

UNIVERSITÀ DI PISA



School of Engineering

Master of Science in Computer Engineering

Internet of Things

PROJECT DOCUMENTATION

SmartPark

Antonio Nunzio Pio Di Noia

Academic Year 2020/2021

Contents

1	Introduction	3
2	Design and implementation	5
2.1	Overview	5
2.2	Vehicle Detection Sensors	6
2.3	Smart traffic lights	7
2.4	Flame detectors	8
2.5	Collector	9
2.5.1	Telemetry database	11
3	Testing	13
3.1	Test 1 - simulated environment	13
3.2	Test 2 - hybrid environment	16

Chapter 1

Introduction

Nowadays parking a vehicle is a real time-consuming task, as the number of cars is constantly growing, especially in large cities. The Internet of Things, in this context, can help in reducing the total time it takes to park your own vehicle and also reducing your emissions, since a lot of vehicle detection sensors, or park lot sensors, were made available in the last years. These kind of sensors, together with the Internet of Things paradigm, could make available systems that help the driver in reaching the nearest free parking lot without wasting time in searching around for it. These sensors come mostly in three shapes: *In-ground sensors*, *Surface mount sensors* and *Overhead indicator sensors*. They are simply able to detect whether a given parking lot is occupied by a vehicle or not, usually exploiting a combination of infrared and magnetic field detection mechanisms. Also, flame detector sensors are available in the market. These kind of sensors are designed to detect and possibly respond to the presence of a flame, or fire, exploiting usually ultraviolet, near infrared, or infrared detecting capabilities, thus allowing flame detection. So, with the help of these three actors, the IoT paradigm, vehicle detection sensors, and flame detectors, it is possible to create smart parking systems that ease the drivers' life, reduce emissions produced by cars driven around searching for a free park lot, and are also provided with security mechanisms to, at least, react as fast as possible to fires.

The objective of this project is to design and implement an IoT control system for a parking, with the capability of making instantly visible to drivers whether in a given parking area there are free park lots or not exploiting vehicle detection sensor and smart traffic lights, and also the capability to detect possible fires so that emergency actions can be taken as fast as

possible. The vehicle detection sensors send their readings, exploiting the CoAP application protocol, to a collector which stores them in a database, but also exploits info coming from all the vehicle detection sensors to update the status of the smart traffic lights, again using CoAP as application protocol. The flame detectors 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 to start the alarm system of the parking.

Based on this system, it could be really useful to design and implement a mobile application which, gathering the information available in the collector and in the database, could guide the driver to the nearest free park lot.

Chapter 2

Design and implementation

2.1 Overview

The system reported in this work is made of two networks: a network composed by vehicle detection sensors and smart traffic lights, exploiting the CoAP application protocol, and a network made of flame detectors, 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.

The two networks are *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, f.i. sending the command to turn the red led to a smart traffic light when it detects that all the park lots are occupied. 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.

