

# Middleware for the IoT: Practical works report



# Practical Work 1 – MQTT

## MQTT: State of the art

- The typical architecture of an IoT system that is based on publishers and subscribers. For example, a temperature sensor (the publisher) can publish an information such as its temperature to a Broker under a certain topic (temperature in this case). Then, the subscribers can go and subscribe to the topic and get the information.
- MQTT is based on the IP protocol TCP. It uses only one weak network bandwidth and its encrypted via TLS/SSL. It facilitates the connections as well as the disconnections and allow many users at the same time.
- There are two different variants of MQTT. The main one that has been used the most is designed for TCP/IP networks. The second one is MQTT-SN that is supposed to work over UDP. However, this version is not extremely popular now, but this could change in the future with more IoT deployments.
- The different topics that will have to be used are the luminosity and the state of the switch. The following figures shows the different cases of publishers and subscribers.

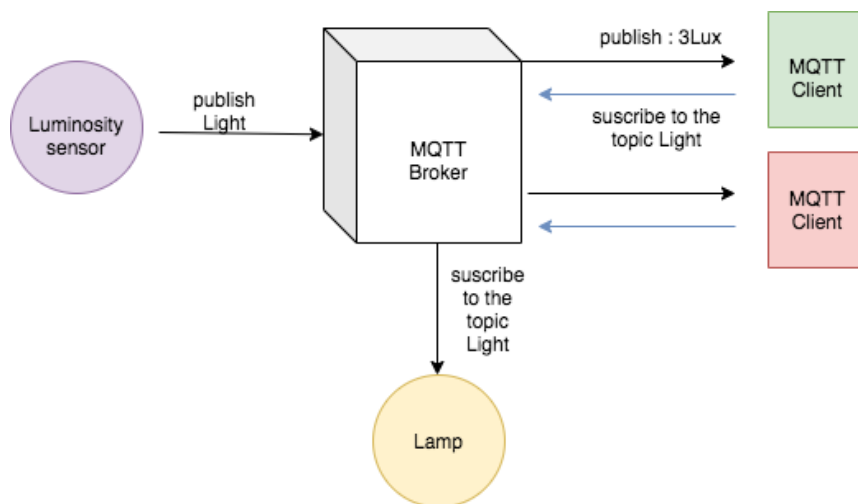


Figure 1: System with a luminosity sensor

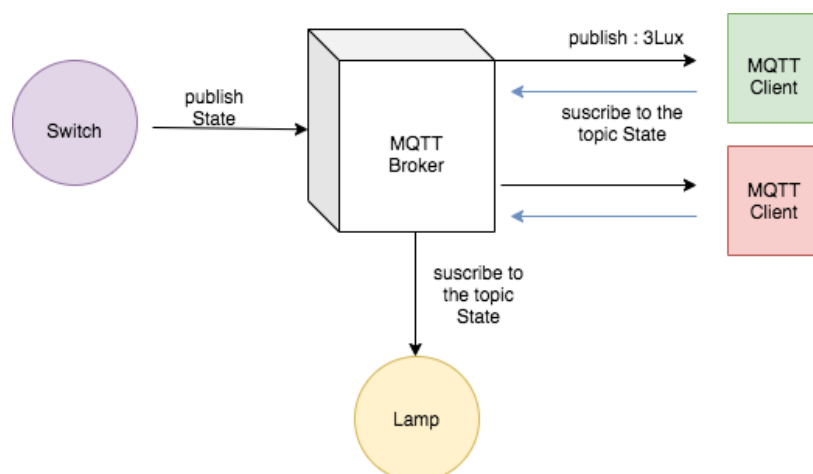
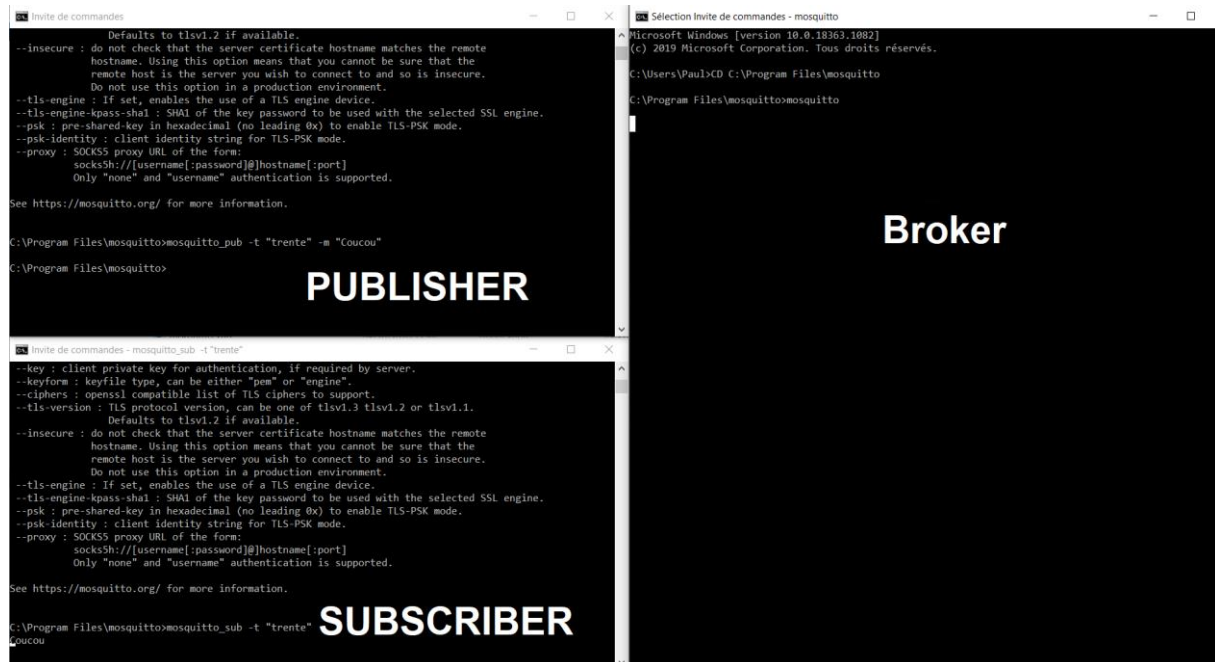


Figure 2 : System used manually

