RoadSense - Requirements and Specifications

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.

The team consists of three members: Luca Di Bello, Georg Meyer, and Paolo Deidda.

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.

Requirements

From the objectives outlined in the previous section, the team had identified the following requirements for the **RoadSense** project:

1. **Road heatmap**: Create an interactive web application that displays road conditions using a heatmap overlay (lighter colors for smoother roads, darker colors for rougher roads).

2. **Alerting system**: The system should automatically create alerts when hazardous road conditions are detected and allow users to view, acknowledge, and resolve these alerts. Users should be also able to manually add alerts on the map.

3. **Anomaly Differentiation**: Distinguish between normal road features (i.e. speed bumps, potholes, manholes, etc) and hazardous road conditions.

4. **Road network coverage**: To achieve an extensive coverage of the road network and minimize discrepancies between single measurements an actual road state a swarm of sensor nodes installed in different cars is needed. Untilization of vehicles owned by public services (i.e. buses, dump trucks, etc.) can be utilized as well as vehicles of private volunteers. Further offers to businesses with large car fleets (i.e. Taxi companies, Ride Share companies, etc.) can be made to further increase measuring devices.

5. **Continuous system calibration**: Each IoT device should be calibrated to the vehicle it is installed in, taking into account the vehicle's characteristics and driving conditions. This calibration should be simple and possibly automatically. This can be archieved by a first calibration phase coupled to an initial parameter set.
6. **Scalability**: The system should be highly scalable to potentially handle data from thousands of vehicles simultaneously.
7. **Centralized Data Management**: The server should be able to aggregate data from multiple IoT devices and store it efficiently for later analysis and visualization.
8. **Optimized Data Transmission**: Since continuous data transmission can be costly for the user, the system should transmit data only when the vehicle is at established access points (e.g., Wi-Fi hotspots).
9. **Power Supply**: The systems should be designed to operate using the vehicle's power source primarily, with a small backup battery to ensure that the data transmission is not interrupted if the vehicle is turned off.
10. **Durability**: The sensor nodes should be robust and weatherproof to withstand various driving conditions to minimize effort and cost of maintenance for the end user.

System design

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.

Sensor Nodes

Specifications

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 14 (very bad) with a value of 15 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**:
    1. **Acceleration in z-Axis** to determine road state and potholes. (Adapt pollingrate to vehicle velocity/ must be high enough)
    2. **Acceleration in x,y-Axis and rotational acceleration** to minimize errors induced from driving scenarios.
    3. **Driving Velocity** to couple shock amplitudes to velocity (through Spring-Damper Model).
    4. **Geographical Position** to reference qualification to current position.
    5. **Driving Direction** to deduce road lane (use gps data).
6. **Transmit Data at established Gatepoints**
    1. Transmitted information Format:
        - (Node ID (2 Byte)) | Position (2 * 4 Byte (SP FP)) | RoadState (0.5 Byte)
    2. Preprocess and save (Position, Quality)-Tuples locally on Node
    3. Automatically establish connection at gatepoints and transmit new gathered data

RT-UML with State Charts

Hardware Components

1. **Microcontroller**:
    - **Arduino Nano 33 IoT** or similar with built-in Wi-Fi capability.
2. **IMU Sensor**:
    - **MPU-6050** or **MPU-9250** accelerometer and gyroscope module.
3. **GPS Module**:

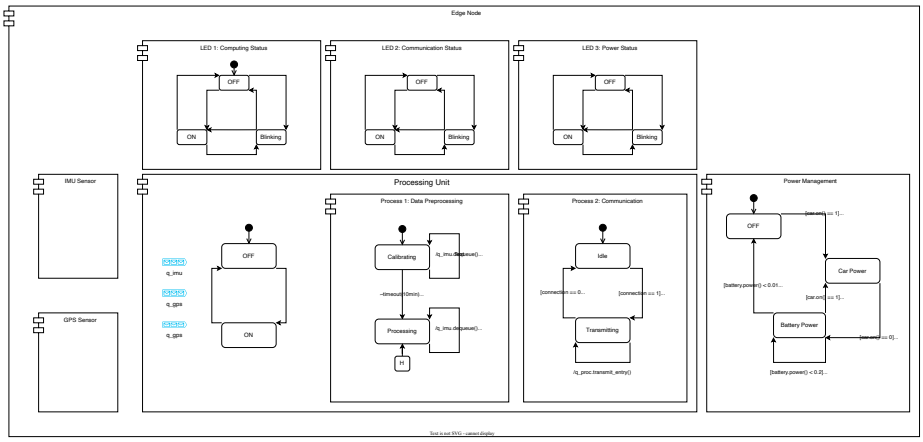

Figure 1: RT-UML of a Sensor Node

- **Ublox NEO-6M** GPS module for accurate positioning.

4. **Power Supply**:
   - **Primary**: Connected to the vehicle's power supply.
   - **Backup**: Rechargeable Li-Po battery with voltage regulation.
5. **Enclosure**:
   - Durable casing with LEDs for status indication (e.g., transmission activity, errors).
6. **Connectivity**:
   - Wi-Fi module (if not integrated) like **ESP8266** or **ESP32**.

