RFC: Rider App Realtime Update PoC

Author(s)	Michael Gerasymenko Andrii Raikov Rolando Santamaría Masó
RFC status	Accepted
PRD	Project Charter - Rider Pulse
Owner(s)	log-deliveries
Review deadline	Nov 5, 2025
Epic	ROTW-3378
Public Slack Channel	#deliveries-rider-pulse

Timeline

Date	Comment
Sep 12, 2025	Document created

Introduction

Our riders are working in a very harsh environment and our duty is to provide them with the best tools possible to complete their jobs.

Rider application and its backend services is a single system that spreads across the cloud and riders mobile devices. We need to make this system strive to be up to date at all times, so the riders, customers and vendors have the most up to date and accurate information.

Rider's application must use the state-of-the art technology to accurately and timely display the most up to date information about rider shifts, assigned and changed orders and potential shift breaks, as well as any other updates needed.

Context & Problem statement

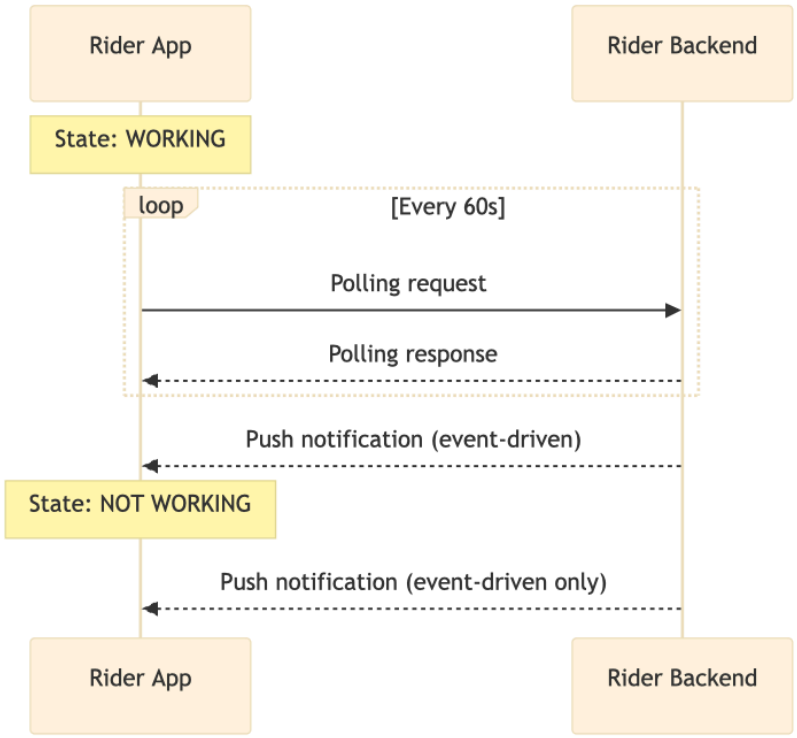
In our design prior to the map centric approach we supported a "pull down" gesture that triggered a status refresh. During the migration we applied the backwards compatibility requirement for the functionalities to ensure we could release the new UI faster, and that's how we came to the conclusion a "button" was needed as we couldn't allow for a pull down gesture on the map.

Along with pull to refresh, the app is receiving push notifications for important updates. This approach proved itself to be not the most efficient.

However, this system proved itself not as reliable as we need it to be.

Remove Refresh Button

We removed the refresh button in the scope of the Remove Refresh Button project and replaced it with smart polling.



Smart polling is refreshing the app continuously over a defined time interval (currently 60 seconds after the last update) during the rider shift. Most of the updates are also sent to the app as push notifications, triggering the refresh of its contents.

This update made it possible to remove the refresh button and impacted some key metrics. However, with this approach, the information displayed in the app still could be outdated by up to 60 seconds and potentially more. This can happen in case the internet connection was lost after the last poll, or an update happened after the poll was completed.

An additional downside of polling is an increased and constant backend load. Even with traffic from the refresh button going away, we could conclude an increase of around 66% of the backend calls.

Idea

We are proposing to develop a proof-of-concept solution on one of the mobile platforms (iOS) based on the technology that would suit functional and non-functional requirements and that would allow confirming the hypothesis that realtime communication would offer the superior rider experience:

- Requests to Home API 🔽 strong decrease by up to 50%
- Acknowledged deliveries 🔼 increase from 82,66% to ca. 90%
- Rider Accepting Time 🔽
- ICE Delivery not seen 🔽
- Stretch
    - Acceptance rate 🔼
- Control metrics
    - Reassignment rate

The initial solution, developed as a PoC, would use the technologies that we are going to use in the final solution.

The initial solution is going to support a subset of requirements from the final set and would be flexible enough to extend and be used by different squads.

Requirements

The following use cases were shared with us by the product team as a set of requirements for the **final state** of the system

Squad	Use Case	Metric	Impact	Covered in PoC
Acceptance Squad	New delivery (normal and b2b including auto-accept)	AR, reassignment, rider accepting time, delivery not seen issues	High	Yes
	Route change	Stacks AR, delivery time	Medium	Yes

	Delivery cancelled	UTR	High	Yes
	Destination updated	Time to Customer	Medium	Yes
	Stacked /b2b order received	AR	High	Yes
On The Way	Rider GPS pings being sent to the DH BE system	Increase efficiency of the location updates	High	
	Payment at the door Feedback	SoftPOS fail rate (QR pay as well in the future)	High	
	Task list up to date, esp. when crossing the geofence	RCR (App issue, e.g. UtcC not displayed), Time spent on tasks (e.g. photo task shown but rider already inside geofence)	High	
	Order Status Component (Food is ready in xx min, Food is Ready)	Time at Pickup	High	
PUDO	Hide Order information	Usage of Food-is-Ready button of vendor	Medium	
	Expanded Bubble (Shift starting, delivery complete, shift ended)	Vibes?	Low	
Rider Experience	Push Notifications?	_	High	Yes, except marketing

