UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Internet of Things

Project documentation

TEAM MEMBERS:
Fabiano Pilia

Contents

Chapter 1

Introduction

1.1 Aim of the project

A smart home is a home environment in which appliances and devices can be automatically controlled remotely from anywhere with an Internet connection using a mobile phone or other connected device. The devices in a smart home are connected to each other via the Internet and allow the user to remotely control functions such as home security access, temperature or lighting.

This is the context of this project, in fact the aim is to create an **intelligent aquarium** that maintains the values of substances in the water at an optimal value for the life of the living creatures inside (aquatic plants and fishes). Keeping these values constant is of crucial importance for the life inside the aquarium, because abrupt fluctuations in these values could lead to diseases or even death for the species. These values are monitored by sensors inside the aquarium and are altered by the release of substances into the water, such as CO2 or osmotic water. The aquarium provides also the possibility to keep the temperature of the water at a certain value since it's very important for the internal ecosystem.

1.2 Monitored values

This section will present the main values to be taken into account when assessing the health of the aquarium, thus the values to be monitored by the sensors, and how to act to alter the values to keep them at an optimal level. The parameters considered refer to a freshwater aquarium.

1.2.1 Ph

The **pH** measures the acidity or basicity of a solution. It has *no units of measurement* and is presented on a scale from 0 to 14. When its values are between 0 and 7 it measures the acidity, while we refers to the basicity of water when these are between 7 and 14. A value of 7 is considered neutral.

The optimum pH value is **6.75**, with an acceptable range between **6.5** and **7**, monitored through a **pH sensor**.

To alter the pH value there are various methods, the one adopted in this project is to release **carbon dioxide** via a dispenser inside the aquarium, but due to the close relationship between pH, kH and Co2 it is only necessary to act when the kH value is stable, otherwise the pH value varies abruptly leading to death of the organisms.

1.2.2 Kh

The **Kh** represents the calcium carbonate concentration which expresses the amount of carbonates and bicarbonates, usually characterised by Calcium and Magnesium in water. The Kh value has a **buffering property** that stabilises the pH by preventing acid or basic changes, and is also used to calculate the CO2 concentration in relation to the pH.

An **high** kH value has more **stability**, but *does not allow* pH changes, while **low** kH values are associated with **instability** that *easily allows* pH changes, but also brings **fluctuations** in the pH value that are very dangerous for the aquarium's ecosystem.

The optimum value is around **5 °** (values between 4.5 ° and 5.5 ° are considered good), monitored through a kH sensor. When the value of the kH is optimum it is possible to modify the pH without incurring in abruptly fluctuations.
In order to stabilize the kH value it is necessary to release **osmotic water** inside the aquarium, which has the characteristic of having the kH equal to **0**.
The osmotic water is contained inside a tank.

1.2.3 Temperature

Another value to take in consideration is the **temperature** inside the aquarium, in order to do this a **smart termometer** is used and to keep it constant is used a **ventilation system** to cool down the water and an **aquarium heater** to increase the temperature.
Normally the temperature must stay around **25 °C**, so values between **24 °C** and **26 °C**, but different values of temperature can be used depending on the type of fish that the aquarium contains (e.g. tropical fishes or freshwater fishes).
The temperature must stay **constant** since the values of CO2 to be released inside the aquarium is related to it.

1.2.4 CO2 and its relation with the temperature, pH and kH

The amount of CO2 inside the water can be obtained through the following formula:

$$\mathbf{CO2[mg/l]} = \mathbf{15,69692} \times \mathbf{kH} \times \mathbf{10^{(pK_a - pH)}}$$

- **15,69692** is a coefficient obtained considering the chemical relation between the CO2 and the substances related to the kH level.
- $\mathbf{pK_a}$ is **temperature-dependent** and calculated as follows:

$$((3404,71/(\mathbf{T} + 273.15)) + (0,032786 \times (\mathbf{T} + 273.15) - 14,8435)$$

  The +273.15 is used to convert the temperature in Kelvin, since the formula requires this measure.

As we can see it's possible to compute the concentration of CO2 in the water by observing the values of **pH**, **kH** and **temperature**.
There isn't the need to monitor the level of CO2 through a sensor, since are sufficent the values of kH, pH and temperature to evaluate it.

When the level of kH and the temperature are constant and inside the desired interval, it's possible to modify the *pH* value using a cylinder filled with carbon dioxide (CO2) that is injected with **small bubbles**. The **more** bubbles are injected, the **lower** the pH. The aim is to find the right amount of bubbles per minute to create a balance and make the pH constant.
To increase the pH value is sufficent to stop or reduce the flow of CO2 and eventually it will increase, since the gas will leave the water, due to the water movement induced by the filtering system.
In conclusion to dispense CO2 through bubbles is used a cylinder that can be regulated based on the values monitored by the sensor.

1.2.5 Intervals of CO2 using the relation with pH, kH and temperature

In the following table we can see the values of CO2 required to keep the pH at a certain value after the stabilization of the kH, in the first case when the temperature is fixed at 25 °C (figure 1.1), that is the base case considered in this project, at 30 °C (figure 1.2) and 18 °C (figure 1.3) to see how the temperature change the values required. The table is filled using the formula mentioned in the previous subsection.

