P2P Distributed Chat: A Decentralized Communication System

Arthur Chen (ac820)
Shuqi Shen (ss1481)
Jay Yoon (jy320)
Jingheng Huan (jh730)

Abstract

This report presents a fully decentralized peer-to-peer (P2P) chat system developed as part of the CS512 Distributed Systems course. Unlike traditional chat architectures that rely on centralized messaging servers, our system utilizes WebRTC DataChannels for direct client-to-client communication. Firebase Firestore is used solely for connection signaling, while all real-time messaging bypasses server infrastructure entirely. We further developed a minimal NestJS backend exclusively to support quick-match pairing. Our system demonstrates distributed coordination, NAT traversal, low-latency communication, and decentralized state management. Experimental evaluation shows message latency consistently below 20 ms under typical network environments. However, challenges related to NAT configurations, signaling race conditions, and session persistence emerged. This paper details the architecture, implementation, evaluation, and lessons learned while building a fully decentralized communication system, offering insights into practical distributed system design in modern web environments.

1. Introduction

Real-time messaging applications play a critical role in modern digital communication. Traditionally, such applications rely on centralized back-end servers that mediate message delivery, manage user states, and coordinate message sequencing. Although this architecture provides control and reliability, it inevitably introduces latency, scalability constraints, and single points of failure. Emerging web standards and decentralized networking technologies challenge the need for centralized mediation, particularly in light of browser-native capabilities such as WebRTC. This project investigates whether real-time communication can be achieved entirely via peer-to-peer (P2P) channels between clients, thereby removing messaging servers from the communication loop.

1.1 Motivation

The primary motivation is to explore how distributed system concepts can be applied to the problem of real-time communication without relying heavily on backend services. As messaging applications scale, server costs and maintenance requirements grow proportionally. Additionally, routing messages through a server increases network latency, especially when users are geographically distant from the server. By using WebRTC DataChannels for browser-based direct communication, we aim to reduce this latency and investigate new forms of decentralized message exchange.

More broadly, the project serves as a hands-on exploration of decentralization principles taught in the course. Topics such as distributed coordination, state consistency, and reliability are usually studied in the context of large-scale backend infrastructure. Our system demonstrates that these challenges also exist in peer-hosted environments and must be carefully addressed even in small-scale applications.

1.2 Problem Statement

Implementing P2P messaging without a backend introduces several obstacles. First, two independent clients must locate each other and initiate a secure connection without prior knowledge of their network addresses. This step, known as signaling, usually requires a dedicated server. We seek to determine whether a real-time database such as Firestore can serve as a substitute while maintaining low overhead.

Second, the system must establish and maintain a communication channel that achieves sufficiently low latency. We investigate whether WebRTC-based communication can consistently outperform traditional server-mediated messaging.

Third, the absence of a centralized server shifts responsibilities such as reconnection, state synchronization, and reliability toward clients. This paper studies how such responsibilities are redistributed and how to mitigate emerging risks.

Formally, we aim to answer:

1. Can two web clients reliably establish P2P communication without a centralized relay server?
2. Is Firestore a viable alternative for signaling compared to a dedicated WebSocket server?
3. What are the trade-offs in reliability, performance, and engineering complexity?

1.3 Objectives

Our project objectives are to:

1. Implement direct P2P communication using WebRTC DataChannels.
2. Use Firestore exclusively for signaling metadata exchange.
3. Evaluate performance across different network configurations.
4. Analyze the implications on distributed system reliability.

Through this work, we demonstrate that decentralized messaging is feasible using modern web technologies, though not without trade-offs.

2. System Architecture

2.1 Overview

The system follows a three-layer architecture:

Client Application Layer (React)
P2P Communication Layer (WebRTC DataChannel)
Signaling Layer (Firebase Firestore)

Unlike centralized chat systems, our server does not participate in message exchange and merely aids in user pairing. Once peers connect, communication proceeds entirely via WebRTC, ensuring minimal routing overhead and full decentralization.

2.2 Signaling Layer

Signaling metadata is exchanged via Firestore in the form of SDP offers and answers. Each room uses:

- rooms/{roomId}

- Fields: offer, answer, state, createdAt
- Subcollections: caller/candidates, callee/candidates

Using onSnapshot, each peer monitors changes in real time, mirroring WebSocket functionality without requiring a dedicated backend.

We occasionally encountered race conditions where ICE candidates arrived before SDP processing was complete. To mitigate this, candidates were temporarily cached until both SDP descriptors were applied.

