UNIVERSITY OF PISA

Internet of Things

Year 2020/2021

Implementation of a temperature regulation system

Project specification

FRANCESCO CAMPILONGO

FRANCESCO DEL TURCO

Link GitHub: https://github.com/FranDT/ProjectIoT

1.0 Introduction

This elaborate is used to present the implementation of a temperature regulation system following the specification of the project, therefore deployed on a MQTT network and a CoAP network.

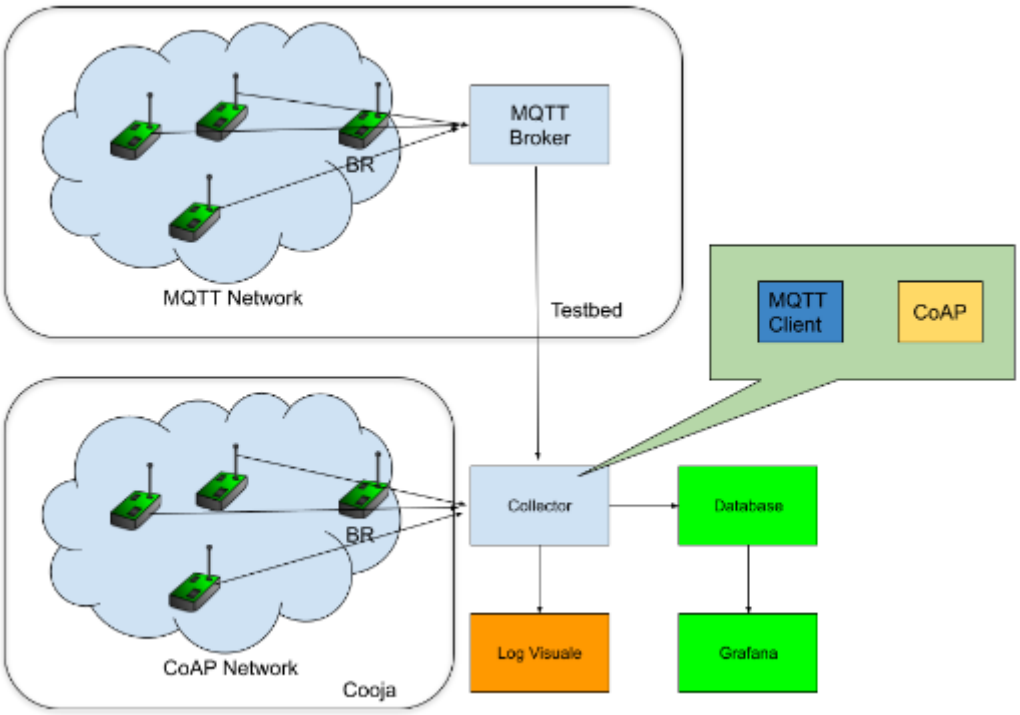
The goal of the system is to recognize, through a thermometer working as sensor, when the temperature of a room exceeds the threshold value, in order to send to the actuator (in this case, we consider it being both able to rise and lower the temperature) a message to regulate the temperature to the standard level. If the temperature of the room gets higher than 25°C or gets lower than 18°C, the corresponding actuator will be activated until the sensed temperature will return to the standard temperature value, equal to 21°C.

This system tries to emulate at best the behavior of a real system, in which the thermometer is usually separated from the actuator, either a conditioner or a radiator: those entities need in any case to be coupled not randomly, but considering the room they're placed in.

The sensors will send the sensed temperature to the Collector, which is in charge of controlling the temperature and sending a message to the actuator in case a regulation is needed. The information about the sensors, actuators and the measurements are saved inside a MySQL database, that is then connected to a Grafana dashboard in order to give the possibility to the system administrator to check the situation in an easy way.

2.0 Architecture

In the following picture we can see a simple scheme of the architecture, as required from the specification of the project:



As we can see, there are three main sections that shape this system: a CoAP Network, a MQTT network and a Collector, connected to a database, that is used as a central hub to collect the information coming from the field devices to be used and controlled in the future.