	Nest Header KPIs		Low	
Rider Availability	Offline - all information on the start now tab (Which zone is having a delay, and what is the payment?)	Mean delay, conversion of start now	High	
	Offline - Access restricted	Fillrate	Low	
	Offline - session starting in 30 mins	On time start	High	
	Infra-level payments in the idle state	Idle time, Delay, RSAT, potential legal/PR risk if what we pay is not what we show	High	
	Cashback component	Idle time	Low	
	State changes	Legal risk, RSAT	High	Yes
	Quest progress (based on acceptance rate + delivery completed)	AR	High	

Product requirements analysis

Data flow

All use cases except for one (Rider GPS pings being sent to the DH BE system) require data to be sent from the backend to the app.

Regular pings including GPS coordinates, connection info for the ISP dashboard and the battery level is a process of continuous location sharing from the rider to the backend. Currently, the data is being sent every 10 seconds. In the future, the requirement would change to send the courier location when the location is changed by at least 100 meters (translates to an update roughly every 4 seconds).

Functional requirements based on the product use cases

The proposed real-time communication platform must meet the following functional criteria to serve as a robust solution for courier-backend messaging:

- **Persistent Connection Lifecycle**: The communication channel must remain active and capable of message delivery as long as the application is running with an internet connection. This is independent of the courier's specific work status (e.g., offline, on-shift, or actively on a delivery).
- **Bidirectional Communication Channel**: The system shall establish a persistent, real-time, bidirectional communication channel between backend services and the courier's mobile application. While the architecture must support full-duplex communication, the initial (PoC) implementation will prioritize server-to-client (push) messaging. This channel will also be used for future message acknowledgements. This feedback will not only confirm message receipt but also act as a heartbeat to ensure the app's connection is live, and it is designed to function alongside the regular GPS updates submitted during the courier's working state.
- **Multi-Tenant and Reusable Platform**: The solution must be implemented as a centralized, domain-agnostic platform that can be utilized by multiple product squads. It must provide logical channel or topic isolation to ensure that messages intended for one functional area do not interfere with others.
- **Reliable State Synchronization**: The platform will be the authoritative source for real-time updates. It must guarantee message delivery over the real-time channel to ensure data consistency between the backend and the client, thereby removing the need for client-side polling for critical state changes.
- **Flexible Payload Structure**: The solution must be payload-agnostic, avoiding unnecessary restrictions on message size or data structure. Reasonable size limitations known from state of the art would apply. This allows individual squads the flexibility to define data contracts that are appropriate for their specific use cases.
- **Coordinated Notification Delivery:** To ensure a low-noise and intuitive user experience, event notifications must be coordinated across delivery channels. For any single server event, the system must guarantee that only one notification is presented to the user.

Delivery via the active WebSocket channel will always suppress the dispatch of a corresponding native push notification for that same event.
- **Fallbacks**: Solution should support multi-level fallback support to reduce criticality and dependency.

Non-functional requirements

The solution must adhere to the following non-functional requirements to ensure security, performance, and scalability:

- **Low Latency**: End-to-end message delivery, from the originating backend service to reception by the client application, must be optimized for minimal latency.
- **Security and Authentication**: All connections must be secure and encrypted using state of the art protocols. The platform must enforce a strict authentication mechanism to ensure that only legitimate and authorized courier applications can establish a connection and receive messages.
- **High Scalability and Resilience**: The system must be architected to concurrently manage connections from the entire active courier fleet (60K simultaneously active riders in one country). It must demonstrate resilience against common network intermittency through robust automatic reconnection logic. Furthermore, the infrastructure must incorporate mechanisms for DDoS mitigation.
- **Tiering:** the new application should be a Tier-2 application based on the Delivery Hero definition.
- **Standardized Message Protocol**: While the payload remains flexible, all messages transmitted through the platform must conform to a uniform envelope or schema. This standardized wrapper shall include essential metadata, such as message type, source, and a timestamp, to facilitate efficient routing, monitoring, and client-side handling.
- **Remove Kong from the chain of calls**: our Logistics gateway should be crossed out from the architecture of the service. This is necessary to support the initiative to decommission Kong. This is also removing an additional layer from the connection chain.

Proposed solutions

Criteria	Socket.IO	Amazon API Gateway (WebSockets)
Management & Scalability	Self-managed We are going to be responsible for provisioning, scaling, and maintaining servers. Scaling requires careful architecture.	Fully managed & serverless AWS handles all infrastructure, scaling, and availability. It scales automatically to handle millions of concurrent connections.
Features & Flexibility	Feature-rich out of the box. It includes automatic reconnection logic, fallback to HTTP long-polling (for restrictive networks), and broadcasting to logical "rooms" or "namespaces."	Lean and protocol-focused. It provides a raw WebSocket connection. Features like reconnection and message retries must be implemented in your client application. Backend routing is handled via Lambda functions.
Long-term support	This approach would most likely require either establishing a squad, or a squad taking ownership over the new service.	Managed approach reduces the load on the team and allows for a lightweight support duty
Ecosystem Integration	Built on a Standard, Flexible Foundation: The solution is built upon the robust and widely-adopted Socket.IO library, ensuring reliable real-time communication. This foundation is backend-agnostic, allowing it to connect with any cloud provider or internal service. While the core engine is standard, the specific adapters for integrating with systems like a message broker will need to be developed and maintained.	Native AWS integration. Seamlessly connects with IAM (for auth), Lambda (for logic), DynamoDB (for connection state), and SNS/SQS, enabling powerful, event-driven architectures with minimal "glue code."

