IoT System for Smart Agriculture

By

Matteo Giorgi

Salvatore Arancio Febbo

Università di Pisa

Department of Information

Artificial Intelligence and Data Engineering,

Pisa, Italy.

(Gennaio 2024)

Contents

Abstract

This project focuses on the implementation of a smart agriculture system, harnessing the potential of the Internet of Things (IoT). We have deployed a network of sensors to monitor environmental humidity, soil condition, and temperature, complemented by actuators dedicated to controlling various environmental aspects. The actuators oversee operations such as opening windows, igniting stoves, activating fans and misting systems, and facilitating irrigation in response to the needs identified by the sensors. This IoT solution aims to optimize agricultural conditions, enabling a more intelligent and automated management of cultivated environments.

List of Figures

CHAPTER 1

Introduction and Motivation

Smart Agriculture, based on the *Internet of Things (IoT)*, represents a crucial frontier in the evolution of the agricultural sector. The growing need to increase efficiency, optimize resources, and address environmental challenges has driven research towards innovative solutions. This project sits at the intersection of technology and agriculture, proposing an intelligent system that uses advanced sensors and actuators programmed in C to collect critical data on the agricultural environment. These devices communicate through a Java-programmed application. Through this connection, the system responds in real-time to detected variations, allowing optimized management of fundamental environmental elements such as **air humidity, soil moisture, and temperature**. The motivation behind this initiative lies in the goal of improving agricultural productivity, reducing waste, and providing cultivators with a *flexible and customizable* tool to tackle the ever-growing challenges in the agricultural sector. In this context, we explore the revolutionary potential of IoT to transform traditional agricultural practices, making agriculture more **sustainable, efficient, and adaptable** to changing environmental conditions.

1.1 Project Objectives

The project aims to develop a dedicated IoT system focusing on **telemetry** and **control**, with a specific use case to be defined. The primary objectives include establishing a network of IoT devices, encompassing **sensor**s for environmental data collection and **actuator**s. Simultaneously, there is a requirement to develop a **cloud** application for acquiring and storing data from **MQTT** sensors into a **MySQL** database. Additionally, the project entails creating a **remote control** application to retrieve information on actuators and sensors from the database, implementing elementary **control logic**. Lastly, the project incorporates a user interface through the command line for **user input** and a web-based interface using **Grafana** for visualizing the collected data.

1.2 Implementation Choices

Before delving into the detailed implementation of the application, it is crucial to outline some strategic choices we have made:

- We opted to develop the Collector and control logic using the Java programming language.
- Concerning data encoding, we chose to use the JSON format. This choice is motivated by the nature of our devices, which are subject to constraints and may experience malfunctions without compromising the integrity of the system. For instance, a farmer can easily check the situation on-site without encountering any issues. Since our application is non-critical, there is no need to resort to a more complex data encoding language like XML.

1.3 Deployment Structure

The ecosystem of **smart agriculture**, particularly concerning the **Internet of Things (IoT)**, has been configured as follows. Sensors have been managed using the **MQTT protocol**, establishing communication with the **Collector** through the **broker**. In practice, sensors register with the broker using their own "**topics**," while the **Collector** subscribes to the "**topics**" of sensors of interest. Subsequently, data from the sensors, stored in **MySQL**, is utilized to control **actuators**. The actuators are operated by the **Collector** using the **CoAP protocol**.

In the context of our application, **air humidity** and **temperature** are crucial for activating the **conditioner** and **window actuators**. Similarly, the **soil humidity sensor**, based on its values, determines whether the **irrigation actuator** should be activated or not.

In the image 3.1, we can get a simplified idea of the architecture of the ecosystem.

