UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Internet of Things

IoT Smart Irrigation System

Project Documentation

TEAM MEMBERS:
Edoardo Fazzari
Mirco Ramo

Academic Year: 2020/2021

Contents

1 Introduction **2**
- 1.1 Proposed Solution . . . 2
- 1.2 Deployment Structure . . . 2
- 1.3 Implementation Choices . . . 3

2 CoAP Network **4**
- 2.1 Temperature Sensor . . . 4
  - 2.1.1 Resources . . . 4
  - 2.1.2 Data Generation . . . 4
- 2.2 Soil Moisture Sensor . . . 5
  - 2.2.1 Resources . . . 5
  - 2.2.2 Data Generation . . . 5
- 2.3 Rain Sensor . . . 6
  - 2.3.1 Resource . . . 6
  - 2.3.2 Data Generation . . . 6
- 2.4 Tap Actuator . . . 7
  - 2.4.1 Resources . . . 7

3 MQTT Network **8**
- 3.1 Devices . . . 8
- 3.2 Aquifer Level Detector . . . 8
  - 3.2.1 Topics . . . 8
  - 3.2.2 Data Generation . . . 8
- 3.3 Reservoir Level Detector and Actuator . . . 9
  - 3.3.1 Topics . . . 9
  - 3.3.2 Data Generation and Actuation Mechanism . . . 9

4 Collector **12**
- 4.1 Command Line Interface . . . 12
- 4.2 MQTT Side . . . 13
- 4.3 CoAP Side . . . 13
- 4.4 Database And Data Visualization . . . 13
- 4.5 Automatic Irrigation System . . . 15

1 — Introduction

Agriculture is one of the most fundamental resource of food production and also plays a vital role in keeping the economy running of every nation by contributing to the Gross Domestic Production. But there are several issues related to traditional methods of agriculture such as excessive wastage of water during field irrigation, dependency on non-renewable power source, time, money, human resource etc. Since every activity now a days is becoming smart, it needs to smartly develop agriculture sector for growth of country. Out project aims at developing a Smart Irrigation System using IoT Technology with the objective of automating the total job, providing adequate water required by crop by monitoring the moisture of soil and climate condition in order to prevent the wastage of water resources, resulting in many advantages for farmers. The irrigation at remote location from home will become easy and more comfortable. In addition, it will not only protect farmers from scorching heat and severe cold but also save their time otherwise required by the to-and-from journey to the field.

1.1 Proposed Solution

Modern agricultural methods propose an alternative way of water provisioning: instead of a daily/weekly irrigation, high-tech agricultural firms are moving towards slow, constant and lossless techniques that allow for water saving and a better quality of irrigation. In this optic, we decided to implement a smart system that is capable to compute real time the water need of the field and so to continuously adjust the water output that will be provisioned to plants. Of course, the success of this requires the water to be always available, but we all know that the seasons in which the need is higher are the ones with the lowest availability of natural water resources and vice versa: traditional irrigation methods tend to waste water abundance during wet periods and to overexploit aquifers during dry periods. In order to avoid that, the smart system makes use of a reservoir, i.e. a big container that collects water from the aquifer when the availability of it is much higher than the actual need, to be used as backup resource when aquifers are not enough. So, more precisely, the system uses provided sensors to determine the exact water output needed by plants; once this value has been computed, the system select from where to take the water and how to handle the reservoir, according to the water policy presented in the next indention. Finally it communicates output level and selected source to the tap actuator that handles the actual irrigation, and a new iteration will be performed very soon in order to guarantee the small but almost constant output. WATER POLICY: if the aquifer flow is enough to cover needs, the source is the aquifer and the water excess is stored in the reservoir unless it is full. If it rains, no artificial irrigation is needed, so we turn-off the system in order to save energy and to allow the aquifer to grow again. If the aquifer is not enough to cover needs, the water will be entirely fetched from the reservoir, in order not to drain the aquifer; of course if the reservoir runs out of water the aquifer will be needed again. If the level detection system goes down, the water is taken from the aquifer.