Cost Model *(Cloudflare CDN cost not considered)*	Infrastructure-based. You pay for the running servers, load balancers, DBs, caches and data transfer. The primary cost is often the engineering time for maintenance and evolving the solution (DevOps/SRE). In terms of Infrastructure Costs, this solution is significantly cheaper. Taking the current Geo Tracking Service as an example, we spend ~14k monthly. Even with a 10x infra cost increase, it would still be cheaper.	Pay-as-you-go pricing for Amazon API Gateway WebSockets offers a cost-effective solution for low-to-medium scale applications. However, at a massive scale, it's crucial to perform a cost analysis to understand the total expenditure. While it significantly reduces the need for maintaining a WebSocket server, you'll still need to build an extension for synchronization and publication purposes. For the Delivery Hero Scale, the cost forecast for the Amazon API Gateway WebSocket API is about ~$187k per month. Other cost factors will have to be introduced afterwards, such as Lambda, DynamoDB and Kubernetes Containers.

Recommendation

After a detailed comparison focusing on long-term financial sustainability and technical control, the **Socket.IO-based solution is the recommended approach.**

We still propose implementing a real-time communication system using a publish/subscribe (pub/sub) architecture fronted by a WebSocket Gateway. This model, built with Socket.IO, provides a scalable, decoupled, and powerful solution that meets all specified requirements in the most cost-effective manner.

Justification

The primary driver for this recommendation is the **significant cost difference at our operational scale.** The forecasted monthly cost for the fully managed AWS solution (~$187k+) is prohibitively high compared to a self-managed Socket.IO cluster. The substantial cost savings from the Socket.IO approach will allow for the allocation of engineering resources to manage the platform while still representing a massive reduction in total operational expenditure.

Furthermore, a self-managed solution offers:

- **Greater Control and Flexibility:** It prevents vendor lock-in and gives us full control over the technology stack, release cycles, and feature implementation.
- **Rich Feature Set:** Socket.IO's built-in features, such as automatic reconnection and network fallbacks, reduce the development burden on client-side teams and provide a more resilient user experience out of the box.

While this approach requires dedicated ownership for infrastructure management, this is a strategic investment. It ensures we have a custom-fit, cost-effective, and highly capable platform that can evolve with our business needs without being constrained by the cost model of a managed service.

Architecture Overview

The architecture is designed for decoupling, security, and scalability. It consists of the following key components that work in concert to deliver messages from backend services to the courier application.

- **Backend Services:** Services from different squads will act as message publishers. They will generate events based on their business logic.
- **Message Broker:** Publishers will send their messages to a centralized, topic-based message broker. This broker decouples the backend services from the client-facing communication layer.
- **WebSocket Gateway:** A self-managed cluster running **Socket.IO**. This service subscribes to the message broker, maintains persistent WebSocket connections with active courier applications, and pushes messages securely to the correct rider.
- **Rider App:** The app establishes a secure WebSocket connection and receives real-time updates pushed by the gateway.

Server Lifecycle

The following are the server side flows in scope:

- **Handling WebSocket connections**: This is the initial phase where a client establishes a persistent, two-way communication channel with the server. Unlike standard HTTP requests, which are short-lived, a WebSocket connection remains open (consuming a socket port for a long period). This requires the server to manage long-lived connections efficiently, scaling horizontally and allocating resources for each active client. The server must be able to upgrade a standard HTTP request to a WebSocket protocol.
- **Client Authentication**: Once a WebSocket connection is established, the server must verify the identity of the client using JWT (DH-IAM). This ensures that only authorized users can access specific services or data. Authentication MUST happen at the beginning

of the connection handshake to avoid potential denial-of-service (DoS) attacks. Common methods involve passing access tokens during the initial handshake.

- **Connection State Management**: The server must monitor the status of each client connection, identifying whether it is active, idle, gracefully closed, or unexpectedly disconnected, along with its associated socket server IP. To enable messaging and resource cleanup of stale connections, the server is also responsible for maintaining a registry of active connections.
- **Rate Limiting**: The server needs to implement a system to track client activity and throttle or reject requests that exceed the defined limits. During the PoC phase, the client will only be allowed to send the Message Processing Acknowledgment messages, eg:

```JSON
{
  "metadata": {
    "eventType": "ack",
  },
  "payload": {
     "messageId": "xxsdfdc3d4-e5f6-7890-1234-000000000"
  }
}
```

- **Topics and Subscriptions Management**: This flow addresses how clients organize their interests to receive relevant messages. Clients can "subscribe" to specific "topics" or channels, and the server maintains a mapping of which clients are interested in which topics. When a new message for a topic arrives, the server knows exactly which clients to send it to. This is a fundamental pattern for building publish/subscribe (pub/sub) systems.
- **Sending Messages to Clients**: This is the core function of the service in a WebSocket-based system. Once an event occurs (e.g., courier route is updated), the server must efficiently route and push the corresponding message to the appropriate client(s). Backend services acting as publishers will use a dedicated HTTP endpoint on the WebSocket server to send messages for a specific client, often identified by an attribute like *riderId*. The WebSocket server then uses this identifier to look up the correct client connection and sends the message over the established WebSocket.
- **Handling Client Acknowledgment Messages**: This flow involves the server receiving and processing messages from the client that confirm receipt of a previous message. This is critical for ensuring reliable message delivery, especially in systems where messages are high-value or must be guaranteed. The server might use acknowledgments to update message status, trigger follow-up actions, or resend messages that were not acknowledged within a specific timeframe. This helps in building a robust and

fault-tolerant communication system. During the PoC phase, the server implementation will be limited to monitoring message delivery via Grafana with Open Telemetry.

