UDC [004.774:338.48](043.2)

D.V. Melnyk
O.N. Romanyuk

METHODS OF IMPLEMENTING CHAT AND COMMUNICATION TOOLS ON A COLLABORATIVE TRAVEL PLATFORM

Vinnytsia National Technical University

Анотація

У тезах досліджуються методи реалізації чату та комунікаційних інструментів на веб-платформі для організації спільних подорожей. Розглянуто та порівняно основні технологічні підходи до забезпечення обміну повідомленнями в реальному часі: HTTP-поллінг, Server-Sent Events, WebSocket та WebRTC. Обґрунтовано вибір WebSocket як оптимального протоколу для групового чату з огляду на вимоги затримки, двонаправленості та масштабованості. Описано архітектуру системи комунікацій із використанням Socket.IO, Redis Pub/Sub та хмарних сервісів сповіщень для забезпечення надійної доставки повідомлень як онлайн-користувачам, так і тим, хто тимчасово відключений. Проаналізовано питання безпеки – JWT-автентифікацію на рівні WebSocket, рольовий контроль доступу до кімнат чату та захист транспортного рівня. Запропоновано комплексну схему реалізації комунікаційного модуля з урахуванням вимог масштабованості та відмовостійкості.

Ключові слова: спільні подорожі, чат у реальному часі, WebSocket, Socket.IO, Redis Pub/Sub, горизонтальне масштабування, push-сповіщення, JWT-автентифікація, безпека комунікацій.

Abstract

This paper investigates methods of implementing chat and communication tools on a web platform designed for organizing collaborative travel. The principal technological approaches to real-time messaging are reviewed and compared: HTTP polling, Server-Sent Events, WebSocket, and WebRTC. The selection of WebSocket as the optimal protocol for group chat is justified with respect to latency, bidirectionality, and scalability requirements. The architecture of the communication subsystem is described, incorporating Socket.IO, Redis Pub/Sub, and cloud notification services to guarantee reliable message delivery to both online and temporarily disconnected users. Security aspects are analyzed, including JWT-based authentication at the WebSocket layer, role-based access control for chat rooms, and transport-layer encryption. A comprehensive implementation scheme for the communication module is proposed, addressing scalability and fault-tolerance constraints.

Keywords: collaborative travel, real-time chat, WebSocket, Socket.IO, Redis Pub/Sub, horizontal scaling, push notifications, JWT authentication, communication security.

Introduction

The rapid growth of sharing economy platforms and mobile-first web applications has created significant demand for services that connect people with aligned interests and goals. Among these, collaborative travel platforms – web services that allow users to find co-travelers, organize joint trips, and coordinate travel logistics – have become an increasingly relevant category of social application [1]. Unlike traditional travel booking systems, which are transactional in nature, collaborative travel platforms are social in their essence: their core value proposition depends on the ability of participants to discover one another, establish trust, and communicate effectively before, during, and after a shared trip.

Communication tools therefore occupy a central role in the architecture of such platforms. Travelers need to confirm participation, agree on departure times, share last-minute route changes, ask questions before meeting strangers in person, and coordinate within a group once the trip is underway. None of these needs can be adequately served by asynchronous communication mechanisms such as email. Instead, they require near-instant, group-aware, and context-bound messaging that is directly integrated into the platform's workflow [2].

The design of such a communication subsystem involves a series of non-trivial technical decisions. The choice of the underlying real-time protocol has direct implications for message delivery latency, server resource consumption, and the complexity of horizontal scaling. The architecture must handle thousands of concurrent connections while ensuring that message access is strictly scoped to confirmed group members. At the same time, the system must gracefully handle offline users without losing messages, and must protect the privacy of travelers who are sharing sensitive personal plans with relative strangers.

Despite the practical importance of this problem, the literature on real-time communication in the specific context of collaborative travel remains limited. Most existing work either addresses generic chat architectures for enterprise messaging [3] or focuses on the higher-level aspects of travel platform design without examining the technical implementation of the communication layer [4]. This paper aims to address this gap by systematically analyzing the available real-time communication technologies in terms of their suitability for a collaborative travel platform and by proposing a concrete architectural design that satisfies the specific requirements of this domain.

The remainder of the paper is organized as follows. Section 1 establishes the domain-specific requirements for communication in collaborative travel platforms. Section 2 presents a comparative analysis of real-time communication technologies. Section 3 describes the proposed architecture of the communication subsystem, including the message flow and scalability strategy. Section 4 addresses security and access control. Section 5 discusses offline delivery and the notification strategy. The paper concludes with a summary of findings and directions for future work.