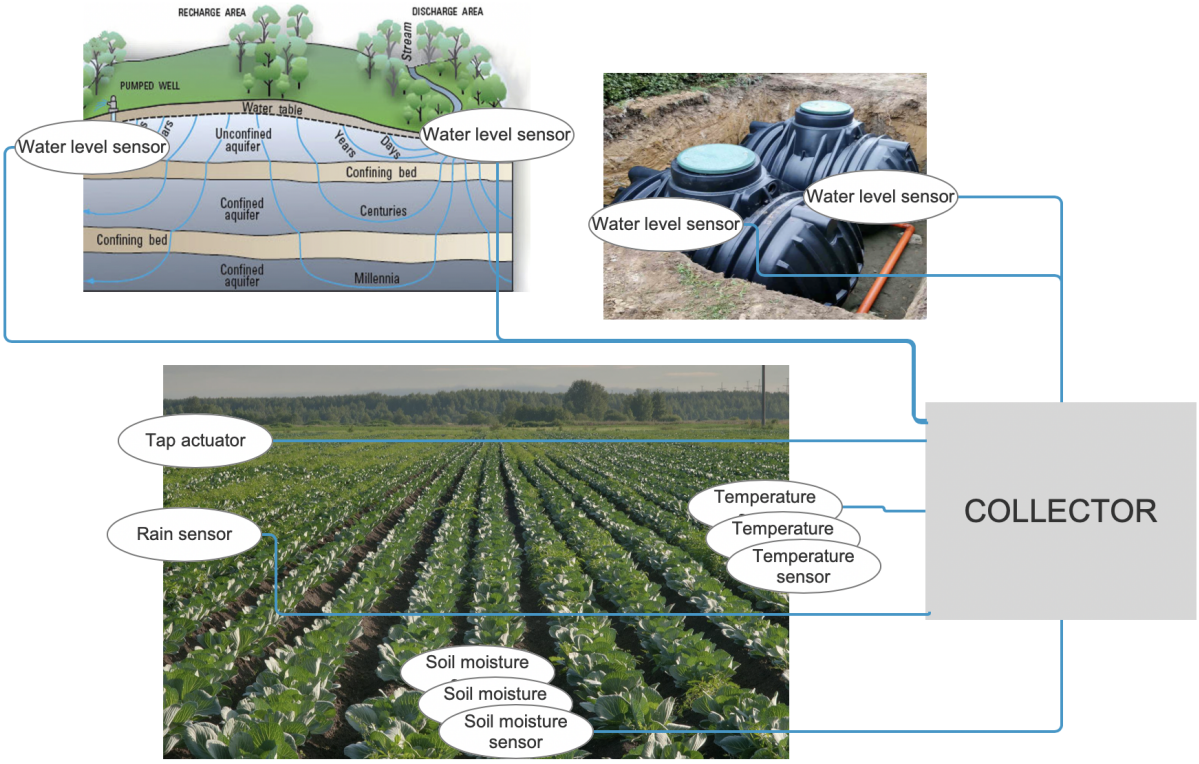
1.2 Deployment Structure

The objective of this project is to adopt a smart irrigation system to water cultivated fields making use of the local water resources, i.e. aquifers and reservoirs, in the way of using the water resources as more eco-friendly as possible (e.g., without disrupting aquifers). Thus, we can consider two different locations to takes care: the **field** and the **water provisioning site**,

For what concern the *water provisioning site*, it is composed of sensors which have the mean of monitoring the water level both of the aquifers and reservoirs. In this way, the system and the user can know where to take water. Although, a single device for source may be enough, we decided to ideally deploy multiple water level sensors in the same source in order to avoid

errors in the monitoring (e.g., in the case of a single device if it is detected that the reservoir is empty, but it is not, the irrigation system will use the water from the aquifer pointlessly). All water level sensors will make use of *MQTT* as explained in the "MQTT Network" chapter.