Example implementation using Socket.IO in Appendix

App Lifecycle

The courier application will manage the WebSocket connection based on its lifecycle state to ensure reliability and battery efficiency.

- **Connection Initiation:** A secure WebSocket (wss://) connection is established immediately after a successful user login when the app enters the foreground. The client authenticates using a short-lived token (e.g., JWT) passed during the connection handshake.
- **Connection Maintenance:** The connection remains active as long as the app is not suspended. A **heartbeat mechanism** (ping/pong frames) will be used to monitor connection health and detect silent drops, allowing the client to proactively reconnect.
- **Automatic Reconnection:** In case of a connection failure (e.g., network change from Wi-Fi to cellular, server restart), the app will automatically attempt to reconnect using a strategy like **exponential backoff** to avoid overwhelming the server.
- **Background State:** When the app moves to the background, the OS may terminate the WebSocket connection. For critical, time-sensitive events (e.g., a new delivery offer), the system will fall back to sending a **native push notification** via comms.

Topics

To ensure messages are correctly routed and isolated, the system will use a topic-based subscription model. Upon connecting, the client will subscribe to a set of topics specific to that rider and their context.

Topic Naming Convention: We will use a clear, hierarchical naming convention:<use-case>.<entity-id>

Example Topics:

- **Rider-Specific Deliveries:** rider.{riderId}
    - Use Case: Pushing new delivery offers, cancellation updates, or destination changes for a specific rider.

This structure ensures that the client only receives relevant information and allows squads to publish events without impacting other parts of the system.

Payload Structure

All messages sent over the WebSocket channel will conform to a **standardized envelope structure**. This ensures consistent message handling on the client side while providing flexibility for the data payload itself.

The message will be a JSON object with two top-level keys: metadata and payload.

- **metadata (Object):** Contains information common to all messages for routing, debugging, and analytics.
    - messageId (String): A unique UUID for the message.
    - eventType (String): A clear, machine-readable identifier for the event.
    - timestamp (String): The ISO 8601 timestamp of when the event was generated.
    - source (String): The name of the originating service
- **payload (Object):** An object containing the data specific to the event. The structure of this object is flexible and is defined by the publishing feature-based on the eventType.

Example: New Delivery Offer Payload

```JSON
{
  "metadata": {
    "messageId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
    "eventType": "...",
    "timestamp": "2025-09-23T09:05:15.123Z",
    "source": "delivery"
  },
  "payload": {
     ...
  }
}
```

Schema Evolution Strategy

To ensure backward compatibility and prevent schema regressions while allowing new fields and event types to evolve over time, we propose the following approach:

1. Backward-compatible evolution

- All additions to existing event payloads (new optional fields) must be non-breaking.
- Existing fields must never be removed or renamed without deprecation.
- Clients should be built to ignore unknown fields, ensuring forward compatibility.

2. Explicit versioning

When a breaking change is unavoidable (e.g., field renames or structural changes), a new eventType will be introduced — e.g., `delivery_offer_created.v2`.

This avoids ambiguity for consumers and supports parallel migration.

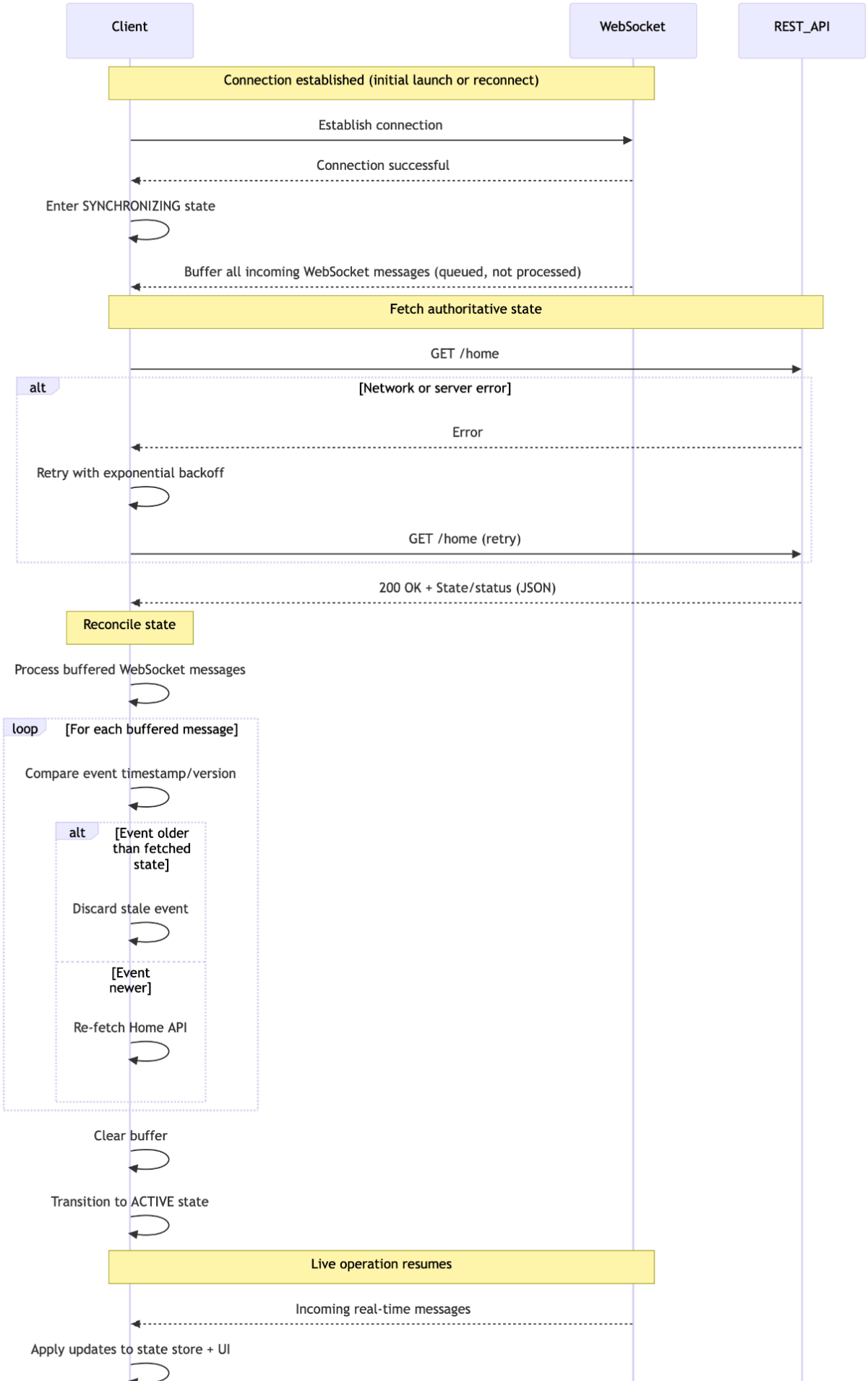
3. Governance
   - All schema changes must go through the event review process (similar to an ADR or API design review).
   - Minor, additive changes can be merged after automated validation; major changes require explicit sign-off and migration plan.

