
UNIVERSITÁ DI PISA
ENGEGNERIA DELL'INFORMAZIONE

# IoT telemetry and control system

# Project Report

**Bruno Augusto Casu Pereira de Sousa**

**Master's degree Computer Engineering – Computer Systems and Networks**
**882II 22/23 Internet of Things**

Pisa, 2023

# Contents

# 1. Introduction

The objective of the project is to develop a telemetry system based on IoT devices. The platform will use several nodes forming a Wireless Sensor Network (WSN) to collect and transmit data (sensor readings) to a remote cloud application, that will store it in a database. The system must also use a control application that reads the information stored in the database and sends commands back to actuators deployed in the WSN. The overall architecture of the system is illustrated in Figure 1:

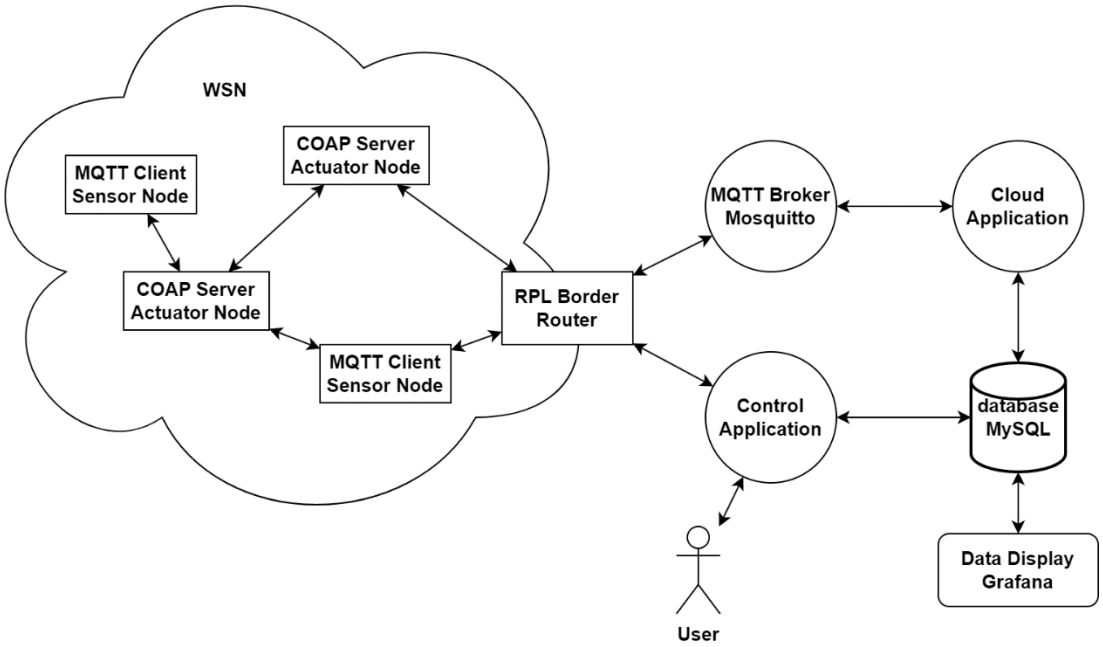


Figure 1 – IoT telemetry and monitoring system architecture.

To diversify the technologies used in the project the Nodes deployed in the monitoring WSN will use the Message Queuing Telemetry Transport (MQTT) protocol and the Constrained Application Protocol (COAP) in its application layer. The sensor Nodes will be MQTT Clients, connected to a MQTT broker (Mosquitto), and the actuator Nodes will be COAP servers exposing a resource.

For the Cloud Application service, a Python based module will subscribe to the topic where the sensors are publishing and, as it receives the messages containing the sensor readings, will write the information in a table on a MySQL database. Once the data is stored a Grafana dashboard will be used to generate a web page, displaying the sensor readings in a chart, allowing a user to observe and analyze the behavior of the system that is being monitored.

Also, a second Python module, called Control Application, will periodically read the data from the database (DB) and by using a simple logic, will issue commands to the actuators deployed. This control application will also allow users to manually send messages to the actuators.

## 2. Use-case scenario

When growing corps indoor, in many cases there is a need to control the temperature inside the greenhouses, to ensure that the plants can develop well and to increase the crop yield. For example, when growing tomatoes indoors, the temperature must be kept below a certain value so that the fruits are not damaged.

To provide such temperature regulation, the most common approach is to use a combination of temperature sensors to collect the local temperature in the crops and a set of cooling fans, that can reduce the temperature inside the greenhouse. A closed-loop control system is generally used in the regulation to active the cooling system when the average temperature is above a certain value.

Figure 2 – Greenhouse farming using fans as a temperature control mechanism.

This type of use-case scenario can benefit from a modern wireless system, that can accurately collect the data and provide automatic control over a large monitoring area. The IoT telemetry system proposed could help to improve the quality of the temperature control over the corps and to provide better statistics and data that could be included in the study of the plants development. This type of platform can even reduce the cost of deploying the monitoring system in a large farming area as it eliminates the use of cables and optimizes of the activation of the fans, allowing remote control over the actuators.

The project developed then aims to provide a WSN and a control system that will collect temperature data using Sensor Nodes and activate or deactivate a set of Actuator Nodes deployed in the WSN, to reduce the temperature in the monitored area according to a set threshold value.

To better adapt the IoT platform described to this application and considering that some of the monitoring areas could be large, the sensors and actuators will be divided in monitoring Sections. The control system then evaluates the average temperature of each

independent section to decide if it is necessary to activate the actuators and the cooling systems on that specific Section. More details of the data formats and identifiers will be explained in the Data Acquisition section of the report.

To illustrate the Node distribution in a possible scenario, Figure 3 shows two perspectives of a greenhouse, where the WSN is placed:

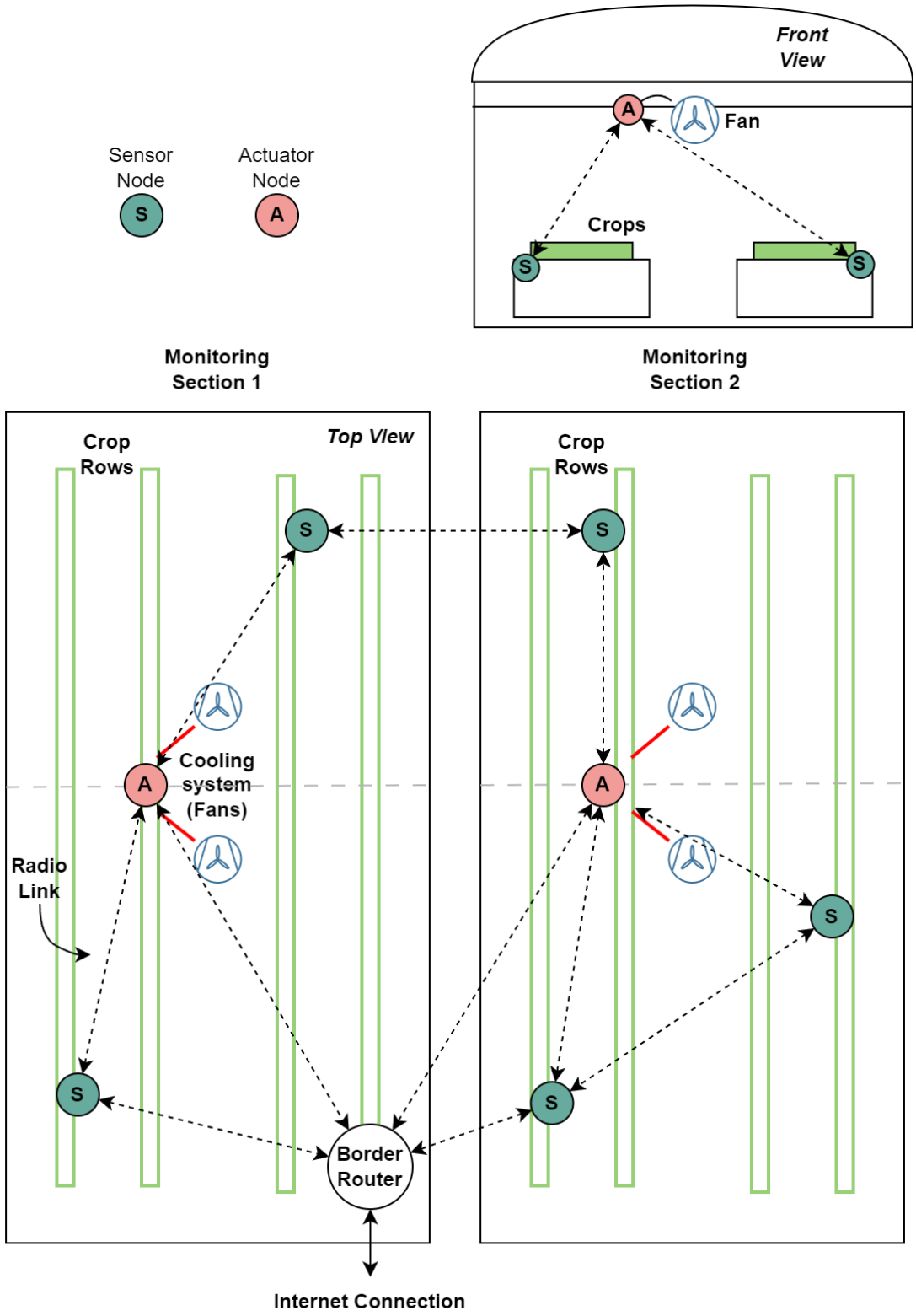


Figure 3 – Example of the node's placement in a WSN deployed in a greenhouse.

# 3. Data Acquisition system

To obtain the temperature readings in the deployed WSN, the Sensor Nodes, the data flow on the system will start from the Sensor nodes transmitting periodically data frames to the MQTT broker. To do so, the sensor will communicate with a border router, that also uses the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL), allowing the sensors messages to reach the external network and connect to the MQTT broker.

As the data frames reach the broker, this component will send to all nodes subscribed to the sensor data topic the message received. A Python module (defined as the Cloud Application) will operate as a MQTT client and subscribe to this topic, receiving all the data frames from the monitoring WSN. The payload of the messages will be parsed and the temperature reading, along with the sender information will be stored in a relational database system (in this application MySQL will be used as the DB system). To illustrate the data path, Figure 4 shows the components used in the monitoring system data acquisition section:

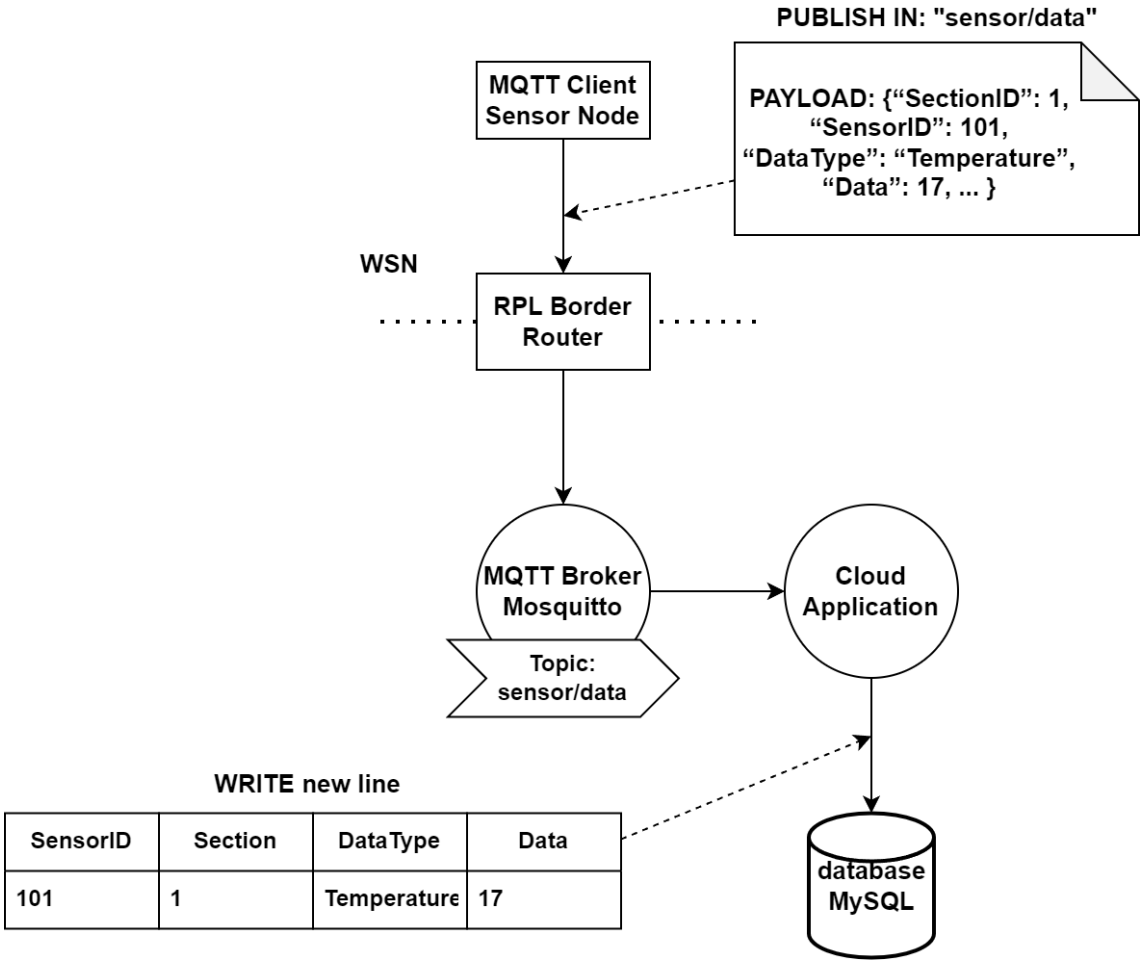


Figure 4 – Data acquisition system, data flow diagram.

## 3.1 MQTT Sensor

The Sensor Node will use a MQTT Client program, based on the example provided by Contiki-NG. By exploiting the many network libraries that Contiki-NG has, the client node can be easily configured, as the network stack is handled in the lower layers.