Firmware Development

- **Sensor Calibration**:
  - Implement routines to calibrate measurements/models from the IMU sensor for accurate readings.
- **Data Sampling**:
  - Sample sensor data at appropriate intervals fitting to vehicle speed (e.g., 50 Hz).
- **Noise Reduction**:
  - Apply Kalman Filter or Complementary Filter to fuse sensor data.
  - Use moving averages or median filters to smooth out transient spikes.
- **Baseline Adjustment**:
  - Establish a baseline for the vehicle's normal vibrations.
  - Adjust subsequent readings by subtracting the baseline values.
- **Wi-Fi Connectivity**:
  - Configure to connect to known Wi-Fi networks.
  - Implement setup mode for inputting Wi-Fi credentials.
- **Data Packaging**:
  - Format data (timestamp, GPS coordinates, vibration metrics) for

transmission.

- **Data Transmission**:
  - Use HTTP/HTTPS protocols to send data to the server.
  - Implement error handling and retries for network issues.
- **Power Management**:
  - Monitor power source and switch between vehicle power and backup battery as needed.
  - Implement sleep modes when the vehicle is not in motion.

Data Processing and Noise Reduction

- **Calibration Period**:
  - Collect initial data to establish the vehicle's baseline vibration patterns.
  - Dynamically adjust the baseline to account for changes (e.g., vehicle load).
- **Filtering Techniques**:
  - **Low-Pass Filter**: Remove high-frequency noise unrelated to road conditions.
  - **High-Pass Filter**: Eliminate low-frequency movements like vehicle tilts.
  - **Band-Pass Filter**: Focus on frequencies corresponding to road-induced vibrations.
- **Statistical Methods**:
  - **Standard Deviation and Variance**: Measure dispersion of vibration data.
  - **Peak Detection**: Identify significant deviations indicating bumps or potholes.
  - **Thresholding**: Categorize road conditions based on vibration intensity thresholds.
- **Edge Computing**:
  - **Data Compression**: Reduce data size by transmitting only significant events.
  - **Event Detection**: Implement on-device logic to detect and report anomalies.
  - **Power Efficiency**: Optimize code to reduce processor load and conserve battery life.

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.

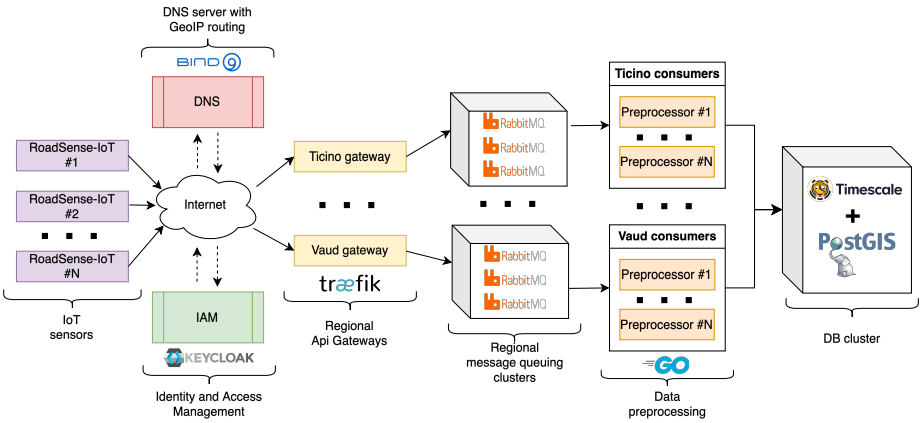

Figure 2: IoT Data Pipeline

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:

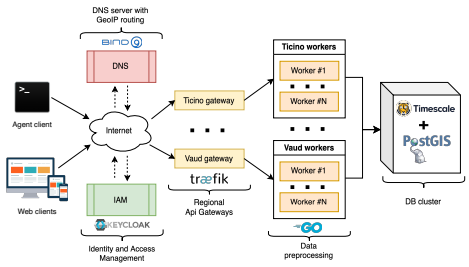

Figure 3: Client-Server Architecture

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 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.

Possible Problems and Solutions

Network Connectivity

- **Wi-Fi Availability**:
  - **Challenge**: Continuous Wi-Fi connectivity may not be available during travel.
  - **Solution**: Store data locally and transmit when connectivity is available.
  - **Alternative**: Use mobile hotspots or integrate GSM modules for cellular data.
- **Data Security**:
  - Encrypt data transmissions to prevent interception or tampering.
- **Error Handling**:
  - Implement robust error handling for network disruptions.

Vehicle Variability

- **Machine Learning Models**:
  - Train models to adjust for different vehicle characteristics.
- **User Input**:
  - Allow users to specify vehicle type during device setup for better calibration.
- **Adaptive Algorithms**:
  - Continuously learn and adapt to the vehicle's behavior over time.

Power Management

- **Sleep Modes**:
  - Implement low-power modes when the vehicle is stationary.
- **Efficient Components**:
  - Use low-power sensors and microcontrollers.
- **Power Monitoring**:
  - Provide alerts when battery levels are low.

Data Volume Management

- **Data Sampling**:
  - Optimize sampling rates to balance data quality and volume.
- **Selective Reporting**:
  - Transmit only aggregated or significant data points.

- **Data Compression**:
  - Compress data before transmission to reduce bandwidth usage.