The *field site* is more articulated and will exploit multiple types of sensors and a single type of actuator. The actuator needed is the one capable of providing the water to the plants, which we called "tap actuator". This will be used in conjunction with the other sensors presents in the fields that will monitor the environment, specifically there will be temperature sensors, soil moisture sensors and a rain sensor. All these sensors and the tap actuator will exploit *CoAP* as explained in the "CoAP Network" chapter.



1.3 Implementation Choices

Before starting explaining the implementation of the application, it is important to list here some implementation choices that we made:

- We decided to implement the **Collector** and the *control logic* using Java.
- Regarding the *data encoding*, we opted for **JSON** since our devices are very constrained devices which can fail without putting at risk the system (e.g., the farmer can check the situation on site without any problem). Our application is not a critical application, thus a data encoding language such as XML is not needed. Anyway, the best solution would be CBOR, but because it is not mature on contiki we did not use it.

2 — CoAP Network

2.1 Temperature Sensor

The temperature sensor measures the local temperature in Celsius (at the Collector level will be given the possibility to display the results in Fahrenheit, see the Collector chapter for further details). The goal of this sensors is to quantify and schedule the water provisioning.

2.1.1 Resources

The temperature sensor exposes two resources: the *temperature_sensor* and the *temperature_switch* resources.

The **temperature_sensor** resource is an observable resource that provides to the clients the temperature acquired by the sensor. The resource not only provides the mere temperature to the clients, but it informs if the temperature is lower or greater than a certain threshold. Hence, the sensor exposes a *PUT* method, in order to set up the lower or the upper bound for the temperature.

The change of the bounds is done at step, the user will specify the threshold that he/she wants to change through the CLI: upper or lower. At the server side the request will be processed checking if the value arrived is consistent (e.g., the new value for the lower bound is not greater than the upper bound actual value), after those controls the parameter is updated.

The **temperature_switch** resource is connected to the *isActive* boolean variable, which indicates if the sensor is operating or not. This is done for turning off the temperature sensor when it is raining in order to save energy, since we do not perform any analysis for irrigating when the weather does that for us. For the reason that we want to change the status of the resources based on the rain sensor, it is implemented a *PUT* method for changing the value of the *isActive* variable.

2.1.2 Data Generation

Data is generated every $8 \times CLOCK\_SECOND$ in order to have a smooth simulation (i.e., without too many records in the Cooja's Log that let the simulation being difficult to understand). The value for the temperature is updated using the following algorithm:

```
static void temperature_event_handler(void)
{
    if (!isActive) {
        return; // DOES NOTHING SINCE IT IS TURNED OFF
    }

    // extimate new temperature
    srand(time(NULL));
    int new_temp = temperature;
    int random = rand() % 8; // generate 0, 1, 2, 3, 4, 5, 6, 7

    if (random <2) {// 25% of changing the value
        if (random == 0) // decrease
            new_temp -= VARIATION;
        else // increase
            new_temp += VARIATION;
    }

    // if not equal
    if (new_temp != temperature)
    {
        temperature = new_temp;
```

```
        coap_notify_observers(&temperature_sensor);
    }
}
```

2.2 Soil Moisture Sensor

Soil moisture sensors measure the water content in the soil and can be used to estimate the amount of stored water in the soil horizon. Soil moisture sensors do not measure water in the soil directly. Instead, they measure changes in some other soil property that is related to water content in a predictable way. Checking the different technologies used for measure soil moisture content, we decide to exploit the *soil water potential*[1].

2.2.1 Resources

The soil moisture sensor exposes two resources: the *soil_moisture_sensor* and the *soil_moisture_switch* resources.

The **soil_moisture_sensor** resource is an observable resource that provides to the clients the soil moisture tension acquired by the sensor. The resource not only provides the mere tension to the clients, but it informs if the value is lower or greater than a certain threshold. Hence, the sensor exposes a *PUT* method, in order to set up the lower or the upper bound for the tension[2].

The change of the bounds is done at step, the user will specify the threshold that he/she wants to change through the CLI: upper or lower. At the server side the request will be processed checking if the value arrived is consistent (e.g., the new value for the lower bound is not greater than the upper bound actual value), after those controls the parameter is updated.