With that, a Finite State Machine is used in the program to track the Sensor Node connection state. The states are checked and updated based on a loop that waits for an event to occur. To generate the events, the MQTT function mqtt_register() will register a set of callbacks that can trigger the FSM (used to identify when the Sensor Node has established a successful connection with the broker, or it has been disconnected). Also, an event timer is used in the program (defined as process_timer), and the expiration time is set according to the state and action necessary. With both these triggers, the FSM will be constantly polled, allowing the system to know its connection status and when to publish a message containing the sensor data. The transitions of the sates are then defined as:

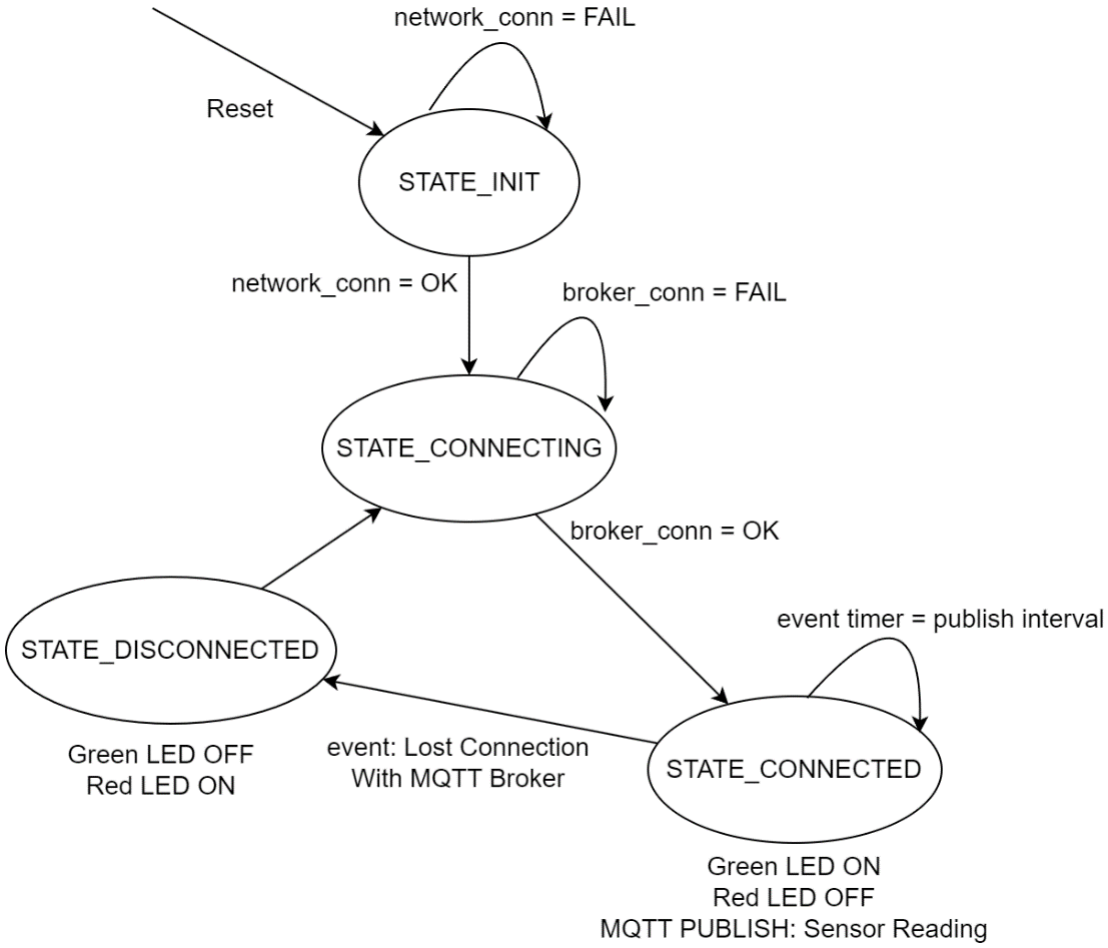


Figure 5 – MQTT Sensor node FSM.

The four states represented have the following properties:

1. **STATE_INIT**: In this state, the MQTT Sensor node will attempt to get a network connection, by using the underlying IP protocols. This connection is obtained once the Border Router is set. The Node will remain in this state and will check again for the connection status using the process_timer trigger that is set for RECONNECT_PERIOD seconds. The RED LED is set on in this state.
2. **STATE_CONNECTING**: In this state the MQTT Sensor node has a network connection and will attempt to connect to the MQTT Broker. If the connection is not established, the Node will retry the connection by setting the process_timer with an interval of RECONNECT_PERIOD seconds plus a random delay value (between 0 and 10 seconds).
3. **STATE_CONNECTED**: In this state, the MQTT Sensor node has obtained network connectivity and has connected to the MQTT Broker successfully. The node will remain in this state until a disconnect event is triggered. In this state the Node will generate and publish a message process_timer is set with an interval of PUBLISH_DATA_PERIOD. The RED LED is set off and the GREEN LED is set on in this state.
4. **STATE_DISCONNECTED**: This is a transition state and is set after the MQTT defined callback for the disconnection event polls the FSM loop. When the Node enters this state, it will configure the notification LEDS (RED on and GREEN off) will set the state to STATE_CONNECTING and set the process_timer with RECONNECT_PERIOD seconds.

When connected, as mentioned above, the MQTT Sensor node will periodically publish a message. This operation is configured to publish in the “sensor/data” topic. The frame that is generated in the node provides all the necessary info for the Cloud Application to identify the sender and the Section that the sensor reading is from. The referred data frame uses a JSON encoded payload. An example of payload is:

```
{
“SectionID”: 1,
“SensorID”: 101,
“DataType”: “Temperature”,
“Data”: 17,
“Platform”: “cooja”,
“MsgNumber”: 10,
“Uptime (sec)”: 82200
}
```

The SensorID and SectionID parameters are both hard coded in the node program and remain constant during the operation (the nodes are not considered to be mobile). When simulating the network in Cooja, a different approach was used, and to obtain its SensorID parameter, the nodes will use its link-local IPv6 as a base to generate the id number. For the SectionID, the nodes with even IPv6 addresses are assigned to Section 1, and nodes with odd IPv6 addresses are assigned to Section 2.

In the project, since the devices used (nRF52840) do not contain actual temperature sensors, the data added to the messages is randomly generated, and it is used only for test purposed and to demonstrate the platform operation.

## 3.2 RPL Border Router and MQTT Broker

The Border Router used in the project is the rpl-border-router example provided by Contiki-NG. No modifications were done in the device, as it is only intended to provide the WSN access to the external network. For the MQTT Broker system, the Mosquitto MQTT Broker software was used, as part of the project specifications. The broker then automatically sets the topic where the sensor nodes are publishing and retransmit the messages the subscribers. In the project deployment the only subscriber to this topic will be the Cloud Application.

## 3.3 Cloud Application

The Cloud Application is a python-based module that works as a MQTT Client, using the "paho" python library. At the start of the program, the application will connect to the MySQL database management system, with a set of defined credentials coded. On the project design, a new database was created, and one table was set to store the sensor readings, defined as *sensor_data:*

```
mysql> USE sensor_db
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+----------------------+
| Tables_in_sensor_db |
+----------------------+
| sensor_data          |
| test                 |
+----------------------+
2 rows in set (0.00 sec)

mysql>
```

```
mysql> SELECT * FROM sensor_data LIMIT 10;
+----+----------+---------+-------------+------+---------------------+
| id | sensorid | section | datatype    | data | timestamp           |
+----+----------+---------+-------------+------+---------------------+
|  1 | 1900     | 1       | Temperature |   16 | 2023-07-08 17:29:15 |
|  2 | 1900     | 1       | Temperature |   20 | 2023-07-08 17:29:18 |
|  3 | 1900     | 1       | Temperature |   17 | 2023-07-08 17:29:21 |
|  4 | 1900     | 1       | Temperature |   17 | 2023-07-08 17:29:24 |
|  5 | 1900     | 1       | Temperature |   18 | 2023-07-08 17:29:27 |
|  6 | 1900     | 1       | Temperature |   18 | 2023-07-08 17:29:30 |
|  7 | 1900     | 1       | Temperature |   21 | 2023-07-08 17:29:33 |
|  8 | 1900     | 1       | Temperature |   19 | 2023-07-08 17:29:36 |
|  9 | 1900     | 1       | Temperature |   17 | 2023-07-08 17:29:39 |
| 10 | 1900     | 1       | Temperature |   19 | 2023-07-08 17:29:42 |
+----+----------+---------+-------------+------+---------------------+
10 rows in set (0.00 sec)

mysql>
```

Figure 6 – Screen capture from MySQL terminal with the application DB

After this first configuration step in the Cloud app, the program will then attempt to connect to the MQTT broker (Mosquitto), once connected, it will subscribe to the "sensor/data" topic. If a disconnection occurs at any point, the application automatically attempts to connected again with the broker. These operations are managed by the callbacks set on_connect() and on_disconnect().