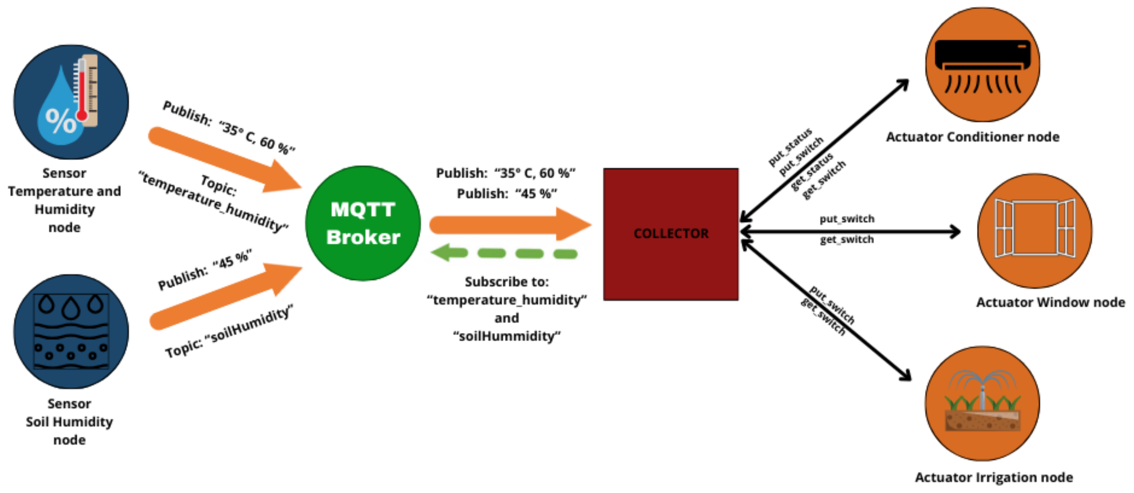


Figure 1.1: Project Architecture

CHAPTER 2

MQTT Network

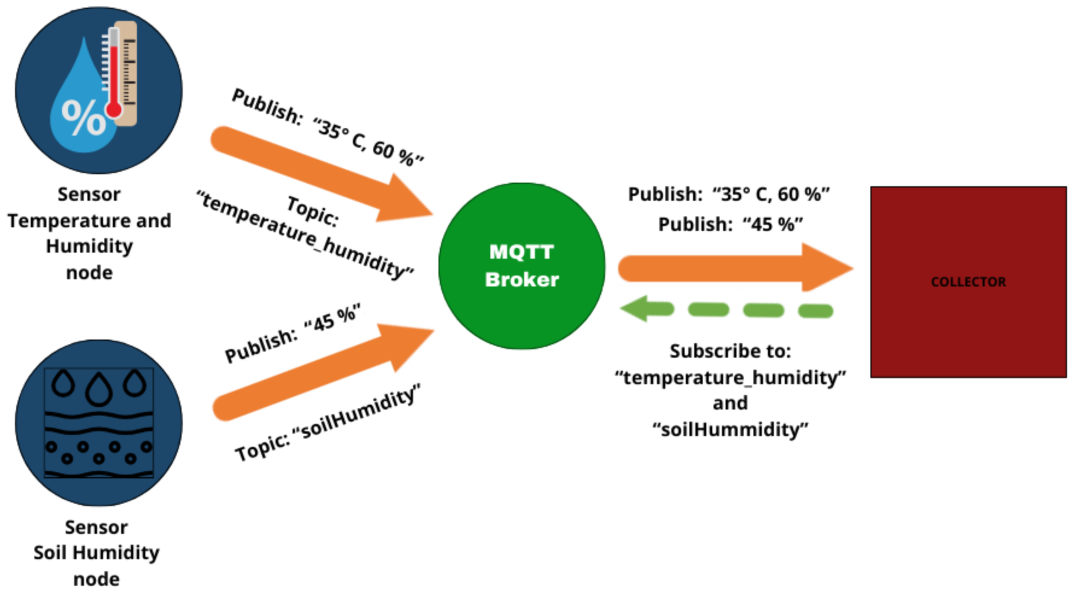

Sensor Temperature and Humidity node
Publish: "35° C, 60 %"
Topic: "temperature_humidity"
MQTT Broker
Publish: "35° C, 60 %"
Publish: "45 %"
COLLECTOR
Subscribe to: "temperature_humidity" and "soilHummidity"
Publish: "45 %"
Topic: "soilHumidity"
Sensor Soil Humidity node

Figure 2.1: MQTT Architecture part

This chapter focuses on the design and implementation of the network based on **MQTT (Message Queuing Telemetry Transport)**, which plays a crucial role in sensor management within our dedicated **IoT system for smart agriculture**. The devices, programmed using the C programming language, are equipped with **MQTT**, renowned for its lightweight nature and versatility. It stands as a fundamental pillar by providing a robust infrastructure for collecting data from a vast array of distributed sensors. Throughout this chapter, we will delve into the design of each sensor, carefully examining how we have seamlessly integrated them with **MQTT** to ensure a reliable and timely flow of data in the context of our smart agricultural project.