1. Domain-Specific Requirements for Communication in Collaborative Travel Platforms

Collaborative travel platforms impose a distinctive and tightly constrained set of requirements on their communication subsystem. Understanding these requirements is a prerequisite for evaluating alternative implementation approaches.

The primary communication scenario is the group chat: a persistent, multi-participant text channel associated with a specific trip. Access to this channel is contingent – a user joins the group chat only upon receiving confirmation from the trip organizer that their application has been accepted. This access control requirement distinguishes travel chat from general-purpose messaging: room membership is not freely chosen but derived from the trip participation graph maintained in the platform's database.

Secondary communication scenarios include: direct messaging between an organizer and a prospective participant during the application process; automated event-driven notifications triggered by business logic (new applications, accepted or rejected invitations, route modifications, trip cancellations); and offline push notifications delivered to users who are not actively connected to the platform.

From the performance perspective, real-time chat in a group travel context imposes demanding latency requirements. Message delivery must be perceived as instantaneous by users who are actively coordinating – a practical threshold of under 100 milliseconds is established in the literature as the boundary of perceptible delay for text communication [5]. The system must simultaneously support at least one thousand concurrent active users without performance degradation, as this is a realistic target for a growing platform that serves users across multiple simultaneous trips.

Persistence is non-negotiable: messages must survive server restarts and must be available to users who join a chat room after its creation (for example, to review the conversation history of a trip they have just been accepted into). The system must also guarantee that no message is permanently lost for a temporarily offline user.

Security requirements add another dimension of complexity. Messages within a travel chat contain information of a personal nature – meeting points, home addresses, phone numbers, photos of participants – and must be accessible only to authorized group members. Any implementation that allows unauthorized access to these channels constitutes not just a technical failure but a potential safety risk to travelers.

2. Comparative Analysis of Real-Time Communication Technologies

Several established technologies are available for implementing real-time communication in web applications. Table 1 summarizes the five most relevant approaches across the dimensions most critical for a collaborative travel platform.

HTTP Polling is the conceptually simplest approach: the client sends a regular HTTP request at fixed intervals to check for new messages [6]. While trivial to implement and universally supported, it generates substantial unnecessary server load during idle periods and introduces latency proportional to the polling interval. For a busy group chat, the latency and bandwidth overhead are unacceptable.

Long Polling improves upon standard polling by keeping the HTTP connection open on the server side until a new message is available, at which point the response is returned and the client immediately opens a new connection. This reduces unnecessary requests but still requires connection re-establishment for every message, and cannot be efficiently implemented across multiple server instances without sticky session routing [7].

Table 1 – Comparison of real-time communication technologies for web applications

Technology	Latency	Scalability	Server Load	Primary Use Case
HTTP Polling	High (1–30 s)	Poor	Very High	Legacy systems, infrequent updates
Long Polling	Medium (1–5 s)	Moderate	High	Basic notifications, simple chat
SSE	Low (<1 s)	Good	Moderate	Server-to-client notifications, live feeds
WebSocket	Very Low (<100 ms)	Excellent	Low	Bidirectional group chat, real-time apps
WebRTC	Very Low (<50 ms)	Limited (P2P)	Minimal	Voice/video calls, peer-to-peer data

Server-Sent Events (SSE) establish a persistent, unidirectional HTTP stream from the server to the client. They are well-suited for delivering notifications and live data feeds, but their inherent one-way nature means that client-to-server messages still require separate HTTP requests. This makes SSE inadequate as a standalone solution for bidirectional chat, though it may serve as a useful complement for notification delivery [8].

WebSocket provides full-duplex, bidirectional communication over a single persistent TCP connection. After an initial HTTP handshake that upgrades the connection to the WebSocket protocol, both the client and the server can transmit messages at any time with minimal framing overhead [9]. Modern implementations such as Socket.IO extend the native WebSocket protocol with higher-level features including automatic reconnection, namespace-based channel isolation, room-based message routing to groups of connected clients, and graceful fallback to long polling for environments where WebSocket connections are blocked by corporate firewalls or older proxies. These characteristics make WebSocket the optimal choice for the group chat functionality of a travel platform.