The **soil_moisture_switch** resource is connected to the *isActive* boolean variable, which indicates if the sensor is operating or not. This is done for turning off the temperature sensor when it is raining in order to save energy, since we do not perform any analysis for irrigating when the weather does that for us. For the reason that we want to change the status of the resources based on the rain sensor, it is implemented a *PUT* method for changing the value of the *isActive* variable.

2.2.2 Data Generation

Data is generated every $8 \times CLOCK\_SECOND$ in order to have a smooth simulation. The value for the tension is updated using the following algorithm (the same of to the one used for the temperature):

```
static void soil_moisture_event_handler(void)
{
    if (!isActive) {
        return; // DOES NOTHING SINCE IT IS TURNED OFF
    }

    // extimate new tension
    srand(time(NULL));
    double new_soilTension = soilTension;
    int random = rand() % 8; // generate 0, 1, 2, 3, 4, 5, 6, 7

        if (random <2) {// 25% of changing the value
            if (random == 0) // decrease
```

[1] *Soil water potential* or *soil moisture tension* is a measurement of how tightly water clings to the soil and is expressed in units of pressure called bars. Generally, the drier the soil, the greater the soil water potential and the harder a plant must work to draw water from the soil.

[2] For the default range value we used the ones indicated here: https://www.metergroup.com/environment/articles/defining-water-potential/

```
                new_soilTension -= VARIATION;
            else // increase
                new_soilTension += VARIATION;
        }

    // if not equal
    if (new_soilTension != soilTension)
    {
        soilTension = new_soilTension;
        coap_notify_observers(&soil_moisture_sensor);
    }
}
```

2.3 Rain Sensor

Rain sensor or rain switch is a switching device activated by rainfall. It is used for water conservation since it is connected to the automatic irrigation system, which will cause the system to shut down in the event of rainfall in order to do not waste water and to reduce energy consumption.

2.3.1 Resource

The only resource provided by the rain sensor is a value indicating if it is raining or not, named **isRaining** and stored as a boolean. Since we are only interested when the status of the variable change, we opt to use the observable pattern provided by CoAP in order to minimize the number of interactions with the sensor.

The only possible action is the **GET** method, which will respond with a text saying *"raining"* or *"not raining"* based on the status of *isRaining*.

2.3.2 Data Generation

Data is generated every $50 \times CLOCK\_SECOND$ in order to have a slow simulation, helpful to see all the possible scenarios the our application can offer. Moreover, rain is a slow phenomenon, thus a higher detection frequency is not needed and would only result in a higher energy consumption. The value of **isRaining** flips (i.e., if it was indicating raining it turns to not raining, and vice versa) with a probability of 50%. This is done in the *rain_event_handler* function in the following way:

```
static void rain_event_handler(void)
{
    // check if raining
    srand(time(NULL));
    int random = rand() % 2; // generate 0, 1

    bool new_isRaining = isRaining;
    if (random == 0) // 50% of changing the value
        new_isRaining = !isRaining;

    // if not equal, notify
    if (new_isRaining != isRaining) {
        isRaining = new_isRaining;
        coap_notify_observers(&rain_sensor);
    }
}
```

In case the value changes, this is notified to all the subscribers.

2.4 Tap Actuator

The tap actuator is the device aim at irrigating the fields.

2.4.1 Resources

The tap actuator should be had four resources: the *tap_intensity*, the *tap_interval*, *tap_where_water* and the *tap_switch*. However, Cooja does not permit that, limiting the maximum number of resources to two. Thus, we joined the resources making just two: the *tap_intensity* (which include *tap_intensity*, *tap_where_water* and the *tap_switch*) and the *tap_interval*.

The **tap_intensity** resource is mainly related to the *intensity* variable, which determines the volume of water to provide at each simulation ($Volume = intensity * computedNeed$). Since we want to pull water from the aquifer or the reservoir, we made this resource observable in the way of better controlling the *water level sensors* in the simulation. The method implemented for this resource are the *GET* and *PUT* methods. The *GET* method respond to the user with a string containing the value of the intensity. On the other hand, the *PUT* method gives to the user the possibility to set up the intensity value updating the current one and to update where to take the water. The message received for the request is in the form of: *A/R* (indicating the aquifer or the reservoir), space, new value for the intensity.