Client-side synchronization strategy

To ensure the courier application always reflects the authoritative state from the backend and to prevent data inconsistencies caused by network interruptions, a robust client-side synchronization strategy will be implemented. This process guarantees that the client application can reliably recover from disconnects and avoid race conditions.

The core principle of this strategy is that the WebSocket channel is used for real-time events and updates, not for initial state delivery. The authoritative, complete state must always be fetched via a dedicated REST API endpoint upon establishing a new connection.

The Synchronization Process


Client
WebSocket
REST_API
Connection established (initial launch or reconnect)
Establish connection
Connection successful
Enter SYNCHRONIZING state
Buffer all incoming WebSocket messages (queued, not processed)
Fetch authoritative state
GET /home
alt
[Network or server error]
Error
Retry with exponential backoff
GET /home (retry)
200 OK + State/status (JSON)
Reconcile state
Process buffered WebSocket messages
loop
[For each buffered message]
Compare event timestamp/version
alt
[Event older than fetched state]
Discard stale event
[Event newer]
Re-fetch Home API
Clear buffer
Transition to ACTIVE state
Live operation resumes
Incoming real-time messages
Apply updates to state store + UI

The client will manage its connection through a simple state machine. The process is initiated every time a connection is established, including the initial app launch and all subsequent reconnections after a network drop.

Step 1: Enter SYNCHRONIZING State

- Immediately upon successful establishment of the WebSocket connection, the client enters a SYNCHRONIZING state.
- **Buffer Incoming Messages:** While in this state, the application **does not** process incoming WebSocket messages immediately. Instead, it adds them to a temporary in-memory queue or buffer. This prevents the UI from updating with new event data before the client has confirmed its foundational state, which could cause inconsistencies.
- **Show Loading Indicator:** The UI should display a global loading indicator or a "Synchronizing data..." message to inform the user that the app is preparing its state.

Step 2: Fetch Authoritative State

- While buffering messages, the client makes an HTTP GET request to a Home endpoint
- This endpoint returns a complete JSON object representing the rider's entire current state
- If this API call fails due to a network error, the client will retry using an exponential backoff strategy. The client remains in the SYNCHRONIZING state until this call succeeds.
- The client must ensure there are no multiple simultaneous or serial jobs to fetch the Home endpoint, fetching commands must be de-duplicated.