2.1 Temperature and Humidity sensors

The temperature and humidity sensors are pivotal components of our dedicated **IoT system for smart agriculture**. These devices are designed to collect essential data about the temperature and humidity in the surrounding environment. The decision to integrate both sensors into a single device was made considering the synergy between them: by working together, **temperature and humidity sensors** do not interfere with each other. This choice is supported by the fact that each **temperature sensor** is associated with a corresponding **humidity sensor**. Placing both sensors in the same location allows not only for efficiency optimization but also for sending a single message encapsulating the values of both potential sensors. This strategy aims to reduce message traffic, thus optimizing communication within the **IoT system** and ensuring the transmission of fewer overall messages. Implemented using the **MQTT (Message Queuing Telemetry Transport)** protocol, the sensor periodically publishes temperature and humidity data to the **MQTT broker**. The topic used for publication is named **"temperature_humidity"**. In addition, this sensor subscribes to the **"condition"** topic, from which it receives information related to temperature and humidity in the case an actuator (such as air conditioner or window) is active and thus influencing the temperature and/or humidity within the greenhouse.

2.1.1 Data Generation

The temperature and humidity sensor generates an initial random value; specifically, the temperature can range between **[0,45]** *degrees*, while the humidity can take a *percentage* value within **[5,95]**. If no actuator is active, the temperature is managed with a random increment within **[-1,+1]**, and the humidity with a value within **[-3,+3]**. In the case where an actuator is active, the sensor receives a message via MQTT specifying the target temperature and/or target humidity along with the respective increment to reach it. The increment is then added to the entity until it reaches the target value. The generated values are formatted into a JSON and sent via MQTT to the broker.

```
...
humidity = 5 + (rand() % 90);
temperature = (rand() % 45);
...
// humidity
if (target_humidity == -1)
{
  // Applica l'incremento al valore corrente di humidity
     [-3,+3]
  humidity += (rand() % 7 - 3);
}
else
{
  // start increasing/decreasing values humidity until it
     reaches the target
  if (humidifying)
  {
    humidity += hum_increment;
    if (humidity >= target_humidity)
    {
      humidity = target_humidity;
      hum_increment = 0;
    }
  }
  else
  {
    humidity -= hum_increment;
    if (humidity <= target_humidity)
    {
      humidity = target_humidity;
      hum_increment = 0;
    }
  }
}
// Assicurati che humidity sia compreso tra 0 e 100
if (humidity < 0)
{
  humidity = 0;
}
else if (humidity > 100)
{
  humidity = 100;
}

// temperature
if (target_temperature == -1)
{
  // Applica l'incremento al valore corrente di temperature
     [-1,+1]
  temperature += (rand() % 2 - 1);
}
else
{
  // start increasing/decreasing values temperature until it
      reaches the target
  if (heating)
  {
    temperature += temp_increment;
    if (temperature >= target_temperature)
    {
      temperature = target_temperature;
      temp_increment = 0;
    }
  }
  else
  {
    temperature -= temp_increment;
    if (temperature <= target_temperature)
    {
      temperature = target_temperature;
      temp_increment = 0;
    }
  }
}
```

2.2 Soil humidity sensor

The soil moisture sensor is a crucial component of our dedicated IoT system for smart agriculture. This sensor is designed to gather fundamental data on soil moisture in the surrounding agricultural environment. Implemented using the MQTT (Message Queuing Telemetry Transport) protocol, the sensor periodically publishes soil moisture data to the MQTT broker. The designated topic for publication is identified as **"soilHumidity"**. In addition, the sensor subscribes to the **"irrigation"** topic, from which it receives information in the event that irrigation is active.This mechanism ensures a consistent flow of information regarding soil moisture, facilitating informed decision-making for optimal agricultural management.

