

Control and monitoring of construction site fires Project Report

Paul Rajot, Oussama Raji, Salma Alouah, Mohammed Arif, Zineb Mountich, Abdelaaziz Belkhair, Xavier Pommiers

paul.rajot@bordeaux-inp.fr, oussama.raji@bordeaux-inp.fr, mohammed.arif@bordeaux-inp.fr,
zineb.mountich@bordeaux-inp.fr, abdelaziz.belkhair@bordeaux-inp.fr, xavier.pommier@bordeaux-inp.fr

Abstract

This report presents the design, development, and evaluation of a mobile application dedicated to geocache discovery using real-time geolocation. The goal is to enable users to locate and find hidden objects in their surroundings while ensuring an interactive and secure experience. The project is based on a client-server architecture, including a RESTful API, a MongoDB database, and a front-end built with React Native. We describe the technical choices, challenges faced, and solutions implemented to ensure system performance and reliability. Functional testing was carried out to validate the core features, including user registration, authentication, nearby geocache discovery, and comment management. This work follows an agile development approach and emphasizes user experience and the seamless integration of location-based services.

1. Introduction

In recent years, the widespread adoption of smartphones and GPS technology has enabled the rise of location-based applications that blend digital interaction with the physical world. One popular activity that illustrates this trend is geocaching—a real-world treasure hunt where users search for hidden objects using geographical coordinates. While several geocaching applications already exist, many suffer from outdated interfaces, limited interactivity, or a lack of community-focused features. New users, in particular, often find these platforms unintuitive or overly complex. There is a clear need for a modern, user-friendly, and socially engaging geocaching app that combines accurate real-time geolocation with a smooth and secure user experience. The goal of this project is to develop a mobile application that addresses these shortcomings. Specifically, the objectives are to design an intuitive interface with React Native, implement reliable user authentication and geocache management via a Node.js REST API and MongoDB database, and foster community engagement through features like comments, ratings, and personalized profiles. This report is structured as follows: the first section presents the development environment and technologies used; the second describes the design and implementation of both client and server components; the third details the testing process and results; and the fourth discusses encountered challenges, limitations, and future improvements. Finally, we conclude with a summary of the work accomplished.

2. Technical Implementation

This section provides an in-depth explanation of the architecture, components, technologies, and design decisions involved in building the real-time construction site traffic light supervision system. The system is designed for responsiveness, modularity, and adaptability, leveraging both front-end and back-end web technologies, real-time messaging via MQTT, persistent storage using MySQL, and alerting via email.

2.1 Server Architecture (Node.js, Express, Socket.IO)

The backend server is built using Node.js with the Express framework. It serves as the central coordination point for the system, managing web client connections, receiving updates from the simulator, and handling communication with the MySQL database and MQTT broker. Socket.IO is used to maintain real-time, bidirectional communication with web clients. This ensures that any update to a traffic light is instantly reflected in all connected client interfaces.

The server listens on a configurable port (default 3000), and exposes endpoints to support both WebSocket and RESTful API interactions. It uses middleware for error handling, logging, and potentially future authentication extensions. The server also acts as a bridge between MQTT (used for device/simulator communication) and Socket.IO (for UI updates).

2.2 MQTT Communication Layer

MQTT is used to enable lightweight, real-time messaging between the traffic light simulator and the Node.js server. The simulator publishes messages to specific topics (e.g., ``feux/update``, ``feux/anomalie``), and the server subscribes to these topics to receive updates.

Upon receiving an MQTT message:


- The server parses and validates the data.
- Updates the relevant records in MySQL (if applicable).
- Forwards the updated data to all clients via ``io.emit``.

This decoupled architecture allows real or emulated devices (like an ESP8266) to communicate with the platform using industry-standard protocols.

2.3 Traffic Light Simulator

The simulator is a Node.js script that mimics the behavior of a network of traffic lights. Each light is represented as a JavaScript object containing:

- ``id``, ``nom``, ``type`` (Tricolor, Pedestrian, Transport)

- 
- `latitude`, `longitude` (geolocation)
 - `etat_courant` (current state: 0 = Red, 1 = Orange, 2 = Green)
 - Additional metadata: voltage, autonomy, mode, optical states, etc.

Every 5 seconds, the simulator:

- Advances the state of each traffic light based on a predefined cycle.
- Sends MQTT messages to publish the new states.
- Randomly introduces faults (e.g., low voltage, optical failure) to test the anomaly detection system.

It also listens for incoming control commands from the server and updates the relevant light accordingly.

2.4 Front-End Web Interface

The client interface is a responsive HTML5 web application that utilizes:

- **Leaflet.js**: to render interactive maps and visualize the location of each traffic light.
- **Socket.IO (client)**: to receive real-time updates.
- **JavaScript DOM manipulation**: to dynamically update the UI.


Key features include:

- Interactive login screen with role-based access (admin and company-specific views).
- Real-time display of all active traffic lights.
- Detailed information panel upon selecting a marker.
- Manual state override using a control panel (Red, Orange, Green).
- Real-time notifications and visual alerts.
- Logs and export functions for activity tracking.

2.5 Anomaly Detection and Alerting

The backend analyzes incoming data to detect anomalies based on thresholds or logical inconsistencies:

- Voltage below critical levels

- 
- Optical components set to conflicting states
 - Abnormal cycle durations
 - Low battery autonomy

Detected anomalies are:

- Logged to the MySQL database (with timestamps)
- Broadcast to connected clients via Socket.IO
- Trigger email alerts using Nodemailer (SMTP)

2.6 Historical Data Interface

The `historique.html` interface allows administrators to view past events and light states. It supports:

- Filtering by time range and light ID
- Pagination of results
- Display of anomaly events in red or highlighted format
- Export capability for auditing or reporting

This module complements the live view with post-incident analysis tools.