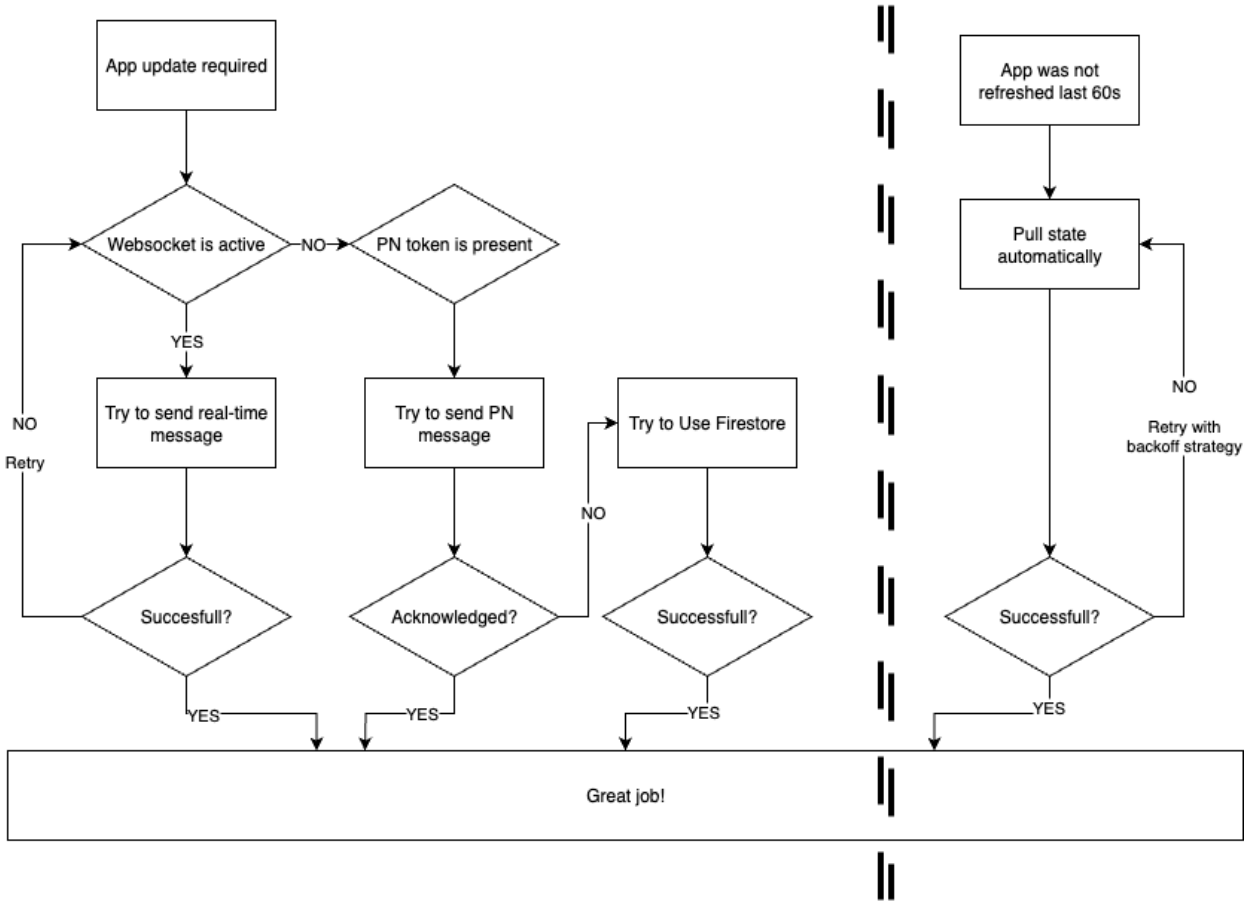
Step 3: Reconcile and Transition to ACTIVE State

- Once the state is successfully fetched from the REST endpoint, the client performs a reconciliation process:
    1. **Atomically Update Local State:** The application's local state store (e.g., Redux, MobX, or a simple state object) is completely replaced with the data received from the API call.
    2. **Process Buffered Messages:** The client then processes the queued WebSocket messages. It should check the timestamp or version of each buffered event against the newly fetched state to discard any stale data (i.e., events that are older than the state snapshot). Any newer events are then applied.
- After the buffer is cleared and the state is consistent, the client transitions to the ACTIVE state.
- **Hide Loading Indicator:** The "Synchronizing" UI is removed.
- **Live Processing:** The client now begins processing new incoming WebSocket messages in real-time, applying updates directly to its state store and UI.

This synchronization flow ensures that no matter when the client connects or reconnects, it always starts from a backend-verified source of truth, creating a resilient and reliable user experience.

Fallback mechanism

We will apply a multi-layer fallback strategy, when first the message is delivered with Websockets, then with Push notification and then with Firestore and we will keep the smart_polling mechanism on mobile.

