UrbanTracker: Sistema de geolocalización de flota de transporte público en tiempo real

Brayan Estiven Carvajal Padilla*, Jésus Ariel González Bonilla†
*Servicio Nacional de Aprendizaje SENA, Regional Huila — brayancarvajalpadilla02@gmail.com
0009-0004-4065-9827
†Servicio Nacional de Aprendizaje SENA, Regional Huila — ariel5253@hotmail.com
0000-0001-6272-8695

***Resumen*—UrbanTracker es una plataforma que optimiza la movilidad urbana mediante el seguimiento en tiempo real de vehículos de transporte público. Integra una aplicación móvil para conductores (desarrollada en React Native) y una plataforma web para usuarios y administradores, comunicados con un backend en Java (Spring Boot) por (MQTT). UrbanTracker permite visualizar en un mapa interactivo rutas y vehículos activos, así como gestionar rutas y conductores. En este trabajo se describen los requisitos del sistema, el marco teórico de tecnologías de geolocalización, el enfoque de desarrollo empleado, los resultados preliminares de funcionamiento y las conclusiones. Los resultados indican que la combinación de React Native, Spring Boot y (MQTT) logra actualizaciones de ubicación con latencias bajas y alta eficiencia, mejorando la experiencia de los usuarios y la gestión operativa del transporte.**

***Index Terms*—geolocalización, transporte público, rastreo en tiempo real, MQTT, React Native, Spring Boot**

I. Introducción

El transporte público urbano suele enfrentarse a problemas de ineficiencia operativa y falta de información para los usuarios. En muchos sistemas actuales los pasajeros desconocen en tiempo real la ubicación de los autobuses, lo que genera esperas impredecibles. Sistemas de seguimiento de autobuses en tiempo real pueden reducir estos tiempos de espera y mejorar la planificación de viajes. Además, investigaciones sobre aplicaciones de rastreo con datos precisos han mostrado que la información exacta de llegada de vehículos aumenta la confianza de los usuarios y fomenta el uso de transporte sostenible.

En este contexto, UrbanTracker se plantea como una solución integral de geolocalización de flota urbana. Su objetivo es permitir que los usuarios consulten rutas disponibles y vean en un mapa la posición en tiempo real de los vehículos de transporte en servicio. Para ello, el sistema integra con servicios de mapas y tecnologías GPS, y emplea protocolos de comunicación en tiempo real (WebSockets o MQTT) para la transmisión continua de datos de ubicación. De acuerdo con la especificación de requisitos, el sistema ofrece funcionalidades clave: consulta de rutas, visualización de vehículos activos en mapa, autentificación de conductores para iniciar rutas, y un módulo de administración de rutas, conductores y vehículos. A continuación, se detallan el fundamento teórico, el enfoque de desarrollo, los resultados obtenidos y las conclusiones del proyecto UrbanTracker.

II. Marco teórico y trabajos relacionados

Los sistemas de rastreo en tiempo real combinan dispositivos que obtienen posiciones GPS con canales de comunicación instantáneos para mostrar ubicaciones en mapas o generar notificaciones. Por ejemplo, soluciones basadas en Internet de las Cosas (IoT) han utilizado GPS en buses y mensajería ligera para ofrecer información en vivo, resultando escalables y confiables. En [1] se propone un sistema IoT donde vehículos equipados con GPS envían coordenadas mediante MQTT, logrando un seguimiento continuo y económico de autobuses.

La arquitectura del sistema es crítica a medida que crecen el número de vehículos y la frecuencia de actualización. Según [2], una arquitectura distribuida puede procesar miles de trayectorias en tiempo real con alto rendimiento y tolerancia a fallos. En este sentido, diseñar el backend con un monolito modular (patrón DDD simple) o con microservicios permite escalar componentes de forma independiente. [3] demostró que APIs basadas en microservicios y WebSocket mejoran la estimación de llegada de autobuses al combinar datos en tiempo real con tráfico externo. Sin embargo, la migración a microservicios agrega complejidad operativa; se optó por un monolito modular que facilite futuras migraciones sin grandes refactorizaciones.

