UNIVERSITÀ DI PISA

Department of Information Engineering (DII)
M.Sc in Artificial Intelligence and Data Engineering

Internet of Things

Vehicle Lights Controller

Giuseppe Aniello, Leonardo Bellizzi

Project code available at:
`https://github.com/LeoBel0799/IoT_Project`

Academic Year 2022/2023

Contents

1 Application

The primary objective of this Internet of Things project is to establish a vehicle car controller system for managing its lighting, including regular headlights and high beams. This application has practical utility in real-world scenarios, as it is designed to operate with real-time data obtained from the vehicle's lights, which serve as actuators.

To achieve the aforementioned goal, it was essential to foster collaboration between the project's two main components: the Remote Control Application and the Cloud Application. The seamless integration of these components is of paramount importance, and a pivotal factor in achieving this integration was meticulous planning of the database structure, which will be discussed in detail later.

The **Cloud Application** has been developed to collect data from various sensors leveraging MQTT, while the **Remote Control Application** is responsible for registering lights within the system leveraging COAP. Based on the data collected from sensors, the Remote Control Application is tasked with the intelligent control of vehicle lights, enabling it to turn them on or off and activate high beams as necessary.

Furthermore, the application offers the functionality to replace malfunctioning lights. Each light's wear level is continuously monitored to determine its operational condition. The wear level is calculated based on the number of times the light has been turned on and off. When a light reaches the point where replacement is necessary, the system initiates the replacement process. Upon replacing a burnt-out light with a new one, the actuator responsible for that light is reset, and all stale data related to the burnt-out light is cleared from the Database. In addition to its core functionalities of real-time lighting control, project demonstrates a commitment to safety and efficiency by incorporating a proactive feature for light replacement. The system diligently monitors the wear and tear of each vehicle light through the continuous tracking of activation cycles. When a light is deemed no longer fit for service, the application seamlessly orchestrates the replacement process, ensuring the vehicle maintains optimal lighting performance.

2 Technologies used

From project guidelines the actuator nodes use the COAP protocol while sensor nodes must use the MQTT protocol. This rules was followed in the project so that sensors communicate with MQTT and actuators with COAP. Following bulleted lists summarize technologies used in the project.

- Ubuntu 18.04 LTS
- Contiki-NG v4.7 on Docker
- C-language
- Java-language
- Docker
- MySQL database
- Californium (Coap Java library)
- Paho (MQTT Java library)
- MySQL Java library
- Grafana to display graphs based on some data

During the testing phase, Cooja was utilized. Following the validation of the program's proper functionality within the simulator, the deployment phase was undertaken on the sensor devices, specifically on the nRF52840 dongle boards.

3 Application design

As said previously three important aspects are:

- Collecting data from sensors
- Logic application on actuators based on data collected from sensors
- Light registration
- Database bridge

3.1 User controller

User can make actions on lights and high beams. Mainly he can turn on or turn off lights and high beams. A menu was built up, and there are up to seven choices available to users.

- 1) Exit Controller
- 2) Bootstrap Lights
- 3) Handle Lights
- 4) Handle Brights
- 5) View Node records
- 6) View Actuator records
- 7) View Light Status records

The application will run continuously until the user stops it by pressing '1', which serves as the Exit Controller.To ensure safety and proper functionality, certain constraints have been implemented. Users cannot control lights and high beams until they have activated them through the second option.

Additionally, users can always check the status of lights and high beams using the last three options, which provide a database view of the data. The sixth option has been designed to help users determine the status of lights during maintenance mode, aiding in identifying when a light needs replacement.

The menu incorporates further constraints to prevent user errors, such as inputting invalid characters or numbers that are not relevant to the application.

3.2 Database design

The design of the database played a pivotal role in the development of this application. It was imperative to transform the application into a real-time system without effectively segregating the data collected by sensors from the actions initiated by users on the actuators.

The database comprises three distinct tables. The first table is responsible for collecting data from the sensors, the second table is utilized for registering the actuators, and the last table was specifically created to record all the actions carried out on the lighting and high beam controls.

Given that the Actuator table accommodates records for both the actuators, including the first light actuator, and relies on the Motion table, which records the sensor data, a "bootstrapped" flag was introduced within the Motion table. This flag serves as an indicator of whether the lights have been initialized. If the lights are marked as "bootstrapped," it signifies that the lighting system has been activated at least once, and consequently, the Actuator table is populated with at least one record. Following the bootstrap event, the Actuator table will continue to be updated with user-initiated actions on the actuators.

3.2.1 Motion table

The motion table contains information collected by the sensor. It includes an *idlight* field indicating a specific light, the *lights* field that could be ON/OFF, a *counter* that tracks the number of times the light has been turned on and off, a *bright* field that could be ON/OFF a *light degree* field indicating three different types of lights: position light, high beam, and low beam. Additionally, there is filed know as *bootstrapped* that indicates if the light was booootstrapped or not, and finally, a *timestamp* indicating the data acquisition time.

