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. The complete source code for this project is available at https://github.com/JayYoon0412/p2p-chat.

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 (Session Description Protocol) offers and answers. This layer acts as the initial coordination mechanism before the direct P2P link is established. Each room acts as a synchronization point using the following data model:

- rooms/{roomId}: The main document containing the session state.
- Fields: offer, answer, state, createdAt, type.
- Subcollections: caller/candidates, callee/candidates for incremental ICE candidate exchange.

The signaling process follows a strict state machine to ensure convergence:

$$\text{waiting} \rightarrow \text{offered} \rightarrow \text{answered} \rightarrow \text{connected}$$

Using onSnapshot listeners, each peer monitors these state changes in real time. This approach mirrors WebSocket functionality without requiring a dedicated stateful backend, effectively leveraging Firestore as a distributed shared memory.

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

2.3 Peer-to-Peer Layer

The core communication channel relies on the WebRTC DataChannel API. We prioritized message ordering and security over raw throughput, leading to the following configuration:

- Channel Name: chat
- Ordered: true (ensures messages arrive in sequence, preventing conversational desynchronization)
- Encryption: DTLS (Datagram Transport Layer Security) is mandated by the WebRTC standard, ensuring end-to-end privacy.

To facilitate connectivity across different networks, we employed public STUN servers from Google (stun.l.google.com:19302) and Twilio (global.stun.twilio.com:3478). These servers allow clients to discover their public IP addresses and port mappings.

The connection state machine monitored by the client is:

$$\text{idle} \rightarrow \text{signaling} \rightarrow \text{connecting} \rightarrow \text{connected}$$

While STUN handles Full Cone NATs effectively, Symmetric NATs proved problematic. A TURN (Traversal Using Relays around NAT) server was omitted to keep the infrastructure minimal, which is a known limitation in our current deployment affecting approximately 13% of cross-network connection attempts in our tests.

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

We chose Firebase Firestore as our signaling mechanism instead of a custom WebSocket server.

- Rationale: Firestore provides a managed, serverless real-time database. This eliminated the need to maintain a stateful signaling server, reducing operational complexity. Its onSnapshot capability provides the necessary event-driven updates for SDP exchange.
- Trade-offs: While simpler to deploy, Firestore writes introduce slightly higher latency (hundreds of milliseconds) compared to raw WebSockets. However, since signaling only occurs once per session, this setup delay is acceptable for the benefit of reduced infrastructure maintenance.

3.2 WebRTC for Communication

WebRTC was selected to achieve true peer-to-peer communication.

- Latency: By bypassing a central relay server, message latency is determined solely by the network path between peers, often resulting in faster delivery than server-routed architectures.
- Privacy: The architectural design ensures that message content never passes through our backend or database, providing inherent privacy (though metadata exists in Firestore).

However, this introduced complexity in handling ICE negotiation failures and connection recovery, tasks typically handled by a central server in traditional architectures.

3.3 Ordered DataChannel

We explicitly configured the WebRTC DataChannel with ordered: true.

- Rationale: In a chat application, message sequence is critical for context. Out-of-order delivery would confuse users.
- Trade-off: Ordered delivery can suffer from head-of-line blocking in poor network conditions, where one lost packet delays all subsequent messages. We accepted this risk to ensure conversational coherency.

3.4 Exclusion of TURN Servers

TURN servers significantly improve connectivity under corporate or symmetric NAT by relaying traffic when direct P2P fails.

- Decision: We intentionally excluded TURN servers.
- Reasoning: TURN servers require high-bandwidth hosting and increase costs significantly. Given the project's scope as an academic exploration of P2P mechanics, we prioritized architectural simplicity and accepted the lower connection success rate in restrictive networks.

3.5 Minimal Server Design

The NestJS backend was intentionally limited to a single responsibility: /quick-match.

- Statelessness: The server maintains an in-memory queue of waiting users but does not persist session state.
- Implication: If the server restarts, the queue is cleared. This decision adheres to the decentralized philosophy—the server is a mere facilitator, not a dependency for ongoing chats. Once paired, users do not interact with the server again.

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 full-stack decentralized chat application. The specific contributions include:

- Frontend Development: Built a responsive React 19 application with TypeScript that manages complex WebRTC state transitions and provides real-time feedback to users.
- Signaling Infrastructure: Implemented a robust signaling protocol on top of Firestore, handling race conditions and state synchronization.
- Backend Engineering: Developed a lightweight NestJS server to handle atomic user pairing for the "Quick-Match" feature.
- Integration & Testing: Conducted cross-network testing to validate NAT traversal strategies and measure latency performance.

Integration testing focused heavily on the signaling phase, ensuring that the transition from the Firestore-mediated handshake to the direct P2P connection was seamless and reliable.

6.1 Use of AI Assistants

We utilized AI-powered coding assistants, specifically Cursor and GitHub Copilot, to accelerate development. These tools were used to:

- Generate boilerplate code for React components and NestJS controllers.
- Debug intricate WebRTC configuration issues and race conditions.

- Refine the technical language and structure of this report.

All architectural decisions, system designs, and core logic were formulated and reviewed by the team to ensure correctness and adherence to project requirements.

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 and Symmetric NATs

While STUN servers successfully resolved IP addresses for most home networks, we encountered significant issues with Symmetric NATs (often found in corporate or university environments). In these cases, the NAT maps internal IP:port pairs to different external ports for different destinations, causing P2P connection attempts to fail. Without a TURN server to relay traffic, these connections could not be established. This highlighted the "real-world" difficulty of pure P2P systems compared to server-relayed architectures.

8.2 Signaling Race Conditions

The asynchronous nature of Firestore updates led to race conditions. For instance, a peer might receive a remote ICE candidate before the remote SDP offer had been fully processed. Solution: We implemented a local buffering queue for ICE candidates. Candidates received early are stored and only added to the RTCPeerConnection once the setRemoteDescription promise resolves.

8.3 Queue Reset on Server Restart

Our decision to use an in-memory array for the quick-match queue meant that any server deployment or restart would clear pending users. Impact: Users currently waiting in the queue would be "orphaned"—stuck waiting for a match that would never arrive. Lesson: Even "stateless" services often require some persistent state (e.g., Redis) to handle service interruptions gracefully.

9. Code Availability

The complete source code for this project, including the client application, server backend, and documentation, is available on GitHub:

https://github.com/JayYoon0412/p2p-chat

Instructions for running the application locally or deploying it are provided in the repository's README.md.

10. 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.

11. 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.

12. 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/