UNIVERSITÀ DI PISA

Computer Engineering

Internet of Things

Sluice gate control system

Project Documentation

Leonardo Poggiani

Academic Year: 2021/2022

Table of Contents

1 | Introduction

The developed project aims to implement an automatic sluice control system of a canal flowing to the sea. In fact, several problems may arise in such channels:

- **Canal level too high:** this can lead to flooding and should therefore be avoided by opening bulkheads to let excess water drain away.
- **Water salinity level too high or too low:** the ecosystem of the canal must be preserved since the water will enjoy a peculiar level of salinity. Therefore, salinity values must be leveled by going to open and close the sluice gates.

With this system, it will no longer be necessary to go and monitor salinity levels constantly as it will be sufficient to set a target value that will be maintained with a configurable tolerance level. Three types of sensors are sufficient for the operation of the system:

- **Border router:** to enable communication between the sensor network and the outside world.
- **Water level sensor:** to be placed inside the channel to enable water level sensing.
- **Salinity sensor:** to be placed inside the channel to enable salinity level detection.

The water level sensor sends readings to a collector using the $CoAP$ protocol where they are stored within a database. The salinity sensor, on the other hand, uses the $MQTT$ protocol.

1.1 Application domain:

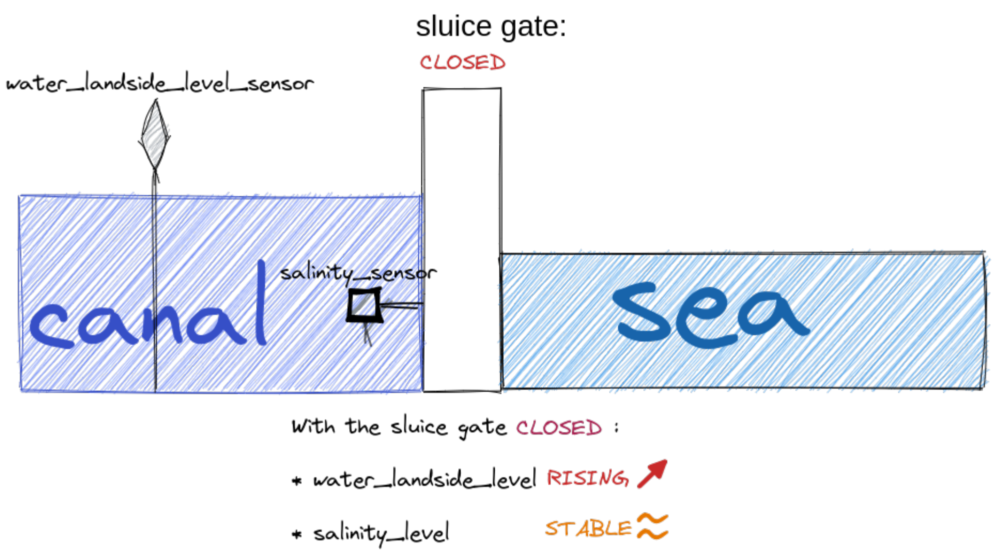
sluice gate:
CLOSED
water_landside_level_sensor
salinity_sensor
canal
sea
With the sluice gate CLOSED :
* water_landside_level RISING
* salinity_level STABLE

Figure 1.1: System behavior with gate closed

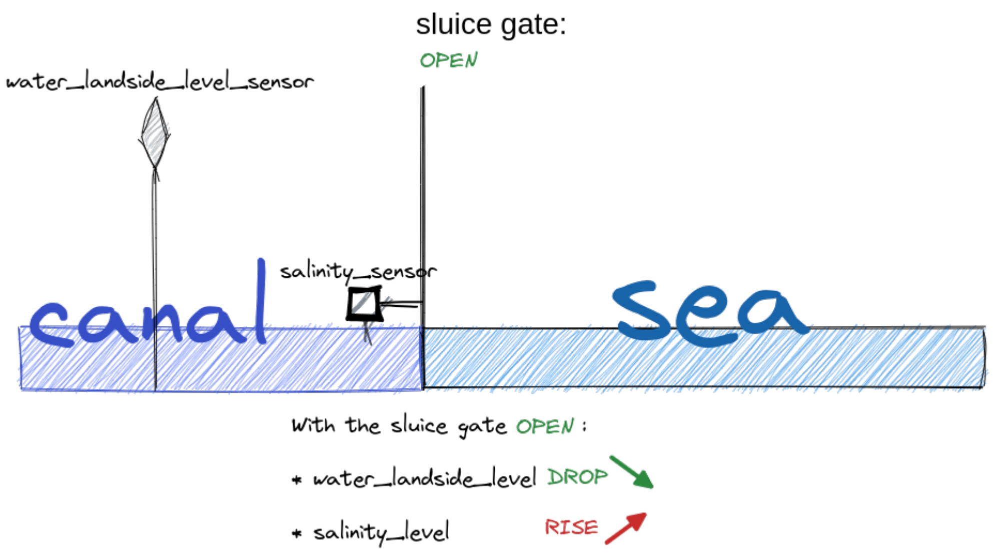
sluice gate:
OPEN
water_landside_level_sensor
salinity_sensor
canal
sea
With the sluice gate OPEN :
* water_landside_level DROP
* salinity_level RISE