Here is an example of a record acquired by the sensor and saved in the MySQL database.

Id	Idlight	Counter	lights	lightsDegree	bright	bootstrapped	created-at
1	1	1	OFF	0	OFF	1	23/09/06 18:41:32

3.2.2 Actuator table

The actuator table contains information following the actuator's actions. It includes the *Actuator's ID*, the *status* of the light and its *bright* after each action, a *counter* indicating the number of times the light has been turned on, a *wear level* indicating the degree of wear of the light calculated as $wearLevel * 0.4$, a flag know as *Fulminated* that alerts when the light is burnt out, and finally, a *timestamp* of the action.

Here is an example of a record saved in the MySQL database after the execution of an actuator action.

Id	IdActuator	Counter	lights	bright	wearLevel	Fulminated	created-at
1	1	1	ON	OFF	0.4	0	23/09/06 18:41:32

3.2.3 Node table

This is the table created after the registration of the two actuators, and it contains their IDs and IPv6 addresses.

Id	Idlight	ipv6
1	1	fd00:0:0:0:206:6:6:6
2	2	fd00:0:0:0:207:7:7:7

3.3 Sensors

Sensors rely on the Motion table, as all the data they sense are continually logged into this table. This data serves as a crucial reference point for the Actuators node. When a user activates one or both lights, the command is executed based on the real-time status of the light, which is determined by the data captured by the sensors at that precise moment. Therefore, the Motion table forms the foundational layer upon which users can control the vehicle lights, turning them on or off as needed.

3.4 Actuators

The Actuators undergo an initial registration process within the application, during which their addresses are recorded and saved in the node table. Subsequently, following the bootstrap process for the lights, an initial record is inserted into the Actuator table. Within this table, data pertaining to both the regular lights and high beams are stored. Two critical types of data are of significance here: the wear level of the light and the counter. The wear level is calculated based on the counter data, serving as a fundamental metric to assess the condition and usage of the lights.

3.5 Use Case Diagram

This paragraph illustrates the use cases of the main components of this project: the sensor, actuator, cloud application, and remote control application, explaining their functions and actions.

3.5.1 Sensor

In Fig.1,the use case of the sensor. The sensor waits for a connection to the broker and then begins to collect data every 15 seconds. Each data point is published on the topic 'light'.

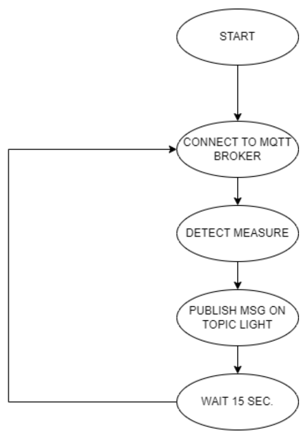


Figure 1: Sensor's use case

3.5.2 Actuator

In Fig.2 the use case of the actuator is illustrated. In the initial phase, there is a registration of the actuator on the server. Once registered and ready to work, it can receive a command from the application, process it, and then await the next command.

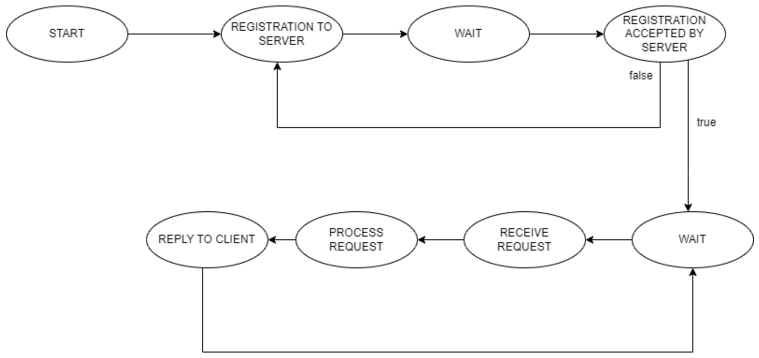


Figure 2: Actuator's use case

3.5.3 Cloud Application

In Fig.3 displays the state machine of the Cloud Application. Its function is to collect data from the sensor and store it in the database.

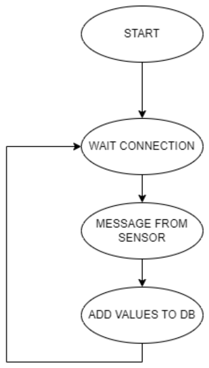


Figure 3: Cloud Application State Machine

3.5.4 Remote Control Application

In Fig.4, the state machine of the remote control application is depicted. There is an initial phase where the actuator needs to register and be saved in the database, followed by another phase where it is possible to insert commands. There are two types of commands: the first type enables actions on the actuator (e.g., handling light and high beam) and subsequently saves the results in the database. The second type allows users to visualize the history of light status by leveraging the database..