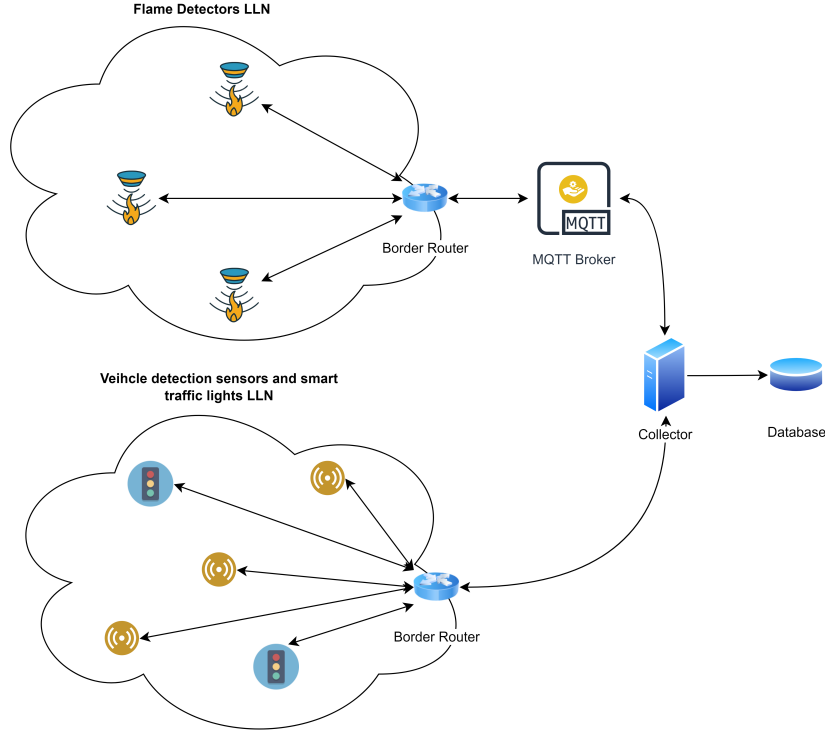


Figure 2.1: System overview

2.2 Vehicle Detection 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. These kind of sensors are placed above each parking lot and they simply report the info about a car arrived (and parked) in the parking lot, or a car that left the parking lot. Moreover, these kind of sensors are equipped with 2 LEDs, green and red, which are exploited to make it instantly, also being far from the lot, visible whether the parking lot is occupied or not.

More in the details, the operation performed by a vehicle detection sensor are the following:

- once it is turned on it waits until it is connected to the network, that is, it has a global IPv6 address and also it is reachable in the network.
- once it is connected, it registers to the collector by issuing a POST to the `/registeredParkSensors` resource, exposed by the collector itself, with payload `{"parkLotID": "PARK_LOT_ID", "occupied": true/false}`. The *occupied* attribute is required because of course a vehicle detection sensor starts its job as soon as the system is on, while instead it registers to the collector only after it has a connection to the border router, and this may take some time during which cars may arrive/leave, so in this way the collector is immediately informed about the lot status.
- during normal operation, it sends an update each time its status changes, that is a car leaves the parking lot or occupies the parking lot. This update is made available through an observable resource, `/lotState`, with payload `{"timestamp": current timestamp, "occupied": true/false}`. Also, the sensor's LEDs change its status: green if the parking lot is free, red otherwise. The arrival/departure of a car is simulated through drawing a random value to be used as the time for which the car will occupy the parking lot.
- if the alarm is active (described later):
 - no more cars are admitted to park inside the parking, so the vehicle detection sensors stop doing their job and shut down their LEDs.
 - if the button present on the sensor is pressed for more than 5 seconds, then the device is restarted. This is meant to be used after the alarm has been managed through a physical intervention.

2.3 Smart traffic lights

Also these devices are IoT devices exploiting the CoAP application protocol, acting both as client and as servers, since they send updates and receive commands from the collector. The operations performed by a smart traffic light are the following:

- once it is turned on it waits until it is connected to the network, that is, it has a global IPv6 address and also it is reachable in the network.
- once it is connected, it registers to the collector by issuing a POST to the `/registeredTrafficLights` resource, exposed by the collector itself, with payload `{"trafficLightID": "TRAFFIC_LIGHT_ID"}`.
- during normal operation, it receives commands from the collector about turning on the red LED if all the parking lots are occupied, green otherwise. These commands are received through the exposed resource `/trafficLight` which accepts only the PUT method.
- if the alarm is activated (described later):
 - the smart traffic light receives a command to start the alarm system, which consists simply in making the red led to continuously blink. This command is received through the `/alarmSystem` resource, which accepts only the PUT method.
 - if the button present on the smart traffic light is pressed for more than 5 seconds, then the device is restarted. This is meant to be used after the alarm has been managed through a physical intervention.

2.4 Flame detectors

These devices are IoT devices which exploit the MQTT application protocol. They act as MQTT client, subscribing and publishing messages to an MQTT Broker. Their role is to detect, with a given probability, a flame in the parking, and this information will be used at the collector side to make a prediction whether or not a fire is present in the parking, and so to possibly start the alarm system. In details, the operation performed by a flame detector are the following:

- once it is turned on it waits until it is connected to the network, that is, it has a global IPv6 address and also it is reachable in the network.
- once it is connected, it subscribes to the `flame-detector-FLAME_DETECTOR_ID/alarm-start` topic, through which it will be able to receive commands about starting the alarm system.

