. - Implement timeouts and keep-alives to detect dead peers. - Use a dedicated reader thread per socket to avoid blocking the main accept loop. - Protect shared maps with ConcurrentHashMap to avoid race conditions.

4.4 Encryption and Key Management

Encryption Approach (Prototype): - AES-256 recommended; AES-128 acceptable for lightweight prototypes. - Use AES in GCM mode (AES/GCM/NoPadding) for authenticated encryption (integrity + confidentiality). - Generate a random IV (12 bytes recommended for GCM) per message and include it (unencrypted or alongside the ciphertext) in the message metadata. - Derive symmetric key via secure channel in production. For this internship prototype, a static key defined in ServerConfig.java is used (NOT recommended for production). Message Encryption Steps: 1. Serialize message JSON to bytes (UTF-8). 2. Generate random IV. 3. Encrypt using AES/GCM with key + IV producing ciphertext + auth tag. 4. Prepend or attach IV to ciphertext (e.g., iv:ciphertext, base64 encoded). 5. Frame the encoded message and send over TCP. Key Storage and Security Notes: - Never commit raw keys to version control. Use environment variables or external secrets manager. - Rotate keys periodically. - For production, use TLS or implement an asymmetric key exchange (RSA/ECDH) to establish per-session AES keys.

4.5 Threading & Concurrency Model

Threading Model: - The Server's accept loop runs on the main server thread and offloads each accepted socket to a new ClientHandler thread (or thread pool). - ClientHandler handles reading from the socket and dispatching messages to the ChatServer logic. - For broadcasting, ChatServer iterates over active ClientHandlers and enqueues outbound messages; writing to sockets should be thread-safe. - Use ConcurrentHashMap for active clients, and synchronized blocks or locks only when updating shared mutable state that isn't concurrent-friendly. Potential Enhancements: - Use a fixed-size thread pool (ExecutorService) to limit resource consumption. - Decouple network IO from processing using producer-consumer queues (LinkedBlockingQueue). - Implement backpressure controls if clients are slow to consume messages.

4.6 Persistence Layer (ChatDatabase)

Design Choices: - Lightweight approach: append-only logs per chat or a single SQLite DB with tables (messages, users, sessions). - Recommended schema for SQLite 'messages' table: id INTEGER PRIMARY KEY AUTOINCREMENT, messageId TEXT UNIQUE, sender TEXT, receiver TEXT, timestamp INTEGER, payload TEXT -- store encrypted base64 payload - Index on timestamp and sender for efficient queries. - For large scale, move to an actual DB server (Postgres/MongoDB) depending on query patterns. Backup & Retention: - Implement periodic backups of DB files. - Provide retention policies (e.g., purge messages older than X days) configurable in ServerConfig.

4.7 GUI Design and Event Flow

GUI Requirements: - Display message list with timestamps and sender labels. - Input box for typing messages + send button + enter-key support. - Panel or list showing active users and connection status. - Notification area for errors (e.g., 'Disconnected', 'Message failed to send'). Event Flow Example (Send Message): 1. User types message and clicks Send. 2. GUI constructs message JSON and hands it to ChatClient.sendMessage(message). 3. ChatClient serializes and encrypts the message and writes framed bytes to socket output stream. 4. Server receives, decrypts, processes, and forwards to recipient handlers. 5. Recipient ClientHandler sends bytes to respective ChatClient instance which decrypts and forwards to GUI for display.

5. Logging, Monitoring & Error Handling

Logging: - LoggerUtil (in 'common') abstracts logging to file and console. - Log levels: INFO (normal operations), DEBUG (detailed), WARN (recoverable issues), ERROR (fatal issues). - Important events to log: server start/stop, client connect/disconnect, auth failures, message delivery failures, exceptions. Monitoring: - Simple metrics: connectedClientsCount, messagesPerMinute, failedDeliveriesCount. - Expose a minimal admin endpoint or CLI commands to print runtime stats for the server. Error Handling: - Validate incoming message format and reject malformed messages with an error code. - Handle IOExceptions on socket reads/writes by closing and cleaning up client state. - Implement retries for transient failures in sending messages; apply exponential backoff. - Ensure database operations are wrapped with try/catch and provide fallback (e.g., write to local failed-ops queue).

