Real-Time Presentation of Courier Movement in Web and Mobile Applications Using Socket.IO and Message Broker Architectures

Anri Morchiladze

Affiliated Associate Professor, International Black Sea University, Tbilisi, Georgia.

Email: *amorchiladze@ibsu.edu.ge*

Abstract

Real-time tracking has become an integral part of modern courier services. Customers expect continuous monitoring of the status and location of their shipments, from the moment they leave the warehouse until delivery. While this functionality appears simple from a user perspective, developing such a system presents significant technical challenges. It must handle a continuous stream of location updates from hundreds of couriers simultaneously, providing this information to customers with minimal latency.

This article presents a real-time courier tracking system based on Socket.IO, a message broker, and a backend service responsible for data storage and synchronization. The proposed architecture decouples the system's core functional components, including data collection, event processing, data persistence, and customer interaction. This modular design enhances scalability, improves system performance, and increases fault tolerance, preventing failures in one component from affecting the entire system. The proposed solution demonstrates how modern event-driven technologies can be combined to provide reliable and efficient real-time tracking for web and mobile applications.

Keywords: real-time systems; Socket.IO; message broker; courier tracking; web applications; mobile applications; event-driven architecture; logistics.

Introduction

Digital transformation has significantly changed the logistics and transportation industry. Modern courier services are no longer limited to parcel delivery; they also rely on continuous location tracking to optimize operational efficiency and provide customers with accurate and up-to-date information about their deliveries.

Traditional client-server communication mechanisms, such as REST, are ill-suited for applictions that require continuous data updates. In a REST-based architecture, the client must repeatedly poll the server to determine whether new information is available. As the frequency of location updates increases, this approach generates unnecessary network traffic, increases the load on the server, and introduces a delay between data generation and presentation.

Real-time communication technologies address these issues by allowing the server to immediately send updates to connected clients when new data becomes available. Among these technologies, Socket.IO is widely used because it provides reliable two-way communication, automatic reconnection, cross-browser compatibility, and seamless transition to alternative transport mechanisms when WebSocket connections are unavailable. Its event-driven architecture makes it particularly suitable for applications that process continuously changing data.

This paper presents a scalable, real-time courier tracking system based on Socket.IO and a message broker for asynchronous message processing. The proposed architecture efficiently handles large volumes of location updates while maintaining low latency and high responsiveness for both web and mobile applications.

Proposed System Architecture

The given architecture consists of these major components:

- Mobile courier application
- Message broker
- Backend processing service
- Database
- Client web/mobile application

The overall data flow is illustrated below on figure 1.

The courier app regularly gets GPS coordinates data through the device's location system. Each location updated is sent to the message broker, instead of directly to the front-end users.

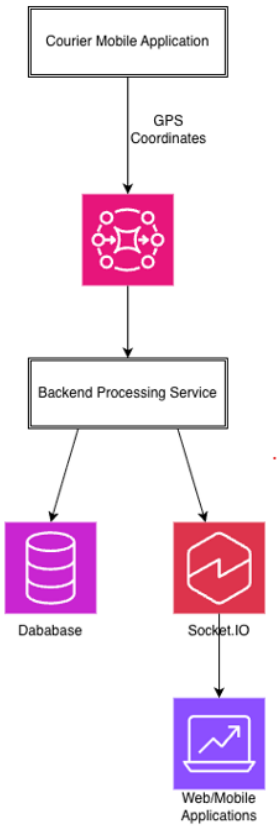


Figure 1: System overall architecture

This message broker guarantees that messages are delivered without any immediate responses. If on the server side, we have multiple nodes, several back-end servers can receive messages divided equally from the queue. This gives us ability, to grow system and handle more tasks by adding more and more nodes. Each node instance performs several operations:

- validates incoming coordinates;
- stores the location in the database;
- updates the latest courier position;
- broadcasts the update through Socket.IO.

Because the backend performs persistence before broadcasting, the system guarantees that displayed positions correspond to stored data.

Comparison of Real-Time Communication Approaches

When building a real-time tracking system, to choose the right communication method is quite important. Using rest api standarts, regular HTTP requests is easy but not efficient for such systems, which requires frequent updates. Long polling can reduce the number of requests, but it still causes extra delays and increased waiting times for front-end users. Socket.IO library, which uses WebSocket with backup options, allows us to have constant two-way communication. Which is better method then pollig http requests. So it is good solution for real-time tracking applications.

Criterion	REST Polling	Long Polling	Socket.IO / WebSocket
Communication model	Request–Response	Request–Response with delayed response	Persistent bidirectional connection
Real-time capability	Low	Medium	High
Network overhead	High	Medium	Low

Server resource consumption	High	Medium	Low
Latency	High	Medium	Very Low
Automatic reconnection	No	Limited	Yes
Scalability	Moderate	Moderate	High
Event-driven communication	No	Partial	Yes
Suitable for courier tracking	Poor	Acceptable	Excellent

The comparison shows that Socket.IO is better for systems that sends location data frequently. It keeps a stable connection, which avoids us to often re-establish HTTP connections. Also, it sends updates immediately when they are ready, which makes the process faster and much efficient.

