University of Pisa

Artificial Intelligence and Data Engineering
Internet of Things

F1 Tyres Temperature Controller

Gabriele Marino 564643
Matteo Pasqualetti 564367

July 20, 2023

Contents

1 Introduction **2**

2 Architecture **3**
2.1 Sensors . . . 3
2.2 Actuators . . . 3
2.3 Controller . . . 3
2.4 Data Encoding . . . 4

3 Analytics **5**
3.1 Database . . . 5
3.2 Grafana . . . 5

4 Implementation's Details **7**
4.1 Controller . . . 7
4.1.1 CLI . . . 7
4.2 IoT Nodes . . . 8
4.2.1 The `TyreSensorMQTT` . . . 8
4.2.2 The `Actuator` Class . . . 8
4.2.3 The `Temperature` Class . . . 8
4.2.4 The `TyreActuatorCoAP` Class . . . 8
4.2.5 The `CoAPRegister` Class . . . 8

Chapter 1

Introduction

Formula 1 racing is a pinnacle of motorsport known for its high speeds, technical expertise, and relentless pursuit of perfection. The performance of a Formula 1 car depends on numerous factors, and tyre temperature is one of the critical parameters that can make a significant difference on the track. Proper tyre temperature management ensures optimal grip, stability, and tyre wear, allowing drivers to push the limits of their cars while maintaining control. Our project focuses on developing an IoT-based solution for monitoring the temperatures of Formula 1 tyres. The system consists of sensor-equipped devices installed on each tyre, capable of collecting temperature data in real-time. These sensors communicate with a central monitoring unit wireless, transmitting the temperature readings for analysis and visualization.

The deployed **sensors** in Formula 1 cars and tyre-warmers play a crucial role in enabling comprehensive monitoring of tyre temperature. These devices are instrumental in capturing real-time data whether the car is on the track or in the box, facilitating automated monitoring and management of temperature. By continuously monitoring tyre temperature, engineers and drivers gain valuable insights into the temperature conditions, empowering them to proactively respond to dynamic changes and ensure optimal tyre temperature conditions.

In addition to providing comprehensive monitoring, our IoT solution incorporates two type of **actuators** specifically designed for real-time interaction with the devices. These actuators include the tire warmers themselves and the steering wheel lights. Leveraging a centralized server, the collected sensor data is analyzed, interpreted, and used to deliver precise instructions to the relevant actuators. This closed-loop control system enables timely and responsive interventions to maintain optimal tire temperatures, especially to ensure compliance with Formula One regulations, where tire temperatures should not exceed 70 degrees Celsius while inside the warmers. Through our system, teams can confidently stay within the allowed limits defined by the sporting regulations, and drivers can effectively manage tire temperature and minimize tire wear on the track.

Chapter 2

Architecture

We have the following sensors:

- **temperature** sensors, used to monitor the temperature while using the tyre warmers
- **temperature** sensors, used to monitor the temperature while the car is on track

and the following actuators:

- **tyre-warmer**, to automatically activate/deactivate the tyre-warmers
- **Steering wheel leds**, to inform the driver about the temperature while the car is on track

Control loop The sensors and actuators work in a closed feedback control loop, the central Controller sets the control system's objective.

2.1 Sensors

The sensors utilize the *MQTT* protocol to transmit data to the controller through the MQTT Broker. The Tyrewarmer sensor sends messages under the topic *TyrewarmerTemp*, whereas the sensors installed on the car employ the topic *TyreTemp* for communication.

2.2 Actuators

The actuators utilize the **CoAP** protocol to receive commands from the Controller and to subscribe to the remote application. The actuator associated with the tyrewarmer can respond to the messages "ENABLE" or "DISABLE" to activate or deactivate the tyrewarmer, respectively, without the need for physical button interaction. Additionally, it can react to the messages "HIGHTEMP" and "LOWTEMP" to dynamically adjust the tyrewarmer based on the measured temperature. On the other hand, the actuator integrated into the car's steering wheel responds to the messages "OVER", "UNDER", and "GREAT" to notify the driver about the current temperature of the tyres, providing real-time feedback.

2.3 Controller

The Controller serves as the central component of the IoT application, responsible for data storage in the database and executing control logic. It effectively manages the diverse sensor-actuator pairs by applying specific policies tailored to each pairing.

The Controller is implemented in *Java*, utilizing Eclipse's libraries for seamless integration with MQTT and CoAP networks.

2.4 Data Encoding

In our sensor network, where resources are limited, we have chosen JSON as the preferred data encoding format. All sensor-generated data is transmitted to the controller in the form of JSON objects. This decision was driven by several factors, with a primary focus on the constrained nature of the sensor devices. 1One key advantage of JSON is its flexibility and simplicity, especially when compared to alternative formats such as XML. XML tends to have a more complex and redundant structure, which makes it less suitable for our specific requirements. By utilizing JSON, we are able to streamline the data representation process, reducing unnecessary complexity and overhead.

Here an example of what the sensor send to the MQTT broker:

```
{
  temperature: 90,
  tyre: 1
}
```

Chapter 3

Analytics

3.1 Database

We're using *mySQL* as database management system. We're storing historic sensors' data in tables, one for each sensor's class, in the form of tuples: unique id, timestamp, and sensor's datum, such as tyre position and temperature.

Table 3.1: MySQL Table: temperature_on_track

id_temperatureontrack	**temperature**	**timestamp**	**tyre_position**
1	90	2023-07-10 15:23:03	1
2	95	2023-07-10 15:23:15	2
3	100	2023-07-10 15:23:23	3
4	100	2023-07-10 15:23:50	4

We have also introduced a table to save the actuators that subscribe to the remote application successfully, through the subscription mechanism illustrated in chapter 4.2.5.