2.2.1 Data Generation

The soil humidity sensor generates a random value between **[0, 99]**. Subsequently, if irrigation is not active, the soil humidity is managed with a random increment within **[-5,+2]**, simulating soil drying. If irrigation is active, a MQTT message is received specifying the target soil humidity and the increment to reach it. The generated values are then sent to the MQTT broker using a JSON format.

```
...
soil_umidity = (rand() % 99);
...
if (target_soil_umidity == -1)
{
```

```c
  // Applica l'incremento al valore corrente di soil_umidity [-5,+2]
  soil_umidity += (rand() % 8) - 5;
}
else
{
  // start incrementing soil_umidity to reach target_soil_umidity
  if (soil_umidity < target_soil_umidity)
  {
    soil_umidity += increment;
  }
  else
  {
    // target_soil_umidity reached
    soil_umidity = target_soil_umidity;
    increment = 0;
  }
}

// Assicurati che soil_umidity sia compreso tra 0 e 100
if (soil_umidity < 0)
{
  soil_umidity = 0;
}
else if (soil_umidity > 100)
{
  soil_umidity = 100;
}
```

2.3 Sensor Device LED States

Each sensor device in the MQTT network is equipped with LED indicators representing different states to provide visual feedback on its network and connection status. The various states and corresponding LED behaviors are as follows:

1. **STATE_INIT:** The initial state where the network is being initialized, and LEDs are turned off.
2. **STATE_NET_OK:** Indicates that the MQTT network is initialized, and LEDs remain turned off.
3. **STATE_CONNECTING:** Represents the phase where the sensor is establishing a connection to the MQTT broker, LEDs remain turned off.
4. **STATE_CONNECTED:** Signifies a successful connection to the MQTT broker, and blue LEDs are illuminated.
5. **STATE_SUBSCRIBED:** Signifies a successful connection to the MQTT broker, and green LEDs are illuminated.
6. **STATE_DISCONNECTED:** Indicates a loss of connection to the MQTT broker, only the red LEDs are turned on.

These LED states allow for quick visual inspection of the sensor devices within the MQTT network, enabling users to assess the network and connection status at a glance.

CHAPTER 3

CoAP Network

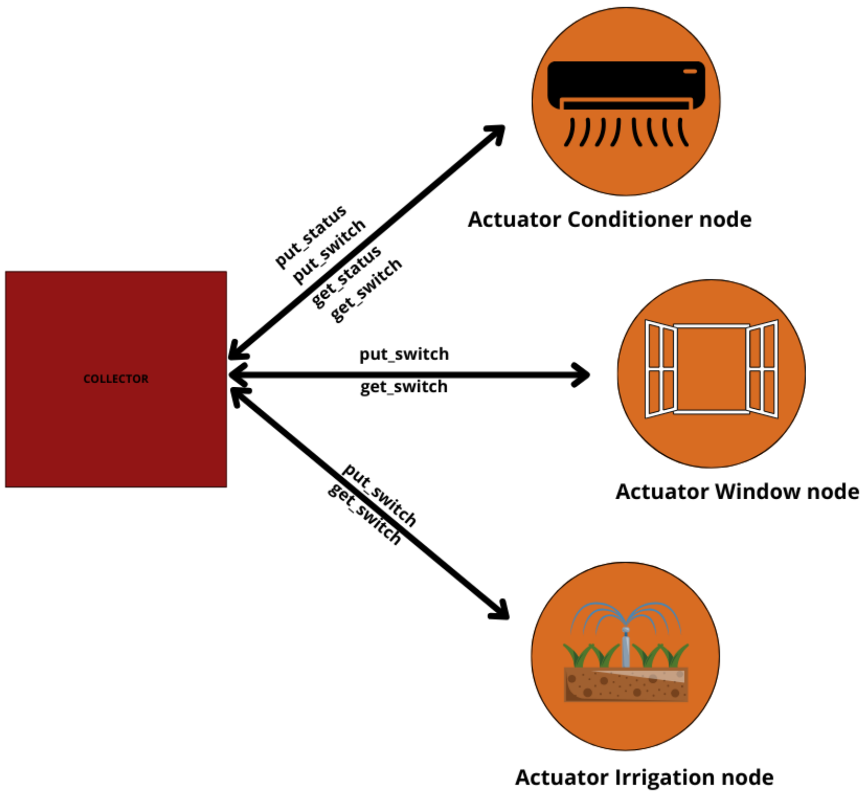


Figure 3.1: CoAP Architecture part

This chapter introduces the network based on **CoAP (Constrained Application Protocol)**, dedicated to the management of actuators within our **IoT system for smart agriculture**. The actuators, programmed in **C**, leverage CoAP, developed for devices with limited resources, providing a lightweight yet efficient solution for controlling and executing actions on the distributed actuators in the system. We will explore the notable features of CoAP that make it particularly suitable for managing actuators in a wide variety of agricultural scenarios, contributing to the creation of an intelligent, dynamic, and highly responsive agricultural environment.