- it publishes a message to the `flame-detector-registration` topic to subscribe itself at the collector side, with payload `{"flameDetectorID": FLAME_DETECTOR_ID}`
- during normal operations, it publishes updates about whether it detected a flame or not to the `flame-detector-FLAME_DETECTOR_ID/flame-detected-updates` topic: if no flame is detected these updates are published once each 10 seconds with payload `{"flameDetected": false}`, otherwise, if a flame is detected, these updates are published once each 2 seconds with payload `{"flameDetected": true}`.
- if a fire is detected by the collector, then the flame detector will receive a message, published by the collector itself, on the `flame-detector-FLAME_DETECTOR_ID/alarm-start` topic with payload `{"alarm": true}`. In this case the flame detector will start making its red LED blinking, and also, it will disconnect from the MQTT broker.
- if the button present on the smart traffic light is pressed for more than 5 seconds while the alarm system is activated, then the device is restarted. Also in this case, this is meant to be used after a human intervention has been carried out in the parking to take actions and, possibly, extinguish the fire.

All the other IoT devices, so vehicle detection sensors and smart traffic lights, cannot be restarted before the flame detector has been restarted after a fire has been put out.

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 vehicle detection sensors and the traffic lights, and with the Flame detectors 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. As explained in the previous sections, before starting

receiving any updates from the 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. In this way when the alarm system is started the collector already knows what devices are connected, and so it can send the signal to each registered device to make it know that the alarm has been triggered. 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:

- `/registeredParkSensors`: it holds and manages the list of registered vehicle detection sensors. This resource accepts only the POST. The format of the messages to register to the collector is `{"parkLotID": "PARK_LOT_ID", "occupied": true/false}` (explained in section 2.2).
- `/registeredTrafficLights`: it holds and manages the list of registered smart traffic lights. This resource accepts only the POST. The format of the messages to register to the collector is `{"trafficLightID": "TRAFFIC_LIGHT_ID"}` (explained in section 2.3).

After the registration of each smart traffic light, the collector sends a command to these devices to make them turning on, issuing a PUT request towards the `/trafficLightLeds` resource exposed by each of these devices with payload `{"mode": "on", "color": "r"/"g"}`. The "color" field is set to "g" (green) or "r" (red) based on the information the collector has, at that moment, received by all the registered vehicle detection sensors. Instead, after the registration of each vehicle detection sensor, the collector subscribes an observe relation towards the `/lotState` resource exposed by each of these sensors as explained in section 2.2. Through this observe relation the collector receives the updates sent out by the vehicle detection sensors about their state (occupied/free). By receiving this info from all the registered vehicle detection sensors the collector is able to understand whether the parking has still some free parking lots or not. In case there are no more free lots, the collector will send a command to the registered traffic lights towards their exposed resource `/traffic-`

cLight, with payload { "color" : "r" } to make them turning on the red LED, to visually signal drivers that there are no more available lots; as soon as a park lot gets free, and so the collector receives the associated update from the given vehicle detection sensor, the collector will send a command to the traffic lights with payload { "color" : "g" }, to make them turning on the green LED.

Finally, as it regards the flame detectors, since as already suggested those are IoT devices exploiting the MQTT protocol to send their updates, the collector in order to receive such updates has to subscribe to the topics where these updates are published by the flame detectors, and also to send command it has to publish them to well defined topics. This part has been described in details in section 2.4.

The collector has also the job of triggering the alarm system when a fire is detected. Here a machine learning technique could be exploited to make this task more reliable. In this project a simpler solution was adopted: the alarm system is triggered when five consecutive updates from the flame detector report that a flame is detected, so to be almost sure that a fire has been spot out, and at the same time not waste too much time before the intervention.

2.5.1 Telemetry 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: `park_lot_status`, `smart_traffic_light_status`, `flame_detector_status`. 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.

- `TYPE_OF_DEVICE_id`: it holds the id of the device that produced, or to which is sent, the update.
- `status`: it holds the value produced by, or sent to, the device.

As it regards the `park_lot_status` and the `flame_detector_status` tables, one row is inserted each time an update is received from one of the associated devices. Instead, as for the `smart_traffic_light_status`, since this is an actuator, so it does not send updates towards the collector, one row is inserted each time the collector detects that the color of the smart traffic lights has to be switched (as described in section 2.5).

