UNIVERSITÀ DI PISA

Internet of Things - Smart Home

Simone Bensi

Table of contents

1. Introduction

A smart home system connects some home objects to automate specific tasks and it is remotely controlled. Smart homes allow users to have greater control of their energy use and can help them become more energy efficient and mindful of ecological factors. In this way users can also save their money.

In particular, the proposed system consists of two networks of IoT devices, and each device is both an actuator and a sensor. One network manages the temperature handling, while the other the luminosity handling.

Deployment structure

The network handling luminosity regulation is made of devices each deployed in a room. In this way we have an actuator and a sensor for each room of the house. House rooms are kitchen, living room, bedroom and bathroom, each of them managed independently because luminosity levels in various rooms of the same house can be much different in every moment. Instead, the network handling temperature regulation is made of one device located in the center of the house. Indeed, the temperature across different rooms in the same house is almost constant.

These networks are respectively composed of many CoAP sensors/actuators and a MQTT sensor/actuator. There is an additional device which acts as a border router allowing sensors to communicate with a collector written in java. Basically, the collector has to receive measurement performed by sensors and to issue commands towards actuators. Furthermore, the collector outputs collected data to the user via a textual log and stores them in a MySQL Database. In order to make possible communication between an MQTT device and the collector, an MQTT broker is needed.

This is the overall scenario.

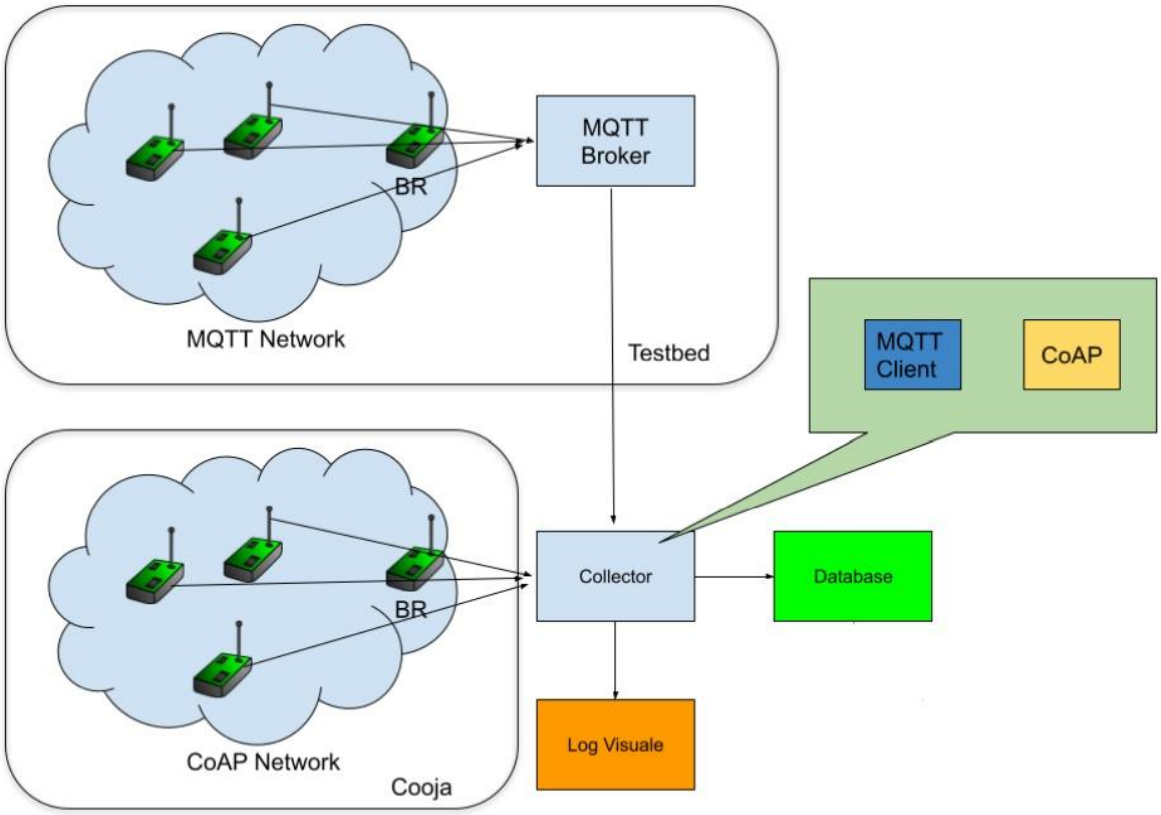


Figure 1: overall scenario

Data Encoding

For data interchange JSON has been chosen as the data encoding language in every message exchange because it's faster than XML. Moreover, JSON encoding is terse, which requires less bytes for transit, in addition to the fact that JSON parsers are less complex, which requires less processing time and memory overhead. Indeed, the main requirements of this application are the low latency and the flexibility in the data interchange, rather than a high level of data integrity and control. We do not have to check the well-formedness of the validity of a document, avoiding associated overhead. For this particular application, if some data get lost or arrive uncompleted to the collector, this is not a crucial problem.

Instead, it is important to offer a pleasant user experience by performing measurements as fast as possible, so that, for example, if the temperature or luminosity drop beyond the specified thresholds the system has to react in a negligible and so imperceptible amount of time.

2. MQTT Network

This network handles temperature and is deployed on real sensors hosted by a testbed and consists of 2 devices: a temperature sensor/actuator put in the middle of the house and a border router which allows the sensor to access the MQTT broker to publish or retrieve information. Also the MQTT broker is deployed on the testbed. In this project the MQTT broker is implemented through the use of Mosquitto.

In particular, the sensor reports every 20 seconds a value that represents the current measured temperature in the house. The actuator takes care of increasing or decreasing temperature according to commands coming to the device from the collector.

The temperature is kept constant from the collector, which periodically receives measurements performed by sensors and based on those values sends commands to the correspondent actuator to maintain temperature in an user specified interval.

Data Publishing

The MQTT sensor periodically reports the temperature in the house by publishing on the topic “temperature”, on which the collector is subscribed.

```
if(state == STATE_SUBSCRIBED)
{
  //publish temperature percentage every 20 seconds
    etimer_set(&publish_timer, DEFAULT_PUBLISH_INTERVAL);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&publish_timer));

    sprintf(pub_topic, "%s", "temperature");

  if(mode == 1)
    temperature = rand() % (TEMPERATURE_VARIATION + 1) + temperature; // if the heater is on the temperature can only increase
  else if (mode == -1)
    temperature = rand() % (TEMPERATURE_VARIATION + 1) + temperature - TEMPERATURE_VARIATION; // if the cooler is on the temperature can only decrease
  else
      temperature = rand() % (TEMPERATURE_VARIATION*2 + 1)  + temperature - TEMPERATURE_VARIATION; // if both cooler and heater are off
                                                                                                  // temperature can either decrease or increase
    sprintf(app_buffer, "{\"temperature\":%d}",temperature);
    printf("%s \n",app_buffer);
  mqtt_publish(&conn, NULL, pub_topic, (uint8_t *)app_buffer, strlen(app_buffer), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
}
```

Figure 2: MQTT data publishing

The temperature sensor behavior is emulated generating every 20 seconds a random integer in order to be able to see how the system works in a reasonable amount of time. In a real context this period may be higher in order to maintain the sensor in sleep mode, allowing it to save energy. Indeed, there is no need to have a high frequency sensing because temperature does not change drastically in a short amount of time.