3.1 Conditioner actuator

The *air conditioner* actuator is designed to flexibly activate the heater, humidifier, and fan, either individually or simultaneously. Four operating modes have been defined:

1. **heater:** activates the heater and fan.
2. **heater_humidifier:** activates the heater, humidifier, and fan.
3. **humidifier:** activates the humidifier and fan.
4. **wind:** activates only the fan.

Based on data received from sensors, along with information from other actuators, the *air conditioner* actuator dynamically selects one of these modes. This customization allows for maintaining an optimal environment for the greenhouse in response to changes in environmental conditions.

3.1.1 Resources

Regarding the air conditioner, there are several accessible operations:

1. **get_status_handler:** Provides information about the current status of the air conditioner.
2. **put_status_handler:** Allows for the modification of the air conditioner's state by updating its parameters.
3. **get_switch_handler:** Permits checking whether the air conditioner is currently on or off.
4. **put_switch_handler:** Enables the turning on or off of the air conditioner.

3.2 Window actuator

The *window actuator* is designed with the specific purpose of controlling the opening and closing of the window in response to data collected from environmental sensors or manual commands provided by the user responsible for system management.

3.2.1 Resources

As for the window, we can interact with it through various operations:

1. **get_switch_handler**: Allows me to determine whether the window is open or closed.
2. **put_switch_handler**: Enables me to open or close the window.

3.3 Irrigation actuator

The *irrigation actuator* is specifically designed to manage the irrigation system based on data received from environmental sensors or manual commands initiated by the user overseeing the system.

3.3.1 Resources

Concerning irrigation, we can access it through various operations:

1. **get_switch_handler**: Allows me to determine whether irrigation is active or not.
2. **put_switch_handler**: Enables me to activate or deactivate the irrigation.

3.4 LED Management in Actuators

3.4.1 Air Conditioner Actuator

The following paragraph illustrates how LEDs are used to indicate different operational states in the air conditioner actuator.

1. **Connection State:** During the network connection process, the red LED flashes to indicate that the device is attempting to establish a connection. Once the connection is successfully established, the red LED remains steadily lit.
2. **Registration State:** When the device attempts to register with the CoAP server, the green LED flashes. This flashing is visible until the registration is successfully completed. Once registered, the green LED remains steadily lit.

3. **Mode State:** Pressing the button allows you to change the operating mode of the air conditioner. Each mode is associated with a different LED color: red for heating, red and blue for heating and humidification, blue for humidification, and green for ventilation.

4. **Conditioner Activation State:** The alteration of the resource's state occurs by holding down the button when the resource is turned off or on, thus transitioning from one state to another.

3.4.2 Irrigation Actuator

The irrigation actuator system utilizes a series of LEDs to indicate different operational states. This section provides an overview of the various conditions and their corresponding light representations.

1. **Connection State:** During the network connection process, the red LED flashes to indicate that the device is attempting to establish a connection. Once the connection is successfully established, the red LED remains steadily lit.

2. **Registration State:** When the device attempts to register with the CoAP server, the green LED flashes. This flashing is visible until the registration is successfully completed. Once registered, the green LED remains steadily lit.

3. **Irrigation Activation State:** By clicking the button, the irrigation switch state is changed, and simultaneously, the LED is set to blue if the irrigation is activated; otherwise, it is turned off.

3.4.3 Window Actuator

The following paragraph illustrates how LEDs are used to indicate different operational states in the window actuator.

1. **Connection State:** During the network connection process, the red LED flashes to indicate that the device is attempting to establish a connection. Once the connection is successfully established, the red LED remains steadily lit.