## MQTT: Test of the broker (Mosquitto)

After running Mosquitto, we opened two different terminals that are going to be our publisher and our subscriber, respectively. As you can see in the screenshot below, after publishing “Coucou” on the topic “trente”, our subscriber to this topic sees the publication.



The screenshot displays three terminal windows. The top-left window, titled 'Invite de commandes', shows the Mosquitto command-line options and the command to start the publisher: `C:\Program Files\mosquitto>mosquitto_pub -t "trente" -m "Coucou"`. The bottom-left window, also titled 'Invite de commandes', shows the command to start the subscriber: `C:\Program Files\mosquitto>mosquitto_sub -t "trente"`, followed by the output 'Coucou'. The right window, titled 'Sélection Invite de commandes - mosquitto', shows the Mosquitto broker running: `C:\Program Files\mosquitto>mosquitto`. The word 'PUBLISHER' is overlaid on the top-left window, and 'SUBSCRIBER' is overlaid on the bottom-left window. The word 'Broker' is overlaid on the right window.

## MQTT: Creation of an IT device with the nodeMCU board that uses MQTT communication

Presentation of the board: NodeMCU is an open-source IoT platform. For our application, the programming language used is Arduino C/C++. The board provides twelve inputs and outputs pins. This product is known to be easy to use and many people use it to develop their first application.