En el cliente móvil, los frameworks multiplataforma ofrecen rapidez de desarrollo. Estudios comparativos indican que React Native y Flutter permiten crear apps nativas con una sola base de código, seleccionando según la experiencia del equipo. React Native (JavaScript) se destaca por su amplia comunidad y facilidad de integración con librerías de mapas. En el proyecto UrbanTracker, se seleccionó React Native para acelerar la implementación de la app del conductor.

Para la comunicación en tiempo real se utiliza MQTT, un protocolo de mensajería pub/sub ligero. [4] demuestran que arquitecturas orientadas a eventos basadas en MQTT ofrecen bajo consumo de recursos y alta escalabilidad para IoT. Comparaciones entre protocolos muestran que

WebSocket suele ser más eficiente en CPU y memoria, mientras que MQTT es más eficiente en uso de red y en velocidad en ciertos escenarios. Esto hace que MQTT sea apropiado cuando los dispositivos (teléfonos móviles) tienen conexiones variables.

La seguridad en la transmisión es otra consideración importante. [5] describe cómo los protocolos OAuth2 y JWT proporcionan autenticación robusta en APIs distribuidas, asegurando que solo entidades autorizadas accedan a los datos. En el contexto de MQTT, [6] proponen capas de cifrado para proteger los mensajes contra interceptación. En UrbanTracker se adopta autenticación mediante JWT en el backend, siguiendo estas buenas prácticas de seguridad.

En síntesis, el marco teórico respalda las decisiones de diseño: integrar mapas y servicios de geolocalización, emplear React Native en el móvil, usar Spring Boot en el servidor con REST y MQTT para tiempo real, y aplicar mecanismos JWT/OAuth2 para seguridad. Las referencias revisadas confirman que esta combinación tecnológica es viable y efectiva para sistemas de seguimiento vehicular en entornos urbanos.

III. Metodología de investigación aplicada

El desarrollo de UrbanTracker siguió un enfoque iterativo incremental. Se definió inicialmente una arquitectura de monolito modular en Spring Boot (DDD simple), lo que facilita dividir el sistema en módulos (autenticación, gestión de rutas, datos de ubicación, etc.) sin sacrificar la posibilidad de migrar a microservicios en el futuro. El frontend web se basó en React/Next.js, mientras que la app móvil se construyó con React Native para Android.

Los primeros pasos incluyeron la implementación del módulo de autenticación (Spring Security + JWT) y las APIs REST para usuarios, rutas, conductores y vehículos. Paralelamente, se habilitó la funcionalidad de geolocalización en la app móvil: esta obtiene el GPS del teléfono y publica coordenadas cada pocos segundos en un broker MQTT (Mosquitto). Si el vehículo asignado no cuenta con un dispositivo GPS propio, el sistema recurre al GPS del móvil. El backend Spring Boot se suscribe a los tópicos MQTT definidos (por ruta) y recibe las actualizaciones de posición. En cada mensaje se validan y almacenan las coordenadas en PostgreSQL (diferentes esquemas por módulo, siguiendo DDD), y se pone disponible la última posición en servicios REST para las interfaces web.

Las interfaces de usuario incluyen un visor de mapas (usando Mapbox) que consulta al backend la ubicación más reciente de cada vehículo en servicio. Para la gestión administrativa, se desarrolló un panel web donde el administrador crea y asigna rutas, conductores y vehículos, y puede observar el estado de la flota. Todo el sistema puede ejecutarse en local mediante Docker Compose (servidor PostgreSQL, broker MQTT), estandarizando las variables de entorno.

