Real-time Chat Application

Submitted by:

Mahitha Pasupuleti

867114134

CPSC-597

Fall 2025

Instructor: Rong Jin

Department of Computer Science

California State University-Fullerton

Table of Contents

Abstract

This project successfully developed an open-source, real-time chat application designed to address the limitations of transparency and customizability found in proprietary solutions like Slack and WhatsApp. The final system is a full-stack application built on a modular architecture using **Node.js, Express, React, and MongoDB**. Key achievements include the implementation of low-latency messaging using **Socket.IO**, secure **JWT-based authentication**, and persistent data storage. The application supports one-to-one messaging and domain-specific group chats (e.g., Technology, General). This platform now serves as a functional communication tool and an educational reference for distributed system design.

Introduction

1. Problem Statement

Real-time chat applications have become integral to how individuals and organizations communicate, but many existing solutions fall short in terms of flexibility, transparency, and educational value. Leading platforms like WhatsApp, Slack, and Discord are closed-source, which limits both customization for specific domains (like healthcare, education, or internal enterprise tools) and the opportunity to understand how such systems are built and secured.

From a development and research perspective, these tools offer little insight into the complexities of real-time communication, such as latency handling, data consistency, state management, and scalability under load. Moreover, the rise in concerns around data privacy and control further emphasizes the need for secure, open, and customizable communication tools.

This project proposes to build a scalable, full-stack real-time chat application from scratch using modular and decoupled components. These include WebSocket-based communication (Socket.IO), session state management with Redis, and secure authentication using JWT. The backend will run on a Node.js and Express.js stack, with MongoDB for data storage. While not a full microservices system, the architecture is modular and scalable, laying a strong foundation for transitioning to a microservices-based design in the future.

Unlike existing proprietary platforms, this project will be fully transparent in its architecture, technology choices, and implementation. It not only addresses core technical challenges like message ordering, responsiveness under load, and secure media handling, but also serves as a learning tool for students and developers interested in distributed system design and secure real-time communication.

The project is both practical and research-oriented: it solves real problems in communication systems while also providing an extensible codebase for future enhancements, such as video/voice integration, CI/CD deployment pipelines, and advanced state synchronization.

2. The Solution

I have delivered a fully functional, transparent, and scalable chat application that bridges the gap between academic theory and production-grade software. Unlike monolithic proprietary software, where backend operations are often treated as "black boxes," this project exposes the internal mechanisms of **real-time state management**, giving developers and auditors full visibility into how data is queued, processed, and delivered.

The solution specifically addresses three critical challenges inherent in modern communication systems:

- **Latency Responsiveness:** By abandoning traditional HTTP polling in favor of persistent **WebSocket connections** (via Socket.IO), the system achieves near-instantaneous, bi-directional communication. This architecture minimizes handshake overhead and ensures that message delivery remains fluid and immediate, even during bursts of high-volume traffic.
- **Architectural Transparency:** The system utilizes a **modular, decoupled design** that strictly separates the React frontend from the Node.js backend services. This structure not only simplifies maintenance and debugging but also provides a flexible foundation for future scalability. It allows individual components, such as the authentication service or

the chat engine, to be scaled or upgraded independently without disrupting the overall system availability.

3. Objectives

The application follows a **Modular Microservices-Based approach**. While designed for eventual full microservices deployment, the current implementation leverages a service-oriented architecture where the backend components are decoupled to ensure scalability and ease of maintenance. The separation of concerns between authentication, messaging, and data persistence guarantees high availability and independent deployment potential.

3.1 High-Level Design

The system relies on a bi-directional communication channel between the **React Client** and the **Node.js Server**.

- **Client:** Handles UI rendering, user state management, and initiates both HTTP requests (for initial data and authentication) and socket events (for real-time communication).
- **Server (API Gateway):** Serves as the unified entry point. It manages standard **REST API endpoints** for user authentication (Login/Signup) and session management, and simultaneously handles the persistent **WebSocket connections** for the real-time Chat Service.
- **Database: MongoDB** is the core persistence layer, storing all user profiles and the complete message history.

The **Communication Flow** for a message is as follows:

1. **Handshake:** The client initiates a WebSocket handshake request to the server.

2. **Authentication:** The server verifies the user's identity by extracting and validating the **JSON Web Token (JWT)**. Only upon successful authentication is the connection upgraded from HTTP to a persistent, secure WebSocket (WSS) connection.
3. **Event Emitting:** When a user types a message and clicks 'Send', the client emits a custom sendMessage event, carrying the message content and the target Room ID.
4. **Persistence & Broadcasting:** The server receives the event, performs necessary validation, saves the message to **MongoDB** for persistence, and then uses the Socket.IO broadcasting mechanism to relay the message instantly to all other users connected to that specific Room ID. This fan-out approach ensures minimal delay.

4. Technology Stack

The project utilizes the MERN stack extended with real-time capabilities.