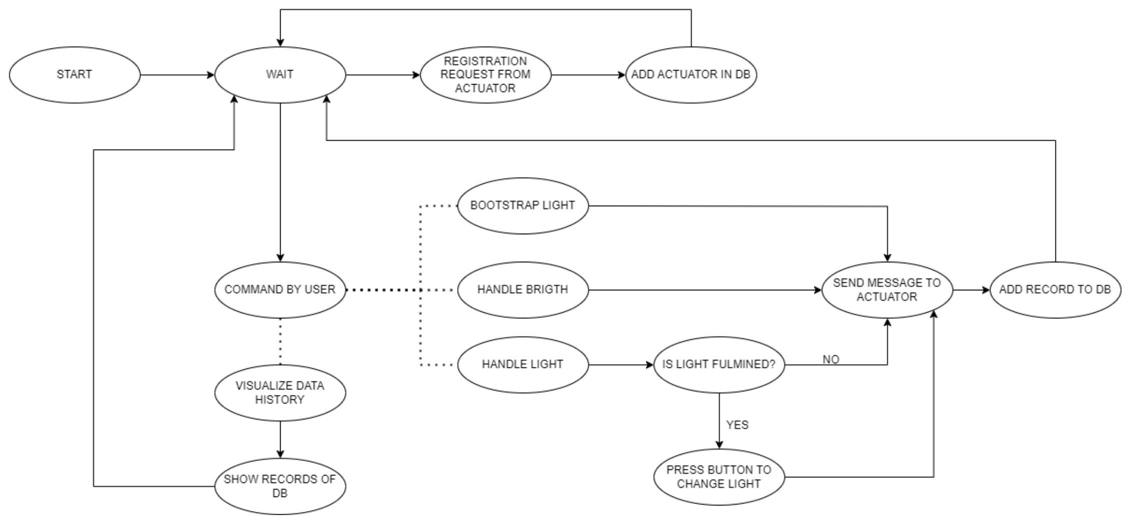


Figure 4: Remote Control Application State Machine

4 Implementation

From a more practical point of view, all business logic is written in Java. For MySQL implementation no script is used but creation and drop of tables are done with Java code. When the application run all tables previously populated are dropped and created again without no data.

For sensors and actuators logic, these parts are written in nesC. There is MQTT part that generates data that are passed to Java part responsible to insert these data into the DB. And there are actuators that use COAP protocol to make some transaction on the lights. Certainly, in the Java component, multiple methods have been established to send requests for turning on and turning off the actuators. These methods facilitate the communication with the actuators and enable the application to control the lighting system effectively.

4.1 Sensor node

MQTT nodes operate without the need for specific addresses; instead, they rely on a broker functioning as a centralized server entity to which MQTT nodes register. The chosen broker is *Mosquitto*, and later in this document, we will provide instructions on its setup.

Each sensor node registers itself to a topic known as *light*. The broker manages messages, which consist solely of a topic and its associated body. Message publication has been configured to occur every 15 seconds, and these messages are formatted in a JSON-like structure.

Whenever a new data record is generated, a red LED indicator on the sensors is activated for the duration of the data transmission process.

4.2 Actuator node

Actuators are required to undergo a registration procedure in order to enroll themselves with the *Remote control application*. Once this procedure is complete, the Border Router is declared as reachable, and the actuators await a registration message. Upon successful registration, the application gains access to two actuators ready to receive user commands for managing lights and high beams. Additionally, the actuators and their respective IPv6 addresses are stored in the Node table within the database. The registration message follows a JSON-like format. However, commands such as ”ON” and ”OFF” for controlling lights and beams are conveyed in plain text. This decision was made for several project-related reasons:

- **Reduced Data Overhead**: Plain text typically incurs less data overhead compared to JSON. When dealing with small or straightforward data transmissions, using plain text can reduce the amount of data exchanged. This advantage is especially pertinent in low-power networks or bandwidth-constrained environments.
- **Simplicity**: Employing plain text simplifies the protocol implemen-

tation and interpretation, especially for resource-constrained devices like IoT sensors. It eliminates the need for a complex JSON parser to process received data.

- **Compatibility with Existing Protocols**:In certain scenarios, communication with devices or systems that use data formats differing from JSON may be necessary. Using plain text facilitates integration in such cases.

Since command to send to actuators and getter are for relatively simple command, JSON is not recommended for the overhead. A COAP node has two resources:

- `bright_handler.c` : This resource manages high beams by either turning on or turning off LEDs on sensors (Turn ON: Blue led - Turn OFF: Led OFF)
- `light_handler.c` : This resource controls lights, enabling the activation or deactivation of LEDs on sensors.(Turn ON: Green Led - Turn OFF: Led OFF)

Apart these two resources,the primary COAP file, `light_node.c` is responsible for actuator registration. This file houses two threads.

The first one, named `light_server`, that start when the application is start up and and it is needed for actuators registration and to activate COAP resource of the the two aforementioned resources alongside the second thread that is in this file and will be explained now.