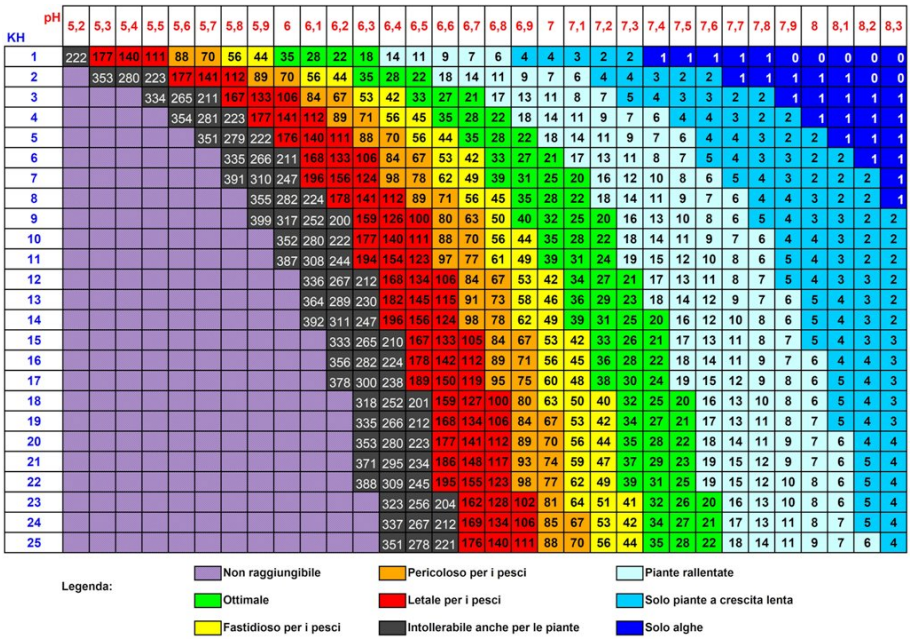

Figure 1.1: CO2 table at 25 °C. used in the majority of aquarium

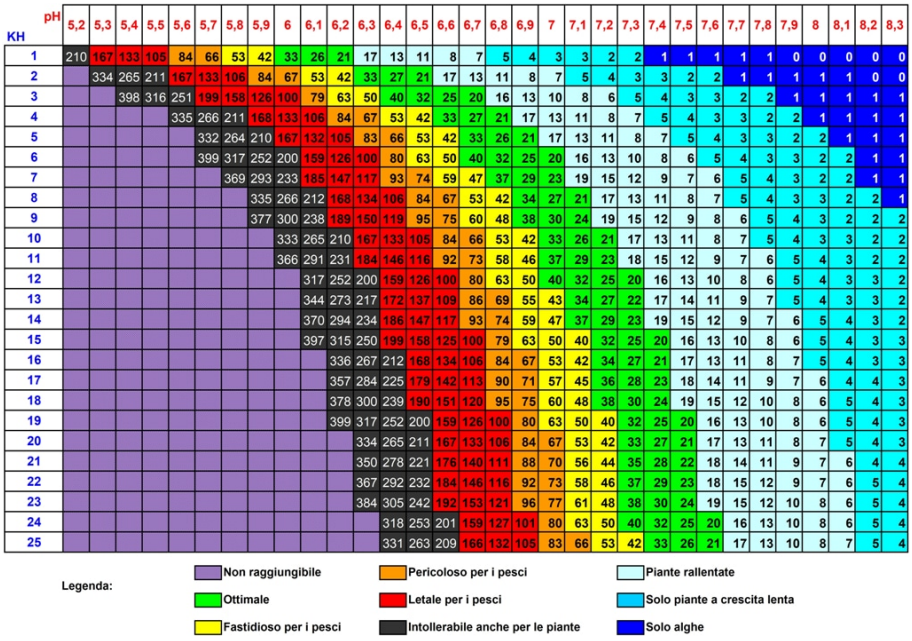

Figure 1.2: CO2 table at 30 °C, useful during summer

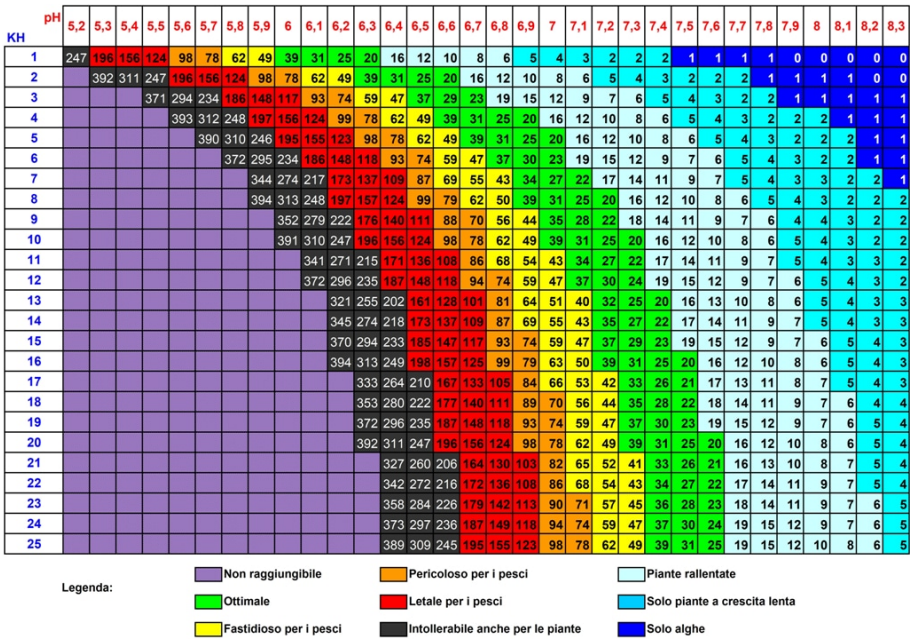

Figure 1.3: CO2 table at 18 °C, useful for example for goldfishes

1.3 Summary of intervals of pH, kH and temperature

The following table shows the ranges within which the values must lie, in the case of a freshwater aquarium; through the configuration file is possible to set the temperature since different fishes require different temperatures, but for fishes that live in the same type of water the level of pH and kH are the same. Remember that the level of CO2 can be obtained through the values of pH, kH and the temperature inside the aquarium.

	min	**opt**	**max**	ε
pH	6.5	6.75	7	0.12
kH (°)	4.5	5	5.5	0.12
temperature (°C)	24	25	26	0.5

Table 1.1: Intervals of values that are optimal for the life in a frehswater aquarium.

Chapter 2

Architecture

In this chapter, the architecture of the system will be presented, highlighting the interactions between the various components and their functionalities.
The overall architecture can be seen in figure 2.1.

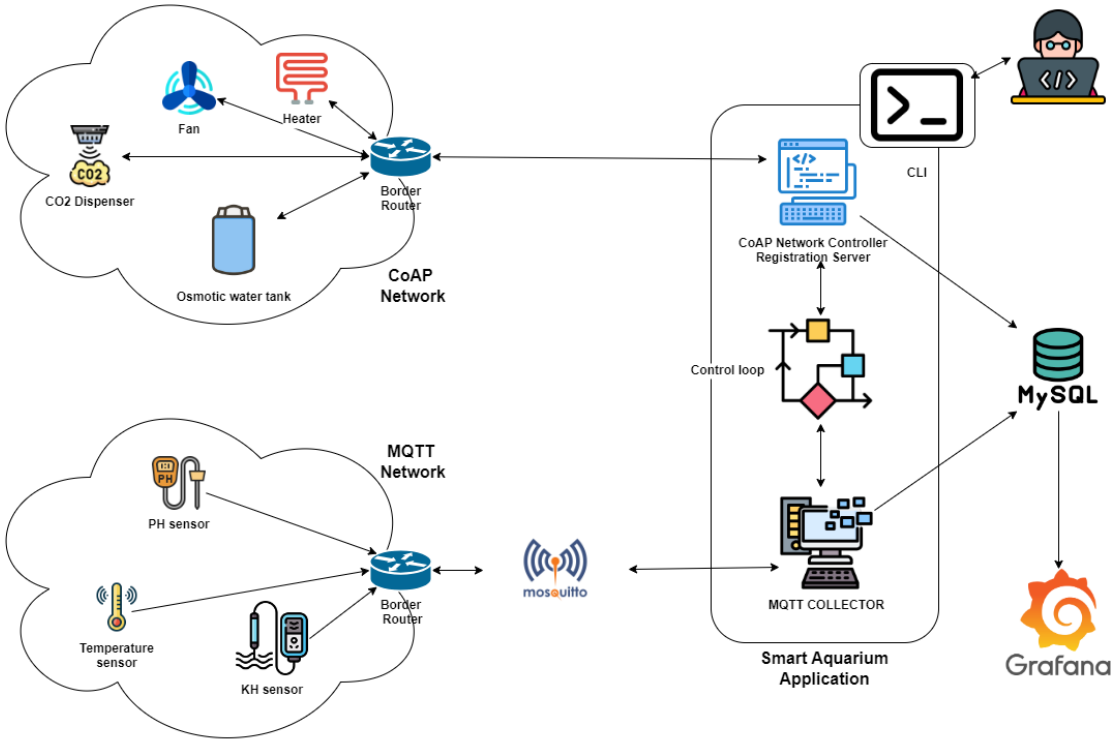

Figure 2.1: Architecture of the system

As shown in the image, the architecture consists of three main blocks:

- **MQTT** Network
- **CoAP** Network
- Smart Aquarium Application

2.1 MQTT Network

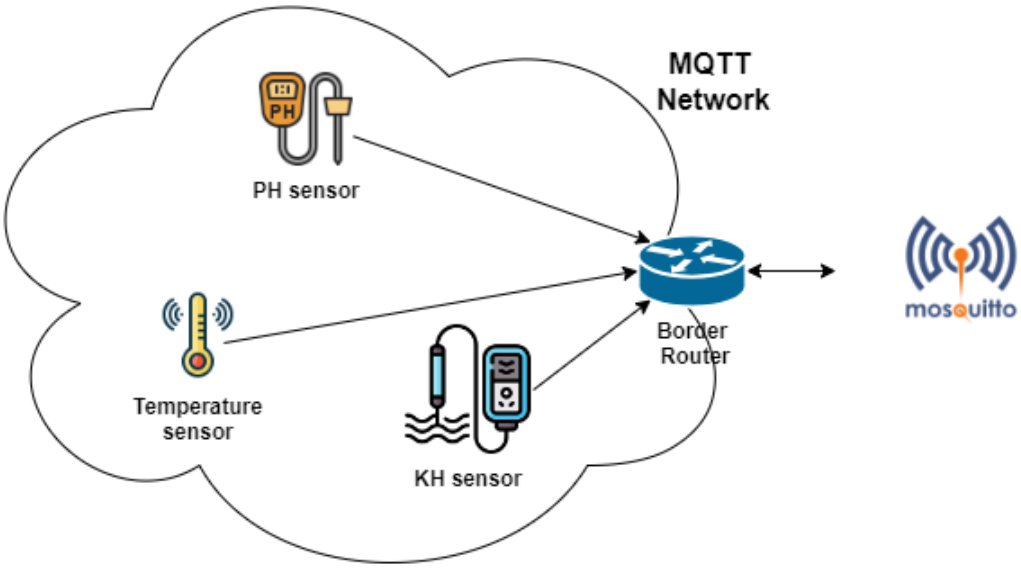

Figure 2.2: MQTT network

The first block of the architecture is the **MQTT network** composed by the following sensors:

- **PH device:** it measures the *pH* value of the water inside the aquarium tank;
- **KH device:** it measures the *kH* of the water;
- **Temperature device:** measures the temperature of the aquarium;
- **Border router**.

Each device is an *MQTT client* that follows the steps in figure 2.3 when it is started: it checks if there is the connectivity with the **border router**, then it connects to the **MQTT Broker**, waits for the *connected event* and finally it subscribes to the **simulation topics** in order to implement the simulation in the correct way (this aspect will be better explained in the next chapters).
After the **start up** phase the sensor enters its **operative** phase that varies depending on the sensor type.

