Università di Pisa

School of Engineering

Master of Science in Computer Engineering

Internet of Things

PROJECT DOCUMENTATION

Smart Ceramic Kiln

Matteo Guidotti
Clarissa Polidori

Academic Year 2020/2021

Contents

Chapter 1

Introduction

The context of our project is large, room-sized ceramic kilns. The maintenance and control of these kilns is unthinkable in a manual way, also there may be a need to control these kilns remotely, as the baking times may be many hours and/or days. But above all, let's think about how much a breakdown in these ovens can be an economic and energy loss if not handled in a timely and efficient manner. In addition, the risk of fire with these high temperatures is very high.

The Internet of Things, in this context, can help. We have integrated sensors for temperature sensing that integrate actuators to regulate the latter; sensors for oxygen percentage sensing that integrate actuators to initiate oxygen emission or filtering systems; also, fire detector sensors that can be designed to detect and possibly respond to the presence of flames, or fire, exploiting usually ultraviolet, near infrared, or infrared detecting capabilities, thus allowing flame detection.

The objective of this project is to design and implement an IoT control system for ceramic kilns, with the capability of making instantly visible to workers whether the firing of the pottery is proceeding in the correct and appropriate manner for the desired type of firing, whether the oxygen level inside the kiln is suitable for the type of pottery, and/or how much The percentage of oxygen is risky or livable for humans, indicating to staff when the level is suitable for them to be able to enter inside the kiln without risk.

The oxygen sensors and the fire detectors send their readings, exploiting the *CoAP* application protocol, to a collector which stores them in a database, but also exploits info coming from all sensors to turn on or off oxygen emitters or filters or activate alarms, again using CoAP as appolication protocol. The temperature sensors instead exploit the *MQTT* application protocol to send their produced data to the collector, which, in addition to store them in a database, will trigger some mechanism to decide whether or not turn on or off heaters inside the kiln.

Chapter 2

Design and implementation

2.1 Overview

The system reported in this work is made of two kinds of sensors in terms of the type of application protocol they use to communicate with the controller: oxygen sensors and flame detectors exploiting the CoAP application protocol, and temperature sensors, that instead use the MQTT application protocol. All these devices are, of course, 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.

An example network is shown in Figure 2.1.

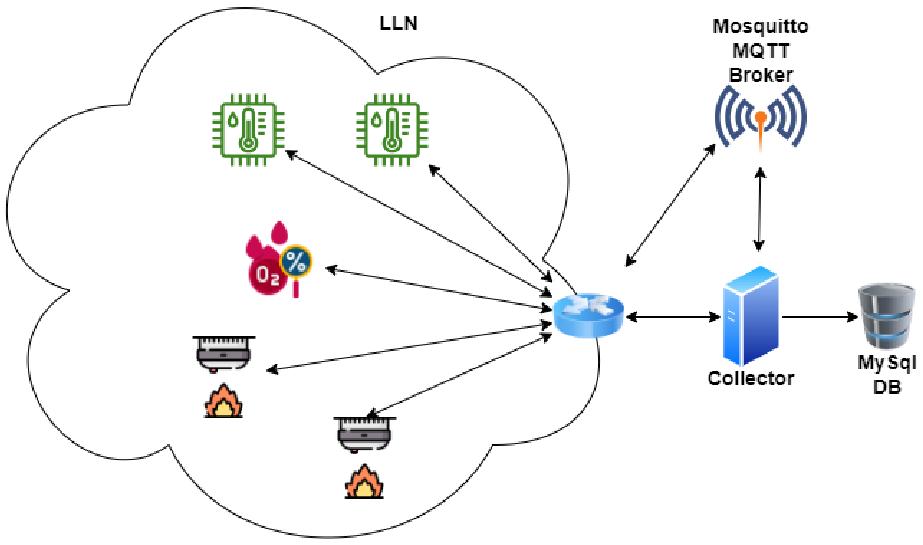


Figure 2.1: System overview

2.2 Temperature sensors

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 temperature sensor can act as subscriber and subscribe to the topic *"heater_state"* so as to receive from the controller the commands to turn on or turn off the heater, the controller acts as a publisher for this topic. The messages published by the controller for this topic are the following:

- `{"heater_on" : true}` to turn on the heater
- `{"heater_on" : false}` to turn off the heater

The sensor also acts as a publisher relative to the topic *"current_temperature* by periodically publishing messages about the current temperature detected in the kiln. Published messages have the following format:

- `{"current_temperature":[value], "heater_state":[on|off]}`

The sensor at a rate of 30 seconds sends the current temperature to the controller as described before. The controller will have asked the user for the target temperature to be kept inside the oven and will turn on or off the heaters inside the oven based on the detected temperature.
The temperature change is emulated in the sensor, with a probability of 1/5 the temperature decreases with the heater off, while with a probability of 4/5 the temperature increases with the heater on.

2.3 Oxygen sensors

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 *oxygen_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:

- `{ "oxygen_value" :  [value] }`

This device also acts as an actuator of an oxygen emitter or filter, the controller that receives the change in oxygen percentage by observing this resource can also send `POST` or `PUT` requests on it to give commands with the following format:

- POST | PUT coap://[fd00::20x:x:x:x]/oxygen_sensor -e "{"type":"emitter|filter", "cause": "CTRL | FIRE | ADMIN", "mode":"on|off"}"

The filtering emission rate varies depending on the reason for which oxygen is being adjusted. Firing control motivations take a slow rate of change, while a fire hazard or an operator that needs to enter the kiln predict a higher rate of change.
The change in oxygen percentage is simulated on the sensor, with filter and emitter off the oxygen level changes with a 10% probability and in 90% of cases remains stable. In case of change with a probability of 2% it increases due to failure or decreases with a much higher probability 98% due to internal furnace combustions.

It acts as a client to register on the controller itself as will be explained later in the section 2.5.

2.4 Fire detectors

The fire detection sensors also use the CoAP protocol for communication with the application. They act as both client and server by exposing the observable resource *fire_detector*. The controller acts as a server by observing the resource and receives through `GET` requests periodically the oxygen relay through a json with the following format:

- `{"alarm":"start|stop"}`

These devices also receive commands from the controller to turn the alarm on or off through POST requests always to the same resource. The requests have the following format:

- POST coap://[fd00::20x:x:x:x]/fire_detector -e {"alarm":"start"}

A fire is simulated on the device, at a rate of 1% the sensor detects flames and turns on the alarm.

It acts as a client to register on the controller itself as will be explained later in the section 2.5.

2.5 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 fire detection sensors and the oxygen sensors, and with the Temperature 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 two resources:

- `/registration`: it holds and manages the list of registered fire detector sensors and oxygen controller sensors. This resource accepts only the `POST`. The format of the messages to register to the collector can be `{"deviceType":"oxygen_controller"}` (explained in section 2.3) or `{"deviceType":"fire_detector"}` (explained in section 2.4).

Only after registration the CoAP sensors begin to do the things described in sections 2.4 and 2.3. The controller subsequently subscribes to the resource exposed by the temperature sensors as described in section 2.2 and starts receiving status messages from them as well. After that, the controller independently takes care of:

- Keeping the oxygen level close to the user-specified target value; the acceptable range is always specified at startup by the user.
- Keeping the temperature the same as the oxygen.
- Turn on the alarm if fire is detected by the fire detector.

It can also receive the following commands from the user:

- *!exit*: exit the program
- *!commands*: list possible commands
- *!changeTargetTemp [new_value]*: set new target temperature value:
- *!changeAccTemp [new_value]*: set new acceptable range for the temperature
- *!changeTargetOxygen[new_value]*:set new target oxygen value
- *!changeAccOxygen [new_value]*: set new acceptable range for the oxygen

- *!stopAlarm* : stop the fire alarm
- *!startAlarm* : start a fire alarm
- *!oxygenFill* : fill the kiln with the oxygen so that a person can enter in

2.5.1 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, three tables were defined, one for each kind of IoT device present in the system: `oxygen, temperature, fire_alarm`. All the defined tables present the same structure:

- `id`: it is simply a unique ID to make it unique each and every row in the table.
- `timestamp`: it saves the timestamp at which the reading/update was collected.
- `value[oxygen,temperature, alarm]`: it holds the value produced by, or sent to, the device.