2.3 Peer-to-Peer Layer

The WebRTC DataChannel uses:

- Channel Name: chat
- Ordered: true
- Encryption: DTLS (native to WebRTC)

The connection state machine is:

$$idle \rightarrow signaling \rightarrow connecting \rightarrow connected$$

NAT traversal is managed using Google and Twilio STUN servers. TURN was omitted to avoid complexity, but its absence reduced connection success rate under restrictive network conditions.

2.4 Client Interface Layer

Developed using React and TypeScript, the interface provides:

- Room creation and joining
- Quick-match peer allocation
- Chat messaging and connection status visualization

The UI reflects WebRTC connection state dynamically and informs users of connectivity issues.

2.5 Server Responsibilities

The NestJS backend supports only quick-match pairing with one HTTP endpoint:

POST /quick-match

If a user is the first to arrive, the server returns waiting + roomId. If paired, the response is paired + roomId. The server does not relay communication.

3. Design Decisions

3.1 Firestore vs WebSocket Signaling

Firestore provides infrastructure-free signaling with no server hosting. While WebSocket servers give more control over signaling timing, Firestore reduced operational complexity at the cost of slight delays due to document writes.

3.2 WebRTC for Communication

WebRTC reduces latency by eliminating server involvement. Browsers also enforce encryption via DTLS and SCTP without requiring additional libraries.

However, WebRTC introduces complexity due to:

- ICE negotiation failures
- Lack of centralized coordination
- Peer connection recovery challenges

3.3 Exclusion of TURN Servers

TURN servers significantly improve connectivity under corporate or symmetric NAT but require hosting and increased costs. Given limited system scope, we prioritized simplicity over reliability and accepted lower connection rates.

3.4 Minimal Server Design

The backend was intentionally limited to fast execution of room allocation and did not manage sessions. This adheres to the decentralized design philosophy but weakens recoverability.

4. Implementation

4.1 Technology Stack

- Frontend: React 19 with TypeScript
- Backend: NestJS (Node.js)
- Signaling Database: Firebase Firestore
- Communication Protocol: WebRTC

4.2 Message Format

```
{
  ”t”: ”chat”,
  ”text”: ”Hello”
}
```

WebRTC supports custom data serialization, and future improvements could introduce typing indicators or delivery receipts.

5. Testing and Evaluation

5.1 Experimental Setup

Tests were conducted with two users across:

1. Same Wi-Fi network
2. Cross-network peers

3. Corporate firewall environments

Latency measurements were recorded using timestamp-based comparisons.

5.2 Results

Scenario	Setup Time	Latency	Success Rate
Same Network	1.2s	< 15ms	100%
Different Network	2.3s	< 20ms	87%
Corporate Firewall	6s+	N/A	40%

5.3 Analysis

Once established, WebRTC connections consistently outperformed centralized alternatives. However, setup failures mainly stemmed from ICE negotiation problems and lack of TURN servers.

6. What We Did

Our team collaboratively developed:

- A React-based chat UI with WebRTC integration
- Firestore-based signaling
- Backend for quick-match pairing
- Automated deployment and testing

Integration testing focused on signaling reliability and user transitions during connection negotiation.

7. What We Learned

We learned that database-driven coordination is viable for decentralized setups. WebRTC enables low-latency communication but requires deep networking knowledge to manage negotiation. Simplifying infrastructure does not simplify failure behavior. Finally, built-in encryption significantly reduces security concerns.

8. Unexpected Problems

8.1 NAT Traversal

Symmetric NAT environments prevented peer discovery despite STUN servers. TURN would likely resolve this but was excluded.

8.2 Signaling Race Conditions

Candidates occasionally arrived before corresponding SDP negotiations, requiring caching and reprocessing.

8.3 Queue Reset on Server Restart

In-memory session handling caused partially matched users to be dropped when the backend restarted.

9. Future Work

1. Add TURN servers for higher reliability.
2. Support mesh or SFU topologies for group chat.
3. Introduce message persistence and offline handling.
4. Improve reconnection strategies.

10. Conclusion

This project demonstrates the feasibility of decentralized browser-to-browser communication using WebRTC for direct messaging and Firestore for lightweight coordination. By eliminating centralized messaging servers, we significantly reduced latency and gained insight into distributed system architectures at the client level. While reliability issues remain under constrained network setups, this work highlights that decentralized systems can achieve robust communication within modern web ecosystems without extensive backend support.

The project deepened our understanding of trade-offs between latency and reliability, and challenged traditional assumptions around centralized architecture in communication applications.

11. References

- WebRTC Specification: https://www.w3.org/TR/webrtc/
- Firebase Firestore Documentation: https://firebase.google.com/docs/firestore
- NestJS Documentation: https://docs.nestjs.com/
- React Documentation: https://react.dev/