2. **Registration State:** When the device attempts to register with the CoAP server, the green LED flashes. This flashing is visible until the registration is successfully completed. Once registered, the green LED remains steadily lit.

3. **Window Activation State:** By clicking the button, the window switch state is changed, and simultaneously, the LED is set to blue if the window is open; otherwise, it is turned off.

CHAPTER 4

Collector

The Collector serves as the central hub of our system, undertaking essential activities such as collecting data from MQTT and CoAP sensors, communicating with devices to execute actions, and storing all gathered data in a MySQL database. These data can then be visualized through Grafana, as outlined in the "Database and Data Visualization" section.

This chapter will delve into the specifics of each subcomponent of the Collector, including the MQTT side, CoAP side, database connection, data visualization, and the automated irrigation system.

4.1 Monitoring - Command Line Interface

The application provides the opportunity to interact directly with the system, allowing users to issue commands to both sensors and actuators. This functionality enables monitoring of environmental conditions and manual intervention when necessary. Below, all implemented commands are presented to streamline the user experience.

1. **!help <command>** - Displays details about a specific command.
2. **!get_conditioner_status** - Shows the status of the conditioner.
3. **!get_conditioner_switch** - Shows the switch status of the conditioner.
4. **!turn_on_heater <temperature> <fanSpeed>** - Activates the heater with specified temperature and fan speed.

5. **!turn_on_heater_humidifier <temperature> <fanSpeed> <humidity>** - Activates the heater-humidifier with specified temperature, fan speed, and humidity.
6. **!turn_on_humidifier <fanSpeed> <humidity>** - Activates the humidifier with specified fan speed and humidity.
7. **!turn_on_wind <fanSpeed>** - Activates the wind with specified fan speed.
8. **!turn_off_conditioner** - Turns off the conditioner.
9. **!get_window_switch_status** - Shows the switch status of the window.
10. **!turn_on_windows** - Opens the window.
11. **!turn_off_windows** - Closes the window.
12. **!get_irrigation_switch_status** - Shows the switch status of irrigation.
13. **!turn_on_irrigation <index>** - Turns on the specified irrigation actuator.
14. **!turn_off_irrigation <index>** - Turns off the specified irrigation actuator.
15. **!print_all_device** - Displays information about all actuator devices.
16. **!get_avg_soil_humidity <number>** - Retrieves the average soil humidity for each nodeId over the last specified number of seconds.
17. **!get_avg_temperature <number>** - Retrieves the average temperature over the last specified number of seconds.
18. **!get_avg_humidity <number>** - Retrieves the average humidity over the last specified number of seconds.
19. **!exit** - Terminates the program.

Before interacting with any actuator or sensor, it is advisable to check, by typing the command **!print_all_device**, whether the device with which we want to interact is present within the output given by the command. Only then should the user proceed; otherwise, the application might encounter issues.

4.2 MQTT Side

The Monitoring system operates as an MQTT client. Its primary responsibility is to receive periodic measurements related to air and soil humidity, as well as temperature, and subsequently provide access to these measurements through user commands. This functionality is executed through the publish/subscribe model. Upon establishing a connection with the MQTT broker (utilizing the Mosquitto process running locally).

The acquired measurements are systematically stored in the MySQL database, with each entry specifying the nodeId, the corresponding value, and the timestamp. This organized storage ensures a comprehensive record of the collected data.

Additionally, if an actuator is active, MQTT messages with the topics "condition" or "irrigation" are sent to force the change in temperature, humidity, and soil moisture, in order to achieve the desired values.

Importantly, the system is designed to be versatile, capable of accommodating any number of MQTT devices. This scalability feature allows for the transparent integration of numerous devices.

4.3 CoAP Side

The Monitoring system serves a dual role as both a CoAP client and a CoAP server. In its capacity as a server, the Monitoring incorporates a dedicated class known as RegistrationServer, an extension of CoapServer. The primary objective of this class is to facilitate the registration of CoAP nodes for the service. This registration mechanism enables the Monitoring to utilize the registered devices as clients, enabling the retrieval of their status and the execution of specified actions outlined in the "Command Line Interface" section and the "CoAP Network" chapter.