Chapter 3

Testing

Two different tests will be carried out, one where all the devices are simulated in the Cooja simulator, and one where again all the devices are simulated in the Cooja simulator but the flame detector, which will be implemented on a real testbed. The first test was added since, as explained before, after the alarm system has been triggered, and after the fire has been put out thanks to a human intervention, the first device that need to be restarted is the flame detector, after which all the other IoT devices can be restarted as well. Given this fact, since it is not possible to physically press the button on the real sensor, it was decided to show the complete behaviour of the system through a simulated test.

For testing purposes, the detection of a flame by a flame detector is triggered based on a decreasing counter, that when reaches 0 will make the flame detector to start sending update notifying about a detected flame.

In the testing environment, the MQTT broker used is Mosquitto. Also, the parking are is thought to be composed of one single area, so one single flame detector is deployed.

3.1 Test 1 - simulated environment

As said, in this test all the IoT devices are simulated in the Cooja simulator, inside a virtual machine running Ubuntu 18.04 LTS, together with the collector and the MySQL database.

The simulated environment consists of five Cooja motes:

- Node 1: a border router

- Nodes 2 and 3: two vehicle detection sensors
- Node 4: a smart traffic light
- Node 5: a flame detector sensor

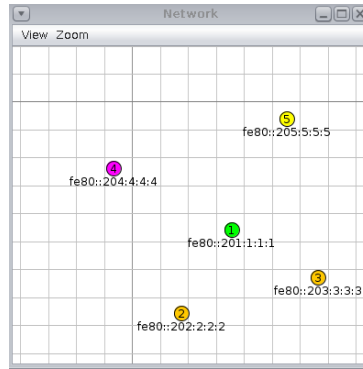


Figure 3.1: Simulated network on Cooja

In order to allow packets to be bridged between the Cooja simulator and the collector, and also the MQTT broker, it is needed to launch the `tunslip6` program which creates a virtual interface, `tun0`, with IPv6 address `fd00::1`.

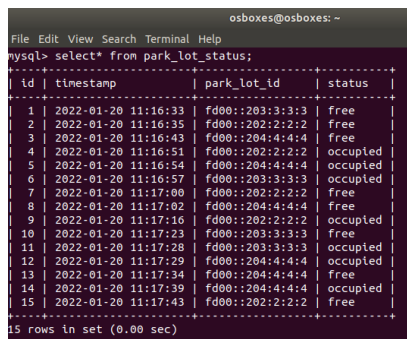
The configuration of this environment is the following:

- the Mosquitto MQTT broker runs locally and listens on port 1883
- as for MQTT, the collector contacts the broker at `localhost:1883`
- the flame detector contacts the MQTT broker at `[fd00::1]:1883`
- as for CoAP, the collector exposes the resources on the address `[fd00::1]:5683`, so vehicle detection sensors and smart traffic light contact the collector at this address
- the database is reachable at `localhost:3306`.

Test flow:

1. The simulation is started. The devices connect to the border router, from which they receive the `fd00:1` prefix. They can thus assign an address to their interfaces:
 - vehicle detection sensors (Nodes 2-3): `fd00::202:2:2:2`, `fd00::203:3:3:3`

- smart traffic light (Node 4): fd00::204:4:4:4
 - flame detector (Node 5): fd00::205:5:5:5
2. during the normal activity, vehicle detection sensors and the flame detector report their updates to the collector, while the smart traffic light receives command from the collector;
 3. 1 minutes after the flame detector started its normal activity, it starts detecting a flame;
 4. after 5 consecutive updates received by the collector from the flame detector stating that the flame is detected, the collector starts the alarm system;
 5. in order to notify the controller, and so to allow all the devices to restart, it is needed to press the button on the flame detector for more than 5 seconds, so the flame detector will restart and start again all its operation;
 6. in order to restart vehicle detection sensors and the smart traffic light it is needed to press the button on each single device for more than 5 seconds, after which the device will start again all its operation;
 7. this time the alarm system will be triggered after 2 minutes;