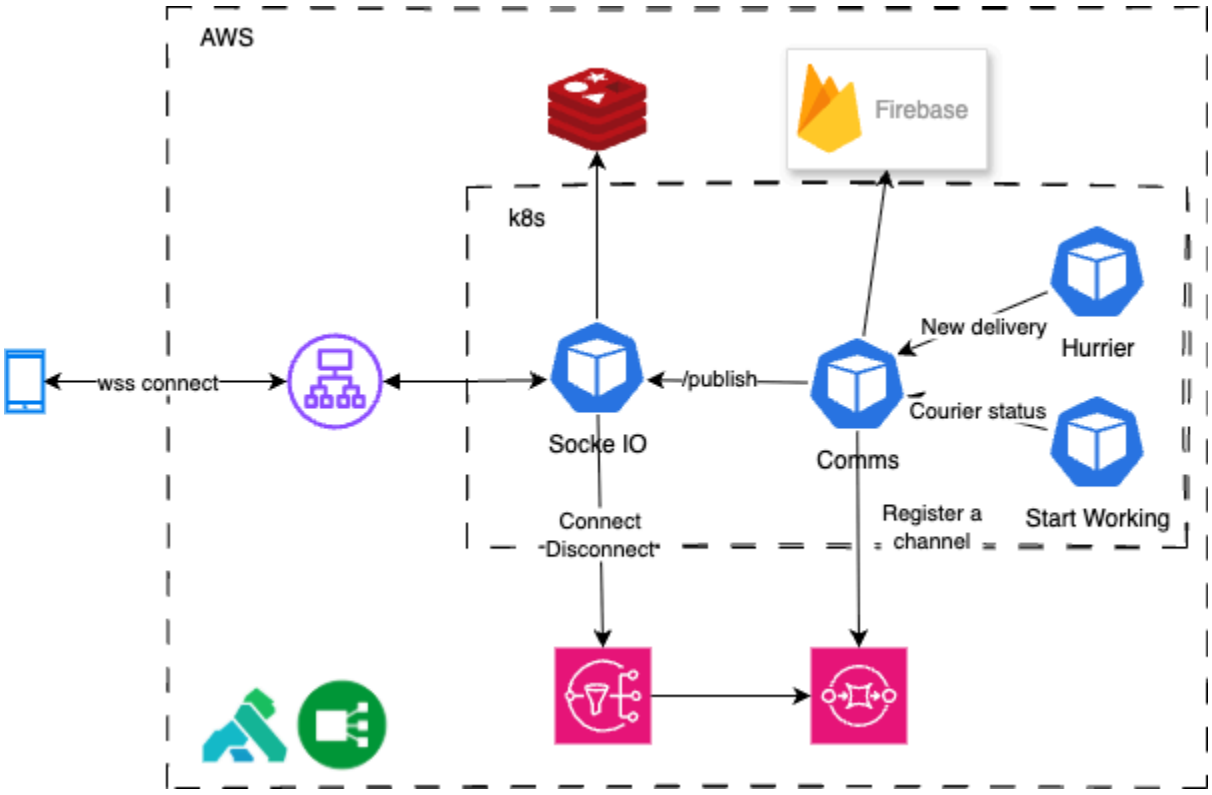


Where? The best place to implement all of these fallbacks is comms-api. It has the concept of “channels” (it can be PUSH_NOTIFICATION, EMAIL, SMS). If a channel exists, this means we have a valid push notification token or email or phone number. We can apply the same logic:

- When the device connects to Websocket, we listen to these events and store “websocket” channels to the system.
- If we lose connection to the device, we also listen to this event and delete this channel.

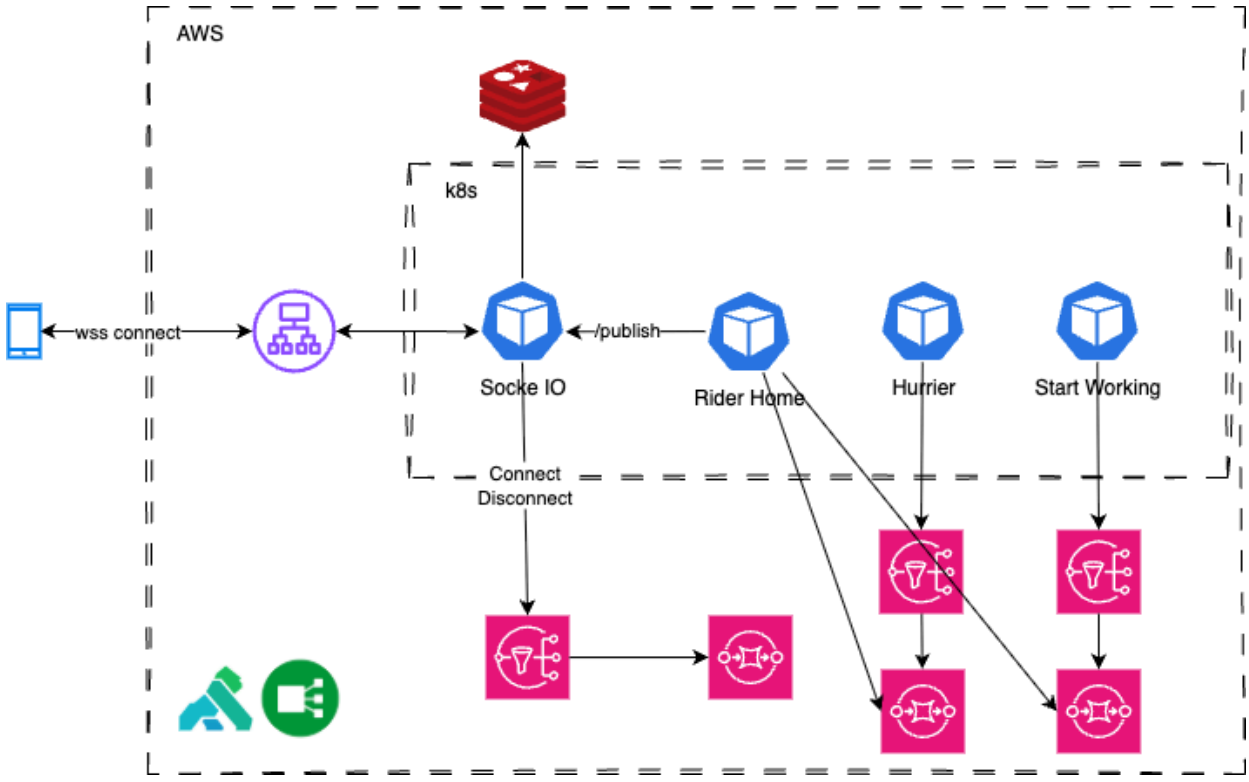
Then we can use the worker to iterate on undelivered messages and change channels to the next available way of communication.

Proposed Architecture (Northstar)



Utilize CHANNEL functionality from comms along with fallbacks to PN and FireStore.

Proposed PoC Architecture



To speed up development of PoC we propose to utilize one (or few) existing services which are already listening to events in Logistics and can call /publish endpoint. We can use Rider Home, which is already listening to multiple Rider related endpoints and extend it to listen to a few more related to Delivery.

Other technologies considered

- AWS IoT
  - Price
- Long polling
  - Missing client-side events, like GPS coordinate updates
- Server side events
  - Missing client-side events, like GPS coordinate updates

Rollout and rollback plan

At the core of this milestone is to keep the operation reliable for our riders.

Realtime solution should allow fallback to smart polling mechanism in case of the repeated permanent connectivity errors.

Rollout

The rollout should be done using the frontend feature flags. Clients on their side should implement a circuit breaker to ensure we are able to cut off the new networking code (one flag per platform).