To make the simulation more realistic, the random number generated cannot be too far from the previous generation, in this case the maximum variation is 2° Celsius. Moreover, to give importance to the current actuator mode, if the cooler is on the temperature can only decrease, if the heater is on the temperature can only increase, while if both are off temperature can freely increase or decrease. Once measured this value is embedded in a JSON message with the following structure.

```
{
    "temperature": 60
}
```

Figure 3: measured temperature JSON message

Actuator

The MQTT device receives remote commands by subscribing on the topic “actuator”. By publishing on the same topic, the collector can regulate the temperature levels in the house, by sending commands which can be either “up”, “null”, “down”.

Each time a message is published on the topic actuator, the MQTT device parse the JSON message and understands what it has to do: if the command is “up” it has to increase temperature because the current temperature is too low; if the command is “null” it has to do nothing because the temperature is already in the user specified interval, thus saving energy; if the command is “down” it has to decrease temperature because the current temperature is too high.

3. CoAP Network

This network handles luminosity and is composed of 4 CoAP sensors, one for each room. Each room is managed independently, because their luminosities do not depend on luminosities of other rooms. There is also an additional device that acts as a border router to gain external access. Each device exposes 2 resources:

- res_light: acts as a sensor for the light measurement in a specific room
- res_bulb: acts as an actuator for the switch on/off three lights in a specific room

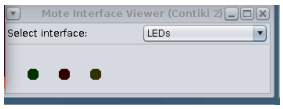
This network has to periodically check the luminosity percentage value in each room of the house and change light level accordingly. More in depth, there are three user defined thresholds which identify three percentage luminosity levels: high, medium and low, which have as default value respectively 75%, 50% and 25%. Data are sent to the collector, which sends a command in which it specifies what bulbs switch on. There are three bulbs (red, green, yellow), which are represented through Cooja leds. These lights represent a lighting source, in particular the more bulbs are on the more the room is illuminated. Moreover, there are four illumination situations:

- Red, green and yellow bulbs are off
- The green light is on while both the red and the yellow bulbs are off
- Both the green and the red bulbs sare on while the yellow bulb is off
- Red, green and yellow bulbs are off

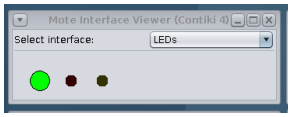
In order to be periodically updated, the collector establishes an observing relation with the light resource of each one of the 4 sensors.

Based on value of current measured luminosity value (let's call it *L*), different scenarios are expected for what bulbs have to be on and what to be off:

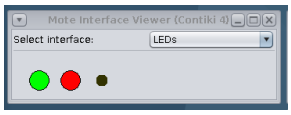
- L > high level



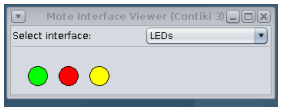
- medium level<L<high level



- low level<L<medium level



- low level<L<medium level



For each room there is the possibility to switch off the system because either the user wants to sleep or he wants to exit the room. He can do this simply by pressing a switch in the room, simulated in this project through a Cooja button. When this button is pressed all room bulbs are always off until the user presses the button again. In this way the user avoids wasting electricity, thus saving money.

CoAP Server

The main code of each sensor first activates both light and bulb resources, than perform a loop in which in each iteration:

- Initialize a timer with a value of 5 seconds. This will be the periodicity through which the sensor will send updates about the measured luminosity.

- Check if the Cooja button has been pressed, and in such case switch the state of the system, activating or deactivating the bulb resource.

The actual main code is the following.

```
while(true)
{
        etimer_set(&et,LIGHT_INTERVAL);
        PROCESS_WAIT_EVENT();

        // every 5 seconds sensor measures light
        if(ev == PROCESS_EVENT_TIMER)
                res_light.trigger();

        // if has been pressed the button I switch the state of the system
        else if(ev == button_hal_release_event)
        {
                btn = (button_hal_button_t *)data;

                // I switch the state of the system
                isSystemActive = !isSystemActive;

                // every time the button is pressed, I have to activate/deactivate the bulb
                res_bulb.trigger();
        }
}
```

Figure 4: CoAP Server main code

The function res_light.trigger() is executed every 5 seconds and triggers the event associated with the observable resource light and notify the observer, which in this case is the collector. In order to make it observable, the resource res_light has to be defined as follows.

```
EVENT_RESOURCE(res_light,
               "title=\"Light\";obs",
               res_get_handler,
               NULL,
               NULL,
               NULL,
               res_event_handler);
```

Figure 5: light resource definition

On this resource only get operations are performed, so the post-put-delete handlers are not defined. Instead, the res_event_handler is called every time the res_light.trigger() is executed by the main code of the CoAP server

which notifies the collector. Before doing this, the res_get_handler is executed. Inside this function, the behavior of the light sensor is emulated using a random function that generates an integer value between 5 lux less than the last generated value and 5 lux more than the last generated value, in order to have a realistic behavior, avoiding abrupt light variations. The code to generate fake light values is the following.

```
// I generate fake light rilevations which cannot be to far from the previous rilevation
light += rand()%(2*MAXIMUM_LIGHT_VARIATION+1)-MAXIMUM_LIGHT_VARIATION;

// the rilevation cannot be higher than 100 neither lower than 0 because it represents a percentage
light = light > 100 ? 100 : light;
light = light < 0 ? 0 : light;
```

Figure 6: light value generation

Then, a JSON message is created and notified to the collector. The structure of the JSON message is the following.

```
{
    "light": 34,
    "node_id": 3
}
```

Figure 7: measured light JSON message

The previous resource manages the luminosity sensing, while the bulb resource acts as an actuator in charge of turning the light on or off in a specific room. The commands to this resource are sent by the collector. The res_bulb is structured in this way:

```
EVENT_RESOURCE(res_bulb,
         "title=\"Bulb: ?POST/PUT mode=off|low|medium|high\";rt=\"bulb\"",
         NULL,
         res_post_put_handler,
         res_post_put_handler,
         NULL,
         res_event_handler);
```

Figure 8: bulb resource definition

Unlike res_light, this resource is not observable and can handle only post and put requests. The res_event_handler is called every time the res_bulb.trigger() is executed by the main code of the CoAP server, namely when the user presses the relative switch. In particular, this function activates the logic to handle lighting if the system was inactive and deactivates it vice versa. The behavior of the light bulb is emulated by the led interface. The res_post_put_handler extracts the post variable, which can be either “low”, “medium”, “high” or “off”, and implements the following logic:

- If “off” is received, than all lights are switched off
- If “low” is received, than green light is switched on, yellow and red lights are switched off
- If “medium” is received, than green and red lights are switched on, yellow light is switched off
- If “high” is received, than all lights are switched on

In general, this message represents the light level to be set accordingly to the current luminosity situation measured by the light resource.