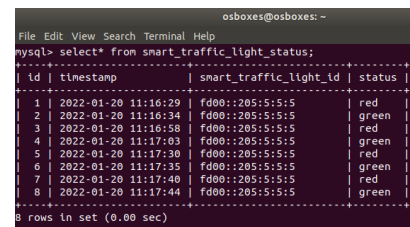


```

osboxes@osboxes: ~
File Edit View Search Terminal Help
mysql> select* from park_lot_status;
+----+-----+-----+-----+
| id | timestamp | park_lot_id | status |
+----+-----+-----+-----+
| 1 | 2022-01-20 11:16:33 | fd00::203:3:3:3 | free |
| 2 | 2022-01-20 11:16:35 | fd00::202:2:2:2 | free |
| 3 | 2022-01-20 11:16:43 | fd00::204:4:4:4 | free |
| 4 | 2022-01-20 11:16:51 | fd00::202:2:2:2 | occupied |
| 5 | 2022-01-20 11:16:54 | fd00::204:4:4:4 | occupied |
| 6 | 2022-01-20 11:16:57 | fd00::203:3:3:3 | occupied |
| 7 | 2022-01-20 11:17:00 | fd00::202:2:2:2 | free |
| 8 | 2022-01-20 11:17:02 | fd00::204:4:4:4 | free |
| 9 | 2022-01-20 11:17:16 | fd00::202:2:2:2 | occupied |
| 10 | 2022-01-20 11:17:23 | fd00::203:3:3:3 | free |
| 11 | 2022-01-20 11:17:28 | fd00::203:3:3:3 | occupied |
| 12 | 2022-01-20 11:17:29 | fd00::204:4:4:4 | occupied |
| 13 | 2022-01-20 11:17:34 | fd00::204:4:4:4 | free |
| 14 | 2022-01-20 11:17:39 | fd00::204:4:4:4 | occupied |
| 15 | 2022-01-20 11:17:43 | fd00::202:2:2:2 | free |
+----+-----+-----+-----+
15 rows in set (0.00 sec)

```

Figure 3.2: park_lot_status table



```

osboxes@osboxes: ~
File Edit View Search Terminal Help
mysql> select* from smart_traffic_light_status;
+----+-----+-----+-----+
| id | timestamp | smart_traffic_light_id | status |
+----+-----+-----+-----+
| 1 | 2022-01-20 11:16:29 | fd00::205:5:5:5 | red |
| 2 | 2022-01-20 11:16:34 | fd00::205:5:5:5 | green |
| 3 | 2022-01-20 11:16:58 | fd00::205:5:5:5 | red |
| 4 | 2022-01-20 11:17:03 | fd00::205:5:5:5 | green |
| 5 | 2022-01-20 11:17:30 | fd00::205:5:5:5 | red |
| 6 | 2022-01-20 11:17:35 | fd00::205:5:5:5 | green |
| 7 | 2022-01-20 11:17:40 | fd00::205:5:5:5 | red |
| 8 | 2022-01-20 11:17:44 | fd00::205:5:5:5 | green |
+----+-----+-----+-----+
8 rows in set (0.00 sec)

```

Figure 3.3: smart_traffic_light_status table

```

osboxes@osboxes: ~
File Edit View Search Terminal Help
mysql> select* from flame_detector_status;
+----+-----+-----+-----+
| id | timestamp          | flame_detector_id | status |
+----+-----+-----+-----+
| 1  | 2022-01-20 11:16:50 | fd00::206:6:6:6   | low    |
| 2  | 2022-01-20 11:17:00 | fd00::206:6:6:6   | low    |
| 3  | 2022-01-20 11:17:10 | fd00::206:6:6:6   | low    |
| 4  | 2022-01-20 11:17:20 | fd00::206:6:6:6   | low    |
| 5  | 2022-01-20 11:17:30 | fd00::206:6:6:6   | low    |
| 6  | 2022-01-20 11:17:40 | fd00::206:6:6:6   | high   |
| 7  | 2022-01-20 11:17:42 | fd00::206:6:6:6   | high   |
| 8  | 2022-01-20 11:17:44 | fd00::206:6:6:6   | high   |
| 9  | 2022-01-20 11:17:46 | fd00::206:6:6:6   | high   |
| 10 | 2022-01-20 11:17:48 | fd00::206:6:6:6   | high   |
+----+-----+-----+-----+
10 rows in set (0.00 sec)