6. Testing Strategy & Results

Unit Tests: - Test EncryptionUtil: encrypt -> decrypt returns original payload. Test with edge cases (empty, large payloads). - Test MessageReceiver: ensure framing and parsing works for partial stream reads. Integration Tests: - Start server locally and run multiple client instances to test multi-client communication. - Simulate network delays and packet fragmentation by sending partial bytes and verifying reassembly. Manual Tests Performed: - Verified connecting 5+ clients concurrently, message broadcast and private messages. - Verified persistence by restarting server and confirming history retrieval. - Simulated abrupt client disconnects and observed server state cleanup. Test Results Summary: - Encryption round-trips succeeded for regular message sizes. - Framing protocol handled partial reads when simulated. - Observed increased CPU/memory when scaling beyond tens of clients on a single-thread-per-connection model; recommended thread-pool improvements for scale.

7. Build, Run & Deployment Instructions

Prerequisites: - Java JDK 11+ (preferably 17) - Gradle or Maven installed (if using provided build files) Build: - Using Gradle: run `./gradlew build` in project root (ensure build.gradle present). - Using Maven: run `mvn clean package` if pom.xml provided. Configuration: - Edit ServerConfig.java or externalize settings in `config.properties`: server.port=9090 server.host=0.0.0.0 encryption.key=BASE64_KEY_PLACEHOLDER db.path=./data/chat.db Run Server (example): - `java -jar build/libs/chat-server.jar` or run from IDE by executing Server.main() Run Client (example): - `java -cp build/libs/chat-client.jar com.example.chat.ChatClient` - Provide username and server host: `--host 127.0.0.1 --port 9090 --username alice` Notes on Deployment: - For production, run server behind a process manager (systemd, Docker container, or Kubernetes). - Bind to internal interface and place behind TLS termination (NGINX) or enable TLS at application layer.

8. Security Considerations

Key Security Points: - Do not hardcode symmetric keys or secrets in source control. Use environment variables or secret stores. - Use AES/GCM for authenticated encryption to prevent tampering. - Validate payload sizes to mitigate denial-of-service attacks via huge messages. - Sanitize any metadata used to write to filesystem to avoid path traversal. - Limit accepted client rate per IP to prevent brute-force or spam. - Implement authentication and authorization for future versions (JWT or OAuth for web integration). Regulatory & Privacy: - If storing user messages, comply with data protection regulations and implement policies for data retention, deletion requests, and backups.

9. Performance Considerations & Optimization

Observations: - Thread-per-connection can exhaust resources with many clients; prefer NIO or thread pools for scale. - Use batched writes where possible to reduce system call overhead. - Use efficient binary protocols (compact framing) to reduce payload size. - Consider message compression for large payloads (images/files) before encryption. Profiling Tips: - Use Java Flight Recorder or VisualVM to profile hotspots. - Monitor GC pauses; tune JVM flags for server workload.

10. Future Work & Extensions

Prioritized Enhancements: 1. Authentication & Authorization: add login, password hashing (bcrypt), and session management. 2. WebSocket Support: add HTTP/WebSocket endpoint to support web clients and mobile apps. 3. Group Chats & Channels: add structured rooms with membership management. 4. TLS/SSL: enable end-to-end TLS for transport layer security. 5. Scalable Persistence: integrate Postgres or NoSQL for horizontal scaling and search. 6. Media Transfer: optimize file transfers with chunked uploads and resumable transfers. 7. CI/CD: add automated tests and deployment pipelines for reliability.

11. Appendix: Sample Configs, Commands, and Snippets

Below are useful snippets and a sample configuration to help testers and developers run the project quickly.

```
# config.properties (example)
server.port=9090
server.host=0.0.0.0
encryption.key=BASE64_KEY_PLACEHOLDER    # store securely in env vars in production
db.path=./data/chat.db
log.path=./logs/server.log
max.clients=1000
message.max.size=65536
```

```
// Example message JSON (before encryption)
{
  "messageId":"uuid-v4",
  "type":"TEXT",
  "senderId":"alice",
  "receiverId":"ALL",
  "timestamp":169...,
  "payload":"Hello world!",
  "metadata":{}
}
```

Sample AES usage (simplified):

```
// EncryptionUtil.java - sample encrypt/decrypt (simplified)
Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
GCMParameterSpec spec = new GCMParameterSpec(128, ivBytes);
cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec, spec);
byte[] cipherBytes = cipher.doFinal(plainBytes);
// transmit: base64(iv) + ":" + base64(cipherBytes)
```

```
# Build (Gradle)
./gradlew clean build
# Run server (example)
java -jar build/libs/chat-server.jar
# Run client (example)
java -jar build/libs/chat-client.jar --host 127.0.0.1 --port 9090 --username alice
```

End of Report