Figure 1.2: System behavior with gate open

2 | Design and Implementation

2.1 Overview

All these devices used are IoT devices, and they are equipped with the Contiki-NG operating system which is an open-source, cross-platform operating system for Next-Generation IoT devices. This is a Low Power and Lossy Networks (LLNs) using the IEEE 802.15.4 standard and the IPv6 protocol. Also, they exploit the RPL protocol which enables the multi-hop communication within the network.

Finally, with the help of a border router, it is possible to send the data out of the LLN. These IoT devices exchange their data with a collector, a program that runs on a standard machine, which is usually deployed on the cloud. So, the job of the collector is to collect the data coming from the IoT devices, and storing them in a database. Also, since the collector has the overall view of what is happening in the system, it can perform some aggregative task like sending actuating commands to the IoT devices.

All the exchanged messages are encoded using the JSON data encoding schema. This because it is quite lightweight, so a really good fit for constrained devices, as IoT devices are, and also more readable and simpler with respect to other data encoding format. More over in this use case it is not required an heavy validation of the messages since the exchanged messages are always short messages with a very low probability to have errors in them.

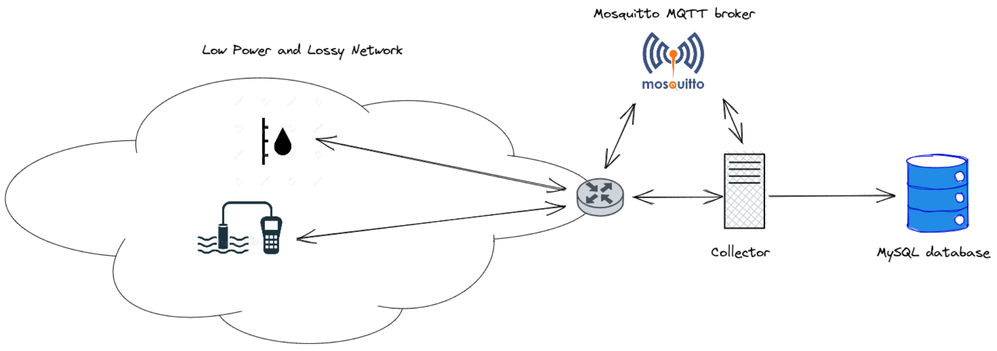

Figure 2.1: System overview

2.2 Level sensor

As previously reported, these are IoT devices exploiting CoAP as application protocol, acting both as client, since they continuously report their readings, and as server, since they also receive commands from the collector. The device acts as a CoAP server by exposing the observable resource level_sensor of which the client (in this case the controller) can become an observer and thus receive status updates through GET response in json format as follows:

```
{"landside_level":"%d.%d", "old_landside_level": "%d.%d"}
```

When the sluice is open the landside water level varies little, while when the sluice is closed the level rises very fast.

2.3 Salinity sensor

These devices are IoT devices which exploit the MQTT application protocol. They act as MQTT client, subscribing and publishing messages to an MQTT Broker. A salinity sensor can act as subscriber and subscribe to the topic "sluice_state" so as to receive from the controller the commands to open or close the sluice, the controller acts as a publisher for this topic. The messages published by the controller for this topic are the following:

```
#define JSON_GATE_OPEN {"sluice_on":true}

#define JSON_GATE_CLOSED {"sluice_on":false}
```

The sensor also acts as a publisher relative to the topic "current_salinity" by periodically publishing messages about the current salinity level detected in the water. Published messages have the following format:

```
{"current_salinity":"%d.%d", "sluice_state": "%s"}
```

2.4 Collector

The collector is a Java program that interacts with all the presented IoT devices: it interacts using CoAP, thanks to the Californium library, with the water level sensors, and with the salinity sensors using the MQTT protocol, this time thanks to the Paho library.

