Report: Roadsense

Edge Computing in the IoT

Luca Di Bello, Paolo Deidda, Nawfal Abdul Malick, Georg Meyer

Semester-project

Due: 20 December 2024

Contents

List of Figures

Chapter 1

System Design

1.1 Introduction

The **RoadSense** project aims to develop an IoT-based system to detect and map road anomalies such as bumpiness and potholes. By installing sensor nodes on multiple vehicles, the system collects and analyzes road vibration data to create a detailed, interactive heatmap of road conditions. This information is invaluable for road maintenance planning, improving driver safety, and providing real-time alerts for hazardous conditions.

Objectives

This project aims to address the following objectives:

1. Develop a cost-effective IoT-based system to detect and map road anomalies.
2. Quantify road bumpiness levels and provide real-time alerts for hazardous conditions.
3. User-friendly interface for stakeholders to visualize road conditions and manage alerts.
4. Improve the accuracy of road conditons by using multiple vehicles for data collection.

1.2 Overview

The **RoadSense** system consists of the following components:

- **Sensor Nodes**: IoT devices installed in vehicles, responsible for collecting inertial data using an Inertial Measurement Unit (IMU) sensor and location data via a GPS module. These nodes should already compute a qualifier for localized road states to minimize data traffic to centralized hubs.
- **Server-Side Application**: Collects, aggregates, and analyzes data from multiple devices, and visualizes road quality using heatmaps.
- **Control Logic**: Defines the behavior of the IoT device in terms of data collection, processing, and communication.

- **User Interface**: An interactive web application allowing stakeholders to visualize road conditions and manage alerts.

1.3 Sensor Nodes

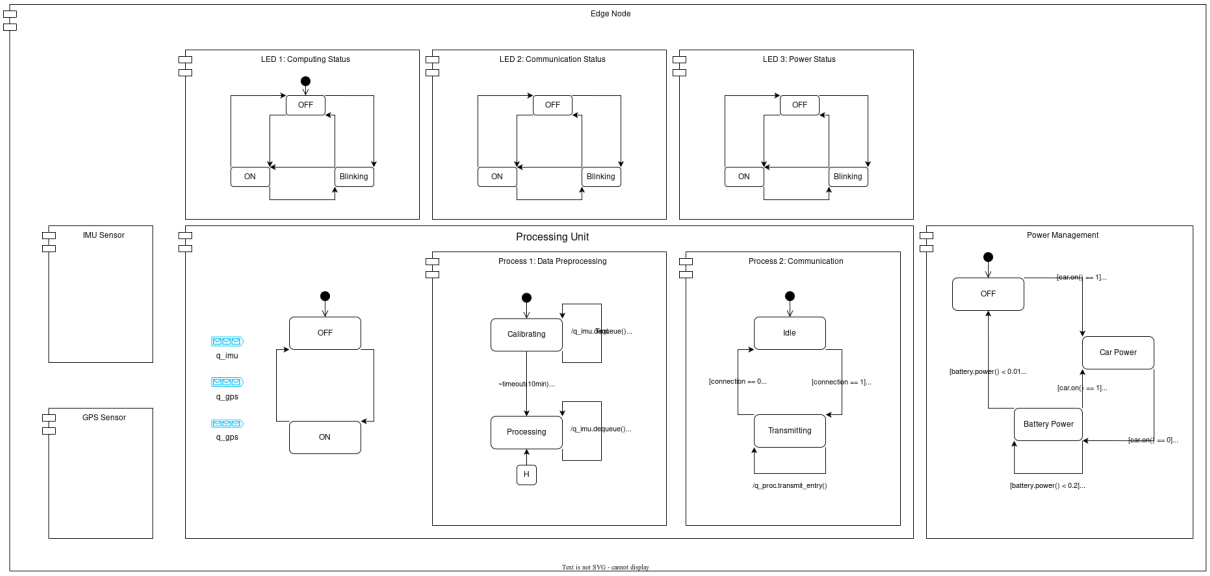


Figure 1.1: RT-UML of a Sensor Node

1. **Cost Restriction per node**: 100 CHF
   As the number of vehicles the nodes will be high, th cost per node needs to be as small possible. To keep cost of installation and parts low, one single node/sensor package will be installed inside the vehicle.