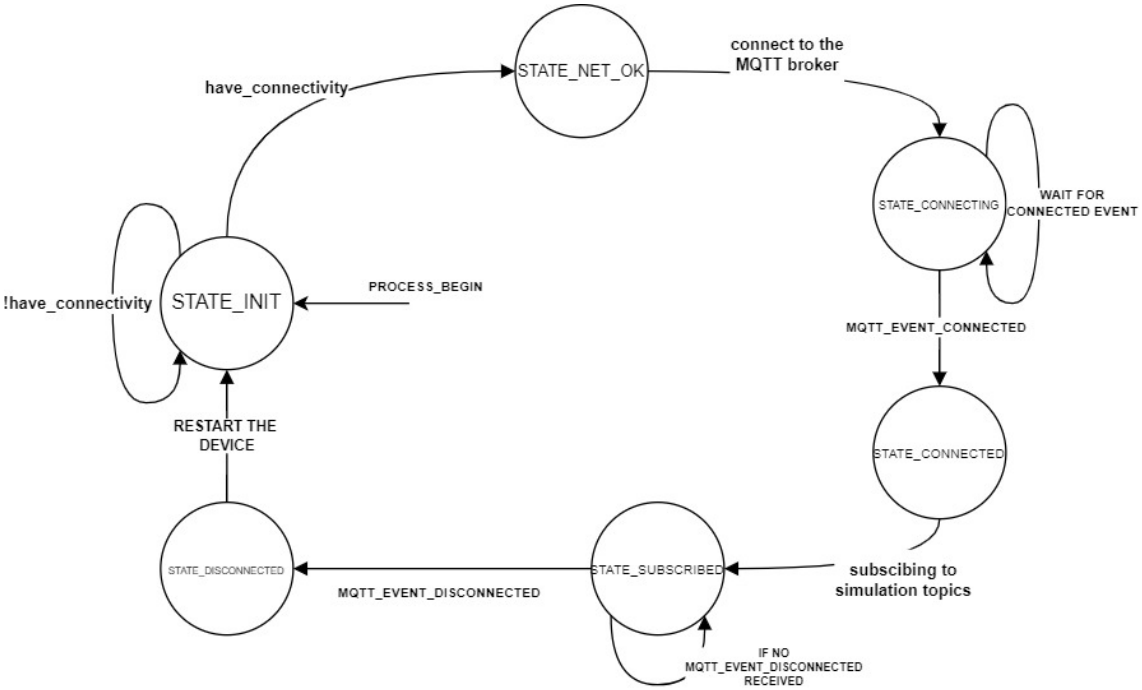

Figure 2.3: State of a MQTT device

2.1.1 PH device

This sensor checks the **pH** value inside the aquarium tank and every **15 minutes** it publishes the current value sensed on the topic "**pH**".
It publishes a JSON message with the following format: **{"pH":float }** where *float* is the current value simulated by the sensor.

Simulation

In order to implement correctly the simulation, the pH device is registered to the "**co2Dispenser**" topic. The application publishes commands that are exploited to change the value of the pH during the simulation, that reflects the behavior of the **CO2** released by the dispenser.
There are **5** different commands and, accordingly, **5** different simulation of the pH variation:

- **OFF**: This is the default mode of the **pH simulation**, it implements a **random variation** of the pH value.
  More in detail it selects with the same probability (1:3) one of the following **3 actions**:
  - Do nothing, hence the pH value remains the same;
  - Decrease the pH value by a value **randomly** selected from the interval **[0.0, 0.05]**;

  - Increase the pH value by a value **randomly** selected from the interval **[0.0, 0.05]**.

  This mode is also used when no significant variation of the CO2 released is detected, so the pH must behave randomly. It is also used when the levels of pH are **around** the optimal value (this is detected on the server and the OFF command is sent).

- **SDEC**: Used when the CO2 is modified in order to **increase (0.05)** the pH level inside the aquarium. It stands for *soft decrease*, characterized by the fact that the new value of CO2 dispensed is not too different from the previous value of CO2.

- **SINC**: It's the opposite of the previous mode, it is used to **decrease (0.05)** in a soft manner the pH level.

- **DEC**: This mode is used when two subsequent variations of CO2 dispensed are up to a certain **threshold**, consequently the pH level is increased by **0.1**.

- **INC**: This mode is the opposite of the previous, in fact the pH value is decreased by **0.1**.

The **change_pH_simulation()** function is called every time a **new pH value** needs to be published on its relative topic, so it is checked the **current mode** and the new current value is generated accordingly.

2.1.2 KH device

This sensor checks the **kH** value inside the aquarium tank and every **15 minutes** it publishes the current value sensed on the topic "**kH**".
It publishes a JSON message with the following format: **{"kH":float }** where *float* is the current value simulated by the sensor.

Simulation

For simulation purposes the device is registered to the "**OsmoticWaterTank**" topic.
The application monitors the values published on the "kH" topic and when its value is **below** or **above**, respectively, the minimum or maximum acceptable values, then the **osmotic water flow** is activated; to implement the correct variation of kH during the flow of osmotic water, the application publishes the following **commands** on the simulation topic:

- **OFF**: The osmotic water is not flowing, so the behaviour of the variation of kH is random.
  More in detail it selects with the same probability (1:3) one of the following **3 actions**:

  - Do nothing, hence the kH value remains the same;
  - Decrease the kH value by a value **randomly** selected from the interval **[0.0, 0.1]**;
  - Increase the kH value by a value **randomly** selected from the interval **[0.0, 0.1]**.

- **INC**: The osmotic water erogation tries to **increase** the kH value, it is done to keep the kH inside the interval in which the **pH** can be modified.
  The variation is constant at each time interval and is equal to **0.2**.

- **DEC**: The osmotic water erogation tries to **decrease** the kH value, it is done to keep the kH inside the interval in which the **pH** can be modified.
  The variation is constant at each time interval and is equal to **0.2**.

These values are checked when a **new kH value** must be generated using the **change_kH_simulation()** function.

2.1.3 Temperature device

This sensor checks the **temperature** in *Celsius degrees* inside the aquarium tank and every **15 minutes** it publishes the current value sensed on the topic "**temperature**". During the **CONNECTED STATE** (see figure 2.3) it registers to two topics: this cause a slower behaviour with respect to the other devices during the startup phase.

Simulation

As anticipated the device is registered to two topics, "**fan**" and "**heater**", where are published commands by the application every time an action towards the *fan* or the *heater* is taken.
The commands received are the same for both topics:

- **on**: used to signal the activation of the **fan** or the **heater**. It is an alternative command, in fact if one of the two actuators was *on* before the reception of the *on* command for the other actuator, then the first is turned **off**.

- **off**: with the same behavior of the *on* command, it signals that the relative actuator is turned **off**.

When a **new value** of temperature has to be generated, then the simulation is activated based on the state of the actuators:

- Neither the fan nor the heater is active, so the device follows a random behavior and draws with the same probability one of the following decisions:
  - Do nothing, hence the temperature value remains the same;

- Decrease the temperature value by a value **randomly** selected from the interval **[0.0, 0.2]**;
- Increase the temperature value by a value **randomly** selected from the interval **[0.0, 0.2]**.

- **fan on**: decrease the temperature by **0.4** Celsius degrees;
- **heater on**: increase the temperature by **0.4** Celsius degrees.

2.2 CoAP Network

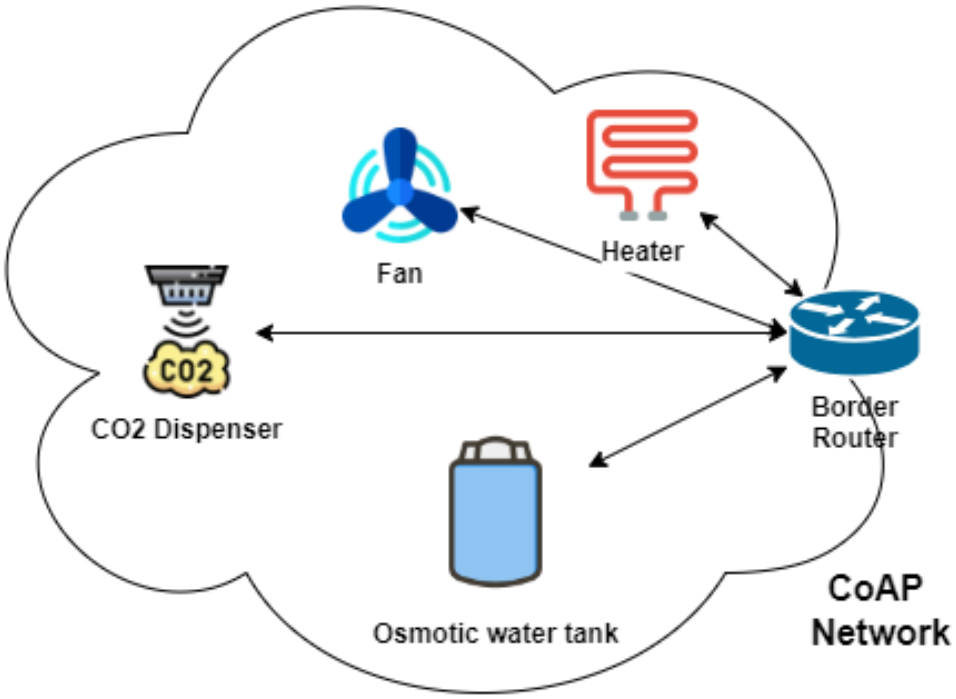

Figure 2.4: CoAP network

The second building block of the architecture is **CoAP Network**. It is composed by different **actuators** that react to the command sent by the **smart aquarium application** in order to apply some modification, that will be better explained in the following subsections, to the **environment**.
All the actuators expose **resources** that are **observed** by the application. This aspect is important since some actuators needs **human maintenance**, and through the observing mechanism the user will be informed about the incumbent intervention.
During the **startup phase** the device follows the steps exposed in figure 2.5.