Since we needed to include *tap_switch* in the resource, and since the *PUT* method was already implemented, we added a *POST* method for updating the switch status. This was the only possible solution for limiting the number of resources to two and we are full aware of the fact that the *POST* method should be used for creating new elements and not for updating variables, but we could not do otherwise.

The **tap_interval** resource indicates the period of time (expressed in *CLOCK_SECOND*) between two activations of the tap. This resource can be retrieved using the *GET* method associated to it, and it can be updated using the *PUT* method. The *PUT* method does not only update the value, but it updates also the intensity (in order not to change the output per second) based on the following formula.

$$\text{intensity} = \text{intensity} * \frac{\text{new\_interval\_value}}{\text{old\_interval\_value}} \tag{1}$$

3 — MQTT Network

3.1 Devices

The MQTT Network will be deployed in the water provisioning site and it is formed by 2 types of node: the Aquifer Level Detector and the Reservoirs Level Detector and Actuator. The role of those devices is to monitor the water level in the two sources, in order to always have it enough for the irrigation needs but without the spoil of natural resources. Each kind of device communicates the sensed levels to the Collector, which will compute the mean of the values to more precisely estimate the actual aquifer and reservoir level.

3.2 Aquifer Level Detector

The Aquifer Level Detector senses the level of water in the Aquifer, in order to estimate the availability. This device senses the height of the water flow in cm and based on that computes the volume of water that is available at each iteration, known the dimensions of the aquifer and the speed of the flow. The user can retrieve at any time the last measurement (the system will do it automatically) and the sensing interval is the same of the **tap_interval**.

3.2.1 Topics

This device is subscribed to the **interval** topic used to change the sensing interval (to adapt it to the tap_interval) and publishes measurements on the **aquifer_level** MQTT topic. The device acts as a MQTT client and handles its connection with the broker.

3.2.2 Data Generation

Data is generated every *interval*, every time the system must decide from where to fetch water. The value for the aquifer sensor is updated according to the following idea: during rainy seasons, the level will be enough to cover the average water need, on the contrary during summer there can be risk of insufficient water coverage. The simulated value is then published as a MQTT message on the **aquifer_level** topic.

```
#define WATER_SPEED 0.0005    /* 0.0005cm/s   https://www.arpa.vda.it/it/acqua/acque-sotterranee/cosa-sono-le-acque-sotterranee */
#define SECTION 200             //2m
#define MAX_LEVEL 60           //60cm

static double simulate_level(){
    boolean summer = false;
    time_t t = time(NULL);
    struct tm tm = *localtime(&t);
    int month = tm.tm_mon;
    if (month >=5 && month<8)  //between June and August
        summer = true;
    srand(time(NULL));
    double availability;  //   cm^3
    if (summer)
        availability = rand()%MEDIUM_NEED;
    else
        availability = MEDIUM_NEED + rand()%(VERY_HIGH_NEED - MEDIUM_NEED);

    //Assuming rectangular aquifer, available water is given by LEVEL * SECTION * WATER_SPEED * INTERVAL
    double level = ((availability/WATER_SPEED)/SECTION)/PUBLISH_INTERVAL;   //cm
    return level<MAX_LEVEL ? level : MAX_LEVEL;
```

```
}

sensed_level = simulate_level();
sprintf(pub_topic, "aquifer_level");
  //Assuming rectangular aquifer, available water is given by LEVEL * SECTION * WATER_SPEED * INTERVAL
available = sensed_level*SECTION*WATER_SPEED*(PUBLISH_INTERVAL/CLOCK_SECOND);
sprintf(app_buffer, "{\"node\": %d, \"aquifer_availability\": %.2f, \"unit\": \"cm^3\"}", node_id, available);
mqtt_publish(&conn, NULL, pub_topic, (uint8_t *)app_buffer, strlen(app_buffer), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
```