WebRTC enables direct peer-to-peer communication between browser clients, making it particularly effective for voice and video calls with very low latency [10]. However, WebRTC requires STUN and TURN server infrastructure to traverse NAT boundaries, adds significant implementation complexity, and offers no advantage over WebSocket for text messaging. It is therefore more appropriate for a future enhancement of the platform rather than its initial communication layer.

Based on this analysis, WebSocket – implemented via Socket.IO – is selected as the primary protocol for the chat subsystem. Its combination of low latency, bidirectionality, room-based routing, and auto-reconnect capabilities aligns precisely with the requirements established in Section 1.

3. Architecture of the Communication Subsystem

The proposed communication subsystem consists of seven components, each with a defined responsibility in the message delivery pipeline. Their roles are summarized in Table 2.

Table 2 – Architecture components of the communication subsystem

Component	Technology / Tool	Role in the Chat System
WebSocket Server	Socket.IO on Node.js / Django Channels	Maintains persistent full-duplex connections for each active user
Message Broker	Redis Pub/Sub or Apache Kafka	Routes messages between multiple server instances; enables horizontal scaling
Persistent Storage	PostgreSQL with message history table	Stores all messages; enables history retrieval and offline delivery tracking
Cache Layer	Redis (key-value store)	Caches active room membership; reduces database queries per message
Notification Service	Firebase Cloud Messaging / SendGrid SMTP	Delivers push and email notifications to offline users as a fallback channel
Load Balancer	Nginx with sticky-session routing	Distributes incoming WebSocket connections across server instances
Frontend Client	React / Vue.js + Socket.IO client library	Renders chat UI; manages connection lifecycle and auto-reconnect logic

The end-to-end message flow within this architecture is illustrated in Figure 1. When a user (User A) types and sends a message, the React frontend emits a 'chat:msg' WebSocket event to the Socket.IO server. Before processing the event, the server validates the sender's JWT token and verifies that the user is a confirmed participant of the target trip room – a database lookup that is cached in Redis to minimize latency. If authorization succeeds, the message is written to PostgreSQL for persistence and simultaneously published to a Redis Pub/Sub channel identified by the trip's unique identifier.

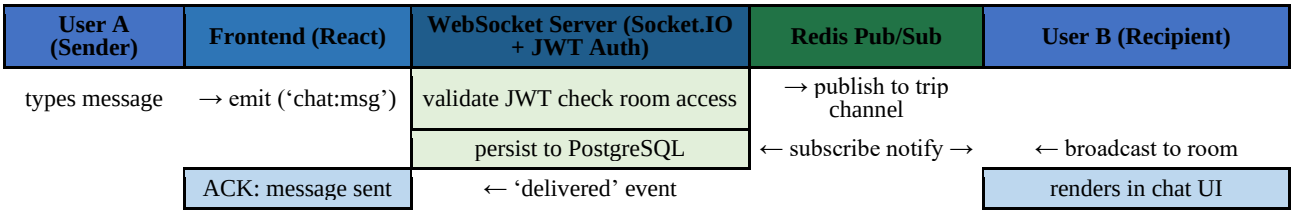

Figure 1 – WebSocket message flow with Redis Pub/Sub for horizontal scaling

All Socket.IO server instances subscribed to that Redis channel receive the publication event and broadcast the message to all clients connected to the corresponding room on their respective instance. User B, connected to any server instance, receives the message in their browser and the chat UI updates in real time. A delivery acknowledgment is returned to User A's frontend, confirming that the message was accepted by the server.

The role of Redis Pub/Sub in this architecture is critical for horizontal scalability. In a single-server deployment, Socket.IO can route messages directly within its own in-process memory. However, as the user base grows and the platform must run on multiple server instances to handle the load – a deployment requirement for any serious web service – each instance maintains only a subset of active WebSocket connections. Without a shared message broker, a message sent by a user connected to server instance A would never reach a user connected to instance B. Redis Pub/Sub resolves this by providing a shared broadcast channel: every server instance subscribes to all active trip channels and rebroadcasts incoming messages to its local connections. The socket.io-redis adapter package implements this pattern transparently, requiring only a Redis connection configuration change compared to a single-instance deployment [11].

For higher throughput at very large scale, Apache Kafka may replace Redis Pub/Sub. Kafka's durable log storage provides the additional capability of message replay, which can simplify the implementation of the chat history feature. However, Kafka introduces substantially greater operational complexity than Redis and is more appropriate for a platform that has outgrown Redis's throughput capacity [12].

4. Security and Access Control

The security model of the communication subsystem must address three distinct threat categories: unauthorized access to chat rooms, interception of messages in transit, and abuse of the messaging channel for spam or denial-of-service attacks.