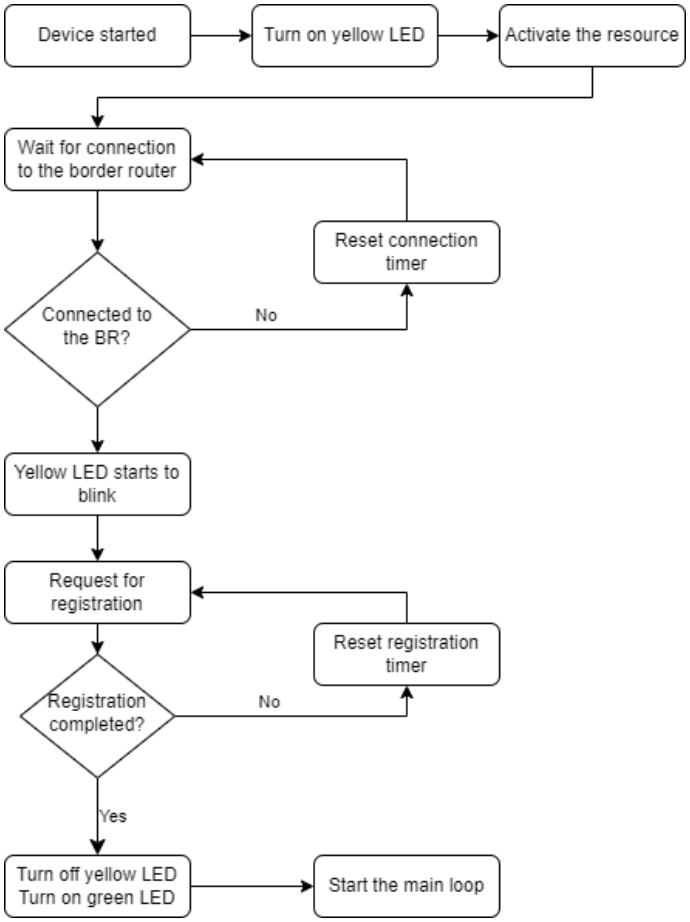

Figure 2.5: Steps followed by the CoAP devices during the startup phase

In the flow diagram can be seen the main steps that a generic CoAP device performs in order to become operative.
Initially it turns **on** the **yellow LED** in order to signal the fact that is *not yet connected* to the **border router**, activate its resource (or resources) and waits for a defined number of seconds to perform a new check on the connection. If the connection is **detected**, then the **yellow led** starts to blink to signal it and the fact that the **registration process** has started.
Through a **blocking request**, inserting JSON message as payload with the format **{"device":device_name }** , the **registration server** is contacted.
The registrations server sends a response upon the reception of a registration message: if there is already a device registered, then it ignores the message, since in the aquarium is present only one actuator per type; while if the **registration server** was waiting for the registration of the device, it sends a response with the message "**registered**".
In the latter case the device **turns off** the *yellow LED* and **turns on** the *green LED* to signal that the **startup phase** has been completed successfully.
Once the registration is completed, the device starts its main loop that varies depending on the device type.

The **CoAP Network** is composed by the following devices:

- **CO2 Dispenser**: it manages the release of CO2 inside the aquarium and exposes a **tank** resource to keep track of the CO2 tank level.
- **Osmotic water tank**: it manages the flow of **osmotic water** inside the aquarium and exposes a **tank** resource to keep track of the water left in the tank.
- **Temperature controller**: it manages two resources, the **fan** and the **heater**.
- **Border router**.

2.2.1 CO2 Dispenser

The CO2 dispenser main loop has **3** main aims:

- Check if the **CoAP Network Manager** has sent a **stop** message, in order to turn off the device and deactivate its resource.
  The message is a **CoAP DELETE** request, handled by the resource that sets a **flag** to signal that the process must break the main loop.
- Wait for a *timer event* to trigger a **periodic event** to the resource, that in fact is a **event resource** (the behaviour of the resource will be better explained later).
- Wait for a *button periodic event*, in fact, if the **tank is almost empty**, the device **turns on** the red LED to signal that the CO2 tank must be re-filled.
  When someone has changed or re-filled the tank then, to signal it to the device, the **button** must be pressed for **5 seconds**.
  During this period the red LED blinks and at the end it is turned off, the green LED is restored and an event is triggered on the resource in order to notify to the observer that the flow of CO2 is activated.

As anticipated, there is a periodic event that cause the **tank level** to be reduced by a certain quantity and the new level must be **sent** to the observer. All these aspects are handled by the **tank resource**. Notice that the flow of CO2 is **always active**.

Tank resource

The tank resource manages the aspects related to the **CO2 tank**, it is reachable by issuing CoAP requests to the **co2Dispenser/tank** endpoint.
At the beginning it initialize the CO2 level at **7000 ml** (number decided to test all the aspects of the resource during the simulation), as minimum level to trigger the need of re-filling **400 ml** (number decided in order to not go under 0 ml during the

simulation) and the CO2 to be dispensed is set to 0, since it must be decided by the smart application according to the pH, kH and temperature values.
The resource handles the following requests and events:

- **DELETE**: set the **stop** flag to *true*;
- **GET**: send a CoAP response containing the JSON message

  {"level":tank_level, "mode":mode }

  Where *level* is the **current tank level** and *mode* can be either **on** or **off**.
- **PUT**: there are two types of request that can be received depending on its attributes:
  - attribute **mode**: if the value received is **on**, then the flow of CO2 is activated (this is done to start the aquarium at the beginning, once the application has received for the first time all the values measured by the sensors and is able to compute the current CO2 to be dispensed);
    if the value received is **off**, the the flow is stopped, done when the aquarium must be turned off.
  - attribute **value**: a **new value** of CO2 to be dispensed is sent by the **smart application**.
  - If none of the previous attribute is present, then a **bad request** response will be sent.
- An **event** is triggered: reduce the tank level by the current CO2 value; if the new value is **below** the minimum threshold, then the red LED is turned on and sets the flag to signal that the tank must be re-filled; finally it notifies all the observers.

2.2.2 Osmotic water tank

The main loop of the Osmotic water tank is the **same** of the CO2 dispenser.
What changes between the two devices is the fact that the CO2 dispenser works with an higher degree of autonomy since the flow of gas is always active, while the *osmotic water tank* is continuously managed by the **smart aquarium application**, in order to keep the **kH** value inside the desired interval.
Another difference is that the amount of water released at each interval is **fixed** and is released if and only if the application has activated the water flow previously.

Tank resource

The tank resource manages the aspects related to the **Osmotic water tank**, it is reachable by issuing CoAP requests to the **OsmoticWaterTank/tank** endpoint.

The initial value of water inside the tank is **5000 ml**, the minimum is **100 ml** that is decided accordingly to the fixed amount of water released during the flow (**50 ml**), in fact is equal to two times it, so if a variation is lost, than the next on will trigger the signal to re-fill the tank, without going to values lower than 0.
The resource handles the same requests as the tank resource of the *CO2 dispenser*, but with some modification:

- **PUT**: only the **mode** attribute is present;
- An **event** is triggered: the trigger has effects only when the flow of osmotic water is active.

2.2.3 Temperature controller

The temperature controller doesn't have a main loop, but essentially, is used to **connect** to the border router, to **register** to the registration server and to activate the **fan resource** and the **heater resource** reachable, respectively, at the endpoints **temperature/fan** and **temperature/heater**.

Fan resource and heater resource

Differently from the previous two resources, the fan and heater ones are not **event resources**. They behaves in the same manner, responding to commands sent by the **smart aquarium application** aimed at changing the status of the actuators. The requests handled by the resources are the following:

- **GET**: sends a CoAP response containing the JSON message

  {"mode":mode }

  Where *mode* can be either **on** or **off**.

- **PUT**: accepts requests with the attribute **mode**, that can be either **on** or **off**, used to start or to stop one of the two devices. In case of requests with different attributes or not well formulated a **bad request** response is sent.