```

Figure 3.4: flame_detector_status table

3.2 Test 2 - hybrid environment

In this test vehicle detection sensors and smart traffic light are again simulated on Cooja, while the flame detector is flashed on a real device, more specifically on a nRF52840 dongle board hosted in a remote testbed. In order to make the dongle executing the flame detector program to be connected to the outside a border router is flashed on another nRF52840 dongle in the testbed. This time the MQTT Broker runs on the remote testbed, so port tunneling is necessary to map the remote port exposed by Mosquitto on the local port of the local machine through which the collector will send its packets to the MQTT Broker. So, the setting is the one presented in Figure 3.5

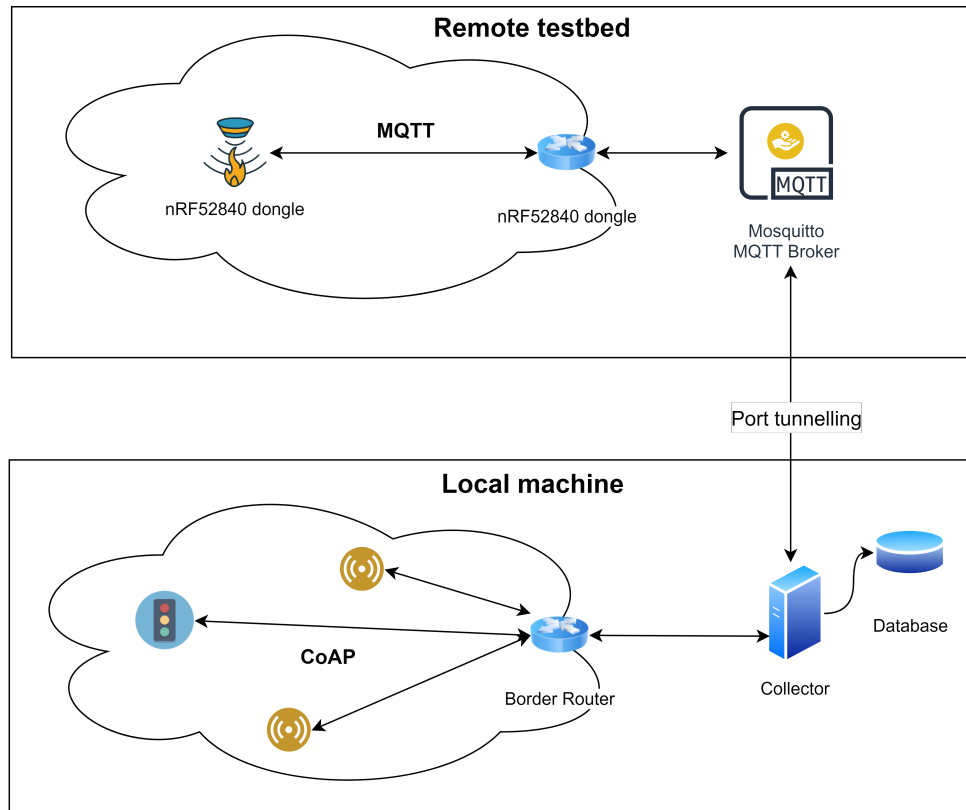


Figure 3.5: Test 2 setting

The configuration of this environment is the following:

- the Mosquitto MQTT broker runs on the remote testbed and listens on address $fd00::1$, port 1883
- the MQTT flame detector contacts the MQTT broker at $[fd00::1]:1883$;
- the collector contacts the remote MQTT broker at $localhost:1883$, and thanks to the port tunnelling, this are mapped on the remote address $[fd00::1]:1883$;
- as for CoAP, the collector exposes the resources on the address $[fd00::1]:5683$, so vehicle detection sensors and smart traffic light contact the collector at this address;
- the database is locally reachable at $localhost:3306$.

Test flow:

1. The simulation is started and the remote devices are programmed. The devices connect

to the border router, from which they receive the fd00:1 prefix. They can thus assign an address to their interfaces:

- vehicle detection sensors (Nodes 2-3): fd00::202:2:2:2, fd00::203:3:3:3
 - smart traffic light (Node 4): fd00::204:4:4:4
2. during the normal activity, vehicle detection sensors and the flame detector report their updates to the collector, while the smart traffic light receives command from the collector;
 3. 1 minutes after the flame detector started its normal activity, it starts detecting a flame;
 4. after 5 consecutive updates received by the collector from the flame detector stating that the flame is detected, the collector starts the alarm system;

In this case, as said, it is not possible to test the restart feature of the system since it is not possible to press on the dongle.