The fire alarm table contains another column *index_fire* that identifies me all the relative arrived notifications of the same fire, useful to make statistics for example about the duration of each of them. As it regards the *oxygen* and the *fire_alarm* tables, one row is inserted each time an update is received from one of the associated devices. Instead, as for the *temperature*, a row is inserted with the average of the values received from all sensors of the same type.

Chapter 3

Testing

3.1 Test - Real sensor boards environment

After simulating the whole system in *Cooja* we flashed the sensors provided to us, we chose to have two temperature sensors, two fire detectors, one oxygen sensor and of course the border router. Collector and MySQL database are inside a virtual machine running Ubuntu 18.04 LTS.

- *nRF52840 Dongle*: a border router
- *CC2650 Launchpad 21, 25*: two fire detector sensors
- *CC2650 Launchpad 18*: an oxygen sensor
- *CC2650 Launchpad 10, 24*: two temperature sensors

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:

- the Mosquitto MQTT broker runs locally and listens on port 1883
- as for MQTT, the collector contacts the broker at *localhost:1883*
- the temperature sensors contacts the MQTT broker at *[fd00::1]:1883*
- as for CoAP, the collector exposes the resources on the address *[fd00::1]:5683*, so fire detectors sensors and oxygen sensors contact the collector at this address

- the database is reachable at *localhost:3306*.

Test flow:

- The simulation is started. The devices connect to the border router, from which they receive the *fd00* prefix. They can thus assign an address to their interfaces:
- During the normal activity, oxygen sensors, fire detectors and temperature sensors report their updates to the collector, while simultaneously receive commands from the collector.
- Every 30 seconds the temperature sensors send the current temperature to the collector.
- Relative to the desired range, the collector sends the command to turn the heater on or off and the temperature starts to rise or fall.
- Whenever the fire detector detects a fire, it activates the alarm and every minute sends the alarm message to the collector until it receives the stop command.
- The collector in case of fire detected activates the oxygen filter and deactivates the alarm only when 10% oxygen is reached
- The oxygen percentage is notified to the collector whenever it changes.
- The collector sends the command to activate the oxygen emitter or filter in case of fire alarm, personnel maintenance, or simply to better regulate firing.

3.2 Representation of simulation data

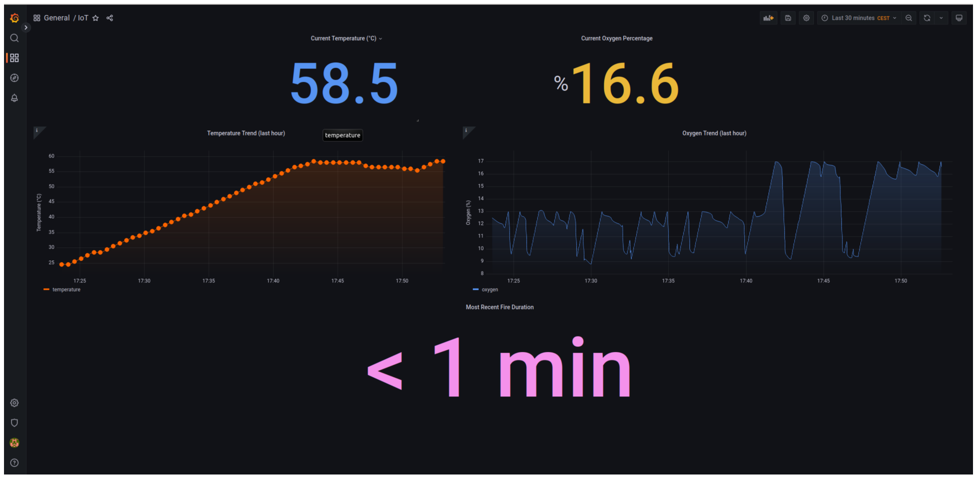
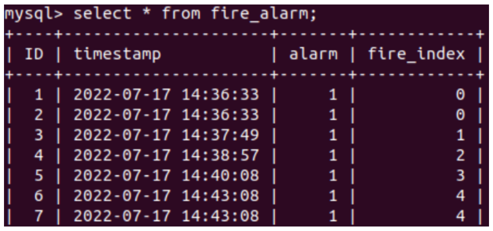
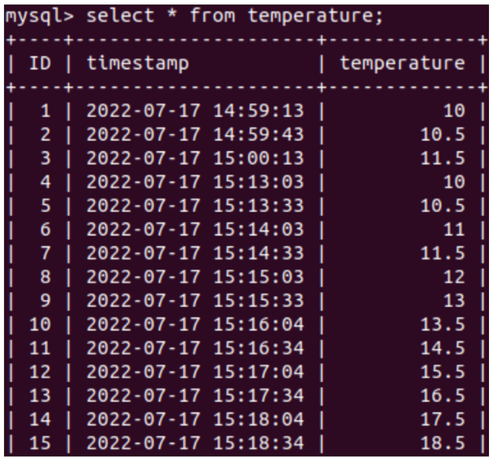