Because of the actual pandemic, we could not finish to develop an application on a real board. The last thing we did was to implement the Arduino program that is supposed to connect the device and the broker. In the next figures, you can see what the modifications we made to the original program example.

```

void setup() {
    // Setup hardware serial for logging
    Serial.begin(HW_UART_SPEED);
    while (!Serial);

    // Setup WiFi network
    WiFi.mode(WIFI_STA);
    WiFi.hostname("ESP " MQTT_ID);
    WiFi.begin("Cisco38658", "");
    LOG_PRINTFLN("\n");
    LOG_PRINTFLN("Connecting to WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        LOG_PRINTFLN(".");
    }
}

void loop() {
    // Check connection status
    if (!mqtt->isConnected()) {
        // Close connection if exists
        network.stop();
        // Re-establish TCP connection with MQTT broker
        LOG_PRINTFLN("Connecting");
        network.connect("192.168.1.114", 1883);
        if (!network.connected()) {
            LOG_PRINTFLN("Can't establish the TCP connection");
            delay(5000);
            ESP.reset();
        }
        // Start new MQTT connection
        MqttClient::ConnectResult connectResult;
        // Connect
        {
            MQTTPacket_connectData options = MQTTPacket_connectData_initializer;
            options.MQTTVersion = 4;
            options.clientID.cstring = (char*)MQTT_ID;
            options.cleansession = true;
            options.keepAliveInterval = 15; // 15 seconds
            MqttClient::Error::type rc = mqtt->connect(options, connectResult);
            if (rc != MqttClient::Error::SUCCESS) {
                LOG_PRINTFLN("Connection error: %i", rc);
                return;
            }
        }
    }
}

```

Figure 3: Modification made to the example code

## Practical work 2 – oneM2M REST API

In this practical work, we interacted with the oneM2M RESTful architecture using Eclipse OM2M.

During this practical, we are going to learn how to use the oneM2M platform, to send requests with the PostMan client, and the monitoring application to watch every exchanged request. We are going to realize the scenario of a connected house, with three sensors, a SmartMeter to have information about the energy (kWh) consumption of the house, a LuminositySensor, to get the luminosity (Lux) in the house and finally, a TemperatureSensor to get the temperature (°C) in the house.

Before starting to send requests, you need to deploy the in-cse and mn-cse on the oneM2M platform.

### Create the sensors on oneM2M

In order to create a sensor on the platform, we need to make a post request to the in-cse URL : <http://127.0.0.1:8080/~in-cse>

Here is the format of the request to create a sensor, the only parameter we need to change is the name of the sensor, which is here “MY\_SENSOR”:

Champ	Valeur
URL	<a href="http://127.0.0.1:8080/~in-cse">http://127.0.0.1:8080/~in-cse</a>
Méthode	POST
En-tête	X-M2M-Origin: admin:admin Content-type: application/xml; ty=2
Corps	<pre>&lt;m2m:ae xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="MY_SENSOR"&gt;   &lt;api&gt;app-sensor&lt;/api&gt;   &lt;lbl&gt;Type/sensor Category/temperature Location/home&lt;/lbl&gt;   &lt;rr&gt;false&lt;/rr&gt; &lt;/m2m:ae&gt;</pre>

<input checked="" type="checkbox"/>	X-M2M-Origin	admin:admin
<input checked="" type="checkbox"/>	Content-Type	application/xml; ty=2

```
1 <m2m:ae xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="MY_SENSOR">
2
3   <api>app-sensor</api>
4
5   <lbl>Type/sensor Category/temperature Location/home</lbl>
6
7   <rr>false</rr>
```

Figure 4 : Creation of a sensor with a POST request on PostMan

After having sent this requests, we can see on the oneM2M tree structure the added sensors:

OM2M CSE Resource Tree

<http://127.0.0.1:8080/~in-cse>

- in-name
  - acp\_admin
  - SDT\_Home\_Monitoring\_Application\_ACP
  - ACP\_Device\_Admin\_1604999688593
  - SDT\_Home\_Monitoring\_Application
  - SDT\_IPE
  - MY\_SENSOR
  - SmartMeter
  - LuminositySensor
  - TemperatureSensor
- mn-name

acpi	AccessControlPolicyIDs
	/in-cse/acp-855184073
cst	1
csl	in-cse
	Supported resource types
	1
	2
	3
	4
	5
	9
	14
	15
	16
	17
	23
	28
poa	Point Of Access
	<a href="http://127.0.0.1:8080/">http://127.0.0.1:8080/</a>