2.3 Smart aquarium application

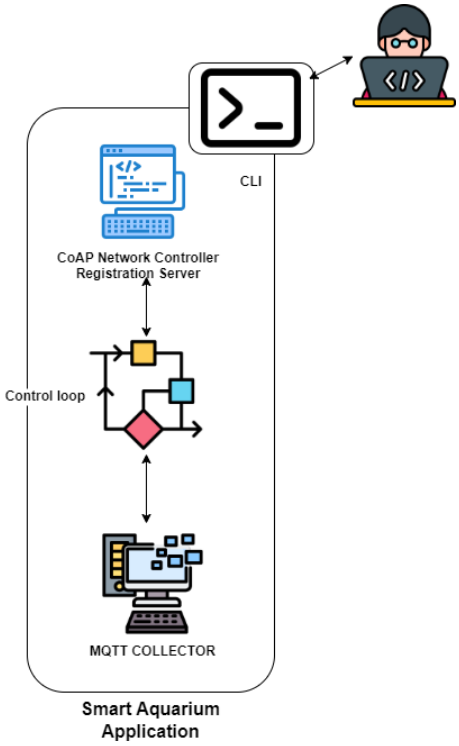

Figure 2.6: Smart aquarium application

The third block that compose the overall system is the **smart aquarium application**, written in **Java**. It is composed by different modules that manage aspects such as the commands sent to the actuators, the observation of the CoAP resources or the storing of the devices status on the database.

2.3.1 Smart application Main Class

It retrieves the configuration parameters from the configuration file, starts the database manager, the MQTT Collector, the CoAP Network Controller (that acts as a registration server to handle the devices on the CoAP network, provides different CoAP clients to interact with the actuators and finally it starts the observers for the resources provided by the devices), starts the thread to check the sensors data and to interact with the actuators and starts a loop to interact with the user receiving commands from the console.

2.3.2 Configuration

The configuration parameters are written to an XML file and are validated by means of its XSD schema, in order to define how the XML document must be written and

to ensure that all the parameters are present.
The file contains all the parameters needed during the life cycle of the application, such as the **names** ot the database tables, the **upper and lower bounds** for the values to implement the *control logic* or the **topics** to publish/retrieve data.
When the application is launched all the parameters are printed on the terminal.

2.3.3 CoaP Network Controller and Registration Server

The CoAP Network Controller is a **CoAP server**, that exposes a **registration resource**.
The devices that want to register to the application have to send a **POST** request to the *resource* containing a **JSON object** with the name of the device associated to the key "**device**".
If the device is in the list of the expected devices and **no other** devices of the same type are registered then it creates:

- a **CoAP client** associated to the device that behaves differently according to the device type.
- an **observe relation** with the resource (or resources) exposed by the device that has requested the registration.

To signal the correct registration it sends back to the device a response containing "**registered**" as payload, otherwise a **bad request** message is sent.

Since the behaviour varies across the different types of devices, to be more clear, is useful to explain it individually for each device.

Osmotic water tank CoAP client and observing behaviour

This class store the current **status** of the osmotic water tank: the **tank level** and the **active flag**.
At the creation it bonds to the endpoint on which the *tank resource* is reachable, read from the configuration files.

It offers the following method to interact with the associated resource:

- **Activate flow**: issues a **PUT** request with **mode=on** as payload and through a **CoAP Handler** manages the response of the device, if it is a *success*, then sets to **true** the **active flag** otherwise is kept to false.
- **Stop flow**: issues a **PUT** request with **mode=off** as payload and through a **CoAP Handler** manages the response of the device, if it is a *success*, then sets to **false** the **active flag** otherwise is kept to true.
- **stop**: sends a **stop flow** request and deletes itself.

The **observe relation** is established after the creation of the CoAP client, through a *CoAP handler* it manages the following response coming from the observed *tank* resource:

{"level":tank_level, "mode":mode }

It firstly reads the "**mode**" key and updates the **active flag** accordingly and then reads the "**level**" key and updates the **tank level** using the float value.
After that it inserts the **status** of the osmotic water tank (only the *tank level*, since when a new value is obtained its *mode* is obviously *on*) into the **database**.

In the following image is possible to see how the *messages* are exchanged during the registration phase and during the observation of the resource with the hypothesis that the CoAP Client has activated the water flow (for clarity this step is not present on the image).

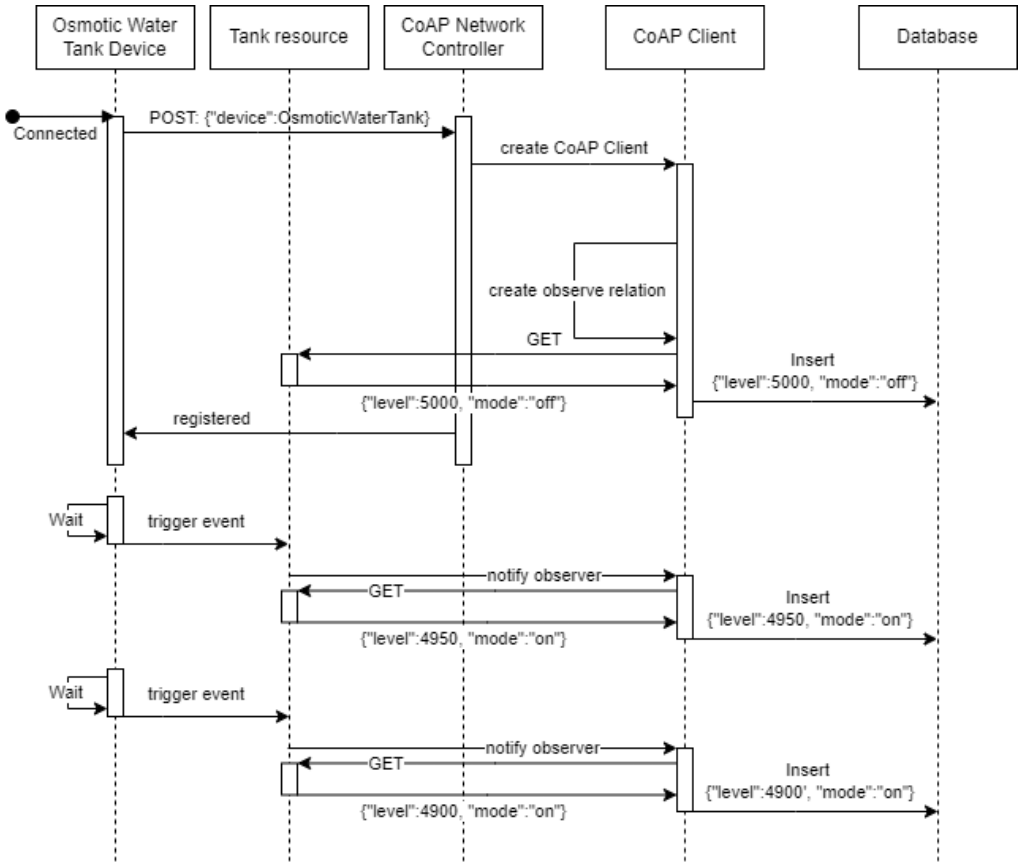

Figure 2.7: Example of messages exchanged between the different components of the application and the osmotic water device.

CO2 CoAP client and observing behaviour

The CO2 Dispenser CoAP Client takes care of the current status of the *CO2 dispenser* device and its *tank resource*.
The status is composed by the **tank level**, the **active flag** (at the beginning the

dispenser must be activated), **current variation** between the current and previous CO2 values and the **current CO2 value**.

At the creation it bonds to the endpoint on which the tank resource is reachable and sets the **current CO2 value** according to the *optimal values* of pH, kH and temperature read from the *configuration file*.

It offers the same **three** functionalities of the previous CoAP Client seen, but additionally it offers the following ones:

- **Compute new CO2**: given the value of pH, kH and temperature as parameters it sets the *current CO2* computed using the formula anticipated in the first chapter.
- **Set CO2 dispensed**: issues a **PUT** request with **value=currentValue** as payload and through a **CoAP Handler** manages the response of the device.
- **Start device**: computes the new level of CO2 to be dispensed and sends it to the device, then activate the flow. This method must be used by the application before starting the *control loop*.

For what concerns the **tank resource**, there's no difference in the setting up of the *observe relation*.
Every time a new status has been received the **current value** of CO2 dispensed and the **tank level** are written into the database.

Temperature controller client

The temperature controller is used to manage **two** different *CoAP Clients*, the first used to interact with the **fan resource** and the other to interact with the **heater resource**.
When the object is created, it instantiates the two clients bounded respectively to the **temperature/fan** endpoint and to the **temperature/heater**, sets their status to **no active** and inserts it into the database.
It offers the following method to interact with the associated resources:

- **Activate fan**: issues a **PUT** request with **mode=on** as payload and through a **CoAP Handler** manages the response of the device, if it is a *success*, then sets to **true** the **active flag** and insert the new status into the database, otherwise is kept to false.
- **Activate heater**: issues a **PUT** request with **mode=on** as payload and through a **CoAP Handler** manages the response of the device, if it is a *success*, then sets to **true** the **active flag** and insert the new status into the database, otherwise is kept to false.