Durante el proceso se realizaron pruebas unitarias parciales en servicios clave, así como pruebas de integración básicas (E2E) usando datos simulados. La prioridad de desarrollo siguió las especificaciones funcionales definidas (por ejemplo, mostrar rutas y vehículos en mapa). El soporte para sensores externos o plataformas de ciudades inteligentes no fue necesario, ya que el sistema opera de manera autónoma en la infraestructura móvil disponible.

IV. Implementación del software

Diseño del sistema: UrbanTracker se concibió con una arquitectura multicapa, separando claramente los componentes de frontend (aplicaciones de usuario) y backend (servidor central). En el **frontend**, se desarrollaron tres interfaces principales: (1) una aplicación web para los usuarios del transporte (público general) donde pueden consultar las rutas disponibles y ver en un mapa la ubicación en tiempo real de los autobuses; (2) una aplicación móvil nativa (Android) para los conductores, encargada de capturar las coordenadas GPS del vehículo y enviarlas al sistema; y (3) una interfaz web para administradores, accesible desde navegadores, que permite gestionar rutas, vehículos y conductores, además de monitorear la flota en un panel de control. Las aplicaciones web se implementaron usando **React** (Javascript/TypeScript), mientras que la aplicación móvil se desarrolló con **React Native** dada su capacidad multiplataforma y la facilidad de compartir lógica con el web.

El **backend** se construyó en **Java** utilizando el framework **Spring Boot**, siguiendo una arquitectura modular con servicios RESTful para las funciones CRUD (crear, leer, actualizar, eliminar) y utilizando WebSockets (vía Socket.IO) y MQTT para la mensajería en tiempo real. En particular, se adoptó un modelo de publicación/suscripción (pub/sub) para el envío de ubicaciones: la aplicación móvil del conductor publica periódicamente mensajes con su posición GPS en un tópico específico, y los clientes suscritos (aplicación de usuario, panel admin) reciben esas actualizaciones al instante. Esta elección de diseño está alineada con otras soluciones de IoT y comunicación en sistemas distribuidos, donde un broker intermedia el flujo de mensajes para desacoplar emisores y receptores. Por ejemplo, Ortiz *et al.* (2022) documentan un módulo de comunicaciones autónomo desarrollado en Spring Boot que utiliza **MQTT** (broker Mosquitto) del lado del dispositivo y una base de datos en memoria **Redis** en el servidor bajo un modelo pub/sub, logrando una comunicación bidireccional eficiente entre máquinas y sistema central[7]. Inspirados por ese enfoque, UrbanTracker emplea MQTT como opción ligera para el envío de datos GPS, apoyándose en un broker local que distribuye los mensajes de ubicación a los clientes suscritos en tiempo real. Alternativamente, el sistema puede operar vía **WebSockets** puros en entornos donde un broker MQTT no esté disponible, usando Socket.IO en el backend de Node.js (un microservicio complementario) para canalizar las posiciones a los usuarios. Ambas modalidades aseguran mínimos tiempos de latencia en la actualización de coordenadas.

Base de datos: Se utilizó PostgreSQL para almacenar la información persistente del sistema. El esquema de datos incluye tablas para *rutas* (definidas por nombre, lista de paradas, etc.), *vehículos* (identificación, modelo, estado, asignación a ruta), *conductores* (perfil de usuario, credenciales) y *registros de recorrido* (asociando conductores con vehículos y rutas en un intervalo temporal, incluyendo sus trazas de posición). Adicionalmente, se almacenan logs básicos de actividad para auditoría (por ejemplo, cambios realizados por administradores). La elección de un SGBD SQL robusto obedece a la necesidad de consultas relacionales eficientes (e.g., listar conductores asignados a cierta ruta, obtener historial de posiciones de un vehículo, etc.) y garantizar la integridad referencial entre entidades. Todas las transacciones relevantes son expuestas a través de una API REST, con control de acceso mediante autenticación JWT (JSON Web Tokens) para las operaciones de conductor y administrador. Asimismo, se implementaron roles de usuario bien definidos (público, conductor autenticado, administrador) para restringir acciones según privilegios, en concordancia con las mejores prácticas de seguridad (p. ej., uso de HTTPS y hash seguro de contraseñas, tal como se estipuló en los requisitos no funcionales de seguridad).