3.3 Reservoir Level Detector and Actuator

The Reservoir Level Detector senses the level of water in the Reservoir, in order to estimate the availability. This device senses the height of the water storage in cm and based on that computes the volume of water that is available at each iteration, known the dimensions of the reservoir. The user can retrieve at any time the last measurement (the system will do it automatically) and the sensing interval is the same of the **tap_interval**. The Reservoir Actuator instead puts or fetches water according to the decisions taken by the controller: if there is water abundance the actuator is asked to store the excess, when water lacks it is asked to fetch from the reservoir.

3.3.1 Topics

This device is subscribed to the **reservoir** topic and the controller can exploit it to change the sensing interval (to adapt it to the tap_interval) or to ask to fetch/store water. Reservoir devices publishes measurements on the **reservoir_level** MQTT topic. The device acts as a MQTT client and handles its connection with the broker.

3.3.2 Data Generation and Actuation Mechanism

Data is generated every *interval*, every time the system must decide from where to fetch water. The value for the reservoir level could be determined mathematically, but the deployment of a sensor allows to deal with possible phenomena like evaporation or breaks. The simulated value however is generated leaving out those phenomena, so it is always equal to the level that was present during the last actuation. The value is then published as a MQTT message on the **aquifer_level** topic. The actuator instead receives commands from the controller through MQTT subscription and adds/removes the quantity of water indicated by the controller.

```
//assuming rectangular reservoirs of capacity 1000 litres
#define WIDTH 200             //2m = 200cm
#define DEPTH 100               //1m = 100cm
#define RES_MAX_LEVEL 50           //50 cm
//Total capacity is 200*100*50 = 1e6 cm^3 = 1000 l

static int sensed_level = RES_MAX_LEVEL/2;
static int capacity = RES_MAX_LEVEL*WIDTH*DEPTH;

/*   The following code is just a simulation of the output of a level sensor and the corresponding actuator that
     puts or fetches the water from the reservoir*/

static int simulate_level(){
    return sensed_level;
}

static int get_capacity(){
    return capacity;
```

```
}

static void put_get_water(int quantity){
//assuming rectangular reservoir
    capacity += quantity;
    sensed_level = capacity/WIDTH/DEPTH;
    if (sensed_level>RES_MAX_LEVEL){
        sensed_level = RES_MAX_LEVEL;
    }
    else if (sensed_level<0){
        sensed_level = 0;
    }
    if (capacity>RES_MAX_LEVEL*WIDTH*DEPTH){
        capacity = RES_MAX_LEVEL*WIDTH*DEPTH;
    }
    else if (capacity<0){
        capacity = 0;
    }
}


LOG_INFO("I try to publish a message\n");
level=simulate_level();
sprintf(pub_topic, "%s", "reservoir_level");
available = get_capacity();
sprintf(app_buffer, "{\"node\": %d, \"reservoir_availability\": %i, \"unit\":
    \"cm^3\"}", node_id, available);
mqtt_publish(&conn, NULL, pub_topic, (uint8_t *)app_buffer, strlen(app_buffer),
     MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
STATE_MACHINE_PERIODIC = PUBLISH_INTERVAL;

static void pub_handler(const char *topic, uint16_t topic_len, const uint8_t *
    chunk, uint16_t chunk_len)
{
  LOG_INFO("Message received: topic='%s' (len=%u), chunk_len=%u\n", topic,
    topic_len, chunk_len);
    if(strcmp(topic, "reservoir") == 0) {
        const char* message = (const char*)chunk;
        char command = message[0];
        char argument[20];
        for (int i = 1, j=0; true; i++, j++){
            argument[j] = message[i];
            if (message[i] == '\0')
                break;
        }
        if (command == 'i'){
            printf("Changing detection interval to: ");
            long interval = atol(argument);
            if (interval<1)
                interval=1;
            printf("%ld\n", interval);
            PUBLISH_INTERVAL = interval*CLOCK_SECOND;
        }
        else if (command == 'l'){
            printf("Received a set_reservoir_level topic command\n");
            int quantity = atoi(argument);
            printf("Changing reservoir water level by: %d\n", quantity);
            put_get_water(quantity);
        }
        else
            printf("Unrecognised command\n");
      }
      else {
```

```
78            LOG_ERR("Topic not recognized!\n");
79        }
80
81    }
```