The second thread, known as `wear_controller` is in charge of handling the case in which a burn out light must be replaced. This process cannot be started at the application's outset through `AUTOSTART_PROCESSES()` but rather only upon receiving data about the burnt light from the server. To activate this thread `process_start()` function must be used. Data from the Java component is transmitted to the COAP node via a POST request to actuators, and this data is tokenized in the `res_post_handler()` function.

Upon receiving data, the thread activates a red light, and after the user presses the button on the actuator, the `res_event_trigger()` function is invoked. This function resets the wear level, counter, and sets the ”fulminated” flag to false, emulating the installation of a new car light. The wear level pertains only to lights; high beams do not have wear levels of their own but instead inherit the wear level of their corresponding lights.

These newly acquired data are transmitted to the server using the `res_get_handler_coap_values()` All transactions are recorded in the Actuators table within the database. Wear levels are calculated in real-time. Furthermore, the next action (ON-OFF) for lights and high beams is determined based on the last recorded state; if the last state was ”OFF,” the new one will be ”ON,” and vice versa. This same principle is applied to high beams. Finally, ON-OFF requests are sent to COAP using a PUT action on actuators, while Wear Levels are transmitted via POST, as there is no need to replace this resource with a new one.

4.3 Application logs

Application is provided alongside info messages useful for user to understand if the application is working properly. Moreover application was tested in all extremes cases in order to reach all those border line situations that could bring application in a failure state.

- **Success message**: [OK]
- **Information message**: [INFO]
- **Error message**: [FAIL]

5 Tests

During the testing phase, a network consisting of five nodes is utilized. Among these nodes, one functions as the border router, three serve as MQTT nodes, and the remaining two operate as CoAP nodes. Two distinct tests are conducted: the first takes place on a simulation platform known as Cooja, which is integrated into Contiki-NG, while the second test involves the use of actual hardware boards, specifically the nRF52840 dongle board.

5.1 Test on Cooja

Below is the list of steps to starts simulation:

- Open the terminal and navigate to the folder where the pom.xml file of the Java application is located. In this project:

  ```
  cd contiki-ng/examples/IoT_Project/project/
  ```

  Then launch the Java application:

  ```
  mvn clean install
  java -jar target/IoT_Project-1.0-SNAPSHOT.jar
  ```

- Run MQTT broker:

  ```
  sudo mosquitto -c /etc/mosquitto/mosquitto.conf
  ```

- Start Contiki container and launch the simulation on Cooja:

  ```
  contikier
  cd tools/cooja
  ant run
  ```

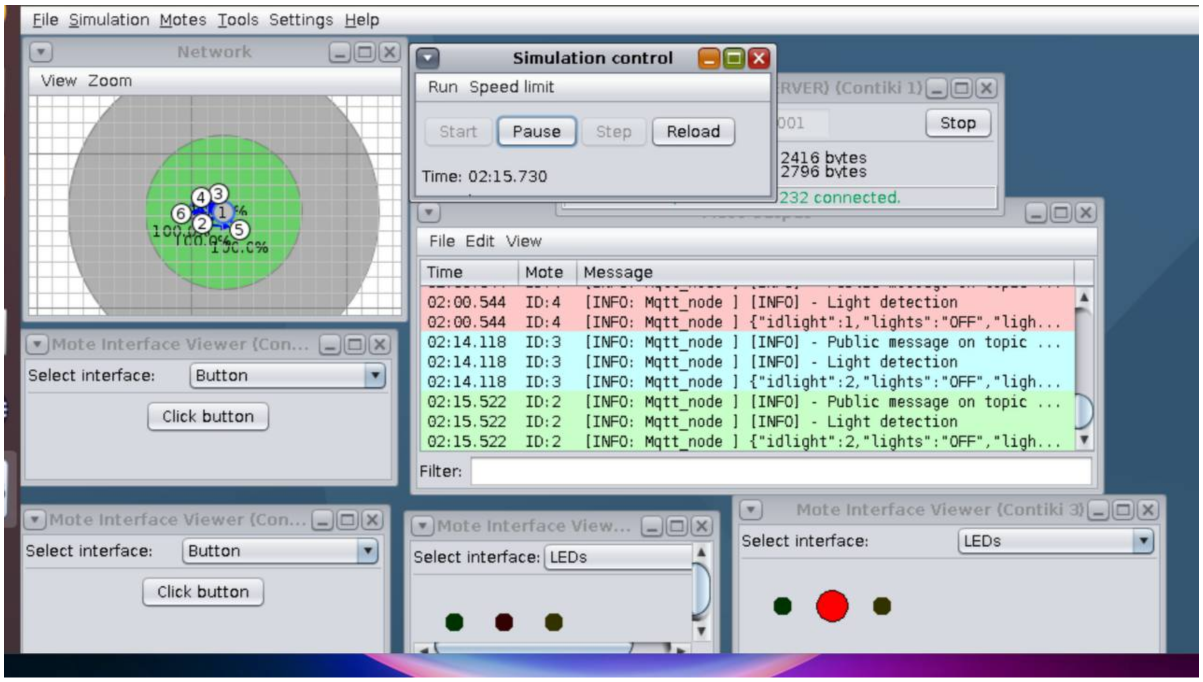
- In Cooja, deploy the following components: one border-router (border-router.c), three sensors (mqtt-node.c), and two actuators (light-node.c).