2. **Quantify felt RoadState**: The node is idealy placed centraly in the car, above one of the axles and mounted securely to the chassis to minimize errors. Roadstate will be quantified in a range of 0 (very good) to 244 (very bad) with a value of 255 for hazardous condition.

3. **Adapt Quantification to different cars and driving states**: A simple linear Mass-Spring-Damper Model is chosen to model the cars factor on the transduced shocks. (While keeping computational effort low.) A first calibration phase coupled to a initial parameter set aims to fit Mass-Spring-Damper Model parameters. Measured data will be fit to quantified values during calibration phase. Further physical quatities other than z-axis acceleration have to be considered to decouple driving induced accelerations from the road state.

4. **High Polling rate of IMU measurements**: As shocks induced by road bumps have a very short period, where the period and amplitude of the induced acceleration is proportional to vehicle speed. The polling rate needs to be chosen high enough to ensure reliable sensor readings for road specific driving speeds.

5. **Sensing of physical quantities**:

(a) **Acceleration in z-Axis** to determine road state and potholes. (Adapt pollingrate to vehicle velocity/ must be high enough)

(b) **Acceleration in x,y-Axis and rotational acceleration** to minimize errors induced from driving scenarios.

(c) **Driving Velocity** to couple shock amplitudes to velocity (through Spring-Damper Model).

(d) **Geographical Position** to reference qualification to current position.

6. **Transmit Data at established Gatepoints**

   (a) Transmitted information Format:
   (Node ID (2 Byte)) | Position (2 * 8 Byte (DP FP)) | RoadQuality (1 Byte) | Unix Timestamp (4 Byte))

   (b) Preprocess and save (Position, Quality)-Tuples locally on Node

   (c) Automatically establish connection at gatepoints and transmit new gathered data

   (d) Packages in MQTT format are sent to the RabbitMQ server

1.4 System Architecture

The **RoadSense** consists of multiple IoT devices installed in vehicles, communicating with a central server designed to be highly scalable to handle data from thousands of devices.

IoT Data Pipeline

The data pipeline has been designed with scalability in mind, allowing for efficient data collection, processing, and data analysis from multiple (potentially thousands) concurrent IoT devices. The following diagram illustrates the pipeline steps, from data ingestion to the storage of processed data.