With the DB and MQTT settings completed, the application will then wait for incoming messages from the broker. As a new message arrives (the on_message() callback is used to identify this event), the application will parse the fields from the JSON payload received, and will do a data inset query on the MySQL data base. The query designed is the following:

```
      sensor_data_insert_query = '''
INSERT INTO sensor_data (sensorid, section, datatype, data) VALUES
(%s, %s, %s, %s)
      '''
```

## 3.4 Grafana and data display

Following the project instructions, to allow users to view and monitor the temperature sensor collected data a web-based visualization tool was used (Grafana). This application allows the deployment of a local server that creates a web page showing in real time charts with the sensor data (can be access from a browser in the URL localhost:3000). To produce the dashboards, the queries had to be adapted to the relational database format used:

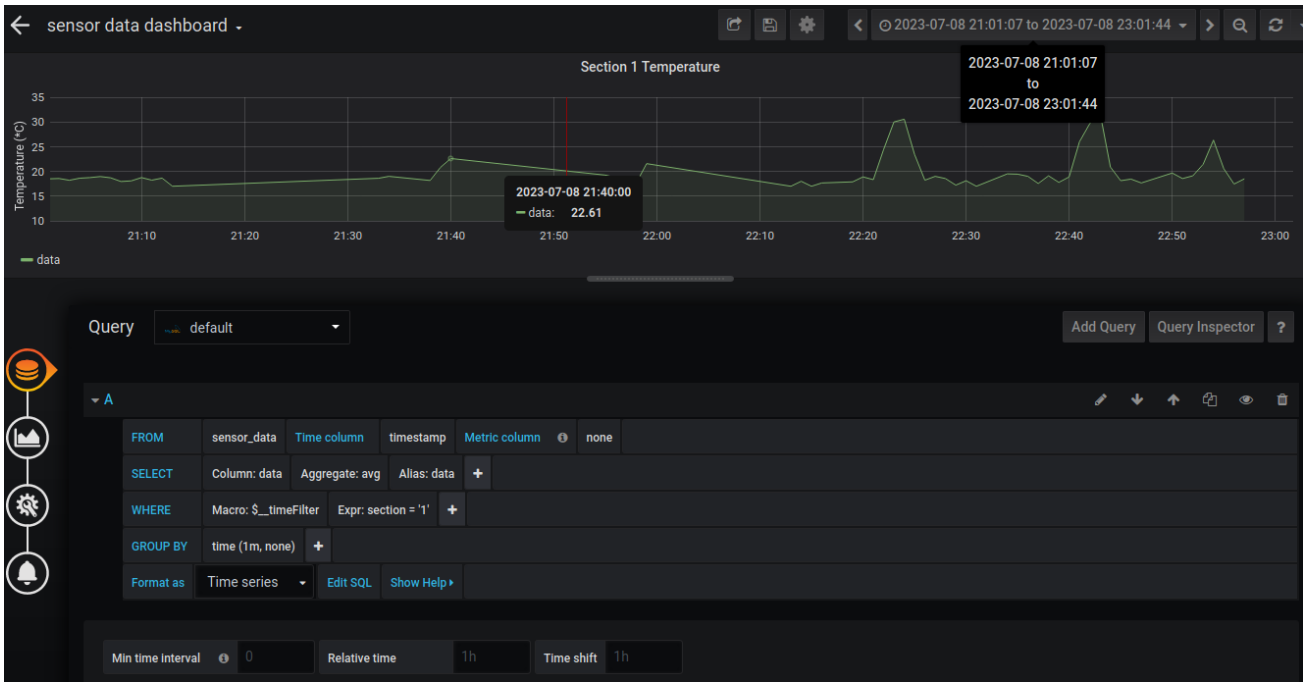


Figure 7 – Screen capture from Grafana query definition

To demonstrate the platform operations 2 Sections were created in the simulation, and for each one a chart was set to show the temperature data on that Section. The images then show the data collected from the tests using the Cooja simulator:

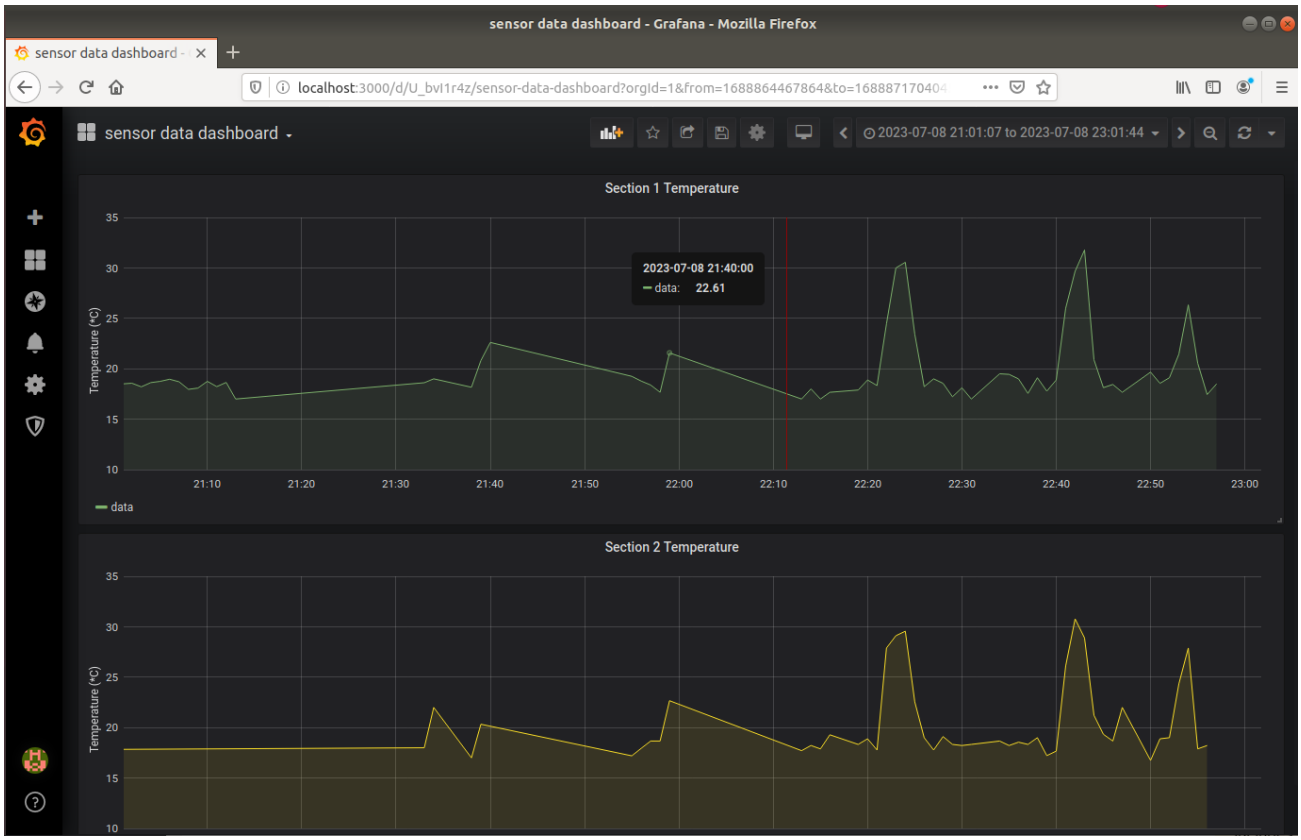

Figure 8 – Screen capture from Grafana sensor data dashboard

# 4. Temperature control system

As the data acquisition section of the monitoring system is defined and operational, the next section will describe the control logic used in the application scenario. As the sensors and actuators are organized by sections, the main parameter that is used to evaluate the environment temperature in the monitored area will be the average temperature measured in the latest sensor readings in a Section.

By combining the data from multiple sensors and calculating an average temperature, the control system will have a better evaluation of the necessity of activating of the cooling system to reduce the temperature. This method also prevents that the decision is made based on outlier values, in case of a malfunction in one of the nodes. Also, by aggregating the latest readings from the database, the system can still monitor the temperature if one or more sensors go down.

The actuators on this platform will then activate or suspend the cooling system and should work as COAP servers exposing a resource. With that, the remote-control application can send POST and GET messages to manage the activation of the temperature regulation system. The actuators will, for the most part, be managed by sending commands from the external network, changing its internal state according to the activation parameter sent in the POST message.

However, to allow more flexibility for the activation of the cooling system, the actuators will also have a physical trigger method (in the deployed MCU platform this is

managed by a button) that changes its internal state (activate or suspend the cooling system). This then grants the users a way to manually set the temperature regulation system, without the need of the whole platform to evaluate the risk condition.

The overall scheme of the component's communication is illustrated in Figure 9:

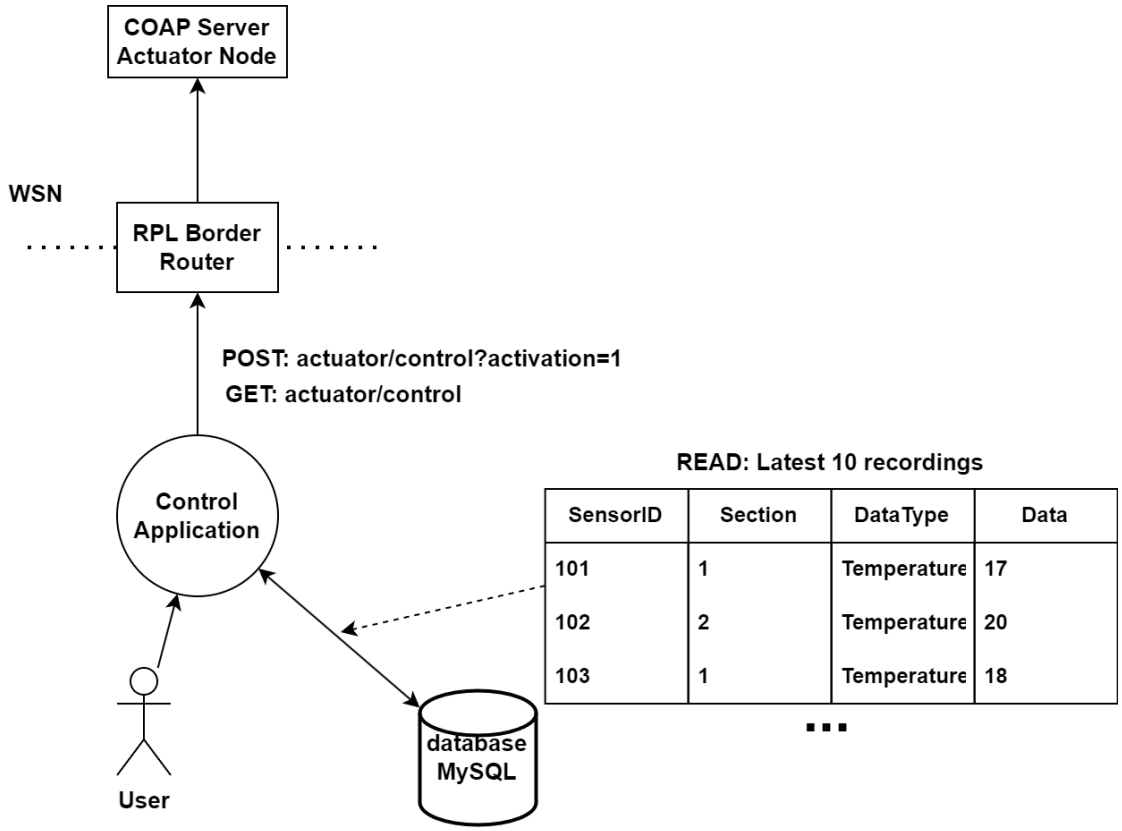


Figure 9 – Control system, data flow diagram.

## 4.1 Control Application

The Control Application uses several python functions to execute the necessary query in the DB, to retrieve the latest sensor readings, and to properly evaluate risk conditions for the crops and set the actuators. The conditions that trigger the activation of the colling system is if the average temperature measurement in a Section is above a certain threshold value.

To obtain the average temperature values, the Control app will execute a loop where in every cycle the component will send a GET message to the COAP server of all Sections, requesting the status of the cooling system (ON or OFF). If a reply message is received, the terminal will print the information on teach Section actuator.

After this initial message, the control application will perform a query in the DB, on the sensor_data table. The query will return the last 10 entries in the table, and a function will parse and combine the temperature values for each Section, compute the average value. The MySQL query designed for this operation is the following:

```
sensor_data_read_query = '''
    SELECT sensorid, section, datatype, data
    FROM sensor_data
    ORDER BY timestamp DESC
    LIMIT 10
'''
```

An important consideration is that the number of entries read on each cycle must be compatible with the number of sensors deployed (in the project design for two Sections with a total of six sensors, the number was set to 10 entries).

In addition to that, the delay between the queries must also be compatible with the use-case application and should be less than or equal to PUBLISH_DATA_PERIOD, in seconds, otherwise information may be lost as the sensors will write more than once before the control system evaluates the average temperature. The monitoring cycle is illustrated in Figure 10:

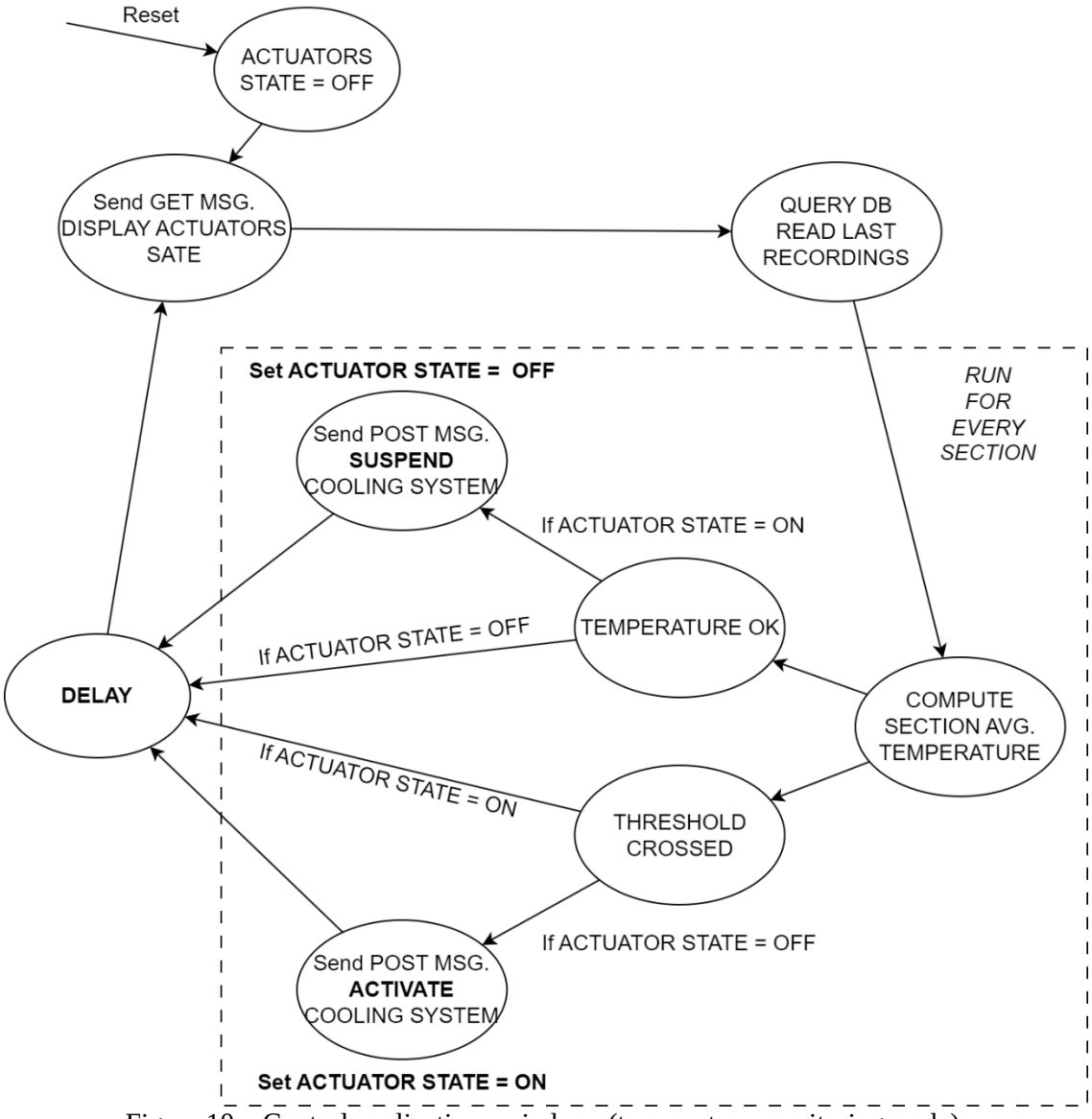


Figure 10 – Control application main loop (temperature monitoring cycle).

Once the average values are compared with the threshold, the control logic will decide if a message to the actuator must be generated. To send the POST and GET messages, the python library "coapthon" was used to build the frames to be sent to the resource. As a requirement, the addresses of the actuator nodes are statically defined in the control app code.

For the message's payload, the GET message is a simple request to the COAP server with no parameter. The expected reply is a string containing the actuator status and the Section identification. An important mention on this property is that the Control Application does not update its internal actuator status based on any reply from the remote nodes. This is done to prevent that the remote-control logic interfere with a manual set on the cooling system.

For example, the temperature of a particular section is below the threshold defined, meaning that the Control App will NOT send an activation message, and the status of the actuator is OFF. However, if a user decides to activate the cooling system (not related to the average temperature status), the Actuator COAP server will have its status as ON. In this case, if the Control App were to read that status, and evaluate that the temperature is OK, it would send a POST message suspending the cooling system. This then would cause a conflict, as it would almost invalidate any user manual activation. Therefore, the Control App keeps an internal status of the cooling system and does not use the status of the actuators in its logic.

Along with the GET message, a POST message is also configured in the control logic, and the frame contains a parameter that is sent to the server. Based on this parameter the remote application can control the state of the cooling system. If the activation parameter is sent with a value of 1, the actuator must activate the fans to reduce the temperature of the Section. If a value of 0 is sent, the fans must be shut down. An example of the activation and suspension of the temperature control system are, respectively:

```
POST:
coap://[fd00::302:304:506:708]:5683/actuator/control?activation=1

POST:
coap://[fd00::302:304:506:708]:5683/actuator/control?activation=0
```

When these messages are sent, the application the updates its internal status only if the COAP server replies with an acknowledge message. If the POST fails, the control application will attempt again to set the cooling system, if it still detects a high temperature value.

The control application can be set to automatically run the periodical temperature checks, but the user can also use the available functions to transmit the POST and GET messages manually and set the state of the actuators remotely. The two functions designed are:

- **post_event(section, action)** : this function takes the Section number parameter (integer) and the activation code to be send in the payload (either 0 or 1). It generates a POST request to the actuator registered on that Section and prints the reply message.

- **get_actuator_status(section)** : this function takes the Section number parameter and sends a GET request to the resource. It will print the response message with the actuator status.

### 4.2 COAP Actuator

The COAP actuator node is based on the Contiki-NG example of the COAP server. In this program a simple loop will start the server and keep the system toggling a LED, to identify that it is waiting for commands. When initialized, the actuator is set to the state OFF (the cooling system is suspended), and the RED LED is set to off.

A resource file is created in the resource folder and exposes a resource that accepts two types of requests, either a GET message or a POST message. The resource address uses Port 5683 and is defined as "actuator/control" as the example:

```
coap://[fd00::302:304:506:708]:5683/actuator/control
```

When receiving a GET message the COAP Actuator must reply with a string containing the actuator status and its Section identification. The possible states are ON, OFF or FAULT (the latest is currently not in use but was kept is a possible expansion on the actuator system).

If a POST message is received the program will check for a mandatory parameter named "activation" that must be sent in the message payload. This parameter can only have two values set, 0 or 1. When the actuator receives a POST message with an activation=1, it must update its internal state and set the cooling system to ON. The reply sent on this POST message is the confirmation that the temperature control system has being activated. In this state the node will turn on the RED LED of the device.

If the parameter sent is activation=0 the COAP Actuator must set its internal state to OFF and reply with a confirmation message. The RED LED is off in this state.

The logic used in the actuator node was made simple, in a way that it prevents the program to go to an unknow or error state due to some misconfiguration, or wrong parameter set. When receiving a payload with wrong encoding or missing parameter the node will respond the message with an error text, but it will not change its internal state.

To complement the operation of the actuator a manual trigger was included in the device program. This was made by adding an event trigger by pressing the device button. When the button is pressed, the device will toggle its internal state.

## 5. Testing: Cooja simulation and nRF52840 deployment

To demonstrate the developed telemetry and control system, the parameters of the sensor and actuator regarding the timing on the readings had to be adjusted. Usually when measuring temperature in large areas, such as the greenhouse environments, the period for collecting the data is around a few minutes, as the temperature values do not change so abruptly. This also reflect the time that the control app reds the information form the DB, as new data is only generated in this interval.