4. Java collector

The collector is entirely implemented in Java. It first initializes the CoAP handler and the MQTT handler, then it interacts with the user by means of the menu. In this menu are listed all collector functionalities, including the possibility to enable the textual log to display data coming from sensors in real time.

Those things are performed in the main contained in the Application file, which also includes the actual code to print the menu and to receive and handle user inputs.

To actually manage MQTT data and CoAP data, respectively the classes MQTTManager and CoAPManager have been defined. They also contain code to set and get thresholds and to obtain last measured values.

CoAP Handler

For what concerns the CoAP network, in order to be periodically updated, the collector establishes an observing relation with all the 4 sensors towards respective light resources. This is performed at the application bootstrap. The following code shows an example for a generic sensor with a generic connectionURI URI.

```
CoapClient client = new CoapClient(connectionURI);
CoapObserveRelation coapObserveRelation = client.observe(
                new CoapHandler()
                {
                        public void onLoad(CoapResponse response)
                        {
                                //parsing
                                JSONParser parser = new JSONParser();
                                JSONObject jsonObject = null;
                                try
                                {
                                        jsonObject = (JSONObject) parser.parse(response.getResponseText());
                                } catch (ParseException e)
                                {
                                        e.printStackTrace();
                                }
                                if(jsonObject != null)
                                {
                                        long light = (Long) jsonObject.get("light");
                                        long node_id = (Long)jsonObject.get("node_id");
                                        rooms[(int)(node_id-2)].setCurrentLuminosity(light);

                                        // current datetime creation
                                        SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
                                        Date d = new Date();
                                        String[] tokens = dateFormat.format(d).split(" ");
                                        String date = tokens[0];
                                        String time = tokens[1];

                                        checkLuminosity(light,node_id);

                                        if(isTextualLogEnabled)
                                                System.out.println("date: " + date + ", time: " + time + ", light: " + light + "node_id: " + node_id +
                                                        ", room: " + ROOM_NAMES[(int)(node_id-2)]);
                                        storeData(light, node_id, date, time);
                                }
                        }
                        public void onError()
                        {
                                System.out.println("----Lost connection----");
                        }
                }
);
```

Figure 9: light data management on the collector

Every time an updating is received, the onLoad function is executed. Here, the JSON message is shown in output if the textual log has been enabled and then parsed. Then, a control logic is executed by the function checkLuminosity and finally the data are stored in the MySql database. The control logic checks in which range measured luminosity falls into and sends a command to switch on only appropriate leds according to what specified before on the specific sensor. To this aim, a post request is issued towards the relative sensor node, on the bulb resource. The following code shows an example for a generic sensor with a generic actuatorURI URI.

```
String actuatorURI ="coap://[fd00::20"+node_id+":"+node_id+":"+node_id+":"+node_id+"]/"+endpointBulb;

String postPayload="";
if(light>rooms[(int)(node_id-2)].getHighLuminosityThreshold())
        postPayload="off";

else if(light>rooms[(int)(node_id-2)].getMediumLuminosityThreshold())
        postPayload="low";

else if(light>rooms[(int)(node_id-2)].getLowLuminosityThreshold())
        postPayload="medium";

else
        postPayload="high";

client = new CoapClient(actuatorURI);
response = client.post("mode="+postPayload,MediaTypeRegistry.TEXT_PLAIN);
```

Figure 10: collector code for issuing commands to CoAP sensors

MQTT Handler

The collector, subscribed to the broker on the topic temperature, executes the following code every time a measurement is published on that topic.

```
public void messageArrived(String topic, MqttMessage message) throws Exception
{
        String json_message = new String(message.getPayload());

        //parsing
        JSONParser parser = new JSONParser();
        JSONObject jsonObject = null;
        try
        {
                jsonObject = (JSONObject) parser.parse(json_message);
        } catch (ParseException e) {
                e.printStackTrace();
        }
        if(jsonObject != null)
        {
                long temperature = (Long) jsonObject.get("temperature");

                // get current date
                SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
                Date d = new Date();
                String[] tokens = dateFormat.format(d).split(" ");
                String date = tokens[0];
                String time = tokens[1];

                if(isTextualLogEnabled)
                        System.out.println("date:"+date+",time:"+time+"temperature:"+temperature);

                temperatureControl(date,time,temperature);
                storeMqttData(temperature,date,time);

        }

}
```

Figure 11: collector code to receive data from MQTT sensor

Once the JSON payload of the message is extracted, it is parsed using a specific Java library. After getting the current date and time, the measurement is shown in output to the user and then stored in the database by the function storeMqttData.

The control logic checks the temperature and sets the mode accordingly.

```
currentTemperature = temperature;

if(temperature > maxTemperature)
        new_mode = -1; // decrease the temperature
else if(temperature < minTemperature)
        new_mode = 1; // increase the temperature
else
        new_mode = 0; // switch off the heater and the cooler
```

Figure 12: mode setting for MQTT network

If the mode is different from the last one, then a new command is sent to the actuator.

```
// if we don't have to change mode, we don't send any message to the sensor
if(new_mode == mode) return;

// I update the mode
mode = new_mode;
switch(mode)
{
        case 1:
                command = "up";
                break;
        case -1:
                command = "down";
                break;
        default:
                command = "off";
}
//publish on topic 'actuator'
try
{
        MqttMessage message = new MqttMessage(command.getBytes());
        mqttClient.publish(publisher_topic, message);
}catch(MqttException me)
{
        me.printStackTrace();
}
```

Figure 13: collector code to issuing commands to MQTT sensors

5. Simulation

The simulation has been performed first deploying CoAP and MQTT sensors, respectively through Cooja and testbed sensors, then starting collector.

CoAP Network Deployment

The simulation has been performed through Cooja, in which CoAP sensors have been deployed as follows.