- **Stop fan**: issues a **PUT** request with **mode=off** as payload and through a **CoAP Handler** manages the response of the device, if it is a *success*, then sets to **false** the **active flag** and insert the new status into the database, otherwise is kept to true.
- **Stop heater**: issues a **PUT** request with **mode=off** as payload and through a **CoAP Handler** manages the response of the device, if it is a *success*, then sets to **false** the **active flag** and insert the new status into the database, otherwise is kept to true.
- **Stop**: stops the fan and the heater and deletes itself.

For this device and its resources **no observe relation** are defined since the status is decided by the *smart aquarium application*.

Closing the CoAP Network Manager

When the **close** method is called, then all the CoAP Client are stopped (stop method of the clients), the *observe relation* are cancelled and, finally, the CoAP Server destroys itself.

2.3.4 MQTT Collector

This class is used to handle the interaction between the **MQTT-based** devices and the **smart aquarium application**. It *subscribes* to the topics in which the sensors will publish their values; It manages the interaction with the database inserting the received values in the correct tables and manages the published messages in order to implement the simulation of the values of the sensors in the correct way.
It keeps track of the *current* values of the different measures and manages the flag to signal that a new value has been published on the relative topic.
When the object is created it **connects** to the **MQTT broker** and it subscribes to the **pH**, **kH** and **temperature** topics.

The functionalities provided by the *MQTT collector* are the following:

- **Read new messages** published on the subscribed topics. When a new message is present, then it is read, parsed from the JSON format to the *Java* class for the sample and finally the retrieved data is **stored** on the **database**.
- **Simulation**: it publishes the commands on the desired topics to be read by the devices to implement the simulation in the correct way.
- **Close**: set the flag to manage the concurrency, this is mandatory since the MQTT collector is used by the **Thread** that runs the *control loop* and by the **CLI**, in this way using an **atomic boolean** and checking it before each operation is possible to know when the MQTT has been closed.

After that it **unsubsribes** itself from all the topics, disconnects from the MQTT broker and close the client.

The *MQTT broker* is implemented using **Mosquitto**, an open source message broker, deployed on the **virtual machine**.

2.3.5 Control loop

Class extending **Thread**, that implements a control loop. The main function of this class is to manage the different devices in order to provide a **safe environment** for the tank life. It *periodically* checks the different values retrieved by the sensors and, when it's needed, it sends commands to the **actuators** aimed at balancing the values in order to keep them inside the safe intervals. In the diagram in figure 2.8 is possible to see the *actions* performed at each *loop*.

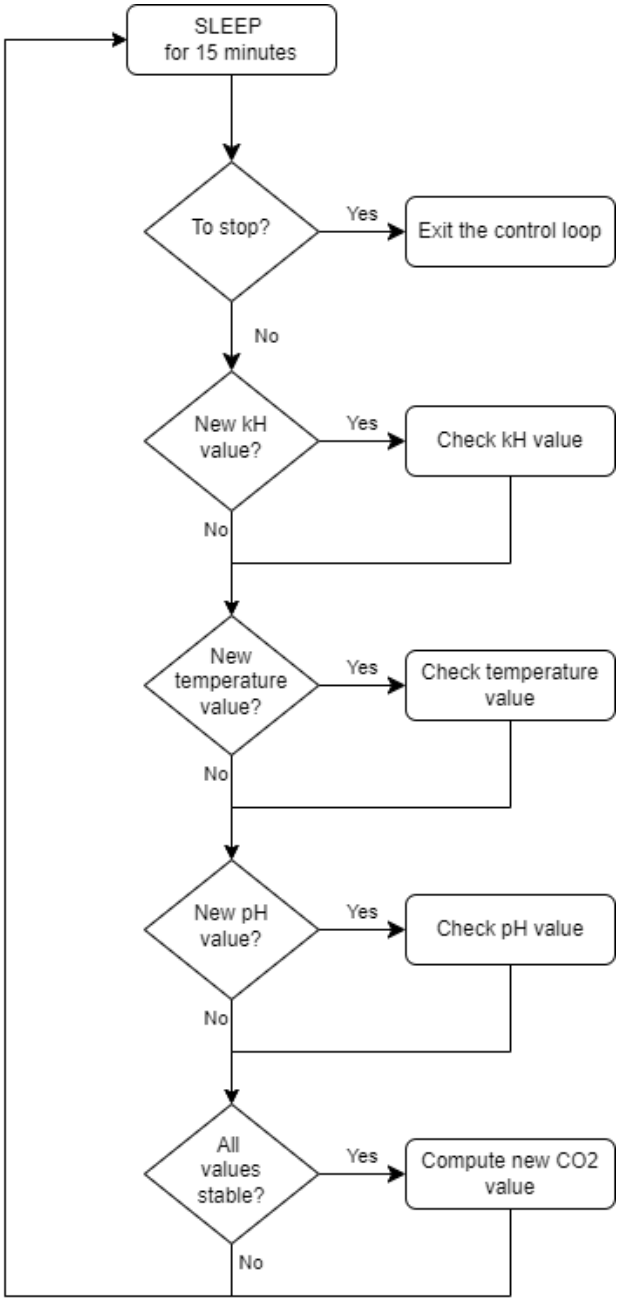

Figure 2.8: Control loop

The first step is to sleep for **15 minutes** in order to synchronize with the devices, (they publish new values every 15 minutes); then it checks if a command to stop the loop has been received, if the flag is set to true then it breaks the loop, otherwise the check of all the values can start.
The **kH value** is checked against the following conditions:

- **kH** $<$ **LB** and **no flow**: in this case the **osmotic water** flow is **activated** in order to **increment** the value of kH inside the aquarium and, accordingly, an **INC** command is published on its topic to change the simulation behaviour.

- **kH** $>$ **UB** and **no flow**: in this case the **osmotic water** flow is **activated** in order to **decrement** the value of kH inside the aquarium and, accordingly, a **DEC** command is published on its topic to change the simulation behaviour.

- **kH** $\in [\bar{\textbf{opt}} - \varepsilon, \bar{\textbf{opt}} + \varepsilon]$ and **flow active**: if consequently to the activation of the **osmotic water** flow the *kH* is inside an interval around its optimal value, then the osmotic water flow is **stopped** and the command **OFF** is published for the simulation.
  In order to not loose the $\pm\varepsilon$ interval around the optimum during the increment/decrement of the kH value, then the epsilon must satisfy the following relation: $\mathbf{2}\varepsilon >$ **maxVariation**.

The **temperature value** is checked against the following conditions:

- **temperature** $<$ **LB** and **heater not active**: if **fan** is active, then it is **turned off**; after that the **heater** is **turned on** to increase the temperature inside the tank and, accordingly, an **on** command is published on the *heater* topic to change the simulation behaviour of the temperature controller.

- **temperature** $>$ **UB** and **fan not active**: if **heater** is active, then it is **turned off**; after that the **fan** is **turned on** to decrease the temperature inside the tank and, accordingly, an **on** command is published on the *fan* topic to change the simulation behaviour of the temperature controller.

- **temperature** $\in [\bar{\textbf{opt}} - \varepsilon, \bar{\textbf{opt}} + \varepsilon]$ and the **fan** or the **heater** is **active**: if consequently to the activation of one of the **devices** the *temperature* is inside an interval around its optimal value, then the **active device** is **stopped** and the command **off** is published for the simulation in the relative topic (**fan** or **heater**).
  In order to not loose the $\pm\varepsilon$ interval around the optimum during the increment/decrement of the temperature value, then the epsilon must satisfy the following relation: $\mathbf{2}\varepsilon >$ **maxVariation** also in this case.

While the previous two checks are quite similar, the one regarding the **pH** value is slightly different due to the fact that the pH value can be modified **if and only if** the *kH* and the *temperature* values are inside their **desired intervals**.
Another complication is the fact that the **pH** value variation is **strongly dependant** on the **current CO2 level** dispensed.

- **pH** $<$ **LB** and **the kH and the temperature are stable**: in this case the following steps are performed:

  - **Compute the new CO2 value** based on the current values of all the three measures and send the new value to the CO2 dispenser.
  - $\left|\mathbf{CO2^{t+1}} - \mathbf{CO2^{t}}\right| <$ **threshold**: send a simulation command to the **temperature controller** to simulate a *soft* **increase** of temperature.

  - $\left|\mathbf{CO2^{t+1}} - \mathbf{CO2^{t}}\right| \geq$ **threshold**: send a simulation command to the **temperature controller** to simulate the **increase** of temperature using an higher value then before.

- **pH** > **UB** and **the kH and the temperature are stable**: in this case the following steps are performed:
  - **Compute the new CO2 value** based on the current values of all the three measures and **send** the new value to the CO2 dispenser.
  - $\left|\mathbf{CO2^{t+1}} - \mathbf{CO2^{t}}\right| <$ **threshold**: send a simulation command to the **temperature controller** to simulate a *soft* **decrease** of temperature.
  - $\left|\mathbf{CO2^{t+1}} - \mathbf{CO2^{t}}\right| \geq$ **threshold**: send a simulation command to the **temperature controller** to simulate the **decrease** of temperature using an higher value then before.