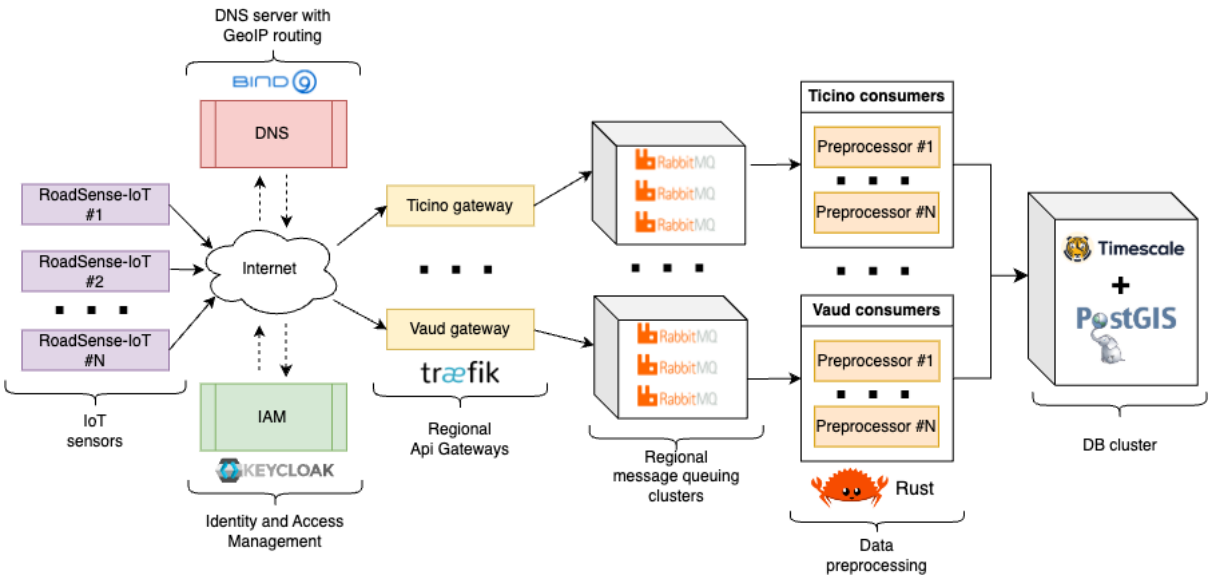


Figure 1.2: IoT Data Pipeline

The pipeline consists of the following components:

1. **Data ingestion**: IoT devices collect vibration, GPS, and other relevant data points. When the vehicle reaches an access point, the data will be transmitted to the server.

2. **Authentication and security**: Each device is authenticated before data transmission to ensure data integrity and prevent unauthorized access. For this purpose, we chose to use Keycloak for identity and access management.

3. **Geographical distribution**: The server leverages DNS-based load balancing to distribute incoming data across regional gateways for efficient processing. We have chosen to use BIND9 for DNS-based load balancing along with GeoIP for geolocation. Each regional gateway will be responsible for routing the user requests to the regional message queuing system. For this purpose, we will use Traefik as the reverse proxy.

4. **Message queues**: Each gateway node processes incoming data and forwards it to a regional queuing system to allow for parallel processing. After evaluating multiple options, we decided to use RabbitMQ as the message queue system. To ensure high availability, we will deploy RabbitMQ in a cluster configuration (refer to the RabbitMQ Clustering Guide).

5. **Data preprocessing**: Each region has a set of preprocessing microservices that consume incoming data from the regional message queuing system, perform data validation, and run initial data processing tasks. These microservices are deployed using containerization technology like Docker and in a future production environment, managed by Kubernetes. Since we want to ensure high-performance data processing, and a small memory footprint, we chose to use Go as the primary language for these microservices.

6. **Data storage**: Processed data is stored in a scalable database system that can handle high volumes of data. Since we are dealing with both date-time and geospatial data, we chose to use TimescaleDB as the database system with the PostGIS extension to support geospatial queries. To ensure data durability and high availability, we will deploy TimescaleDB in a clustered configuration.

Client-Server Architecture

This section describes the client-server architecture of the system, focusing on the interaction between the web application and the server-side components. The following diagram illustrates how the client (web browser) interacts with the server to load and visualize collected data:

The pipeline consists of the following components:

1. **Clients**: The application users (agents or web browsers) interact with the web application to view road conditions, manage alerts, and access other features.

2. **Authentication**: To manage user access and permissions, users need to authenticate themselves. For this purpose, we will use the same instance of Keycloak that

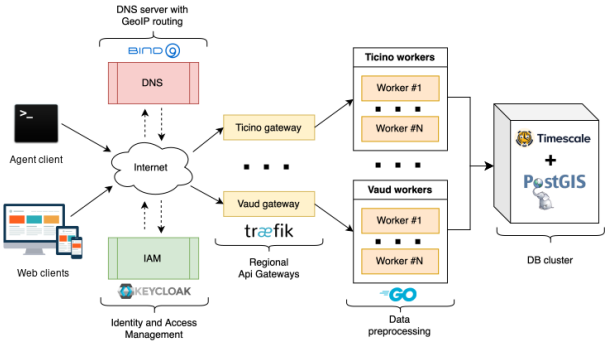


Figure 1.3: Client-Server Architecture

is used for IoT device authentication. To have a greater division between IoT devices and users, we will employ different realms in Keycloak (refer to the Keycloak Documentation).

3. **Geographical distribution**: API requests are routed to the regional API gateways using DNS-based load balancing. We will use the same BIND9 + GeoIP + Traefik setup as described in the IoT data pipeline section.