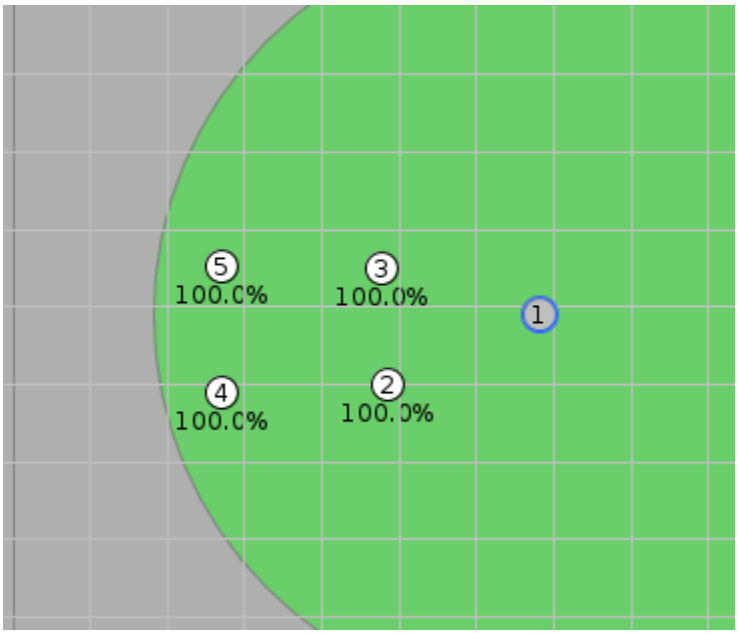


Figure 14: CoAP network deployment

The deployment is composed of 5 Cooja sensors. The mote sensor number 1 is the border router, then we have one sensor for each room: kitchen (mote number 2), living room (mote number 3), bedroom (mote number 4), and bathroom (mote number 5). At the beginning of the simulation, the network requires some time to build the RPL's DODAG. After this, the collector starts its execution establishing observing relations with the 4 CoAP servers.

MQTT Network Deployment

The MQTT network deployment has been performed using a testbed sensor. In particular, the sensor acts as a border router on the port /dev/ttyACM58, while on the port /dev/ttyACM7 acts as both the temperature sensor and actuator. To make possible communication between the mqtt-client and the collector, also Mosquitto has to be runned. What is displayed on the border router and on the MQTT client is shown below.

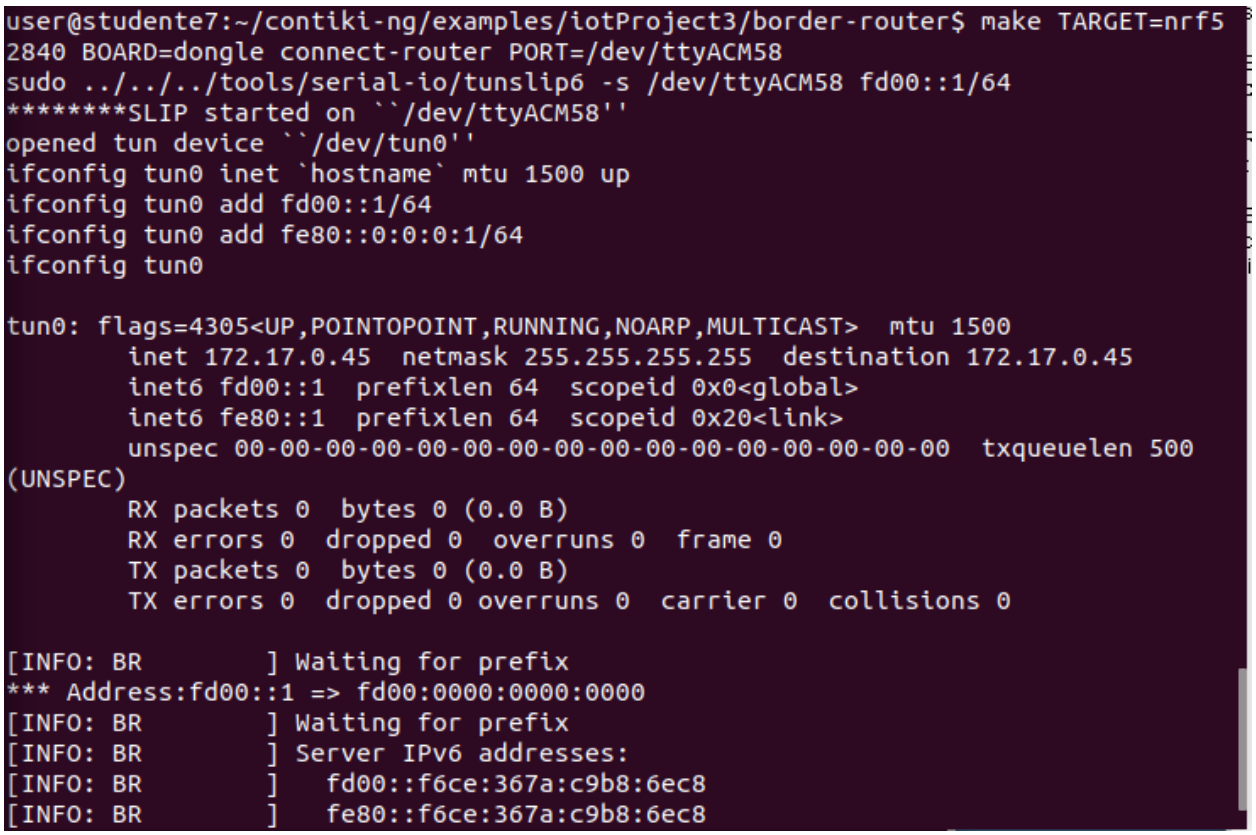


Figure 15: border router deployment on testbed

```
user@studente7:~/contiki-ng/examples/iotProject3/Mqtt-sensors$ make TARGET=nrf52840 BOARD=dongle login PORT=/dev/ttyACM7
rlwrap ../../../tools/serial-io/serialdump -b115200 /dev/ttyACM7
connecting to /dev/ttyACM7 [OK]
Connecting!
Application has a MQTT connection
Subscribing!
Application is subscribed to topic successfully
{"temperature":21}
{"temperature":21}
{"temperature":19}
{"temperature":18}
{"temperature":16}
{"temperature":17}
{"temperature":16}
{"temperature":14}
{"temperature":14}
{"temperature":14}
{"temperature":13}
{"temperature":15}
```

Figure 16: MQTT client deployment on testbed

In particular, the MQTT client shows the last measured temperature by the sensor.

Java Collector

When the collector is executed it first shows a menu to the user, then it receives user commands and performs relative actions.