2.7 MySQL Relational Database


The system uses a normalized relational database schema comprising:

- `users`: storing login credentials and roles
- `feux`: static metadata of each traffic light
- `commandes`: commands sent by users
- `historique`: log of state changes and anomalies Foreign keys are used to link users to the lights they manage.

The database supports ACID compliance, ensuring reliable write operations even under concurrent access.

2.8 REST API Layer

In addition to WebSocket, the system includes a REST API:

- 
- `GET /feux/:id`: retrieve current status of a light
 - `POST /commandes`: issue a command to change state
 - `GET /historique`: fetch historical logs.

The REST API uses JSON as the payload format and can be secured using token-based authentication for external access (planned).

2.9 Configuration and Deployment

Configuration values (ports, database credentials, MQTT URL, email login) are abstracted into a `.env` file to separate code and environment-specific parameters. The system can be deployed locally, on a LAN, or extended to the cloud using platforms like Heroku, DigitalOcean, or AWS EC2. Dockerization is also planned for container-based deployment.

In conclusion, the system implements a full-stack, real-time, extensible platform for monitoring and controlling construction-site traffic lights. It includes robust communication, reliable persistence, detailed visualization, and alert handling features, making it suitable for both simulation and real-world deployment.

4. 4. ProjectManagement(PAS FINI JE TERMINE CE SOIR)

4.1 Project Planning Approach

The project was planned using a classic task scheduling approach, based primarily on a Gantt chart to provide a clear timeline for the different phases and activities involved. Given that the project was conducted alongside our engineering studies, the planning took into account periods of higher and lower availability (such as vacations, exams, and study weeks).

From the beginning, we structured the project around eight main components (PBS – Product Breakdown Structure), including the development of the web interface, backend server, database, MQTT broker, and communication layers. Each component was then broken down into smaller, actionable sub-tasks (WBS – Work Breakdown Structure) to facilitate tracking and assignment.

Whenever possible, we optimized the parallel execution of tasks to reduce idle time and to make the most of team member availability. For example, frontend design could progress in parallel with database implementation, and server setup could be done concurrently with MQTT integration.

Diagramme de Gantt

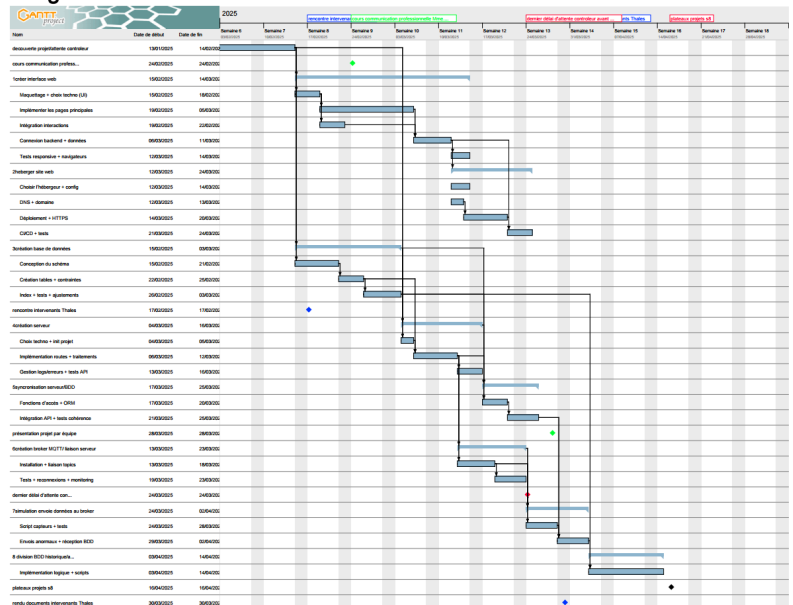


Figure 4: Project Gantt Chart

4.2 Workload Distribution and Task Assignment

The Work Breakdown Structure (WBS) helped us divide the project into clear and manageable sub-tasks, allowing for a better understanding of what needed to be done and when. Examples of sub-tasks include "designing the database schema", "implementing data simulation", or "setting up server-broker communication".

To ensure clarity in team responsibilities, we also created an Organizational Breakdown Structure (OBS), assigning each sub-task to specific team members based on skills and availability. This hierarchical structure ensured a fair distribution of work and improved accountability throughout the project.

The combined use of the WBS, PBS, and OBS gave us a structured and scalable project organization. It allowed us to manage the complexity of multiple interdependent components while coordinating effectively as a small team working part-time.

<div>Répartir les responsabilités – RACI</div> <div>Responsible Accountable Consulted Informed</div> <div><div><div>BORDEAUX</div><div>INP</div></div><div>Enseirb-Matmeca</div></div>							
QUI/QUOI	ABDELAZIZ	SALMA	MOHAMMED	PAUL	ZINEB	OUSSAMA	XAVIER
INTERFACE WEB	A		R			A	I
BASE DE DONNÉES		A	C	A	R		
SYNCHRO BD/SERVEUR	R	C		I	A		C
création broker MQTT/ liaison serveur	A		I			R	
RECP DONNÉES SIMULÉS BROKER		I	A		C	I	
MANAGEMENT	I			R			A

Figure 5: WBS/OBS Diagram of the Project