- Before start the simulation go to the project folder and run the border router:

```
contiki-ng/examples/IoT_Project/sensor-actuators/rpl_border_router
make TARGET=cooja connect-router-cooja
```

- Now is possible start the simulation on Cooja.

An example is shown in Fig.5.



```
Time       Mote  Message
02:00.544  ID:4  [INFO: Mqtt_node ] [INFO] - Light detection
02:00.544  ID:4  [INFO: Mqtt_node ] {"idlight":1,"lights":"OFF","ligh...
02:14.118  ID:3  [INFO: Mqtt_node ] [INFO] - Public message on topic ...
02:14.118  ID:3  [INFO: Mqtt_node ] [INFO] - Light detection
02:14.118  ID:3  [INFO: Mqtt_node ] {"idlight":2,"lights":"OFF","ligh...
02:15.522  ID:2  [INFO: Mqtt_node ] [INFO] - Public message on topic ...
02:15.522  ID:2  [INFO: Mqtt_node ] [INFO] - Light detection
02:15.522  ID:2  [INFO: Mqtt_node ] {"idlight":2,"lights":"OFF","ligh...
```

Figure 5: Simulation on Cooja

5.2 Test on real board

To use physical sensors instead of the Cooja simulator, certain terminal commands need to be executed. Initially, it is essential to clean the environment of the dongles and subsequently flash the dongles with the application code. It is imperative to emphasize that the cleaning and flashing phases must be carried out within a Docker container named "Contikier." Given that there are one Border Router (BR), two MQTT sensors, and two CoAP sensors, a separate container must be initiated for each of them. Consequently, one terminal instance per sensor should be opened, and the provided commands should be executed. However, prior to launching the containers, the dongles must be attached to the Virtual Machine.

Moreover each sensor that will be attached will be its port that is:

```
/dev/ttyACMz
```

Where z is the number from 0 to last sensor attached and will be a sequential number for each dongle. Port number is important to give sensor a reference point. In this project:

- ACM0 - Border Router
- ACM1 - Sensor (MQTT)
- ACM2 - Sensor (MQTT)
- ACM3 - Actuator (COAP)
- ACM4 - Actuator (COAP)

Like said before in each folder (BR,MQTT,COAP) contikier must be access in this way:

```
osboxes@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/rpl_border_router$ ...
    contikier
user@osboxes:¬/contiki-ng$ cd ...
    examples/IOT_Project/sensor-actuators/rpl_border_router/
user@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/rpl_border_router$
```

After this access, cleaning must be once for each folder:

```
user@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/rpl_border_router$ ...
    make TARGET=nrf52840 clean
user@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/rpl_border_router$ ...
    make TARGET=nrf52840 distclean
```

After cleaning now the flashing phase must be execute:

```
user@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/rpl_border_router$ ...
    make TARGET=nrf52840 BOARD=dongle border-router.dfu-upload ...
    PORT=/dev/ttyACM0
```

```
user@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/mqtt-sensor$ ...
    make TARGET=nrf52840 BOARD=dongle mqtt-node.dfu-upload PORT=/dev/ttyACM1
```

Repeat last command one more time into a second new terminal to flash ACM2, starting first contikier like showed before.

```
user@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/coap$ ...
    make TARGET=nrf52840
BOARD=dongle ligh-node.dfu-upload PORT=/dev/ttyACM3
```

Repeat last command one more time into a second new terminal to flash ACM4, starting first contikier like showed before. So for cleaning phase only three terminals, for flashing five terminals.

Subsequently to the aforementioned procedure, application could be start up. Before starting the application, launch Java Jar with these two commands inside Project folder with *Pom.xml* file:

```
mvn clean install
java -jar target/IoT_Project-1.0-SNAPSHOT.jar
```