Table 3.2: MySQL Table: actuators

id_actuators	**type**	**timestamp**	**tyre_position**	**ipv6_addr**
1	res_wheel_led	2023-07-08 09:23:03	1	coap://[fd00:0:0:0:205:5:5:5]
2	tyrewarmer	2023-07-08 09:23:15	2	coap://[fd00:0:0:0:204:4:4:4]
3	tyrewarmer	2023-07-08 09:23:23	3	coap://[fd00:0:0:0:203:3:3:3]
4	res_wheel_led	2023-07-08 09:23:50	4	coap://[fd00:0:0:0:206:6:6:6]

3.2 Grafana

Our project incorporates a real-time monitoring and visualization through the implementation of Grafana dashboard. This powerful tool allows us to access and monitor the data stored in our database, enabling us to gain valuable insights into the trends and patterns of the monitored parameters.

By leveraging Grafana, we have created a user-friendly interface that presents the data in a visually appealing and intuitive manner. The dashboard provides real-time updates, allowing engineers to stay informed about the current status of the monitored parameters. The ability to

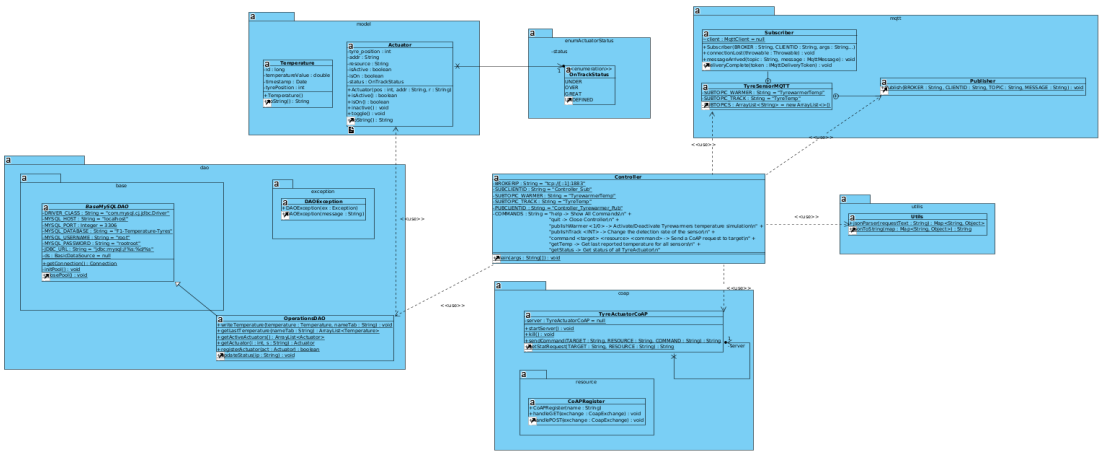
view data trends in real-time empowers decision-making, to respond promptly to any deviations and deciding a strategy in real time.

Grafana also allows the visualization of historical data, providing a comprehensive view of the data over time. This capability is useful for performing in-depth analysis and identifying patterns within a circuit or a specific lap. By visualizing historical data, engineers can gain insights into laps or circuits variations, identify potential behaviours for temperature behavior, and make informed decisions about strategy and driving protocols.

Chapter 4

Implementation's Details

4.1 Controller



4.1.1 CLI

The following are the functions that are accessible to the user through the Command Line Interface:

- **help**: shows all possible commands.
- **quit** terminates the Controller execution.
- **publishWarmer <1|0>**: enables or reset the temperature simulation (USED FOR DEBUG ONLY).
- **publishTrack <seconds>**: used to update the sampling frequence of the car's sensors.
- **command <target> <resource> <command>**: used to send a command to a specific Actuator (USED FOR DEBUG ONLY).
- **getTemp**: returns the last registered temperature within 5 minutes for each tyre, for each sensor type.
- **getStatus**: returns the status of all registered actuators.

4.2 IoT Nodes

The code represents a modular and adaptable system for managing different types of sensors in an Internet of Things (IoT) environment. Let's break down the code and explain its functionality:

4.2.1 The `TyreSensorMQTT`

The `TyreSensorMQTT` defines the behaviour for a topic manager, which is responsible for parsing MQTT messages and executing callbacks. It includes the following methods:

- `messageArrived`: Takes an `MqttMessage` and converts it into a proper object. The specific implementation of the object depends on the topic received, and so, on the type of sensor which sent the message.
- `connectionLost`: Executes a connection recovery, whenever the Controller lose the connection from the MQTT Broker.

4.2.2 The `Actuator` Class

The `Actuator` class is used to virtually represent an Actuator within the system. It includes all the fields representing an Actuator:

- $\texttt{tyre}_{p}osition$: $Indicates on which tyre the actuator is associated to.$ `addr` : $The IPv6 address of the Actuator.$
- `resource`: The type of the Actuator.

4.2.3 The `Temperature` Class

The `Temperature` class is used to virtually represent a registered Temperature from a sensors. It includes all the fields representing a Temperature:

- `temperatureValue`: Indicates the temperature registered from a sensor.
- `timestamp`: When the temperature has been registered.
- `tyrePosition`: The type from which the temperature comes from.

4.2.4 The `TyreActuatorCoAP` Class

The `TyreActuatorCoAP` class is used to send CoAP commands to the Actuators. It includes the following methods:

- `sendCommand`: Send a POST message to give a command to a specific Actuator.
- `getStatRequest`: Send a GET message to retrieve the status of the Actuator.

4.2.5 The `CoAPRegister` Class

The `coAPRegister` class is used to create a CoAP server to exposes a resource that Actuators can use to register within the system:

- `handleGET`: Used by already registered Actuators to inform the Controller they are still alive.
- `handlePOST`: Used to send a brand-new registration.