The menu appears as follows.

```
Commands list:
       setTempRange [lower_bound] [upper_bound] --> set range of allowed temperatures expressed in Celsius degrees
       setMinTemp [lower_bound] --> set the minimum temperature allowed expressed in Celsius degrees
       setMaxTemp [upper_bound] --> set the maximum temperature allowed expressed in Celsius degrees
       setLuminosityThresholds [low_luminosity_value] [medium_luminosity_value] [high_luminosity_value] {room_name}--> set all luminosity thresholds expressed in lux. If room name is not specified thresh
olds are set for all rooms
       setLowLumThreshold [luminosity_value] [room_name] --> set the low luminosity threshold expressed in lux for the specified room
       setMediumLumThreshold [luminosity_value] [room_name] --> set the medium luminosity threshold expressed in lux for the specified room
       setHighLumThreshold [luminosity_value] [room_name] --> set the high luminosity threshold expressed in lux for the specified room
       getCurrTemp -- > get the last measured temperature expressed in Celsius degrees
       getCurrLum [room_name]--> get the last measured luminosity expressed in lux in the specified room
       getTempRange --> get range of allowed temperatures expressed in Celsius degrees
       getLumThresholds [room_name] --> get all luminosity thresholds expressed in lux for the specified room
       showTextLog --> data received from sensors are shown as textual log
       quit --> quit the program
Insert a command or type 'help'
>
```

Figure 17: collector menu

Basically the user can set several parameters used in the application logic and he can get both current parameters and last measurements. He can also show the textual log through which the Collector shows last measured values. If the user wants to issue a command to the Collector, it is suggested to first interrupt the textual log by typing ‘x’, in order to make

command typing easier. Whenever he wants the user can type “quit” to stop the application running.

Below are shown main functionalities the collector offers.

- Set temperature range: collector allows to set the temperature range used to assess if the measured temperature is either too or too high. This can be performed by updating both lower and upper bound with the unique command “setTempRange”, or separately with either “setMinTemp” or “setMaxTemp”.
- Set luminosity thresholds: collector allows to set luminosity thresholds used to assess in which range measured luminosity falls into. This can be performed by updating all those thresholds through the command “setLuminosityThresholds”, or separately with either “setLowLumThreshold”, “setMediumLumThreshold” or “setHighLumThreshold”.
- Get last measured values: collector allows to get last measured values of temperature and luminosity respectively through “getCurrTemp” and “getCurrLum” commands.
- Get actual parameters: collector allows to get actual parameters used for temperature and luminosity handling respectively through the “getTempRange” and “getLumThresholds”.

Use-case

An example has been performed to explain how the overall system works. First of all CoAP and MQTT networks have been deployed as previously said, and the collector has been executed as well.

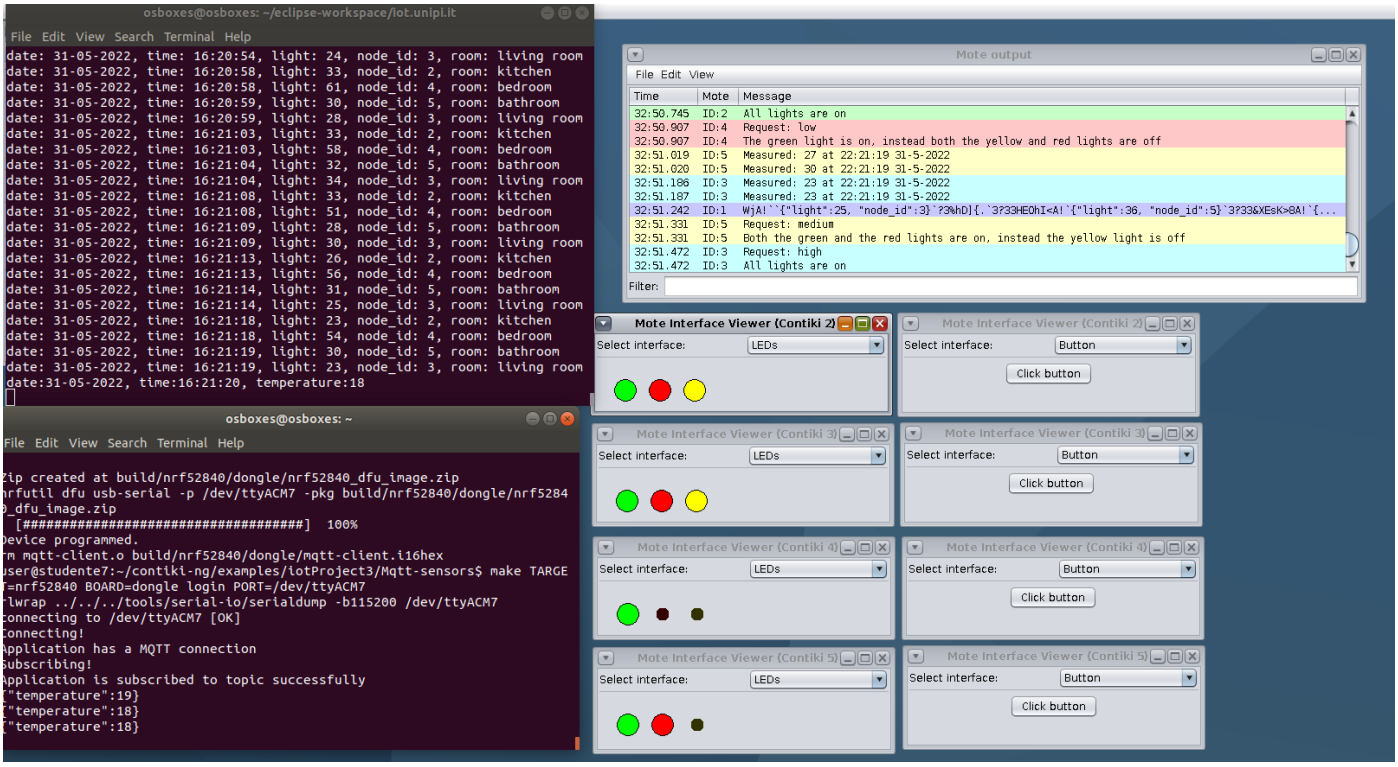


Figure 18: simulation start phase

- On the top-left we can see the collector execution with textual log enabled
- On the top-right we can see Cooja motes output
- On the bottom-left we can see the MQTT client execution
- On the bottom-right we can see Cooja motes leds and buttons

Considering default luminosity thresholds, we can see the current luminosity situation for each room:

- The kitchen (namely mote number 2) has a luminosity value of 23 lux, therefore all lights are switched on because this value is lower than the default low luminosity threshold for this room
- The living room (namely mote number 3) has a luminosity value of 23 lux, therefore all lights are switched on because this value is lower than the default low luminosity threshold for this room
- The bedroom (namely mote number 4) has a luminosity value of 54 lux, therefore only green light is switched on because this value is

both higher than the default medium threshold and lower than the default high threshold for this room

- The bathroom (namely mote number 5) has a luminosity value of 30 lux, therefore only green and red lights are switched on because this value is both higher than the default low threshold and lower than the default medium threshold for this room

For what concerns the temperature, this value is included in the default temperature range, so for the moment the Collector does not send any command to the MQTT client.