4. **API Workers**: Each region has a set of API worker microservices that will handle incoming API requests, query the database, and return the requested data to the client. These microservices will be deployed using containerization technology like Docker and managed by Kubernetes in a future production environment. Furthermore, they will be developed using Go to enhance the application performance.

5. **Database cluster**: API workers connect to the database cluster containing the aggregated IoT data, allowing them to retrieve the necessary information for the client requests.

Chapter 2

System Implementation

2.1 Overview

<todo>

2.2 Prototype Car

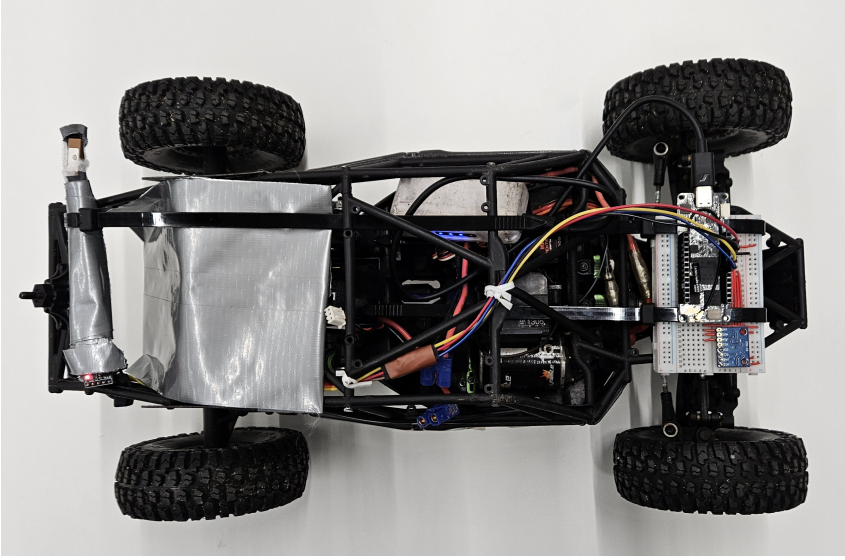


Figure 2.1: RoadSense Prototype RC Top View

Hardware Components

1. **Microcontroller**:
   Arduino Portenta H7 with built-in Wi-Fi capability.

2. **IMU Sensor**:
   GY-521 with MPU6050 6DOF (3-Axis Gyro and 3-Axis Accelerometer).

3. **GPS Module**:
   DFRobot GPS + BDS BeiDou with output of position and speed.

4. **EMF Shielding**:
   DIY using aluminum foil to shield the GPS module from electromagnetic interference.

Wiring

	Sensor	Portenta H7	Description
GY-521	VCC	3.3V	Power supply (3.3V)
	GND	GND	Ground
	SDA	SDA (Pin 11)	I2C Data line (SDA)
	SCL	SCL (Pin 12)	I2C Clock line (SCL)
DFRobot GPS	VCC	3.3V	Power supply (3.3V)
	GND	GND	Ground
	TX	RX (Pin 13)	Serial data transmit line (TX from GPS to RX on Portenta H7)
	RX	TX (Pin 14)	Serial data receive line (RX from GPS to TX on Portenta H7)

Table 2.1: Pin Connections for Sensors with Arduino Portenta H7

2.3 Prototype Embedded Firmware

In this section, we provide an overview of the core components and implementation details of the prototype embedded firmware developed for the sensor node. The firmware orchestrates sensor data acquisition, road quality analysis, and reliable data transmission to an external system. The following subsections summarize the primary files and their responsibilities:

- **Mainfile (roadsense-embedded.ino):** Initializes the system, sets up multithreaded operations, and manages the data flow between sensor acquisition and network transmission.
- **roadqualifier.h:** Contains the logic for measuring, calibrating, and quantifying road segment quality using integrated sensors, along with persistent calibration data handling.
- **RabbitMQClient.h:** Handles WiFi connectivity and MQTT-based communication, enabling the sending of computed road quality metrics to a RabbitMQ server.