Real-Time Communication Using Socket.IO

Socket.IO is create to have constant, two-way communication between clients and servers. Despite from the traditional HTTP polling method, the server can send updates immediately when new location data will be available.

The communication process consists of these four stages:

1. Client establishes a Socket.IO connection.
2. Backend subscribes the client to a courier-specific channel.
3. New coordinates arrive through the message broker.
4. Backend broadcasts the updated location to subscribed clients.

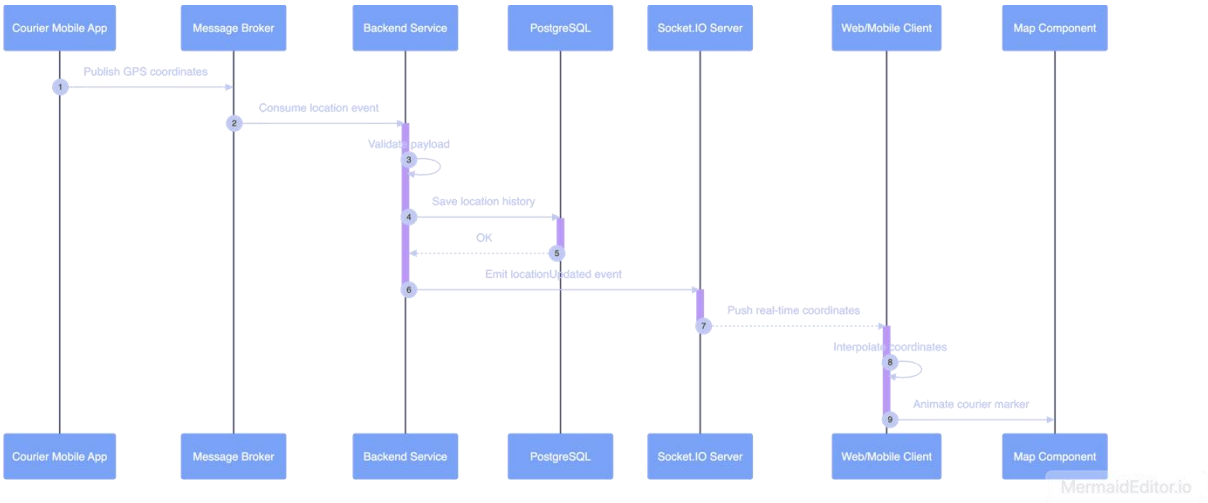
This method of sending data, based on events, helps us to reduce the extra network traffic, because the information is only sent when there's something new to share.

Socket.IO features:

- reconnection after network interruptions;
- heartbeat monitoring;
- event acknowledgements;
- namespace separation;
- room-based subscriptions.

These features simplify implementation to improve communication stability.

Figure 2: Sequence Diagram



Backend Event Processing Pipeline

Let’s define the steps for the backend, how to handle received data from the message broker.

1. Parse the received location event.
2. Validate timestamp and coordinate correctness.
3. Store historical data.

4. Update the latest courier position.
5. Emit a Socket.IO event to subscribed clients.

A simplified processing algorithm is:

1. receive location event
2. validate coordinates
3. write to database
4. update current position
5. emit socket.io event
6. clients receive update

We keep database tasks separate from communication logic. This helps the system to make saving operation in different threads: To save data on storage and to communicate via network to send or receive data.

Performance Analysis

The given architecture splits the work into separate services:

- capturing location data
- handling messages
- storing information long-term,
- ommunicating with clients such that, each get their own data.

The message broker represents a central component of the proposed architecture, acting as an intermediate layer for handling a continuous stream of incoming GPS location updates. Instead of allowing every courier device to communicate directly with the primary backend service, location events are first published to the broker and then consumed by backend instances.

This approach removes the dependency on a single processing point and enables multiple backend service instances to process messages concurrently. As a result, the system can distribute the processing workload across multiple consumers, improving scalability and preventing the backend from becoming a bottleneck when the number of active couriers increases.

That decoupling brings a few concrete performance gains:

- asynchronous processing of GPS events;
- reduced response latency through persistent Socket.IO connections;
- horizontal scalability of backend services;
- reliable delivery of location events;
- fault isolation between system components;
- efficient client subscription using Socket.IO rooms or namespaces.

The frontend improves performance much by updating the only parts of the screen that are visible to the user, instead of re-render the whole screen. Instead of redraw the marker every time new coordinates data will be received, interpolation techniques create smooth movement effects, which keeps the CPU usage low. The parts increase the performance of any front-end parts, for desktop and even for mobile sides.