A runbook must be prepared to explain the feature implications and configuration:

- Define the feature in a simple words
- Contain direct links to the monitoring dashboards,
- Include all information on the mitigation and its hierarchy (backend first, mobile as a last resort): feature flags and links to the flags configurations
- List the known potential issues

Rollout must be communicated with all relevant stakeholders and conducted based on the common rollout rules defined: staged rollout should happen region per region, starting with only one. In every region, the rollout should start with a small country, following the existing procedure defined in Firebase Feature rollout Policy for Mobile apps

Rollback

Rollback should be performed with the client first. Client side circuit breaker should be used as a last resort mitigation scenario.

Discussions

-

Observability

Observability must provide confidence during the rollout.

Footer

Appendix

Example WebSocket Server implementation using Socket.IO

The following is a demonstrative example, further modifications are necessary:

TypeScript

```typescript
import express from 'express';
import { createServer } from 'http';
import { Server, Socket } from 'socket.io';
import { createAdapter } from '@socket.io/redis-adapter';
import Redis from 'ioredis';
import jwt, { JwtPayload } from 'jsonwebtoken';
import pino from 'pino';

// -------------------- Type Definitions --------------------

interface DecodedToken extends JwtPayload {
    riderId: string;
    zoneIds: string[];
}

interface AckMessage {
    metadata: {
        eventType: 'ack';
    };
    payload: {
        messageId: string;
    };
}

interface RiderSocket extends Socket {
    riderId: string;
    zoneIds: string[];
}

// -------------------- Configuration --------------------

const logger = pino();
const PORT = process.env.PORT || 3000;
const JWT_PUBLIC_KEY = process.env.JWT_PUBLIC_KEY || 'your-public-key-here or
DH STS integration...';
const REDIS_NODES = [
    { host: 'redis-node-1', port: 6379 },
    { host: 'redis-node-2', port: 6379 },
```

```
];
const REDIS_OPTIONS = {};

// -------------------- Express and Socket.IO Setup --------------------

const app = express();
const server = createServer(app);
const io = new Server(server, {
    cors: {
        origin: '*',
        methods: ['GET', 'POST']
    }
});

app.use(express.json());

// -------------------- Redis Cluster Adapter --------------------

const pubClient = new Redis.Cluster(REDIS_NODES, REDIS_OPTIONS);
const subClient = pubClient.duplicate();
io.adapter(createAdapter(pubClient, subClient));

logger.info('Redis Adapter connected for horizontal scaling.');

// -------------------- Authentication Middleware --------------------

io.use((socket, next) => {
    const token = socket.handshake.auth.token as string;
    if (!token) {
        return next(new Error('Authentication error: Token missing.'));
    }
    try {
        const decoded = jwt.verify(token, JWT_PUBLIC_KEY) as DecodedToken;
        (socket as RiderSocket).riderId = decoded.riderId;
        (socket as RiderSocket).zoneIds = decoded.zoneIds;
        logger.info({ riderId: decoded.riderId, zoneIds: decoded.zoneIds },
'Socket authenticated.');
        next();
```

```
    } catch (err) {
        next(new Error('Authentication error: Invalid token.'));
    }
});

// -------------------- Rate Limiting Abstraction --------------------

const useRateLimiter = (socketId: string, messageSize: number) => {
    return new Promise<boolean>((resolve) => {
        const isAllowed = Math.random() > 0.1;
        resolve(isAllowed);
    });
};

// -------------------- WebSocket Connection Lifecycle --------------------

io.on('connection', (socket) => {
    const riderSocket = socket as RiderSocket;
    logger.info({ id: riderSocket.id, riderId: riderSocket.riderId, zoneIds:
riderSocket.zoneIds }, 'Client connected.');
    riderSocket.join(`acceptance.delivery.rider-${riderSocket.riderId}`);
    riderSocket.join(`rider-availability.state.rider-${riderSocket.riderId}`);
    riderSocket.zoneIds.forEach(zoneId => {
        riderSocket.join(`rider-availability.zone-info.${zoneId}`);
    });

    riderSocket.on('message', async (message: AckMessage) =>
        const isAllowed = await useRateLimiter(riderSocket.id, message.length);
        if (isAllowed) {
            // process AckMessage here
        } else {
            logger.warn({ riderId: riderSocket.riderId }, 'Rate limit exceeded
for ACK.');
        }
    });
    riderSocket.on('disconnect', () => {
        logger.info({ id: riderSocket.id }, 'Client disconnected.');

       // leave rooms
```

```
    });
});

// -------------------- HTTP Endpoint for Publication --------------------
// Validation abstraction is omitted for simplication



app.post('/publish', (req, res) => {
    const { topic, riderId, zoneId, payload } = req.body;

    if (!payload || !topic) {
        return res.status(400).send('Payload and topic are required.');
    }

    let targetRoom: string;
    if (topic.startsWith('rider-availability.zone-info.') && zoneId) {
        targetRoom = `rider-availability.zone-info.${zoneId}`;
    } else if (topic.startsWith('acceptance.delivery.rider-') && riderId) {
        targetRoom = `acceptance.delivery.rider-${riderId}`;
    } else if (topic.startsWith('rider-availability.state.rider-') && riderId)
{
        targetRoom = `rider-availability.state.rider-${riderId}`;
    } else {
        return res.status(400).send('Invalid topic or missing identifier.');
    }

    io.to(targetRoom).emit('message', {
        metadata: {

            // other headers
```

```
            timestamp: Date.now(),
            topic,
        },
        payload,
    });

    logger.info({ topic: targetRoom }, 'Message published.');
    res.status(200).send('Message published.');
});

// -------------------- Server Start --------------------

server.listen(PORT, () => {
    logger.info(`Socket.IO server listening on port ${PORT}`);
});
```