Authentication at the WebSocket layer is implemented using JSON Web Tokens (JWT). Each WebSocket connection attempt must include a valid, server-issued JWT in the handshake headers. The Socket.IO server validates the token's signature and expiration before allowing the connection to be established. Invalid or expired tokens result in an immediate connection rejection [13]. This mechanism ensures that the WebSocket layer inherits the authentication state of the platform's REST API, without requiring a separate credential management system.

Room-based access control enforces that a user may join only those Socket.IO rooms corresponding to trips in which they hold the status of a confirmed participant. Upon a 'join room' event, the server performs a database query – cached in Redis – to verify the user-trip relationship before admitting the client to the room. Organizers receive an additional permission flag stored in the room's Redis state that grants them the ability to remove other participants from the chat, consistent with their elevated role in the platform's permission model. Participants who are subsequently removed from a trip (for example, if the organizer revokes their confirmation) are immediately disconnected from the corresponding room via a server-side 'kick' event.

Transport-layer security mandates that all WebSocket traffic is transmitted over WSS (WebSocket Secure), the TLS-encrypted variant of the WebSocket protocol. Since WSS operates over the same TLS infrastructure as HTTPS, this requirement is automatically satisfied by terminating TLS at the load balancer or web server

level and proxying the upgraded connection internally. All WebSocket traffic traversing the public internet is therefore encrypted, protecting message content from network interception.

Rate limiting protects the chat system against spam and message flooding. Each authenticated user is subject to a per-connection rate limit of ten messages per ten-second window, enforced at the Socket.IO server level using a token bucket algorithm. Messages that exceed this limit are silently dropped and the sender receives an error event. Repeated violations within a session trigger a temporary suspension of the offending WebSocket connection, and persistent abuse patterns are logged for administrative review.

5. Offline Delivery and Notification Strategy

A fundamental challenge in any mobile-oriented communication system is the intermittent connectivity of users. A traveler may be in transit, in an area with poor network coverage, or simply have the browser tab closed when an important message arrives. A robust communication subsystem must guarantee that such users do not permanently miss messages.

The proposed architecture handles offline delivery through a three-tier strategy. The first tier is Socket.IO's built-in session resumption: when a user disconnects unexpectedly (for example, due to a network interruption), Socket.IO maintains the session state for a configurable timeout period, typically thirty minutes. If the user reconnects within this window, the session is seamlessly resumed and any buffered events are delivered. This tier handles the majority of temporary disconnections without requiring application-level intervention.

The second tier addresses longer disconnections. Every message persisted in PostgreSQL is stored with a per-user delivery status flag. A background job, running at regular intervals, queries the database for messages that have been stored but not marked as delivered to specific recipients. When such gaps are detected, the job dispatches push notifications via Firebase Cloud Messaging for users who have registered a service worker, and email digests via a transactional email service for those who have not. This ensures that users who have been offline for hours or days receive a notification upon their next interaction with the platform.

The third tier concerns the specific scenario of trip-level event notifications – accepted applications, rejected invitations, route changes, and trip cancellations – which are generated not by user messages but by the platform's own business logic. These notifications follow the same two-channel delivery pattern: real-time delivery to connected clients via WebSocket events, and email fallback for offline users. The notification service subscribes to a dedicated event queue (separate from the chat message queue) and handles the templating and delivery of these structured messages.

It is important to distinguish clearly between the roles of WebSocket and email/push in this architecture. WebSocket is the primary real-time delivery channel: fast, low-overhead, and bidirectional. Email and push notifications are fallback channels that ensure eventual delivery to offline users – they are not intended as substitutes for in-app messaging but as a safety net that prevents information loss when the primary channel is unavailable.

Conclusions

This paper has analyzed the methods of implementing chat and communication tools on a collaborative travel platform, addressing the specific technical and domain requirements of this application category.

The comparative analysis of five real-time communication technologies demonstrated that WebSocket, implemented via Socket.IO, is the most appropriate foundation for the group chat functionality of a travel platform. It provides the sub-100 ms delivery latency required for perceived real-time interaction, supports bidirectional communication, enables efficient room-based message routing, and handles auto-reconnection transparently. HTTP Polling and Long Polling introduce unacceptable latency and server load for this use case, SSE is structurally limited to unidirectional delivery, and WebRTC is better suited to voice and video scenarios rather than text messaging.