4 — Collector

The *Collector* is the main component of our system. The Collector performs different activities: it collects the data form both MQTT and CoAP sensors; it communicates with the devices in order to perform actions; and it stores all the data collected in *MySQL* in the way they can be visualized through **Grafana** as indicated in the "Database And Data Visualization" paragraph.

In this chapter, all the subcomponents of the Collector will be explained, which are: MQTT side, CoAP side, DB connection, Data Visualization, and Automation Irrigation System.

4.1 Command Line Interface

The *Collector* exposes a **Command Line Interface**, through which the user can interact with the system. The commands are printed on the terminal when the application starts and they are the following (otherwise indicated the commands are related to CoAP node):

- *getDevicesList*: show list of all available sensors (both CoAP and MQTT).
- *getTemp*: get the last temperature registered.
- *setTemp* $< l/u >< value >$: set desired temperature for specified bound (l for lower, u for upper). Value is an integer expressed in Celsius/Fahrenheit.
- *setUnit* $< F/C >$: change temperature measure unit in C (Celsius) or F (Fahrenheit).
- *getWeather*: get if the rain sensor feels rain or not.
- *getSoilTension*: get the last soil tension registered.
- *setSoilTension* $< l/u >< value >$: set desired tension for specified bound (l for lower, u for upper).
- *getTapInterval*: get the interval which the tap operates.
- *getTapIntensity*: get the intensity which the tap operates.
- *setTapInterval* $< seconds >$: set the interval which the tap operates in seconds (acts also on MQTT nodes).
- *setTapIntensity* $< value >$: set the intensity which the tap operates (values is a double).
- *getWaterLevels*: print the water levels of aquifer and reservoir (MQTT network related).
- *start*: start the automatic irrigation system (Java Thread).
- *stop*: stop the automatic irrigation system (Java Thread).
- *help*: print the commands list.
- *quit*: quit the program.

4.2 MQTT Side

The Collector works as MQTT client: its role consist on receiving periodic measurements about the water levels and make them available to user commands. It is also in charge of sending commands to the reservoir actuator. This mechanism is implemented through the publish/subscribe model: once the Collector is connected with the MQTT broker (Mosquitto process running on user@iot.dii.unipi.it), it starts publishing commands and receiving the updated measurements. In order to enhance the modularity, 2 separate classes handle respectively the communication with the aquifer and with the reservoir, according to topics specified above. Collected measurements are stored into a $< nodeId, value >$ Hashmap, and the actual level is taken as the mean of the levels.

The system can potentially handle any number of MQTT devices.

4.3 CoAP Side

The Collector plays the roles of both CoAP client and CoAP server. As far as server functionality is concerned, the Collector has a class (called RegistrationServer, which extends CoapServer) which has the task of allowing CoAP nodes to register for the service. In this way, the Collector can use the registered devices acting as a client in order to obtain the data generated by them and also perform actions as indicated in the "Command Line Interface" paragraph and in the "CoAP Network" chapter.

To ensure greater modularity to the code, it was decided to implement dedicated classes for each device, the implementation details are left out here. What is important to underline, however, is the number of devices that can be registered in the system: we thought that in our environment it is possible to insert one or more temperature sensors, one or more soil sensors, a rain sensor and a single tap actuator. The Collector is able to scale autonomously as the temperature and soil sensors number increase thanks to the use of a *List of CoapClient* objects assigned for each resource exposed by the sensors.

4.4 Database And Data Visualization

Since we need to save the data obtained, we have implemented a class called *IrrigationSystemDbManager* dedicated to communicating with MySQL to store the data obtained from the devices every time they change. In fact, each device as indicated in the "CoAP Network" chapter has an observable resource, which generate get requests that are captured by our Collector that will call a dedicated function in the *IrrigationSystemDbManager* class to store the data properly.