- **pH** $\in [\bar{\mathbf{opt}} - \varepsilon, \bar{\mathbf{opt}} + \varepsilon]$: if thanks to the change of the *CO2 dispensed* the **pH** has reached an interval around its optimum value, then a simulation message is sent to the **pH device** in order to implement the simulation correctly.
  In order to not loose the $\pm\varepsilon$ interval around the optimum during the increment/decrement of the pH value, then the epsilon must satisfy the following relation: $\mathbf{2}\varepsilon >$ **maxVariation**.

The last step of the **main loop** is to check if **all the measures** are inside their *desired* intervals. If the condition is satisfied then a **new CO2 value** is computed.

2.3.6 Command line interface

The **fourth** module that composes the application is the **Command Line Interface (CLI)**.
After the overall system setup, the registration of the devices and the starting of the system, then all the **available commands** are printed on the terminal. The available commands are the following:

- **:get status**
- **:get temperature status**
- **:get pH status**
- **:get kH status**
- **:get osmotic water tank status**
- **:get CO2 dispenser status**
- **:get fan status**

- **:get heater status**
- **:get configuration**
- **:help**
- **:quit**

These commands enable the user to know the status of the system and the one of all its components at each time, without waiting for a new LOG on the terminal.

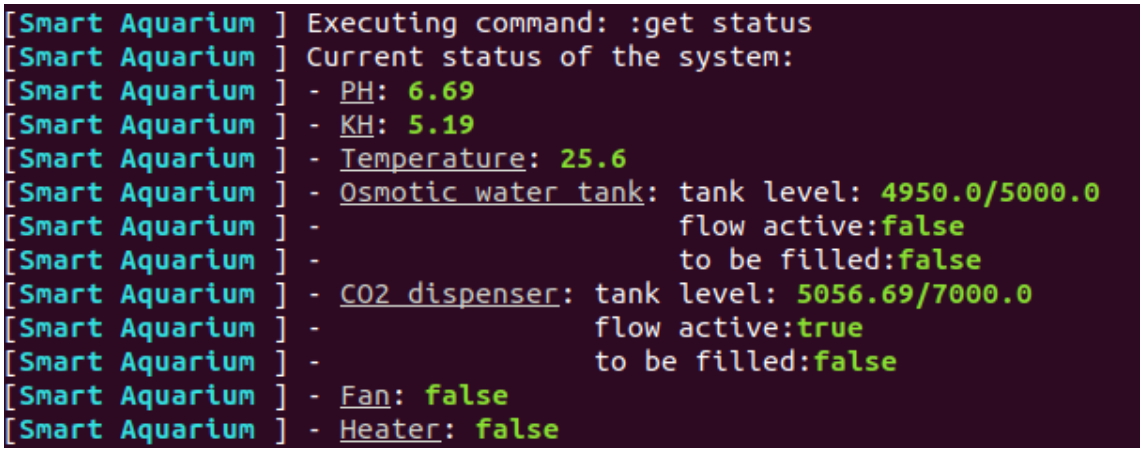

Figure 2.9: Example of the execution of a command

2.3.7 Logging

All the actions and events that happen during the life of the system are printed out on the terminal; different colors are used for each module. An example is depicted on the following figure 2.10;

```
[MQTT Collector ] Inserted {"pH":6.67} in PH.
[MQTT Collector ] Inserted {"temperature":25.4} in Temperature.
[MQTT Collector ] Inserted {"kH":5.58} in KH.
[Smart Aquarium ] Osmotic water tank [ mode = on ].
[MQTT Collector ] Inserted {"pH":6.64} in PH.
[MQTT Collector ] Inserted {"temperature":25.6} in Temperature.
[MQTT Collector ] Inserted {"kH":5.38} in KH.
[CoAP Controller] Inserted {"mode":"on","level":4700.0} in OsmoticWaterTank.
[MQTT Collector ] Inserted {"pH":6.64} in PH.
[MQTT Collector ] Inserted {"temperature":25.8} in Temperature.
[MQTT Collector ] Inserted {"kH":5.18} in KH.
[CoAP Controller] Inserted {"Level": 1222.49,"Value": 43.523872} in CO2Dispenser.
[Smart Aquarium ] Changed CO2 dispensed [ value = 41.39174 ].
[MQTT Collector ] Inserted {"pH":6.59} in PH.
[MQTT Collector ] Inserted {"temperature":25.8} in Temperature.
[MQTT Collector ] Inserted {"kH":4.98} in KH.
[Smart Aquarium ] Osmotic water tank [ mode = off ].
[Smart Aquarium ] Changed CO2 dispensed [ value = 44.649128 ].
[MQTT Collector ] Inserted {"pH":6.59} in PH.
[MQTT Collector ] Inserted {"temperature":26.0} in Temperature.
[MQTT Collector ] Inserted {"kH":4.95} in KH.
[CoAP Controller] Inserted {"Level": 1177.84,"Value": 44.649128} in CO2Dispenser.
[MQTT Collector ] Inserted {"pH":6.6} in PH.
[MQTT Collector ] Inserted {"temperature":26.2} in Temperature.
[MQTT Collector ] Inserted {"kH":4.93} in KH.
[Smart Aquarium ] Fan [ mode = on ].
[CoAP Controller] Inserted {"active": true} in Fan.
[MQTT Collector ] Inserted {"pH":6.6} in PH.
[MQTT Collector ] Inserted {"temperature":25.8} in Temperature.
[MQTT Collector ] Inserted {"kH":4.93} in KH.
[MQTT Collector ] Inserted {"pH":6.63} in PH.
[MQTT Collector ] Inserted {"temperature":25.4} in Temperature.
```

Figure 2.10: Example of the log generated

2.4 Database

Figure 2.11: MySQL database and Grafana

The **MySQL** database is deployed on the **virtual machine**. The access to the database is handled by the **Database manager** class, that provides a **prepared statement** for each different type of operation (are all **INSERT** operations issued to different tables).

The **MQTT Collector** and the **CoAP Network Manager** insert the data coming from the devices and the commands sent to the actuators.
The database is accessed also by **Grafana** in order to implement several graphs to monitor the *status* of the smart aquarium.

The structure of the different tables inside the **smart aquarium** database can be seen in figure 2.12.

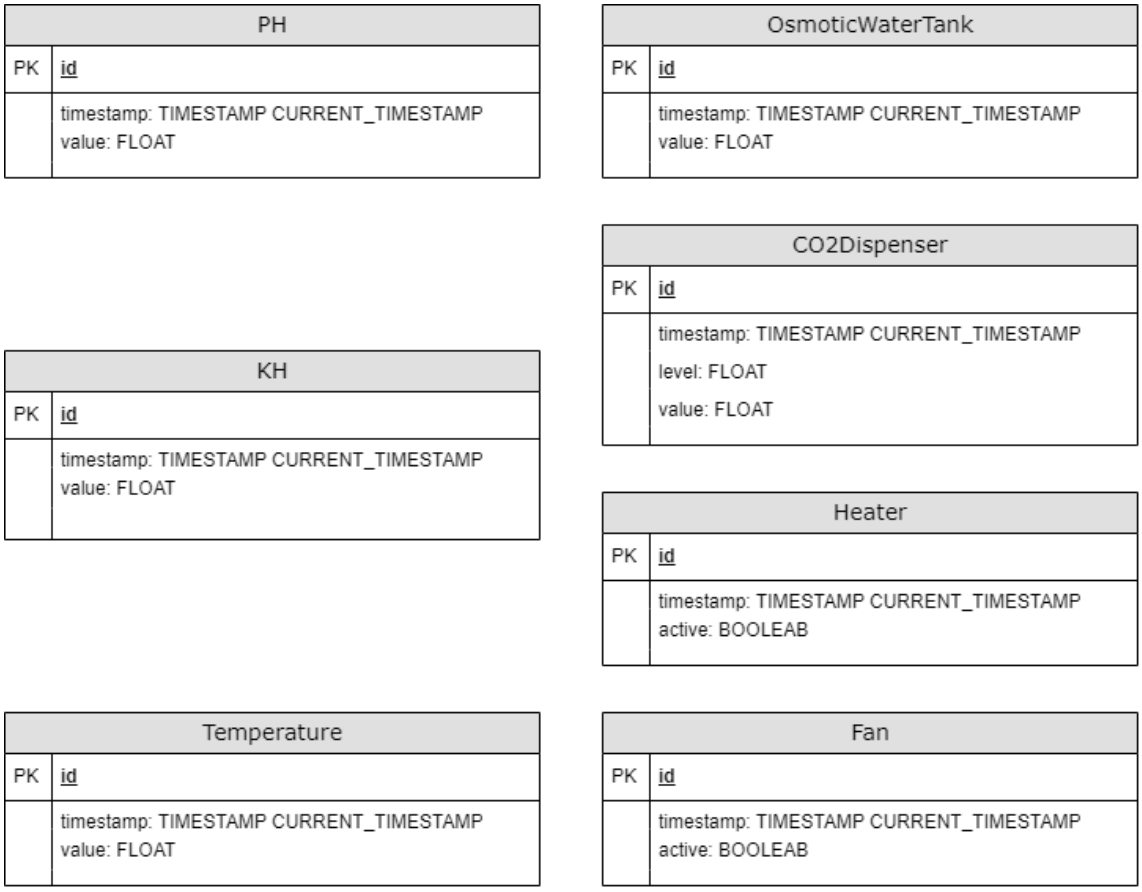

Figure 2.12: Smart aquarium database

There are no *relations* between tables.