Studies show that using WebSocket for communication can reduce network costs by over 50% compared to standard REST API requests, especially in the situation when updates happen much more often than every five seconds. Additionally, publisher–subscriber systems help scale better by separating in two parts: producers and consumers. The producer which sends messages from who receives them, allowing for quicker and much more efficient process.

Real-Time Courier Visualization and Animation

Updating marker positions directly with each new GPS event can cause chaotic or quick movement since GPS data usually comes in every seconds.

To make the movement visualization much smoother, the frontend keeps a list of incoming coordinates to process them gradually.When a new coordinate is received:

- it is appended to the queue;
- the animation engine interpolates intermediate positions;
- the courier marker moves continuously along the calculated path.

Linear interpolation can be expressed as:

$$P(t) = P_o + t(P_1 - P_0), \qquad 0 \leq t \leq 1$$

where:

- P_o - represents the previous position,
- P_1 - represents the new position,
- t - is the interpolation coefficient.

The animation updates the courier marker multiple times between received GPS coordinates, creating visually continuous movement path while preserving positional accuracy.

Additional improvements is added:

- heading calculation for marker rotation;
- speed estimation;
- adaptive animation duration;
- route smoothing;
- map viewport adjustment.

These techniques improve user interface and experience without increasing the network traffic.

Performance Evaluation and Scalability Considerations

This suggested design offers multiple benefits when it comes to scaling.Image the situation when the message broker handles sudden spikes in GPS data, stopping the backend systems from becoming overwhelmed.Second, the backend can easily add more servers, at this moment the data is splitted equally for each one, which means each node works on its own to get and share updates.Third, Socket.IO keeps connections open, which reduces repeatable HTTP requests and the extra work necessary for communication.

When it's okay to have less precise data over time, the database can be made more efficient by grouping data manipulation commands or doing them in the background process. The apps on the front-end only deal with location updates that are attached with the couriers they're tracking, which helps reduces the workload for rendering.

Overall latency consists of:

- GPS acquisition,
- message broker delivery,
- backend processing,
- database persistence,
- Socket.IO transmission,
- frontend rendering.

In practical deployments, total end-to-end latency can remain below one second under normal operating conditions, providing users with near real-time tracking.

Discussion and Practical implications

The proposed architecture follows established principles of distributed systems design. A message broker is used to separate mobile applications, backend services, and client applications, allowing each component to develop independently without creating dependencies between them. This approach also supports horizontal scalability, as additional backend service instances can be deployed without requiring modifications to client applications.

System reliability is further enhanced by asynchronous message processing. Messages are temporarily stored by the broker, allowing processing to continue even if one of the backend services is unavailable. This mechanism reduces the risk of message loss and improves overall system resiliency.

To improve the user experience, the client application interpolates the courier's position between successive location updates, rather than displaying abrupt jumps on the map. This provides a smoother animation that more accurately depicts continuous movement and makes tracking easier for users. While Socket.IO offers slightly more protocol overhead than native WebSockets, it offers a number of practical advantages, including automatic reconnection, event-based communication, namespace support, and compatibility with browsers that don't fully support WebSockets. These features simplify application development and improve the reliability of real-time data exchange in production environments.

Conclusion

Real-time courier tracking systems must handle frequent location updates while maintaining low latency and reliable operation under high request volumes. This paper presents an architecture that combines a message broker for asynchronous event processing with Socket.IO for two-way real-time communication between the server and client applications. Historical location data is

stored to support route visualization and analysis, while the frontend applies interpolation techniques to ensure smooth visualization of courier movements.

Separating data collection, event processing, storage, and presentation enables independent development and scaling of individual system components. As a result, the proposed architecture can support logistics platforms with large numbers of simultaneously active couriers. Future enhancements may include predictive courier movement models, adaptive update rates based on vehicle speed, machine learning-based route optimization, and distributed geospatial storage solutions for large-scale deployments.

REFERENCES

[1] I. Fette and A. Melnikov, *The WebSocket Protocol*, RFC 6455, Internet Engineering Task Force, 2011.

[2] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[3] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[4] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, 2007.

[5] Socket.IO Documentation. https://socket.io/docs/

[6] RabbitMQ Documentation. https://www.rabbitmq.com/

[7] Apache Kafka Documentation. https://kafka.apache.org/documentation/

[8] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] M. Richards, *Software Architecture Patterns*. O'Reilly Media, 2015.

[11] M. Richards and N. Ford, *Fundamentals of Software Architecture*. O'Reilly Media, 2020.

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd Edition. Addison-Wesley, 2018.

[13] Node.js Foundation, *Node.js Documentation*. https://nodejs.org/

[14] PostgreSQL Global Development Group, *PostgreSQL Documentation*. https://www.postgresql.org/docs/

[15] OpenStreetMap Foundation, *OpenStreetMap Documentation*. https://wiki.openstreetmap.org/

[16] Leaflet Contributors, *Leaflet JavaScript Library Documentation*. https://leafletjs.com/