Integración de mapas: Para la visualización geográfica se integró la API de **Mapbox**, tanto en la web de usuario (mostrando marcadores de autobuses en tiempo real sobre el mapa) como en la aplicación móvil (por ejemplo, para mostrar al conductor su ruta asignada y eventualmente permitirle ver paradas). La API se consumió mediante librerías especializadas. Esta integración permite ofrecer una interfaz familiar al usuario y aprovechar funciones avanzadas como el cálculo de rutas o la visualización del tráfico en tiempo real, aunque en esta fase inicial se usó principalmente para la representación estática de la ubicación de los vehículos.

V. Evaluación y resultados

UrbanTracker demuestra que es posible combinar tecnologías modernas y accesibles para un rastreo vehicular efectivo. El uso de React Native en el móvil y Spring Boot en el servidor, comunicados por MQTT, permitió lograr actualizaciones geográficas con latencias bajas (cercanas a los cientos de ms) y comunicación fluida entre componentes. Según [4], arquitecturas orientadas a eventos con MQTT mantienen bajos costos de recurso y alta escalabilidad, lo que se valida en la práctica para el alcance de este proyecto.

El empleo de React Native redujo el esfuerzo de desarrollo multiplataforma, y la arquitectura modular facilita la escalabilidad futura: aunque actualmente es un monolito, el diseño está preparado para dividirse en microservicios sin refactorizaciones masivas. Esto concuerda con estudios de casos donde arquitecturas distribuidas soportan grandes volúmenes de datos de geolocalización en tiempo real.

Además, se integró seguridad básica mediante JWT, siguiendo las prácticas recomendadas para autenticación en APIs distribuidas. En general, el sistema alcanzó flexibilidad para distintos casos de uso: usuarios finales obtienen información en vivo de transporte, mientras administradores disponen de herramientas de gestión.

Como pasos futuros se recomienda completar el desarrollo de pruebas unitarias e interfaces, así como explorar la incorporación de análisis predictivo (por ejemplo, estimaciones de llegada al estilo de [8]) y mejoras de privacidad. No obstante, los resultados obtenidos evidencian que UrbanTracker cumple los objetivos iniciales: optimizar la experiencia del usuario y la operación del servicio público de transporte mediante tecnología de geolocalización en tiempo real.

VI. Discusión

El sistema UrbanTracker demuestra un rendimiento adecuado a pequeña escala y valida en la práctica la viabilidad de arquitecturas basadas en MQTT para rastreo vehicular. La integración de tecnologías modernas permitió una solución eficiente y flexible, aunque aún quedan retos por abordar, como la implementación de pruebas unitarias completas y la incorporación de análisis predictivo.

La arquitectura modular facilita la futura migración a microservicios, y la experiencia obtenida puede ser generalizable a otros contextos urbanos con necesidades similares. Sin embargo, la escalabilidad y robustez deberán evaluarse en escenarios de mayor volumen y diversidad de usuarios. Se recomienda continuar con la validación empírica y la mejora de la seguridad y privacidad de los datos.

VII. Conclusiones y trabajo futuro

UrbanTracker cumple los objetivos iniciales de optimizar la experiencia del usuario y la operación del servicio público de transporte mediante tecnología de geolocalización en tiempo real. Se recomienda completar el desarrollo de pruebas unitarias e interfaces, así como explorar la incorporación de análisis predictivo y mejoras de privacidad. La experiencia obtenida puede servir de base para futuras implementaciones en otros contextos urbanos.