Figure 3.1: Grafana dashboard



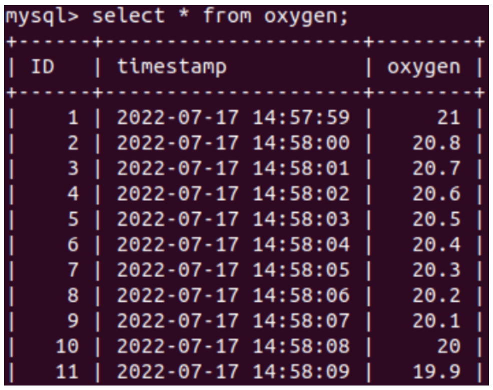
```
mysql> select * from fire_alarm;
+----+---------------------+-------+------------+
| ID | timestamp           | alarm | fire_index |
+----+---------------------+-------+------------+
|  1 | 2022-07-17 14:36:33 |     1 |          0 |
|  2 | 2022-07-17 14:36:33 |     1 |          0 |
|  3 | 2022-07-17 14:37:49 |     1 |          1 |
|  4 | 2022-07-17 14:38:57 |     1 |          2 |
|  5 | 2022-07-17 14:40:08 |     1 |          3 |
|  6 | 2022-07-17 14:43:08 |     1 |          4 |
|  7 | 2022-07-17 14:43:08 |     1 |          4 |
```

Figure 3.2: Fire_alarm table



```
mysql> select * from temperature;
+----+---------------------+-------------+
| ID | timestamp           | temperature |
+----+---------------------+-------------+
|  1 | 2022-07-17 14:59:13 |          10 |
|  2 | 2022-07-17 14:59:43 |        10.5 |
|  3 | 2022-07-17 15:00:13 |        11.5 |
|  4 | 2022-07-17 15:13:03 |          10 |
|  5 | 2022-07-17 15:13:33 |        10.5 |
|  6 | 2022-07-17 15:14:03 |          11 |
|  7 | 2022-07-17 15:14:33 |        11.5 |
|  8 | 2022-07-17 15:15:03 |          12 |
|  9 | 2022-07-17 15:15:33 |          13 |
| 10 | 2022-07-17 15:16:04 |        13.5 |
| 11 | 2022-07-17 15:16:34 |        14.5 |
| 12 | 2022-07-17 15:17:04 |        15.5 |
| 13 | 2022-07-17 15:17:34 |        16.5 |
| 14 | 2022-07-17 15:18:04 |        17.5 |
| 15 | 2022-07-17 15:18:34 |        18.5 |
```

Figure 3.3: Oxygen table



```
mysql> select * from oxygen;
+------+---------------------+--------+
| ID   | timestamp           | oxygen |
+------+---------------------+--------+
|    1 | 2022-07-17 14:57:59 |     21 |
|    2 | 2022-07-17 14:58:00 |   20.8 |
|    3 | 2022-07-17 14:58:01 |   20.7 |
|    4 | 2022-07-17 14:58:02 |   20.6 |
|    5 | 2022-07-17 14:58:03 |   20.5 |
|    6 | 2022-07-17 14:58:04 |   20.4 |
|    7 | 2022-07-17 14:58:05 |   20.3 |
|    8 | 2022-07-17 14:58:06 |   20.2 |
|    9 | 2022-07-17 14:58:07 |   20.1 |
|   10 | 2022-07-17 14:58:08 |     20 |
|   11 | 2022-07-17 14:58:09 |   19.9 |
```

Figure 3.4: Temperature table