Component	**Technology**	**Rationale**
Frontend	React.js	Component-based UI for dynamic state management.
Backend	Node.js + Express	Event-driven, non-blocking I/O is ideal for high concurrency.
Protocol	Socket.IO	Enables real-time, bi-directional event-based communication.
Database	MongoDB	Schema-less storage for flexible message history.
Auth	JWT + Bcrypt	Stateless authentication and secure password hashing.
Caching	Redis	(Optional Integration) Used for session caching and performance.

5. Key Features & Implementation

Most platforms abstract this away without giving developers a chance to study or modify the logic. However, understanding and building this yourself offers deep insight into distributed state systems, event-driven architectures, and pub/sub mechanisms.

5.1 Project Activities

FR-1: User Authentication

- Implementation: Users register via email/password. Passwords are hashed using bcrypt before storage.
- Security: On login, the server issues a JSON Web Token (JWT). This token is passed in HTTP headers for REST calls and validated during the Socket handshake.

FR-2: Real-Time Messaging

- Implementation: Leverages socket.io-client on the frontend and socket.io on the server.
- Latency: Messages are delivered instantly without page refreshes.
- Visual Indicators: The UI updates immediately upon receipt of a receiveMessage event.

FR-3: Group and One-to-One Chats

- One-to-One: A unique room ID is generated based on the two user IDs (e.g., userA_userB).
- Group Chat: Pre-defined rooms (Technology, General, Innovation) are created. Users "join" these rooms upon selection in the UI.

FR-4: Message Persistence

- Workflow: Every message sent via Socket is asynchronously written to the MongoDB messages collection.

- History: When a user enters a chat, a REST API call retrieves the chat history, preserving the context.

6. Database Schema

The application’s data persistence layer is built on MongoDB, managed via Mongoose. The schemas below represent the structure used to enforce data integrity and manage relationships across the application.

6.1. Schema 1: User (Auth & Profile)

This schema handles secure user credentials and token management.

Field	**JSON Type**	**Description**
username	String	Unique, required, indexed for fast lookups.
password	String	Required. Stored exclusively as a **Bcrypt Hash** (never plain text).
refreshToken	String	Used for generating new **JWTs** without re-login.
createdAt	Date	Timestamp recording user registration.

6.2. Schema 2: Message (One-to-One Chat)

This schema stores all private messages between two individuals.

Field	**JSON Type**	**Description**
conversationId	String	The unique Room ID generated from the two user IDs.

senderId	ObjectId (Ref: User)	The ID of the user who sent the message.
recipientId	ObjectId (Ref: User)	The ID of the intended recipient user.
message	String	The message content (maximum 500 characters).
status	String (Enum)	Tracks state: 'sent', 'delivered', or 'read'.
timestamp	Date	Records the exact time of message creation.

6.3. Schema 3: Group Message (Channel Content)

This schema stores messages specific to defined group channels.

Field	**JSON Type**	**Description**
groupId	String (Ref: Group)	The identifier linking the message to the group metadata.
senderId	ObjectId (Ref: User)	Reference to the user who authored the message.
senderUsername	String	Stored directly for optimal display performance in group chats.
message	String	The message content.

timestamp	Date	Time of creation.

6.4. Schema 4: Group (Channel Metadata)

This schema defines and manages the structure and membership of all group channels.

Field	JSON Type	Description
name	String	The displayed name of the channel (e.g., "Technology").
members	Array [ObjectId (Ref: User)]	A list of all user IDs currently subscribed to the group.
admins	Array [ObjectId (Ref: User)]	A list of user IDs with administrative privileges within the channel.
createdAt	Date	The date the group was initialized.

7. Installation & Setup Guide

7.1 Prerequisites

Ensure the following software is installed: **Node.js (v14+)**, **npm**, **MongoDB**, and **Git**.

7.2 Step-by-Step Setup

7.2.1. Clone the Repository

- Bash
- git clone https://github.com/Mahitha-pasupuleti/Nihira.git
- Main URL : https://github.com/Mahitha-pasupuleti/Nihira

7.2.2. Backend Service Configuration

- **Navigate:** `cd ../backend`
- **Install Dependencies:** `npm install`
- **Create Environment File:** Create a file named `.env` and populate configuration details (MONGO_URI, JWT_SECRETS, PORT).
- **Start the Backend Server:** `npm run dev`

7.2.3. Frontend Client Initialization

- **Navigate:** `cd ../frontend`
- **Install Dependencies:** `npm install`
- **Start the Client Application:** `npm run dev`

7.2.4. Frontend Client Initialization

- Open http://localhost:5173 in your browser.
- Register a new account.
- Open a second browser (in incognito mode) and register a second account.
- Select the **"General"** channel to test group chat or select the user from the sidebar to test private messaging.

Conclusion

The "Real-time Chat Application" project has met all proposed objectives. The final deliverable is a robust, secure, and scalable communication platform. By utilizing a modular architecture, we ensured that the system is not only functional for current requirements but also extensible for future enhancements such as video calling or file sharing. This project stands as a successful implementation of modern distributed system principles, providing full transparency and control that proprietary systems lack.