To allow a quick visualization of the system operation those times had to be set to much lower values, in the rage of seconds. That said the overall parameters used in the simulation and in the demonstration using the nRF52840 device, the values are:

- MQTT Sensor:
  - PUBLISH_DATA_PERIOD = 5s
  - RECONNECT_PERIOD = 3s
  - SIMULATION_THRESHOLD_COUNT 8 < msg_n < 13
    Where the threshold count refers to a custom setting that increases the temperature values in messages number 9 to 12 sent by the sensor nodes.
- Control Application:
  - data_check_interval = 5s
  - TEMP_THRESHOLD = 22

To simulate the monitoring platform developed, a Cooja simulation was designed, with a total of 6 Sensor nodes and 2 Actuators, divided in 2 Sections (the RPL border router was also added as a Cooja mote). To connect the border router to the external network a command must be executed in the rpl-border-router folder. This then provides the Cooja nodes connectivity. The nodes were placed in a way that they only see a few neighbors, forcing the usage of the routing protocol to send the messages through the network deployed in the simulation.

Considering the use-case for the project a high-density node distribution in the target application scenario is not necessary as the overall temperature inside the greenhouse should not vary to much depending on the position, meaning that a high granularity sensor network is not necessary to evaluate the internal average temperature.

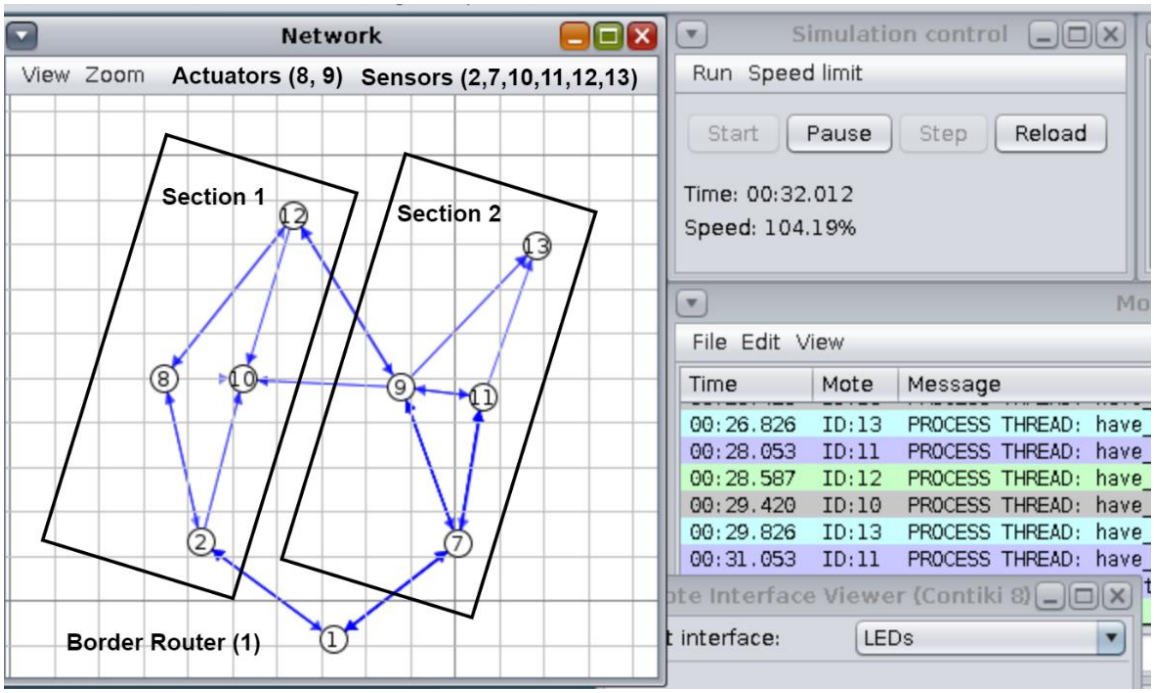


Figure 11 – Node placement in Cooja simulation (edited screenshot).

As the nodes obtain connectivity through the border router, the Mosquitto broker is initialized along with the Cloud app, allowing the published messages to be received by the app and to be stored in the DB. In the simulation execution it is noticeable that nodes that are placed far from the border router take longer to connect to the broker, as the messages must be routed in the WSN. This effect is noticed as the nodes closer to the border router have their GREEN LED turned on before the ones placed far away. Starting the simulation, the operation of the nodes can be traces using the messages printed on the simulator interface, as well as the LED and button interaction provided.

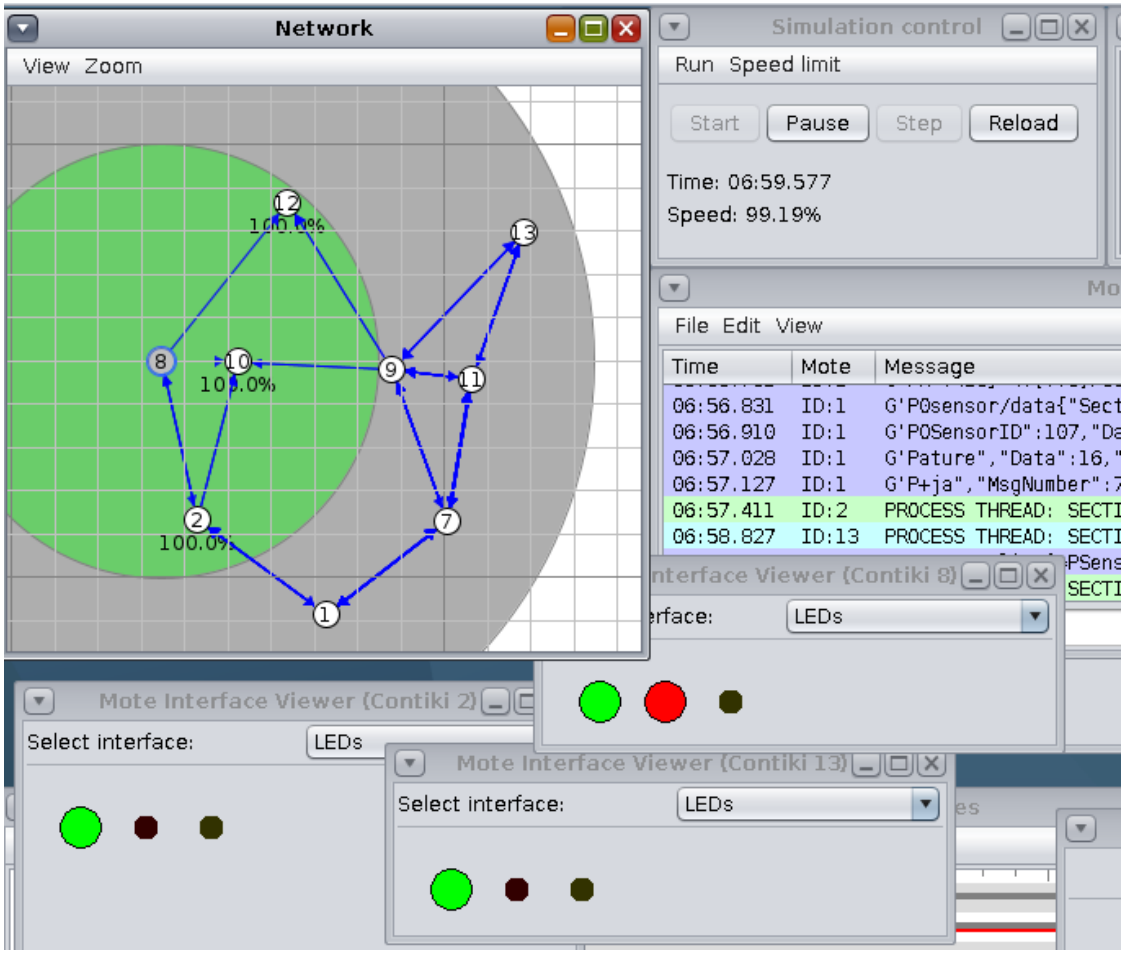

Figure 12 – Node LED and button interaction in Cooja simulation.