2.1 CoAP Network

A CoAP Network has been created as part of the project and deployed through the Cooja simulator, simulating the presence of 5 nodes: a border router, corresponding to the default border router we can find inside the Contiki-ng/examples/rpl-border-router folder; 2 sensor nodes, corresponding to two thermometers that will have to control two different rooms; 2 actuator nodes, that will be tightly coupled with one of the two sensors (ideally, the one in the same room, done through population of the database before the start of the system).
Sensor nodes will expose a single observable resource define as /temp: thanks to this, after that the node has registered its presence with the Collector, it will sense and send periodically the temperature value of the room, that will be analyzed by the Collector in order for it to decide whether to activate the corresponding actuator or not. Since we work in a simulated environment, the regulation of the temperature is virtually done inside the sensor node itself, with the actuator nodes that will just print a message to show that they received an activation or suspension message from the Collector.

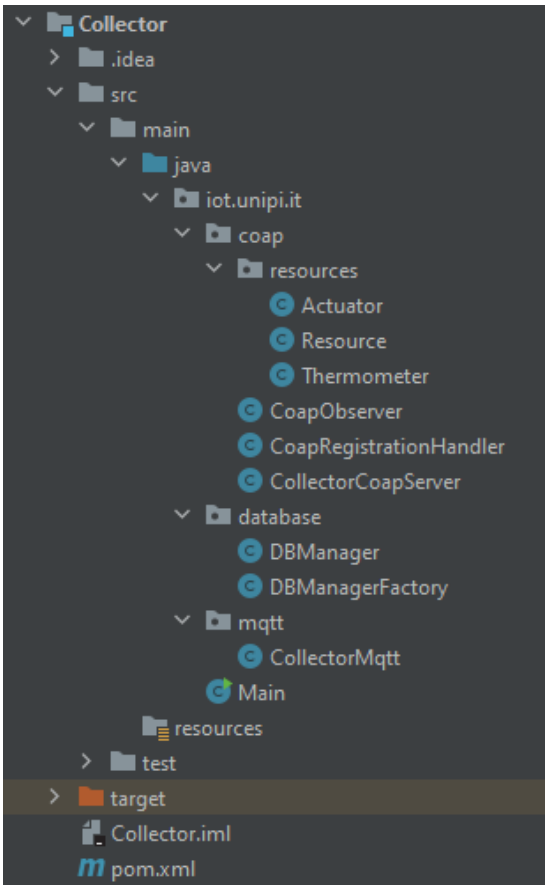
Actuator nodes will expose a single resource named /act: a POST request towards such resource will trigger the execution of the corresponding handler (the only one defined for such resource), in which we will check the option “mode” to understand if we need to turn the actuator on or off; in case the request is to turn on the actuator, the option “value” will be checked, so that if it is equal to “up” the actuator will try to rise the temperature, while if it is equal to “down” it will try to lower the temperature.
As we can see, in this network sensors will only sense the temperature, without doing anything beside communicating the temperature to the Collector; the actuators instead will just wait until they receive a POST request from the Collector, performing at that point the corresponding action. No interaction is present between nodes.

2.2 MQTT Network

The MQTT Network has been deployed on the testbed, as shown at chapter 3.0: in particular, we’ve deployed a couple sensor-actuator and a border router; the MQTT broker has been deployed on the testbed as well.
Sensor nodes are not subscribed to any topic, but are in charge to periodically publish a message containing the sensed temperature on the “temperature” topic: these messages will arrive eventually to the Collector, that is subscribed to the same topic and will receive all the messages published by the sensors (to understand which sensor has published each message, the MQTT sensor ID is included in the message).
Actuators nodes instead are not in charge of publishing anything, but each of them is subscribed to a different topic called “actuator_*actuatorID*“: once the Collector has received a message about the “temperature” topic, it will check the ID of the sensor who published it in the first place; if the temperature needs to be regulated, the Collector will search for the corresponding actuator and will publish a message on the corresponding topic, so that the right actuator can be activate/suspended. In this case, the options “mode” and “value” are passed to the actuator inside the message itself.
The broker is implemented using Mosquitto and, thanks to port forwarding, is able to receive messages both from the Collector deployed locally and from the sensors/actuators deployed on the testbed.