To enhance the modularity of the codebase, a decision was made to implement individualized classes for each device. Within each of these classes, both get and put methods were implemented to facilitate interaction with CoAP servers. The design choice ensures a more organized and maintainable code structure.

The system allows for the registration of an arbitrary number of actuators, although a specific configuration has been proposed. This includes one device for the air conditioner, one for the window, and two for the irrigation system as we have two sensors dedicated to soil humidity.

The scalability is achieved through the utilization of a List of CoapClient objects, allowing the system to adapt seamlessly to an expanding array of actuators.

4.4 Database Management

We have implemented a robust data management system using a MySQL database to store information from our sensors, facilitating actuator management. Sensor data from the broker to the collector is organized into specific tables. For sensor data insertion, we have created the following queries:

```
-- For temperature
INSERT INTO temperature (nodeId, value) VALUES (?, ?);

-- For humidity
INSERT INTO humidity (nodeId, value) VALUES (?, ?);

-- For soil humidity
INSERT INTO soilHumidity (nodeId, value) VALUES (?, ?);
```

To handle actuators, we developed specific queries to retrieve meaningful information:

```
-- For soil humidity
SELECT nodeId, AVG(value) AS average_humidity
FROM soilHumidity
WHERE timestamp >= NOW() - INTERVAL ? SECOND
GROUP BY nodeId;

-- For temperature
SELECT AVG(value) AS average_temperature
FROM temperature
WHERE timestamp >= NOW() - INTERVAL ? SECOND;

-- For ambient humidity
SELECT AVG(value) AS average_humidity
```

```
FROM humidity
WHERE timestamp >= NOW() - INTERVAL ? SECOND;
```

These queries enable dynamic and responsive management of actuators based on updated and time-averaged data. The database plays a central role in collecting, organizing, and retrieving information, contributing to the robustness and efficiency of our automation system.

4.5 Automatic Agricolture System

The **Automation** application represents an intelligent automation system designed to continuously monitor environmental conditions through an **IoT** network. Its main goal is to make real-time decisions based on data collected by sensors and dynamically apply control actions to actuators.

Firstly, the application connects to temperature, humidity, and soil humidity sensors via **MQTT** and **CoAP**, receiving periodic data. These data are then processed to assess current environmental conditions, with a focus on temperature, humidity, and soil humidity.

Depending on the detected conditions, the application takes a series of actions. For example, if humidity is low, it might activate the **heating** or **humidifier**. If the temperature is high, it might activate **ventilation** or open **windows**. Additionally, the application dynamically manages **irrigation** based on soil needs.

This entire process is designed to ensure an optimal environment for cultivation, continuously adapting to changes in environmental conditions. The application also leverages the modularity of the **IoT** network, allowing it to handle a potentially unlimited number of devices distributed in various zones.

In essence, the **Automation** application serves as the brain of the **IoT** system, making informed decisions to ensure effective resource management in an efficient and environmentally friendly manner.

Here is a logical overview of how automation has been implemented.

```
if (get_humidity <= low_humidity) {
    if (get_temperature <= low_temperature) {
        // Disable Window
        // Activate Heater and Humidifier
    }
    else if (low_temperature < get_temperature &&
       get_temperature < high_temperature) {
        // Activate Humidifier
        // Disable Window
    }
    else if (get_temperature >= high_temperature) {
        // Activate Humidifier and Window
    }
}
else if (low_humidity < get_humidity && get_humidity <
   high_humidity) {

    if (get_temperature <= low_temperature) {
        // Activate Heater
        // Disable Windows
    }
    else if (low_temperature < get_temperature &&
       get_temperature < high_temperature) {
        // Disable Air Conditioner
        // Disable Window
    }
    else if (get_temperature >= high_temperature) {
        // Activate Window and Wind
    }
}
else if (get_humidity >= high_humidity) {

    if (get_temperature <= low_temperature) {
        // Activate Heater
        // Disable Windows
    }
    else if (get_temperature >= low_temperature) {
            // Activate Wind and Window
        }
}

selectSoilHumidity(timeSec, nodeId, get_soil_humidity);

if (get_soil_humidity.get(j) <= low_soil_humidity) {
        // Activate Irrigation
} else {
        // Disable Irrigation
}
```

4.6 Data Visualization on Grafana

In this section, we will explore how Grafana is used to visualize the data stored in the database. The sensors used in our setup are of three types: one for temperature, one for air humidity, and two for soil humidity.