After 1 minute and 25 seconds the system changed as follows.

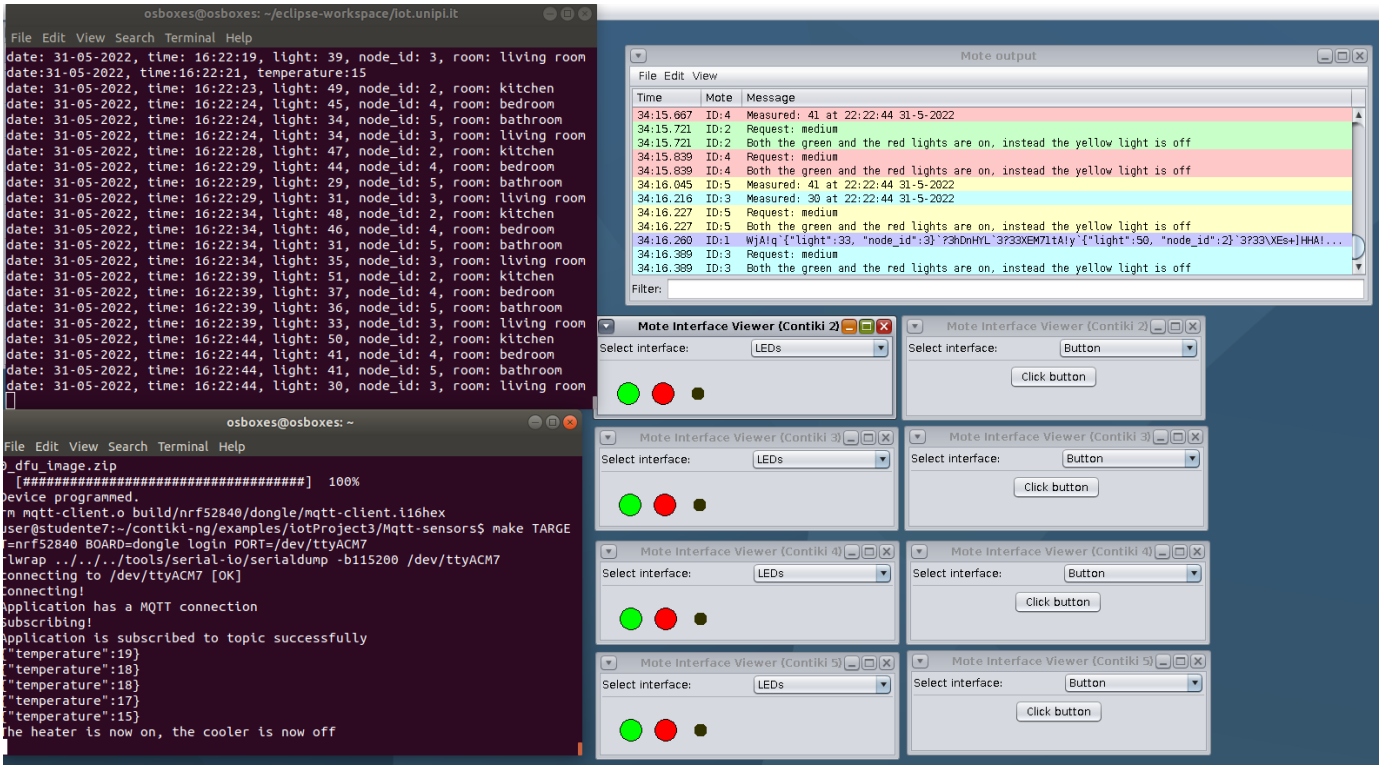


Figure 19: first change in MQTT actuator mode

The current luminosity situation for each room evolved as follows:

- The kitchen (namely mote number 2) has a luminosity value of 50 lux, therefore yellow light is switched off because this value is both higher

than the default low threshold and equal than the default medium threshold for this room

- The living room (namely mote number 3) has a luminosity value of 30 lux, therefore yellow light is switched off because this value is both higher than the default low threshold and equal than the default medium threshold for this room
- The bedroom (namely mote number 4) has a luminosity value of 41 lux, therefore the red light is switched on because this value is both higher than the default low threshold and lower than the default medium threshold for this room
- The bathroom (namely mote number 5) has a luminosity value of 41 lux, therefore no light has been switched because this value is both higher than the default low threshold and lower than the default medium threshold for this room

For what concerns the temperature, this value has dropped below the default temperature lower bound, so the heater has been activated.

After another 55 seconds the living room button was pressed simulating the user exiting from this room. The living room situation changed as follows.

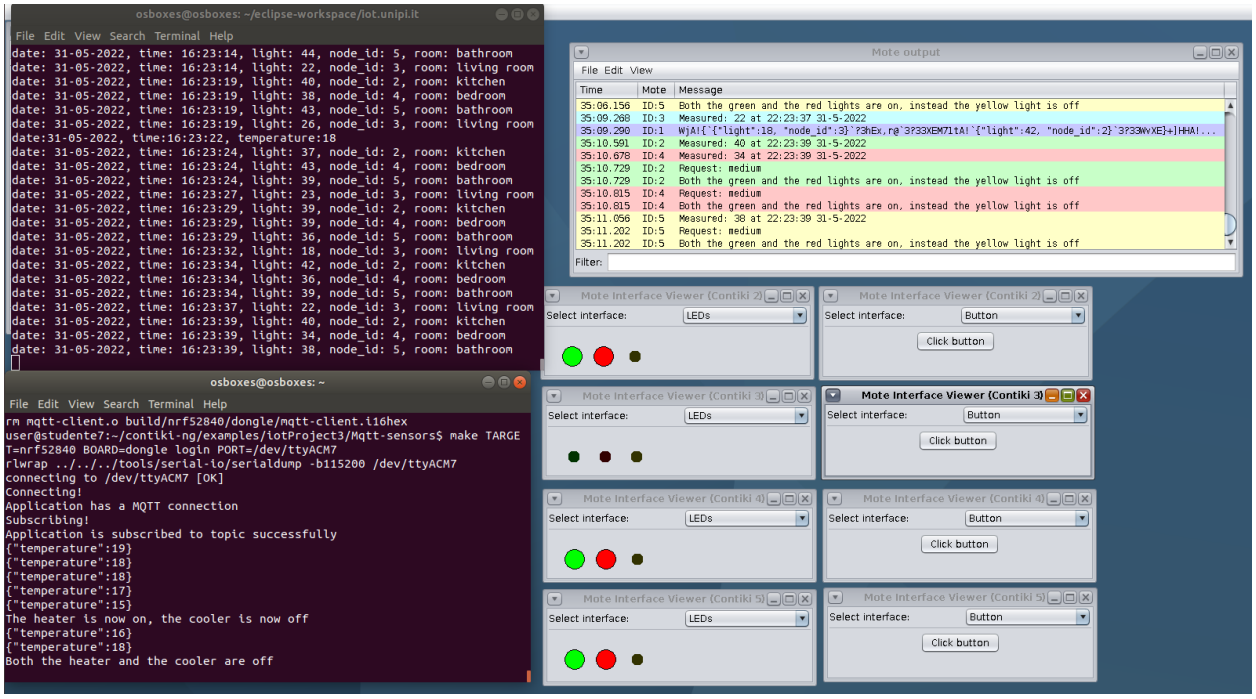


Figure 20: first button press

Regarding the living room luminosity state, we can see that despite the last measured luminosity value was 22 lux, all lights are switched off. Please notice that if the user did not press the button, then all lights would be on.