2.3 Collector section

The collector section includes multiple parts inside itself, since it has to implement a MQTT part, a CoAP part and a DB handling part. This section is implemented as a Maven project and is written in Java: the structure of the section is the following:



Inside the POM file we've declared the dependencies that we're going to use for our project: in particular, we have Californium and Paho to handle the two networks, the MySQL connector to handle the database and the JSON dependency for data encoding.

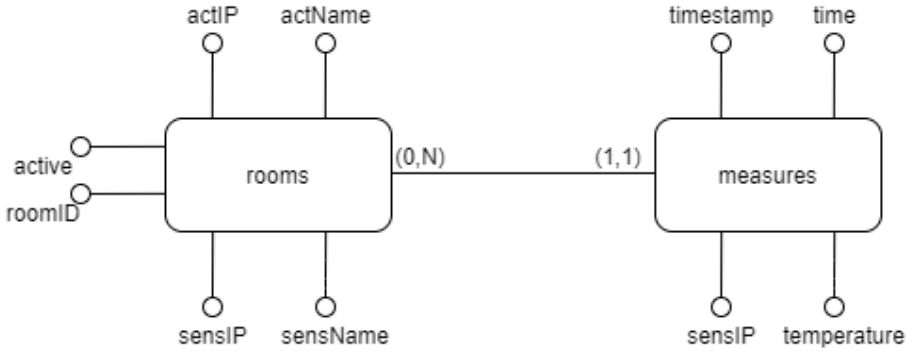
Since we need to handle two different networks from the same Java application, we decided to have two different threads, each one dedicated to one network: this division is performed inside the Main class, in which we also initialize the DBManager class, which is handled as a singleton.

The CoAP part of the Collector starts with the CollectorCoapServer, that is a class implementing the CoapServer class and to which a single "registration" resource is added, corresponding to the CoapRegistrationHandler class: the sensors and actuators from the CoAP Network will issue a POST request towards this resource in order to register themselves to the application. Inside the CoapRegistrationHandler class we will have a handlePOST function that will be triggered every time a registration request from a node arrives: the Collector will save the address of the source node and will issue a GET request toward the "/.well-known/core" resource of the node, that will reply with the list of resources exposed by the node. Since each node has one single resource, the Collector will check if the node is a thermometer or an actuator and will register it to the database consequently; if the node is a thermometer, we will also start observing that resource, waiting for the incoming message with the new measurements. The CoapObserver implements the observation

itself: each time a message arrives from a sensor, it is analyzed to understand whether it's necessary or not to send a message to the corresponding actuator, which information needs to be retrieved from the database. We can see that two resources are defined, Thermometer and Actuator: these classes both extend the Resource class, that defines a nodeAddress field (for the IP address) and a resourceName field; the Actuator class also defines an isActive field, that is checked before trying to send information to the corresponding actuator to make sure it has already registered.
The CollectorMqtt class is very similar to the CoapObserver class in its functioning, but initially it obviously has to create an MqttClient instance to connect to the broker and subscribe to the "temperature" topic.
Finally, all the queries to be issued to the database are declared inside the DBManager class, in which the actual connection to the database is performed as well. Since we need just one DBManager instance, we used the DBManagerFactory class to create a singleton instance to be retrieve by the different classes when they need it.

2.4 Database

The following picture shows a simple diagram of the DB schema:



As shown, we will have two tables:

- The "rooms" table is used during registration to insert the name of the CoAP resources and to check if a room has both the sensor and the actuator up and running (for MQTT, it is assumed they are always ready to run, with the name of the resource saved either in actName or sensName, while sensIP and actIP remain NULL);
- The "measures" table is used to save the measurements done by the sensors, associating the IP instead of the roomID not to have too many accesses to the database during the insertion phase. In this table we have also a "timestamp", that is the timestamp coming from the sensor and is equal to the number of seconds elapsed at the time of the measurements since the start of the process running on the sensor, while the "time" field contains the insertion time inside the database of

the measurement, that is required for Grafana, that asks for a time value while the timestamp we retrieve inside the sensor is a long.