4.6.1 Temperature Sensor Dashboard

The temperature sensor dashboard on Grafana provides an overview of temperature variations over time. Figure 4.1 illustrates the temperature dashboard.

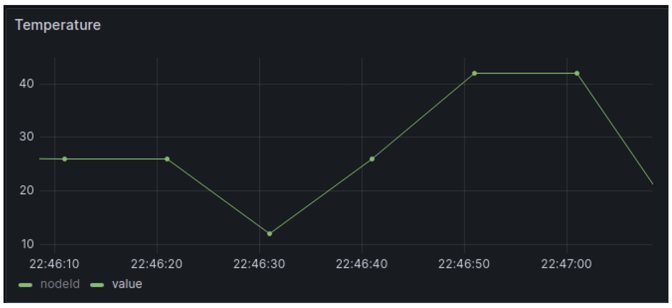


Figure 4.1: Temperature Sensor Dashboard

4.6.2 Air Humidity Sensor Dashboard

The air humidity sensor dashboard focuses on visualizing variations in air humidity levels. Figure 4.2 illustrates the air humidity dashboard.

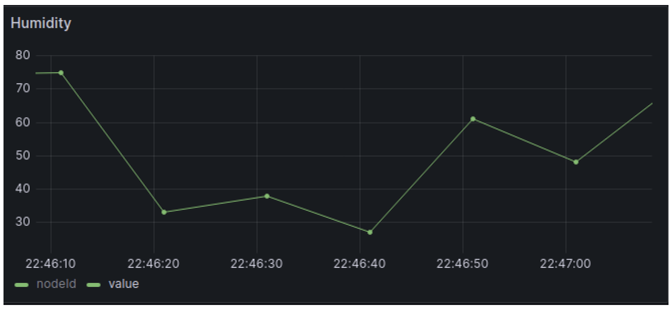


Figure 4.2: Air Humidity Sensor Dashboard

4.6.3 Soil Humidity Sensor Dashboard

The soil humidity sensor dashboard is more complex as it monitors 2 nodes. Figure 4.3 displays the trend of soil humidity values for both nodes, while the red line, when hovered over, indicates the correspondence of the value to the node.

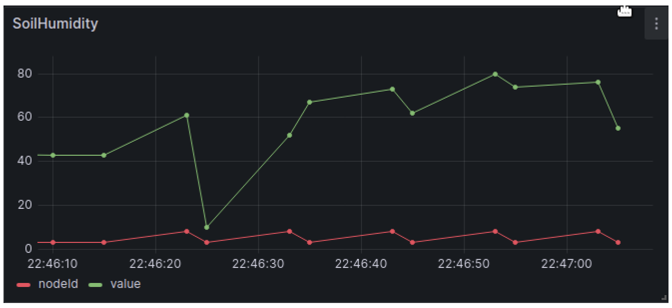
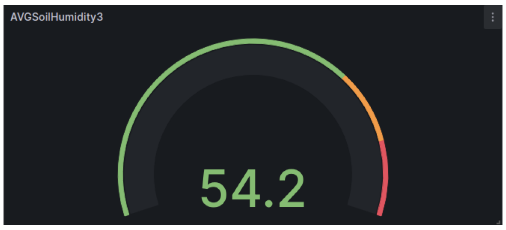
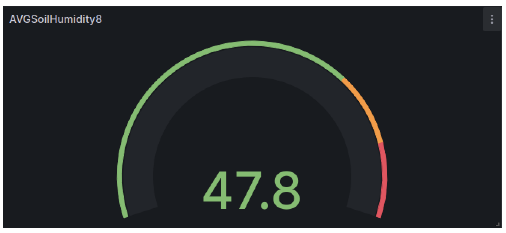


Figure 4.3: Soil Humidity Sensor Serial Dashboard

Two other dashboards for soil humidity (4.4a and Figure 4.4b) show the average values for each node.



(a) Soil Humidity Sensor Average Dashboard (Node 1)



(b) Soil Humidity Sensor Average Dashboard (Node 2)

Figure 4.4: Comparison of Soil Humidity Sensor Average Dashboards for Node 1 and Node 2