The button was pressed again to reactivate the luminosity control for the living room, simulating the user re-entering that room. Then some terminal commands were issued after disabling textual log in order to type the command in an easy manner.

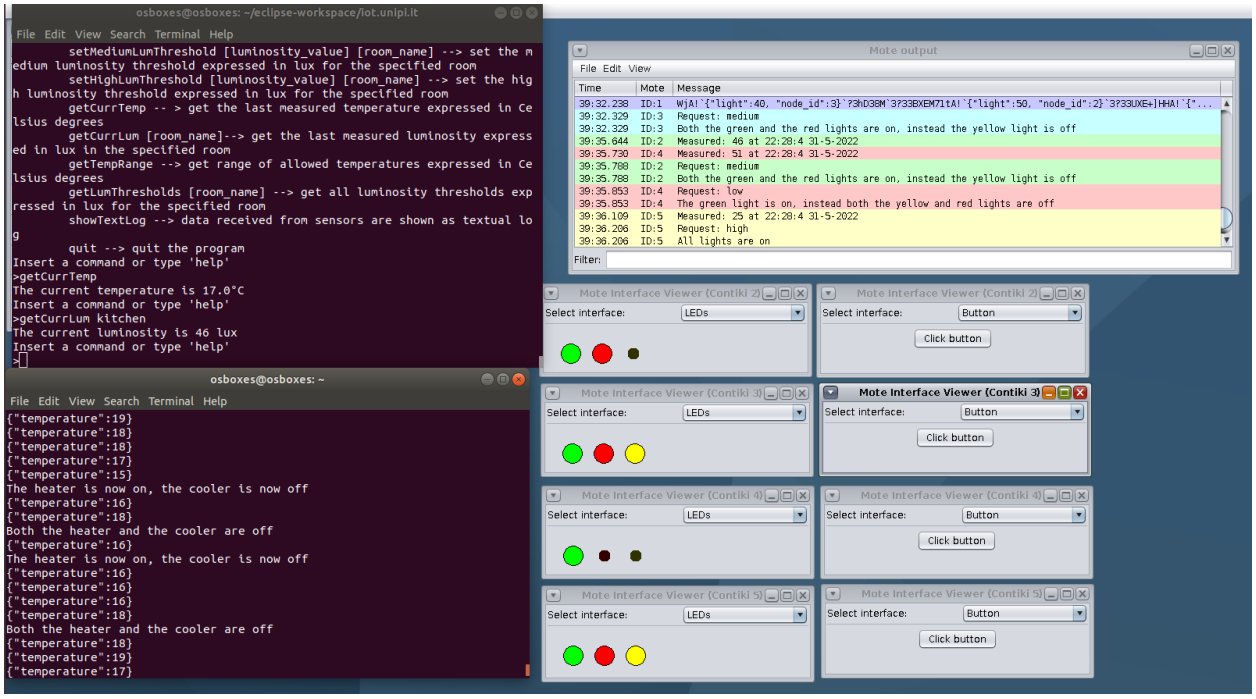


Figure 21: command issuing from user to collector through menu

In particular, the command “getCurrTemp” was issued to get the last measured temperature, that in this case is 17°. Then, the command “getCurrLum” was issued to get the last measured temperature in the specified room, that in this case is 46 lux for the kitchen.

At the end a command for changing a room threshold was issued.