The proposed seven-component architecture – comprising a Socket.IO WebSocket server, Redis Pub/Sub message broker, PostgreSQL persistence layer, Redis cache, cloud-based notification service, Nginx load balancer, and React frontend client – satisfies the key performance, scalability, security, and reliability requirements of the platform. In particular, the Redis Pub/Sub adapter enables horizontal scaling to multiple server instances without application-level changes, addressing the requirement for concurrent support of large user populations.

The security model, based on JWT authentication at the WebSocket handshake, room-level access control derived from confirmed trip participation status, WSS transport encryption, and rate limiting, provides a

layered defense that protects travelers' personal information and prevents unauthorized access to group communication channels.

The three-tier offline delivery strategy – session resumption, background delivery tracking with push notifications, and email fallback – ensures that no message is permanently lost regardless of a user's connectivity state, satisfying the reliability expectations of a platform whose users depend on timely communication to coordinate real-world activities.

Future research directions include the integration of end-to-end encryption at the application layer for enhanced message privacy, the evaluation of Apache Kafka as a replacement for Redis Pub/Sub under high-throughput production conditions, and the extension of the communication layer to support voice and video calls via WebRTC for richer in-trip coordination capabilities.

REFERENCES

1. Botsman R., Rogers R. What's Mine is Yours: The Rise of Collaborative Consumption. New York: Harper Business, 2010. 280 p.
2. Wang D., Park S., Fesenmaier D.R. The Role of Smartphones in Mediating the Touristic Experience. Journal of Travel Research. 2012. Vol. 51, No. 4. P. 371-387.
3. Kleppmann M., Kreps J. Online Event Processing. ACM Queue. 2019. Vol. 17, No. 1. P. 1-11.
4. Buhalis D., Law R. Progress in information technology and tourism management: 20 years on and 10 years after the Internet. Tourism Management. 2008. Vol. 29, No. 4. P. 609-623.
5. Card S.K., Moran T.P., Newell A. The Psychology of Human-Computer Interaction. Hillsdale: Lawrence Erlbaum Associates, 1983. 469 p.
6. Grigorik I. High Performance Browser Networking. Sebastopol: O'Reilly Media, 2013. 406 p.
7. Loreto S., Saint-Andre P., Salsano S., Wilkins G. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202. IETF, 2011. URL: https://www.rfc-editor.org/rfc/rfc6202 [Accessed: 18.03.2026].
8. Garcia-Morales V. Server-Sent Events and Real-Time Web Applications. Journal of Web Engineering. 2014. Vol. 13, No. 3-4. P. 250-271.
9. Fette I., Melnikov A. The WebSocket Protocol. RFC 6455. IETF, 2011. URL: https://www.rfc-editor.org/rfc/rfc6455 [Accessed: 18.03.2026].
10. Sredojev B., Samardzija D., Pleskonjic D. WebRTC Technology Overview and Signaling Solution Design and Implementation. Proceedings of IEEE 18th International Conference on Engineering of Computer Based Systems. 2015. P. 9-14.
11. Sanfilippo A. Scaling Socket.IO to Multiple Nodes. Socket.IO Documentation. 2023. URL: https://socket.io/docs/v4/using-multiple-nodes/ [Accessed: 18.03.2026].
12. Kreps J., Narkhede N., Rao J. Kafka: A Distributed Messaging System for Log Processing. Proceedings of the NetDB Workshop. 2011. P. 1-7.
13. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. IETF, 2015. URL: https://www.rfc-editor.org/rfc/rfc7519 [Accessed: 18.03.2026].

Мельник Дмитро Валерійович – студент групи 1ПІ-22б, факультет інформаційних технологій та комп'ютерної інженерії, Вінницький національний технічний університет, Вінниця, e–mail: dmitromelnik304@gmail.com.

Романюк Олександр Никифорович – доктор технічних наук, професор, професор кафедри програмного забезпечення, завідувач кафедри програмного забезпечення, Вінницький національний технічний університет, м. Вінниця, e–mail: rom8591@gmail.com.

Melnyk Dmytro V. – student of group 1PI-22b, Faculty of Information Technologies and Computer Engineering, Vinnytsia National Technical University, Vinnytsia, e-mail: dmitromelnik304@gmail.com.

Romanyuk Oleksandr N. – Doctor of Technical Sciences, Professor, Professor of the Software Engineering Department, Head of the Software Engineering Department, Vinnytsia National Technical University, Vinnytsia, e–mail: rom8591@gmail.com.