We can therefore see that for each room we may have multiple measurements, with each measurement associated to only one room.

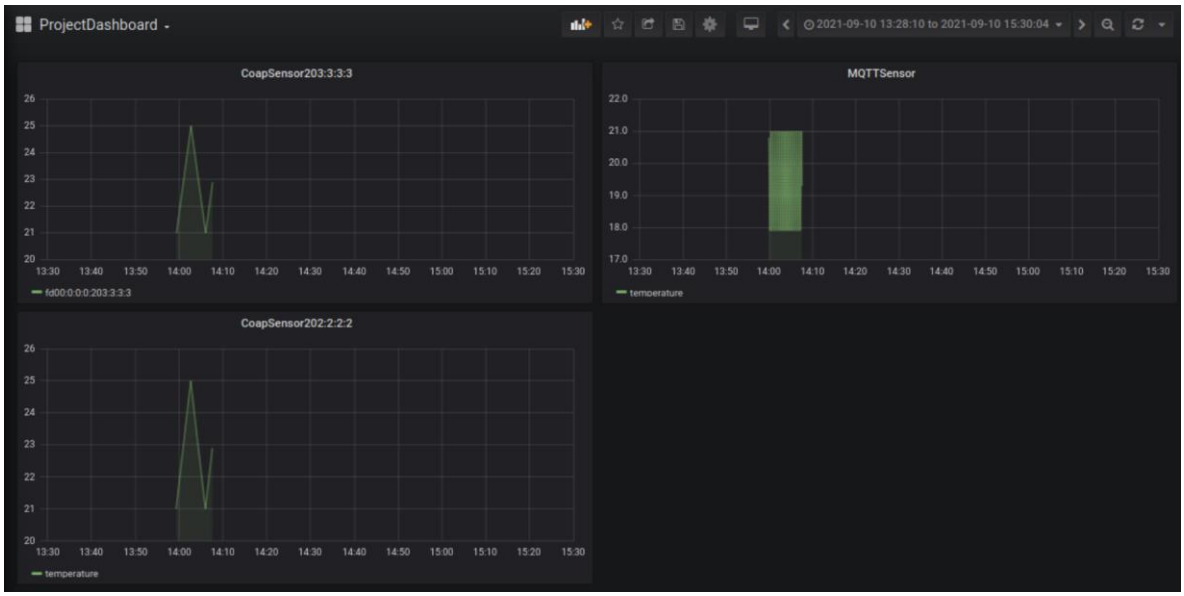
2.5 Data Encoding

As requested in the project specification, we studied the best solution in terms of data encoding for our project and we came to the following conclusion:

- For the communication of the measurements in both CoAP and MQTT, we decided to use a JSON messages since we have to insert multiple fields inside the message itself, which is very easy with JSON;
- If the Collector has to send a message to an actuator in the CoAP Network, it will use a POST request where the payload will contain the parameter as plain text, since they're easier to be parsed inside the actuator when using CoAP;
- If the Collector has to send a message to an actuator in the MQTT Network, it will use JSON for the same reason of point 1.

2.6 Grafana

In order to visualize easily the measurements done by each sensor, we have connected our database instance, in particular the "measures" table, to a Grafana dashboard presenting one panel for each room, showing the changing of the temperature (y-axis) during the time (x-axis), as we can see in the following picture:



As we can see, as soon as the temperature hits one of the threshold, the corresponding actuator starts working in order to bring it back to the standard level.

3.0 Deployment and execution

The code for the project can be found inside the git repository reported in the index page.