Its job is, as the name suggests, to collect the data generated by the IoT devices and perform some collective task exploiting its complete view of the system. Finally, it has also to store these data in a database. Before starting receiving any updates from CoAP IoT devices, those have to register to the collector. This is needed at the collector to keep track of what devices are up and running and are part of the whole system. Another side advantage of registration is that it becomes unnecessary for the IoT devices to insert each time in their messages their own ID, since this is sent once for good in the registration message, thus reducing the bandwidth.

As it regards the CoAP protocol, the collector behaves both as client and as server, receiving registration messages and updates, and sending commands. It exposes a resource:

- */registration:* it holds and manages the list of registered level controller sensors. This resource accepts only the POST. The format of the messages to register to the collector can be:

  ```
  {"deviceType":"level_controller"}
  ```

There are also some actions that the user may perform in order to change the behavior of the system:

- **!exit:** close the application.
- **!commands:** print the list of commands.
- **!changeTargetLandsideLevel:** change the target level for the landside water.
- **!changeAccLandsideLevel:** change the acceptable range for the landside water.
- **!dischargeWater:** when the water level is too high, the user can flush the water in excess.

2.5 Database

This is one of the main blocks of the collector. Indeed, the collector has to store the collected data in a database so to be able to perform some operation on the historical data, such as show some plot or to perform some analytics and so on, maybe to improve the quality of the offered service.

In this system, the database used is MySQL, a classical relational database, since the quantity of reported data is not so huge, in which cases a more appropriate choice would have been a time series database.

In particular, two tables were defined, one for each kind of IoT device present in the system: salinity and water_level. All the defined tables present the same structure.

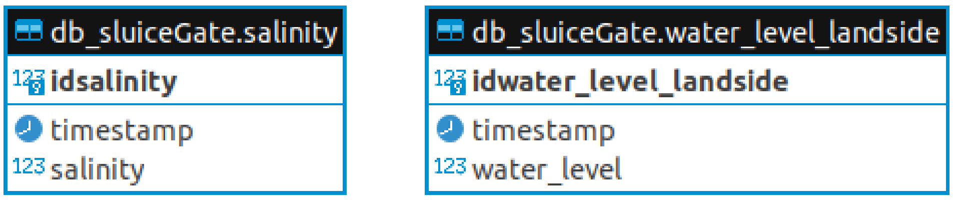

Figure 2.2: Database structure

3 | Testing

3.1 Testing environment

After simulating the whole system in *Cooja* we flashed the sensors provided to us. Collector and MySQL database are inside a virtual machine running Ubuntu 18.04 LTS. We used three *nRF52840 Dongle*: one as *Border router*, one as *salinity sensor* and another one as *water level sensor*.

In order to connect a sensor to a laptop as an IEEE 802.15.4 transceiver it is needed to launch the tunslip6 program which creates a virtual interface, *tun0*, with IPv6 address fd00::1. The configuration of this environment is the following:

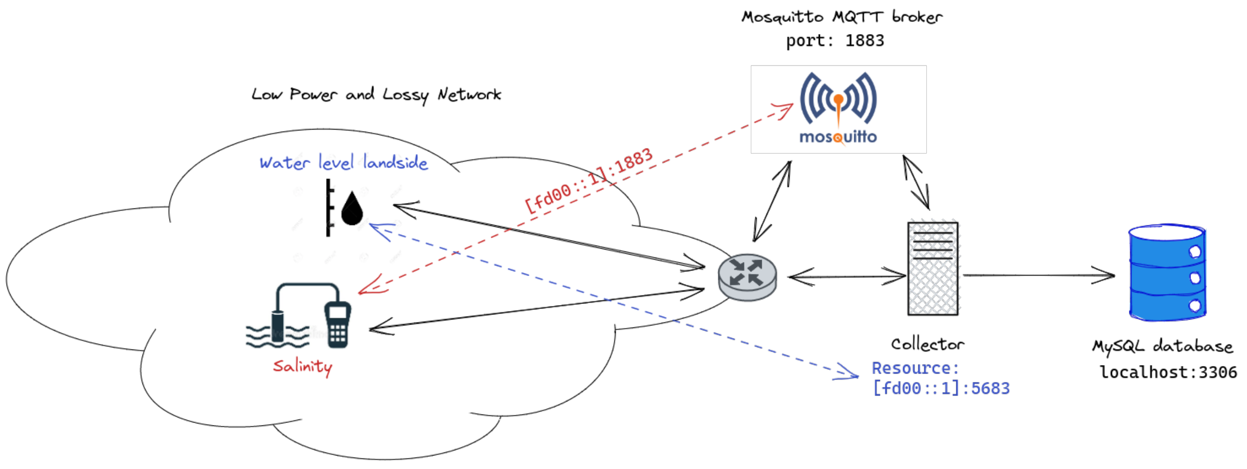

Figure 3.1: Testing environment