Once the data frames are received by the Cloud app the information in the payload is printed. The Control app has a similar behavior, and it prints the reply messages received by the COAP server as the monitoring cycles are execute and the requests are sent. A screenshot of the logs from those applications are shown in Figures 13 and 14:

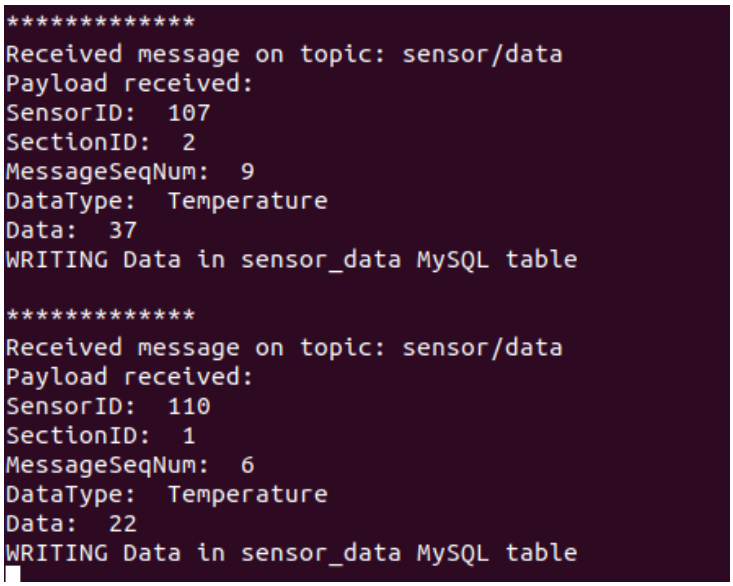


Figure 13 – Cloud application terminal.

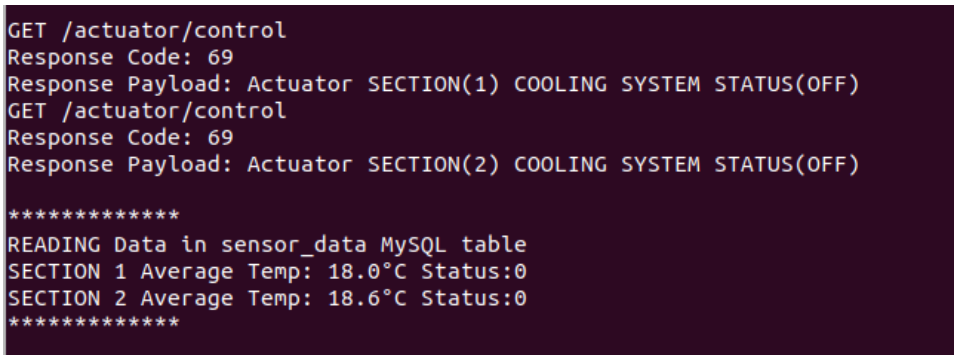


Figure 14 – Control application terminal.

After the simulation in Cooja, the system was tested using the Nordic nRF52840 dongle boards. The overall setup for this test used 3 dongles, one operating as the border router (necessary to convert the frames sent over the radio into the USB port), one as a Sensor node and one as an Actuator node. With that, the complete data path that the monitoring system developed uses could be tested. The behavior of the platform when deployed in the dongles was like the simulation, and the information generated by the sensor node was properly sent to the Cloud application and stored in the database. Also, the commands form the Control application were able to configure the state of the actuators, demonstrating the complete operation of the telemetry and monitoring system. An illustration of the test setup is in Figure 15:

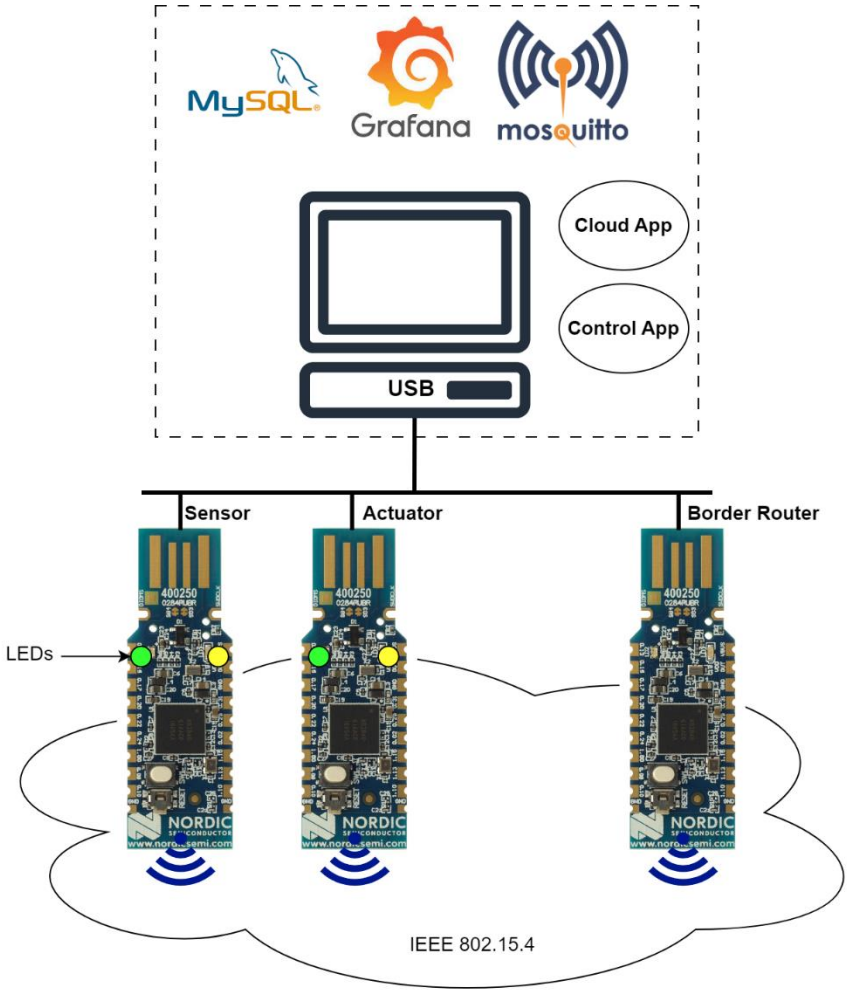


Figure 15 – Dongle test setup.

# 6. Conclusions

The simulation environment shows that the integration between the Data acquisition system and the Monitoring system were working without issues. The Sensor nodes were able to transmit the data frames over the WSN and the DB access was properly done with the designed queries. The conclusion on the system is that the platform developed can accurately collect data from a WSN and to send commands over the network to control the remote devices. By studying the protocols used in the network and physical layers it is possible to conclude that this system could be deployed over a large greenhouse environment, as the nodes were able to transmit the messages even when positioned far away from the border router node.

To deploy the developed monitoring system in a production environment some additional features must be present, such to detect fault in the sensor and actuators, as well as to track the number of packs that are lost. A secure protocol can also be added to the frames sent in the WSN to prevent intrusions and attacks. With some improvements the monitoring system can be adapted and provide a better quality of service for applications similar to the use-case defined in the project.