3.1 Database and Collector

If you're deploying the database on a local machine, considering that you already have MySQL correctly installed, you need to create a new database and the tables with the following command:

```
CREATE DATABASE project;
CREATE TABLE rooms(roomID INT NOT NULL, sensIP VARCHAR(100), sensName
VARCHAR(100), actIP VARCHAR(100), actName VARCHAR(100), actActive INT DEFAULT
0);
CREATE TABLE measures(sensIP VARCHAR(100) NOT NULL, temperature FLOAT NOT NULL,
timestamp INT NOT NULL, time TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

At this point, insert the values in the "rooms" table in order to have the following population:

```
mysql> select * from rooms;
+--------+---------------------+-------------+---------------------+-------------+-----------+
| roomID | sensIP              | sensName    | actIP               | actName     | actActive |
+--------+---------------------+-------------+---------------------+-------------+-----------+
|      0 | fd00:0:0:0:202:2:2:2 | NULL        | fd00:0:0:0:204:4:4:4 | NULL        |         0 |
|      1 | fd00:0:0:0:203:3:3:3 | NULL        | fd00:0:0:0:205:5:5:5 | NULL        |         0 |
|      2 |                     | f4ce365e24c0 |                     | f4ce36bf4d7c |         1 |
+--------+---------------------+-------------+---------------------+-------------+-----------+
```

To deploy the collector, it is necessary to move inside the *ProjectIoT/Collector* folder: from here, run the following commands:

```
mvn clean install
mvn package
java -jar target/Collector-1.0-SNAPSHOT.jar
```

After this, the collector should be running. Note that is necessary to run the border router for MQTT before running the collector, since it needs to connect to the MQTT broker.

3.2 CoAP Network

To deploy the CoAP network, it is necessary to open the Cooja simulator first, using the following commands:

```
contikier
cd tools/cooja
ant run
```

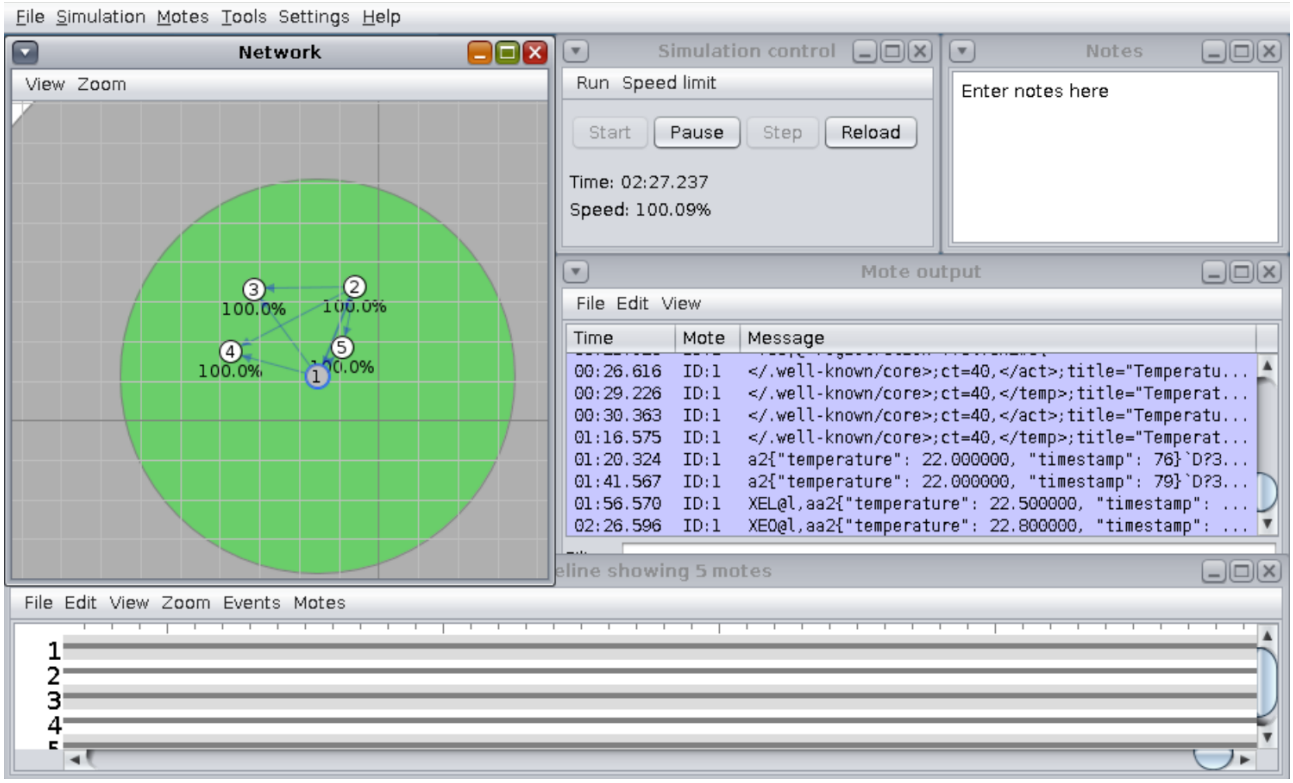
After doing this and after creating a new simulation, it is necessary to deploy the border router through Cooja: you can find it at *ProjectIoT/CoapImplementation/BR/border-*

router.c, that must be compiled in Cooja and then created and assigned correctly as a server instance.

After deploying the border router, deploy two sensors first (*ProjectIoT/CoapImplementation/Sensors/sensorNode.c*) and then two actuators (*ProjectIoT/CoapImplementation/Sensors/actuatorNode.c*): at this point, from a new terminal go inside the border router folder shown before in order to run tunslip6, using the command:

```
make TARGET=cooja connect-router-cooja
```

At this point, you can start the simulation to check with the CoAP network already operative.



3.2 MQTT Network

To deploy the MQTT network, it is necessary to move the MQTT directory inside the testbed, in the right directory; this can be done using the following commands from the ProjectIoT folder:

```
scp -r -P 2007 -i /path/key MqttImplementation/ user@iot.dii.unipi.it:~
ssh -i /path/key -p 2007 user@iot.dii.unipi.it
mv MqttImplementation/ contiki-ng/examples
```

The /path is the path from you local machine to the ssh key used to access the testbed. From the same terminal, run the border router with the following commands:

```
cd contiki-ng/examples/MqttImplementation/BR
make TARGET=nrf52840 BOARD=dongle border-router.dfu-upload PORT=/dev/ttyACM7
make TARGET=nrf52840 BOARD=dongle connect-router PORT=/dev/ttyACM7
```

At this point, open another terminal in which Mosquitto will be launched. In order for Mosquitto to receive messages from the Collector running on the local machine, it is necessary to connect to the testbed via ssh using a port forwarding system, using the following commands:

```
ssh -L 1883:127.0.0.1:1883 -p 2007 -i /path/key user@iot.dii.unipi.it
sudo mosquitto -c /etc/mosquitto/mosquitto.conf
```

You can now deploy the actuator and the sensor (in this order): for the actuator, open a new terminal and run the following commands:

```
ssh -i /path/key -p 2007 user@iot.dii.unipi.it
cd contiki-ng/examples/MqttImplementation/Sensors
make TARGET=nrf52840 BOARD=dongle mqtt-actuator.dfu-upload PORT=/dev/ttyACM58
```

For the sensor, open another new terminal and run the following commands:

```
ssh -i /path/key -p 2007 user@iot.dii.unipi.it
cd contiki-ng/examples/MqttImplementation/Sensors
make TARGET=nrf52840 BOARD=dongle mqtt-sensor.dfu-upload PORT=/dev/ttyACM9
```

Note that the functioning of this system depends on the sensor ID and the actuator ID: if you want to flash on devices that are different from the ones specified above, you will have to change the entry of the database corresponding to roomID equal to 2, inserting the sensor ID and the actuator ID of the nodes you're running the node on. Note also that you may need to change the */etc/mosquitto/mosquitto.conf* file, adding the following lines if not already present:

```
allow_anonymous true
listener 1883 fd00::1
listener 1883 localhost
```