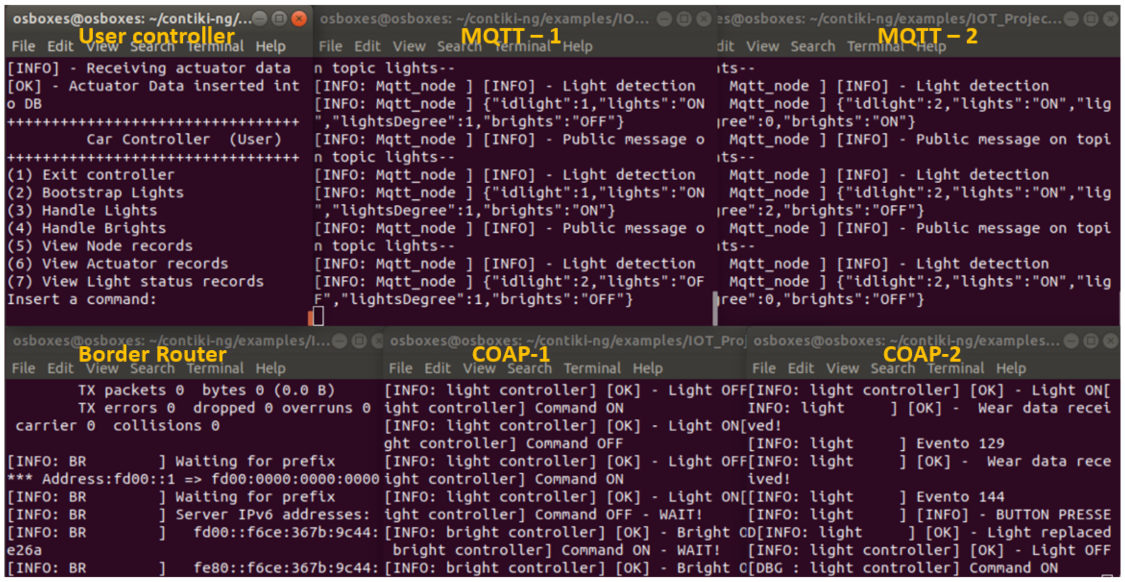
Now open all terminals except that one of Border Router, outside container and type the following command:

```
osboxes@osboxes:¬/contiki-ng/examples/IOT_Project/sensor-actuators/(mqtt-coap)$make ...
    login TARGET=nrf52840 BOARD=dongle PORT=/dev/ttyACMz
```

Where z are all serial port (ACM1,ACM2,ACM3,ACM4) After last commands, open a new terminal in Border Router folder and type the following command to start up the application:

```
osboxes@osboxes:~/contiki-ng/examples/IOT_Project/sensor-actuators/rpl_border_router$
make TARGET=nrf52840 BOARD=dongle PORT=/dev/ttyACM0 connect-router
```

In Fig.6 an image that shows the terminal of the boards and in Fig.7 an image of four out of five boards used for application. Board not in image is the BR that has no led logic.


osboxes@osboxes: ~/contiki-ng/...
User controller
File Edit View Search Terminal Help
osboxes@osboxes: ~/contiki-ng/examples/IO...
MQTT – 1
File Edit View Search Terminal Help
es@osboxes: ~/contiki-ng/examples/IOT_Projec...
MQTT – 2
dit View Search Terminal Help
osboxes@osboxes: ~/contiki-ng/examples/I...
Border Router
File Edit View Search Terminal Help
osboxes@osboxes: ~/contiki-ng/examples/IOT_Proj
COAP-1
File Edit View Search Terminal Help
osboxes@osboxes: ~/contiki-ng/examples...
COAP-2
File Edit View Search Terminal Help

```
[INFO] - Receiving actuator data
[OK] - Actuator Data inserted int
o DB
+++++++++++++++++++++++++++++++++
        Car Controller  (User)
+++++++++++++++++++++++++++++++++
(1) Exit controller
(2) Bootstrap Lights
(3) Handle Lights
(4) Handle Brights
(5) View Node records
(6) View Actuator records
(7) View Light status records
Insert a command:
```

```
n topic lights--
[INFO: Mqtt_node ] [INFO] - Light detection
[INFO: Mqtt_node ] {"idlight":1,"lights":"ON
","lightsDegree":1,"brights":"OFF"}
[INFO: Mqtt_node ] [INFO] - Public message o
n topic lights--
[INFO: Mqtt_node ] [INFO] - Light detection
[INFO: Mqtt_node ] {"idlight":1,"lights":"ON
","lightsDegree":1,"brights":"ON"}
[INFO: Mqtt_node ] [INFO] - Public message o
n topic lights--
[INFO: Mqtt_node ] [INFO] - Light detection
[INFO: Mqtt_node ] {"idlight":2,"lights":"OF
F","lightsDegree":1,"brights":"OFF"}
```

```
its--
 Mqtt_node ] [INFO] - Light detection
 Mqtt_node ] {"idlight":2,"lights":"ON","lig
ree":0,"brights":"ON"}
 Mqtt_node ] [INFO] - Public message on topi
its--
 Mqtt_node ] [INFO] - Light detection
 Mqtt_node ] {"idlight":2,"lights":"ON","lig
ree":2,"brights":"OFF"}
 Mqtt_node ] [INFO] - Public message on topi
its--
 Mqtt_node ] [INFO] - Light detection
 Mqtt_node ] {"idlight":2,"lights":"ON","lig
ree":0,"brights":"OFF"}
```

```
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0
 carrier 0  collisions 0

[INFO: BR        ] Waiting for prefix
*** Address:fd00::1 => fd00:0000:0000:0000
[INFO: BR        ] Waiting for prefix
[INFO: BR        ] Server IPv6 addresses:
[INFO: BR        ]   fd00::f6ce:367b:9c44:
e26a
[INFO: BR        ]   fe80::f6ce:367b:9c44:
```

```
[INFO: light controller] [OK] - Light OFF
ight controller] Command ON
[INFO: light controller] [OK] - Light ON
ght controller] Command OFF
[INFO: light controller] [OK] - Light OFF
ight controller] Command ON
[INFO: light controller] [OK] - Light ON
ight controller] Command OFF - WAIT!
[INFO: bright controller] [OK] - Bright C
 bright controller] Command ON - WAIT!
[INFO: bright controller] [OK] - Bright C
```

```
[INFO: light controller] [OK] - Light ON[
INFO: light      ] [OK] -  Wear data recei
ved!
[INFO: light      ] Evento 129
[INFO: light      ] [OK] -  Wear data rece
ived!
[INFO: light      ] Evento 144
[INFO: light      ] [INFO] - BUTTON PRESSE
D[INFO: light      ] [OK] - Light replaced
[INFO: light controller] [OK] - Light OFF
[DBG : light controller] Command ON
```