2.5 Source Code - Github

The source code of the project is available in the following GitHub page:

https://github.com/Fabi8997/smart-aquarium

Chapter 3

Implementation on real devices and data encoding

3.1 Implementation on nrf52840 dongle

In order to run all the functionalities presented in the previous chapters, it is required to made some adaptation to the code.
Are available three **nrf52840 dongle** (figure 3.1) devices that must implement a **CoAP device**, an **MQTT device** and a **rpl border router**.

Figure 3.1: nrf52840 dongle

The *first problem* lies in the fact that it's not possible to work with **floats** in a proper way, in fact during the tests it was not possible to insert them inside a string using the **sprintf** function, and *inaccuracies* due to error propagation were common. To solve this problem, **two** solutions are used: when the calculations are done with a float, then it is divided into **two parts** held by **two integers**: the *integer* part and the *fractional* part. On the other hand, if the computations that were previously performed using **floats** are now performed using their initial value *multiplied* by **100**, then to convert them to floats to be sent via JSON messages, the

integer is *divided* by **100** to obtain the integer part, and the remainder constitutes the fractional part.

```
//To handle the error propagation and the fact that the floats
//cannot be sent, then are used integers,
//then converted into float strings
bool co2_flow = false;
int co2_tank_level = 700000; //float 7000.0
static int minimum_tank_level = 40000; //float 400.0
bool co2_to_be_filled = false;
int co2_value = 0; // float 0.0

//To cast the int to a float and print it into a JSON string
snprintf((char *)buffer, COAP_MAX_CHUNK_SIZE,
    "{\"level\":%d.0%d , \"mode\":\"%s\"}",
    (int)(co2_tank_level/100), co2_tank_level%100,
    mode);
```

The *second aspect* that must be handled using real sensors is the fact that the **yellow** LED doesn't work. To solve this problem the **blue LED** is used instead.

The last *problem* encountered is the fact that all the **CoAP devices** and the **MQTT devices** must be merged in a single device for both the network types. The new devices obtained are:

- **coap-device**: it presents all the features provided by the devices in the CoAP network, embedded in a single device.
- **mqtt-device**: it is the union of all the MQTT devices, the only difference is that a **green LED** blinks every time a new message is published on its topic. To avoid problem caused by the **internal buffer** shared, then a **token passing** mechanism has been proposed in order to know which simulation must be performed during the current turn (figure 3.2).

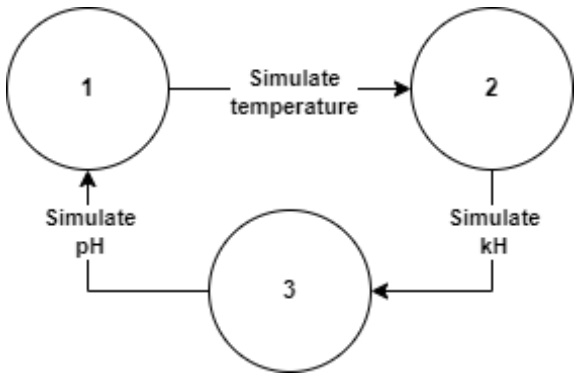

Figure 3.2: Token passing mechanism

3.2 Data encoding

As anticipated in the previous chapters, in this project are used two types of **data encoding** (plus a third type used to exchange simulation messages).

The **JSON** format is used to send data from the **devices** to the **collectors** (both *CoAP* and *MQTT*).
The main aspect that drove the choice of this data type is that it is a **lightweight** data encoding, coupled with the fact that the sensors and actuators have **limited resources**.
The messages generated are very short, consisting mainly of a **key** and a **value**, so the simple format provided by JSON is more appropriate in this case than the more complex syntax required by*XML*, for example.

The **XML** format is used to build the configuration file, as previously explained.

A simple **text** format is used to send *simulation* messages. The different types of messages have already been explained in the previous chapters. This format is used to distinguish the messages generated for the simulation, as they are **fictitious** messages that wouldn't exist in a real environment.

Chapter 4

Grafana

The dashboard created with Grafana using the data read from the database is shown below, focusing on each panel that composes it.

4.1 Sensors panels

The first three panels show the values obtained from the sensors. The **red dotted** lines are the **lower bound** and the **upper bound**, while the **yellow dotted** lines are the intervals around the optimum value (**light green** dotted line).

PH Panel

Figure 4.1: Grafana pH panel

KH Panel

Figure 4.2: Grafana kH panel

Temperature Panel

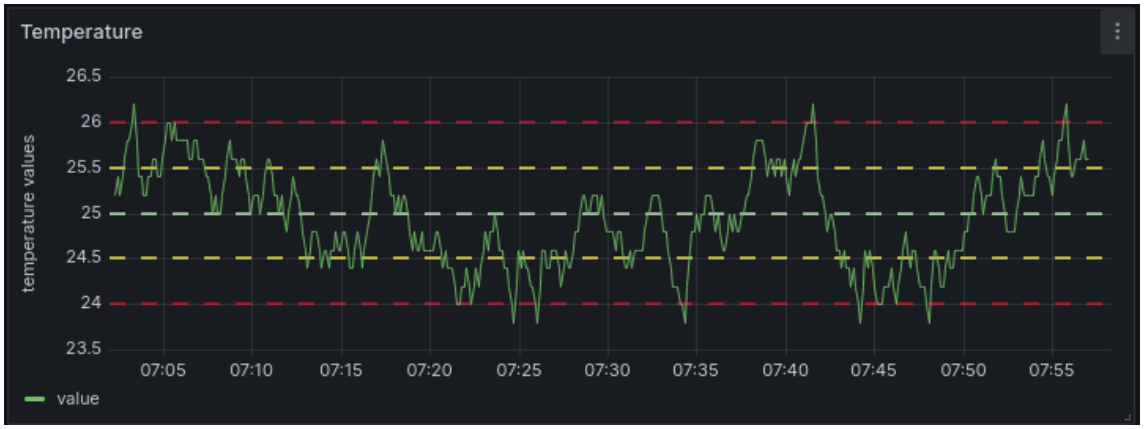

Figure 4.3: Grafana temperature panel

4.2 Actuators panels

The following are the panels relative to the actuators.

CO2 Dispenser panels

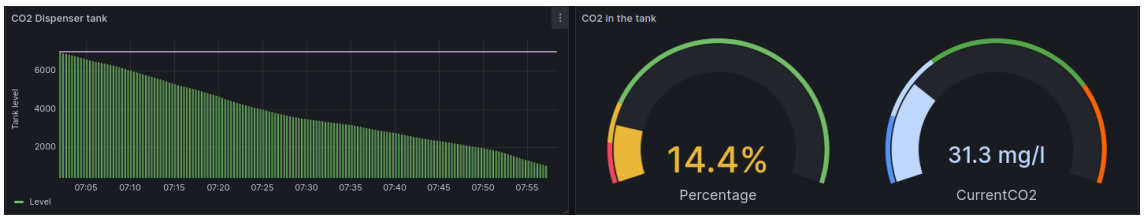

Figure 4.4: Grafana CO2 panel

On the left is the graph of the CO2 level over time, the **pink line** is the maximum capacity of the tank. On the right are, in order, the panel relating to the **percentage** of CO2 remaining in the tank and the **current level** of CO2 being supplied.

Osmotic water tank panels

Figure 4.5: Grafana Osmotic water tank panels

On the left is the graph of the osmotic water tank level over time, the **pink line** is the maximum capacity of the tank. On the right are the panel relating to the **percentage** of osmotic water remaining in the tank.

Osmotic water tank panels

Figure 4.6: Grafana Fan and Heater panel

The **green** line is associated to the activation of the **fan**, while the **yellow** line is associated to the **heater**.

4.3 Temperature, fan and heater relation

The following image, extracted thanks to the dashboard by focusing on a specific interval, shows the temperature variation when the fan or heater are active and how they're stopped when the interval around the optimum is reached.

Figure 4.7: Relation between Fan, Heater and temperature

4.4 KH and Osmotic water flow relation

The following image, extracted thanks to the dashboard by focusing on a specific interval, shows the kH variation when the osmotic water is flowing and the flow is stopped when the interval around the optimum is reached.

Figure 4.8: Relation between kH and the osmotic water flow