By clearly defining these components, the firmware maintains a modular structure, simplifying development, testing, and future enhancements.

2.3.1 Mainfile (roadsense-embedded.ino)

The main Arduino `roadsense-embedded.ino` file serves as the central entry point for the embedded firmware running on the sensor node. Its primary tasks involve initializing system components, orchestrating two concurrent threads for road data acquisition and transmission, and managing communication buffers.

- **Initialization and Setup:** At startup, the main file initializes serial communication for debugging. It then sets up the `RoadQualifier` instance, which prepares sensor input (e.g., IMU and GPS readings) for analyzing road quality. If initialization fails, the system reports this via serial output.
- **Multithreading using Mbed OS:** Leveraging Mbed OS RTOS features, the firmware runs two threads concurrently:
  1. *Road Segmentation Thread*: Periodically calls `roadQualifier.qualifySegment()` to compute the quality of a road segment. Upon success, it stores the resulting `SegmentQuality` record into a thread-safe circular buffer.
  2. *Data Transmission Thread*: Establishes and maintains a WiFi connection, then continuously reads from the circular buffer to transmit data using a `RabbitMQClient`. If no data is available, it waits until new records arrive.
- **Circular Buffer for Data Storage:**
  A custom circular buffer, protected by a mutex, ensures safe concurrent access from both threads. If the buffer is full, the oldest entry is overwritten, preventing blocking conditions and ensuring efficient memory usage.
- **Data Transmission via RabbitMQ:**
  Once connected to WiFi, the data transmission thread publishes buffered `SegmentQuality` records to an external system through the `rabbitMQClient`. This design decouples data acquisition from network-related issues, allowing both to operate independently.
- **Watchdog and Timing:**
  Although not fully explored in the provided snippet, the code includes a watchdog timer and uses `ThisThread::sleep_for()` to handle timing and maintain system responsiveness.
- **Main Loop:**
  The `loop()` function remains empty, as the system relies on RTOS threads for ongoing tasks. All main logic thus resides in separate threads defined in the setup phase.

2.3.2 roadqualifier.h

The `roadqualifier.h` file encapsulates the logic and data structures required to process road quality measurements from connected sensors, manage calibration and data persistence, and ensure system readiness. This file defines the `RoadQualifier` class, which serves as the core of the road quality analysis functionality.

- **Sensor Abstraction and Dummy Modes:** The code supports both actual hardware operation and dummy sensor modes for testing without physical IMU or GPS devices. Conditional compilation flags (e.g., `DUMMY_MPU` and `DUMMY_GPS`) select between real and simulated sensor inputs. This approach allows for development and debugging of other modules without actual available sensors.

- **Road Segment Qualification:** The `RoadQualifier` class provides a `qualifySegment()` method to measure a predefined road segment's quality. It uses acceleration data (from the MPU6050 or dummy equivalent) and position/speed data (from a GPS module or dummy object) to compute a `SegmentQuality` metric. If a valid segment is detected, it returns a quantized quality value mapped into a byte range.
- **Calibration Handling and Flash Memory:** The file includes routines for:
  - *Calibration*: Acquiring accelerometer data over a specified timeframe to determine minimum and maximum values, ensuring that subsequent measurements are interpreted correctly.
  - *Persistent Storage*: Using Mbed's `FlashIAPBlockDevice` and related helpers (`FlashIAPLimits.h`) to store and retrieve calibration parameters (e.g., minimum and maximum acceleration differences) in non-volatile flash memory.
  - *Deletion of Calibration Data*: Providing a function `deleteCalibrationFromFlash()` to erase previously stored calibration information, enabling reset or re-calibration scenarios.
- **Quantification and Mapping:** A dedicated `quantifyToByte()` function maps computed acceleration differences into a 0–255 byte range based on the caputred calibration data. This allows for easy interpretation, efficient storage and transmission of road quality metrics.
- **Initial Setup and Readiness Checks:** The `begin()` method initializes sensors, loads or creates calibration data, and ensures a stable GPS fix before considering the system ready. The `isReady()` method provides a quick way to confirm that the `RoadQualifier` is fully operational.
- **GPS and IMU Integration:** Functions such as `waitForValidLocation()` and `waitForValidSpeed()` ensure that the system obtains reliable, fresh data from the GPS before proceeding. The IMU (or dummy MPU) data is read at each iteration, feeding the computation that identifies peak acceleration differences along the measured road segment.