Figure 6: Real sensors simulation

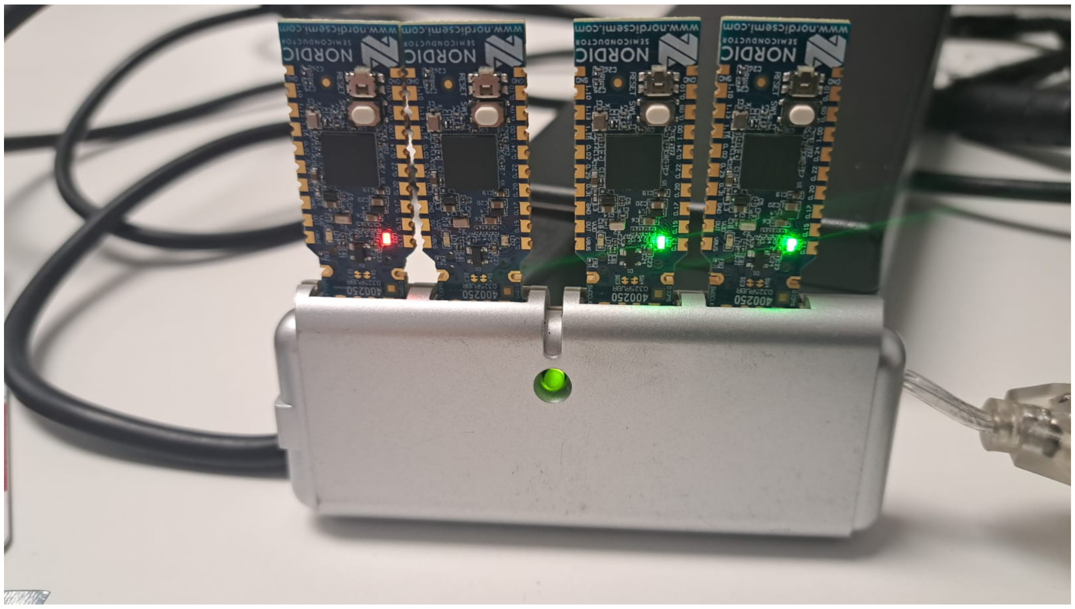


Figure 7: Sensors lights during execution

6 Grafana

Grafana is an open-source platform dedicated to monitoring and observability. It enjoys extensive usage for the visualization and analysis of data from diverse sources, empowering users to construct interactive and adaptable dashboards. These dashboards provide valuable insights into the operational efficiency and functionality of systems, applications, and infrastructure. To craft these informative dashboards, Grafana has the capability to establish connections with databases, with MySQL being a notable example in this context. In Fig.8, the wear level of an actuator over time is depicted. It increases until it reaches a certain threshold, after which its value returns to 0 thanks to user action of press button to replace burn out light.

In Fig. 9, the number of time that a light is turn on by an actuator.

Finally, in Fig.10, the status of light and brightness of a certain sensor are shown over time.