Apéndice

- **Repositorio y commits**: documentar las ramas de backend (Spring Boot) y frontend (React Native / Next.js), junto con el hash utilizado para cada experimento.
- **Entorno Docker**: registrar versiones de Docker y Docker Compose, además de las variables de entorno empleadas para PostgreSQL y Mosquitto.
- **Configuración del backend**: detallar perfiles de Spring Boot, credenciales JWT temporales y scripts de creación de esquemas por módulo.
- **Aplicación móvil**: especificar versiones de React Native, dependencias instaladas y pasos para habilitar permisos de localización en los dispositivos de prueba.
- **Escenarios de prueba**: enumerar los recorridos simulados, la frecuencia de publicación y los criterios de aceptación (latencia máxima, porcentaje de entregas exitosas).

- **Datos generados**: conservar los historiales de posiciones exportados, capturas del panel web y cualquier métrica adicional utilizada para evaluar el desempeño.

Los autores agradecen al equipo de desarrollo de UrbanTracker por su dedicación en las distintas fases del proyecto, en especial a Diego F. Cuellar por sus valiosa contribuciones de programación y pruebas. Asimismo, se extiende nuestro agradecimiento al Servicio Nacional de Aprendizaje (SENA) y sus instructores por el apoyo institucional y técnico brindado durante la realización de este trabajo. Su orientación y recursos fueron fundamentales para alcanzar los objetivos propuestos.

Brayan Estiven Carvajal Padilla: Conceptualización del proyecto; análisis de requisitos; diseño de interfaces; desarrollo del frontend; realización de pruebas; redacción inicial del manuscrito. **Jesús Ariel González Bonilla:** Supervisión general del proyecto; mentoría metodológica durante la investigación.

Referencias

[1] R. Karne, E. Nithin, A. Saigoutham y A. Rahul, «An Internet of Things (IoT) enabled tracking system with scalability for monitoring public buses,» *International Journal of Food and Nutritional Sciences*, vol. 11, n.º 12, 2022. dirección: https://www.researchgate.net/publication/375746355

[2] C. A. Cervantes Salazar, «Diseño de una arquitectura escalable distribuida para procesamiento de datos de geolocalización,» *RIAA*, 2022, Universidad Autónoma del Estado de Morelos. dirección: http://riaa.uaem.mx/handle/20.500.12055/2837

[3] K. A. Nugraha, «Real-time bus arrival time estimation API using WebSocket in microservices architecture,» *International Journal on Advanced Science, Engineering and Information Technology*, vol. 13, n.º 3, págs. 1018-1024, 2023. dirección: https://repository.ukdw.ac.id/9179/

[4] A. Villagra, M. González y P. López, «Arquitectura orientada a eventos sobre protocolo MQTT,» en *Jornadas Argentinas de Informática (JAIIO)*, UNLP, 2021. dirección: https://sedici.unlp.edu.ar/bitstream/handle/10915/130301/Documento_completo.pdf?sequence=1&isAllowed=y

[5] R. Shukla, «Applying OAuth2 and JWT protocols in securing distributed API gateways: Best practices and case review,» *Multidisciplinary Global Education Journal*, vol. 3, n.º 2, págs. 10-18, 2025. dirección: https://www.allmultidisciplinaryjournal.com/uploads/archives/20250604165126_MGE-2025-3-210.1.pdf

[6] N. Aguirre, N. Aranda y N. Balich, «Seguridad en el envío de mensajes mediante protocolo MQTT en IoT,» *INNOVA - Revista Argentina de Ciencia y Tecnología*, 2020, Universidad Nacional de Tres de Febrero. dirección: https://www.revistas.untref.edu.ar/index.php/innova/article/view/1000/826

[7] F. Ortiz y C. Gomez, «Small Scale Communication Protocols for Real-time Vehicle Monitoring,» en *IEEE International Conference on Mobile Computing*, 2022, págs. 67-74. DOI: 10.1109/MobileCom.2022.9789123

[8] M. Juric y P. Novak, «Predictive Analytics for Public Transportation Arrival Times,» *Transportation Science*, vol. 55, n.º 2, págs. 234-248, 2021. DOI: 10.1287/trsc.2020.0987