2.3.3 RabbitMQClient.h

The `RabbitMQClient.h` file manages the communication between the sensor node and an external RabbitMQ server over MQTT. It encapsulates WiFi connectivity handling, MQTT client operations, and the formatting and publishing of road segment data into a consistent interface.

- **WiFi Connectivity Management:** The class attempts to connect to one of several predefined WiFi networks. It continually checks WiFi status and provides a method `isConnectedWiFi()` to confirm a successful connection. By iterating through a list of credentials, the code increases the likelihood of establishing a network connection in various deployment environments.
- **MQTT Integration for RabbitMQ:** The `RabbitMQClient` uses the `PubSubClient` library to communicate over the MQTT protocol. It sets up the MQTT server (RabbitMQ host, port, user, and password) and ensures a persistent connection. The

`connect()` method and the internal `ensureConnected()` helper function handle reconnection logic and error reporting.

- **Error Handling:** In case of connection failures or publishing errors, the class stores the MQTT state code, accessible via `getErrorCode()`. This mechanism aids in debugging and understanding the cause of communication issues.
- **Publishing Data and Callbacks:** To send road segment quality data, the class provides:
  - *`publishSegmentQuality()`*: Converts a `SegmentQuality` struct into a JSON-formatted message and publishes it to a designated MQTT topic.
  - *`sendDataCallback()`*: A method suitable for periodic or callback-driven operations, connecting to the RabbitMQ server (if not connected) and publishing freshly acquired segment data.
- **Integration with the Firmware:** By abstracting away the details of WiFi and MQTT connections, `RabbitMQClient` allows other parts of the firmware—such as the road qualifier threads—to focus solely on data acquisition and retrieval. The communication logic remains modular, enabling future changes to the network stack or message format without altering the core road quality logic.

2.4 Prototype Network

<todo>

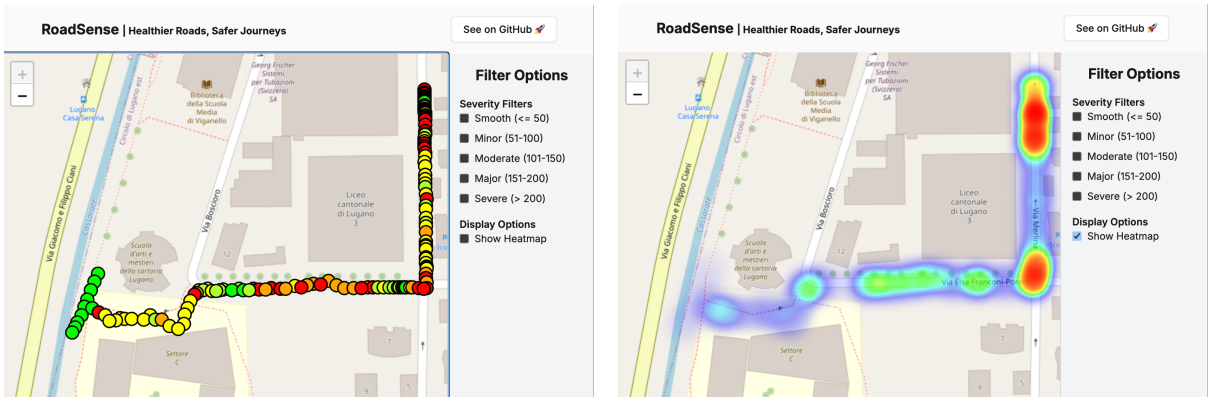
2.5 Prototype Frontend

<todo>

Chapter 3

Results

3.1 Demonstrations



(a) Point Quality Data (b) Data Distribution Heatmap

Figure 3.1: Exemplary Datavisualization with Roadsense Web Application