The data that are saved in the DB are the following (here the sql code to generate the tables):

```
CREATE TABLE IF NOT EXISTS `iot_irrigation_system`.`rain` (
  `idrain` INT NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `isRaining` TINYINT NOT NULL,
  PRIMARY KEY (`idrain`))
ENGINE = InnoDB;


CREATE TABLE IF NOT EXISTS `iot_irrigation_system`.`soilMoisture` (
  `idsoilMoisture` INT NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `soilValue` DOUBLE NOT NULL,
  PRIMARY KEY (`idsoilMoisture`))
```

```
ENGINE = InnoDB;


CREATE TABLE IF NOT EXISTS `iot_irrigation_system`.`tap` (
  `idtap` INT NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `intensity` DOUBLE NOT NULL,
  `interval` INT NOT NULL,
  PRIMARY KEY (`idtap`))
ENGINE = InnoDB;


CREATE TABLE IF NOT EXISTS `iot_irrigation_system`.`temperature` (
  `idtemperature` INT NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `tempValue` TINYINT(1) NOT NULL,
  PRIMARY KEY (`idtemperature`))
ENGINE = InnoDB;


CREATE TABLE IF NOT EXISTS `iot_irrigation_system`.`waterLevelAquifer` (
  `idwaterLevel` INT NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `nodeId` VARCHAR(10) NOT NULL,
  `waterLevel` DOUBLE NOT NULL,
  PRIMARY KEY (`idwaterLevel`))
ENGINE = InnoDB;


CREATE TABLE IF NOT EXISTS `iot_irrigation_system`.`waterLevelReservoir` (
  `idwaterLevel` INT NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `nodeId` VARCHAR(10) NOT NULL,
  `waterLevel` DOUBLE NOT NULL,
  PRIMARY KEY (`idwaterLevel`))
ENGINE = InnoDB
```

Talking about **Data Visualization** we used *Grafana* for displaying most of the information presented above[3]. An example of what we plot is shown in the following figure:

[3]We didn't plot the tap interval, since it can be obtained by subtracting two adjacent point in the tap intensity graph.

4.5 Automatic Irrigation System

The Automatic Irrigation System is the hearth of the proposed solution: it is in charge of periodically computing the water need based on data reported by the CoAP sensors and selecting the water source based on the measurements of the MQTT devices. The Automatic Irrigation System is a parallel Java Thread that can be started or stopped at any time by the user. At every iteration the system:

- Collects into a Parameters object all the retrieved data needed to estimate the water need
- Computes the need based on the following formula:

```
if(p.isRaining){
  Logger.log("[Irrigation System]: It's Raining, no irrigation is needed");
  continue;
}

switch(p.soilStatus){
   case TOO_LOW:
       level = Levels.LOW;  //low soil moisture means wet soil, so little
    water is needed
       break;
   case NORMAL:
       level = Levels.MEDIUM;
       break;
   default:
       level = Levels.HIGH;
}
if (p.temperatureStatus == BoundStatus.TOO_HIGH)
     level = level.increaseLevel();
```

- Determines the actual tap output, given by *level*tapIntensity*.
- Determines the water source, based on the water policy reported in the first paragraph and implemented as follows.

```
if (need>p.aquiferLevel)
    WhereWater=RESERVOIR;
WhereWater=AQUIFER;
```

- Sends a command to the reservoir to fetch/store water

```
if (source == RESERVOIR){
     rc.changeReservoirLevel(0-quantity);
     Logger.log("\tFetched "+quantity + " from the RESERVOIR");
}
else{
     rc.changeReservoirLevel(p.aquiferLevel-quantity);
     Logger.log("\tFetched "+p.aquiferLevel+"cm^3 from the aquifer");
     Logger.log("\t" + quantity + "cm^3 of them are output of the tap,");
     Logger.log("\t" + (p.aquiferLevel-quantity) + "cm^3 of them are stored
     in the reservoir");
}
```