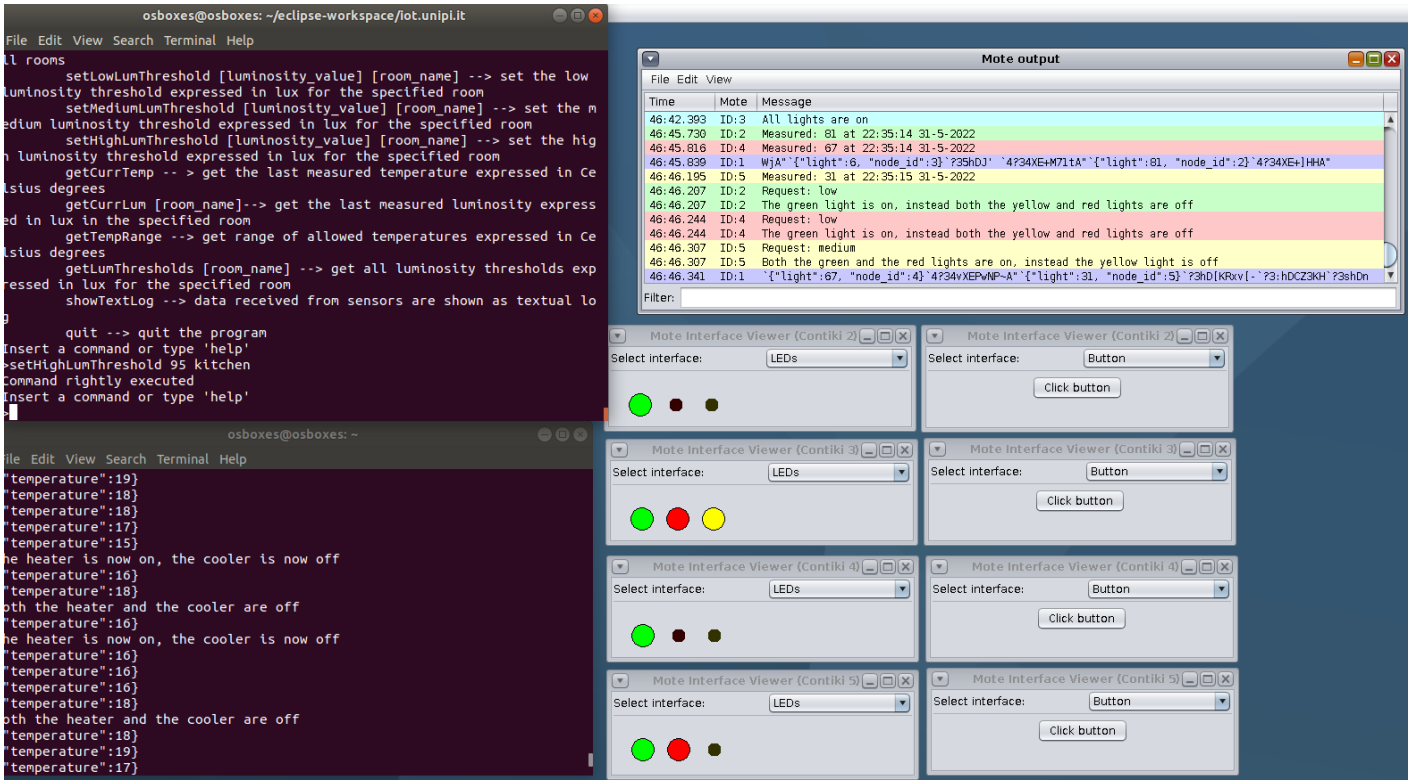


Figure 22: threshold changing

In particular, the command “setHighLumThreshold” was issued in order to update the high luminosity threshold for the specified room. In this case, a high luminosity threshold of 95 lux was specified for the kitchen. Please notice that if this threshold was not updated all lights would be off because the last measured luminosity value for the kitchen is 81 lux.

Furthermore, we can see that the update has been performed correctly by typing the command “getLumThresholds” specifying the kitchen, as follows.

```
Insert a command or type 'help'
>setHighLumThreshold 95 kitchen
Command rightly executed
Insert a command or type 'help'
>getLumThresholds kitchen
Luminosity thresholds are:
        low luminosity: 25 lux
        medium luminosity: 50 lux
        high luminosity: 95 lux
```

Figure 23: getting current thresholds

Regarding data recording, every time a new measurement is sent to the Collector by some sensors, it adds this measurement to the respective table of a MySql database. We have in particular two tables, one for MQTT data and one for CoAP data.

The MQTT table has as columns date, time and temperature value for each measurement, and it appears like this.

31-05-2022	16:11:56	19
31-05-2022	16:11:58	21
31-05-2022	16:12:01	21
31-05-2022	16:12:03	23
31-05-2022	16:12:06	22
31-05-2022	16:14:33	20
31-05-2022	16:15:04	19
31-05-2022	16:15:34	19
31-05-2022	16:16:05	20
31-05-2022	16:16:35	22
31-05-2022	16:17:06	24
31-05-2022	16:17:36	24
31-05-2022	16:18:07	23
31-05-2022	16:20:19	19
31-05-2022	16:20:50	18
31-05-2022	16:21:20	18
31-05-2022	16:21:51	17
31-05-2022	16:22:21	15
31-05-2022	16:22:52	16
31-05-2022	16:23:22	18
31-05-2022	16:23:53	16
31-05-2022	16:24:23	16
31-05-2022	16:24:54	16
31-05-2022	16:25:24	16
31-05-2022	16:25:55	18
31-05-2022	16:26:26	18
31-05-2022	16:26:56	19
31-05-2022	16:27:27	17

Figure 24: MQTT table in MySql

Instead, the CoAP table has as columns date, time, light value, node id and correspondent room for each measurement, and it appears like this.

15-05-2022	06:01:47	46	5	bathroom
15-05-2022	06:01:48	50	3	living room
15-05-2022	06:01:48	50	3	living room
15-05-2022	06:01:48	45	5	bathroom
15-05-2022	06:01:48	52	3	living room
15-05-2022	06:01:49	45	4	bedroom
15-05-2022	06:01:53	44	4	bedroom
15-05-2022	06:01:53	46	2	kitchen
15-05-2022	06:01:53	47	5	bathroom
15-05-2022	06:01:53	54	3	living room
15-05-2022	06:01:57	41	2	kitchen
15-05-2022	06:01:58	41	4	bedroom
15-05-2022	06:01:58	46	5	bathroom
15-05-2022	06:01:58	50	3	living room
15-05-2022	06:02:02	44	2	kitchen
15-05-2022	06:02:03	38	4	bedroom
15-05-2022	06:02:03	49	5	bathroom
15-05-2022	06:02:03	52	3	living room
15-05-2022	06:02:07	47	2	kitchen
15-05-2022	06:02:08	39	4	bedroom
15-05-2022	06:02:08	46	5	bathroom
15-05-2022	06:02:08	57	3	living room
15-05-2022	06:02:12	47	2	kitchen
15-05-2022	06:02:13	44	4	bedroom
15-05-2022	06:02:13	45	5	bathroom
15-05-2022	06:02:13	60	3	living room
15-05-2022	06:02:17	42	2	kitchen
15-05-2022	06:02:18	46	4	bedroom
15-05-2022	06:02:18	44	5	bathroom
15-05-2022	06:02:18	65	3	living room
15-05-2022	06:02:22	40	2	kitchen
15-05-2022	06:02:23	51	4	bedroom
15-05-2022	06:02:23	47	5	bathroom
15-05-2022	06:02:23	69	3	living room

Figure 25: CoAP table in MySql

How to run the project

To run the project you have to put the folder *SmartHome* in *contiki-ng/examples*

First deploy the MQTT client on remote testbed, typing on terminal the following command:

sudo ssh -i keys -p 2007 user@iot.dii.unipi.it

Travel to contiki-ng/examples/SmartHome and set the PAN_ID to 7 for both the mqtt sensor and the border router inside the respective project-conf.h. Then enter in the border-router folder and deploy the border router with the following commands:

make TARGET=nrf52840 BOARD=dongle border-router.dfu-upload PORT=/dev/ttyACM58

make TARGET=nrf52840 BOARD=dongle connect-router PORT=/dev/ttyACM58

Start the mosquitto broker on another terminal with the following command:

sudo mosquitto -c /etc/mosquitto/mosquitto.conf

Open another terminal and in the mqtt-client folder type the following commands:

make TARGET=nrf52840 BOARD=dongle mqtt-client.dfu-upload PORT=/dev/ttyACM58

make TARGET=nrf52840 BOARD=dongle login PORT=/dev/ttyACM58

Now open Cooja, deploy first the border router and then the other 4 CoAP sensors/actuators. The border router must have a serial socket. Then, use the tunslip6 with the following command, executed in the border router's folder:

make TARGET=cooja connect-router-cooja

At this point the simulation can be started on Cooja.

To enable the communication between the Java Collector running locally and the MQTT Broker running on the testbed, run this command on the local VM:

ssh -L 1883:[fd00::1]:1883 -p 2031 -i keys user@iot.dii.unipi.it

To run the collector, travel to collector folder, then issue following commands:

mvn install

java -jar target/iot.unipi.it-0.0.1-SNAPSHOT.jar