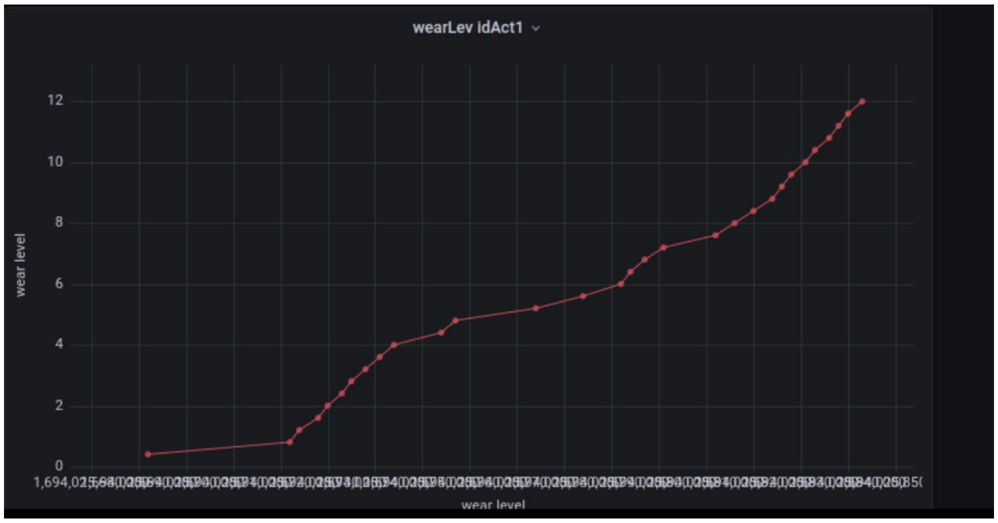


Figure 8: Trend WearLevel of a light

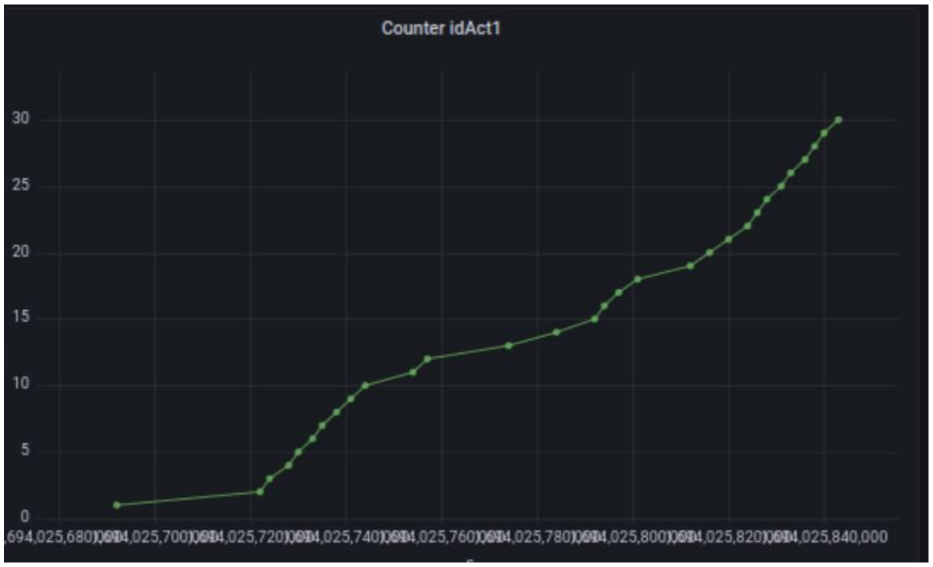


Figure 9: Trend counter of a light

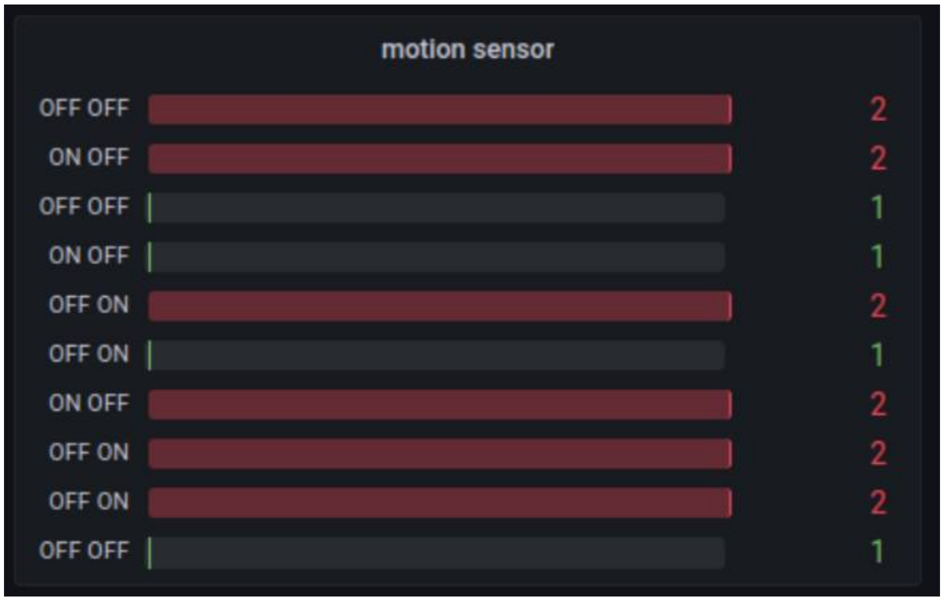


Figure 10: Sensors status