Figure 5 : Screenshots of the new sensors on the oneM2M platform

Now we are going to create two containers for each sensor, one called DESCRIPTOR, to publish on it some details about technical parameters of the sensor, and the type of data it measures, and another one called DATA, on the one we will publish content instances, like the measures of a real sensor. In order to create such containers, you must write a POST request, changing the type into type 3, and of course, change the body of it with the corresponding parameters (just the name in this case):

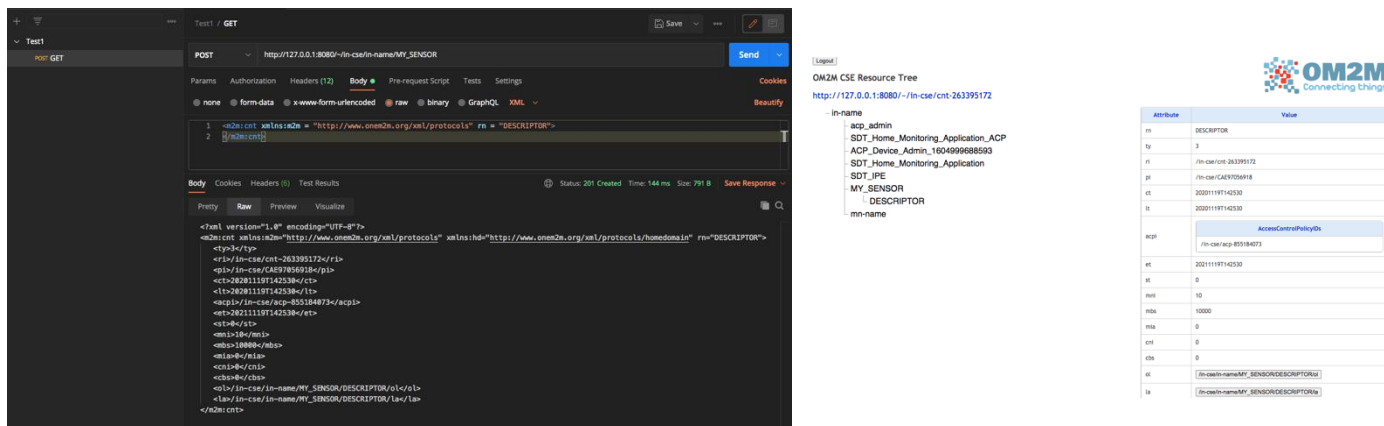


Figure 6 : PostMan requests for creating the containers

Now that these containers are created, we can publish a first content instance on each of them:

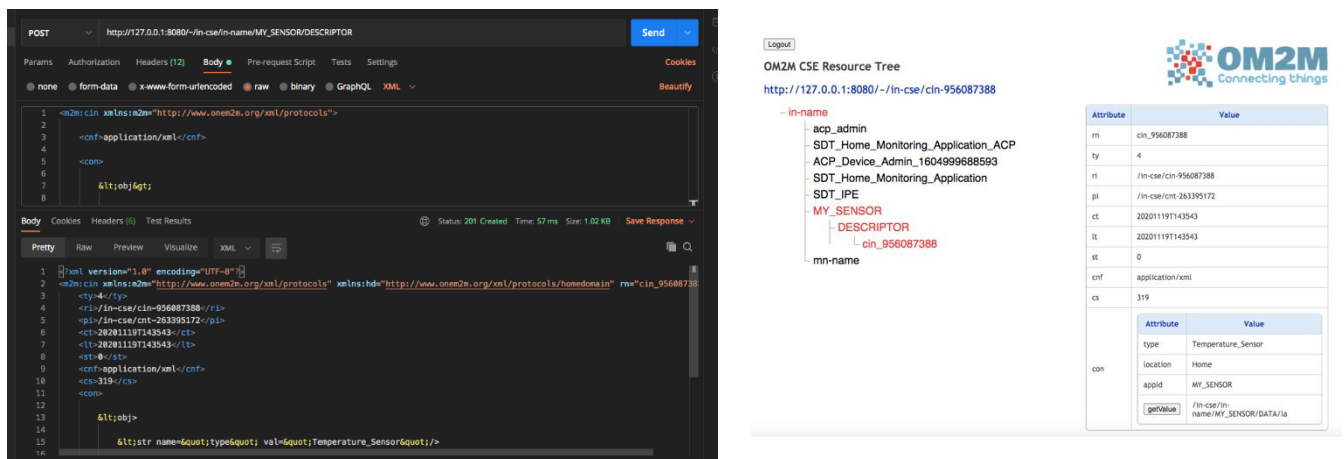


Figure 7 : PostMan requests to publish a content instance on a container

As always, we must make sure that the publication has well been made on the oneM2M platform:

The figure consists of two main parts. On the left, a screenshot of a Postman interface showing a POST request to the URL `http://127.0.0.1:8080/-/in-cse/in-name/MY_SENSOR/DESCRIPTOR`. The request body is an XML document. On the right, the OM2M CSE Resource Tree is displayed, showing a hierarchy where `MY_SENSOR` is a child of `SDT_IPE`, which is a child of `ACP_Device_Admin_1604999688593`. Below the tree, a table lists attributes and their values for the resource.

Attribute	Value
m	cin_956087388
ty	4
ri	/in-cse/cin-956087388
pi	/in-cse/cnt-263395172
ct	20201119T143543
lt	20201119T143543
st	0
cnf	application/xml
cs	319

Figure 8 : Confirmation of creation on PostMan and on oneM2M

From now on, our sensor publishing set up is ready. All we need is to subscribe to it.

First, we need to run the monitoring application “monitor.jar”, to listen to every new post from the sensor. We can see on the next screenshot that the monitoring application is listening on the port 1400.

```
[chiaraaubert:Downloads chiara$ java -jar monitor.jar
Starting server..
The server is now listening on
Port: 1400
Context: /monitor
```

We need to send a new post request, to create a subscribe and receive upcoming events about the sensors.

Here is the structure of the request we need to follow for this subscribing:

The figure consists of two main parts. On the left, a table lists the fields and values for a POST request. On the right, the OM2M CSE Resource Tree is displayed, showing a hierarchy where `DATA` is a child of `SDT_IPE`, which is a child of `ACP_Device_Admin_1604999688593`. Below the tree, a table lists attributes and their values for the resource.

Champ	Valeur
URL	<code>http://127.0.0.1:8080/-/in-cse/in-name/MY_SENSOR/DATA</code>
Méthode	POST
En-tête	X-M2M-Origin: admin:admin Content-type: application/xml;ty=23
Corps	<code>&lt;m2msub xmlns:m2m="http://www.onem2m.org/xml/protocols" m="SUB_MY_SENSOR"&gt;</code> <code>&lt;nu&gt;http://localhost:1400/monitor&lt;/nu&gt;</code> <code>&lt;nct&gt;2&lt;/nct&gt;</code> <code>&lt;/m2msub&gt;</code>

Figure 9 : Structure of the subscribe POST request

After repeating the subscribe operation to every sensor, we get this configuration on the oneM2M platform:

From now on, we can send new content instance requests to the data topic, as if the sensor were publishing data, and we have a notification with the data on the monitoring application like this:

Figure 10 shows the final publishing-subscribing set up. On the left, a tree view displays the configuration hierarchy. On the right, a table lists the attributes and their corresponding values.

Attribute	Value
rn	DATA
ty	3
ri	/in-cse/cin-845206916
pi	/in-cse/CAE503574314
ct	20201119T151339
lt	20201119T151339
acpi	AccessControlPolicyDs
et	20211119T151339
st	1
min	10
mb	10000
ml	0
cnf	1
cs	201
oi	/in-cse/in-name/TemperatureSensorDATAis
la	/in-cse/in-name/TemperatureSensorDATAis

Figure 10 : Final publishing-subscribing set up

At the left a screenshot of the publication of the TemperatureSensor, for a detected temperature of 30°C, and at the right, a capture of the notification from the subscriber, where in the parameter data, we also can read the data 30°C.

Figure 11 shows the publication of data and notification on the monitoring application. On the left, a table lists the attributes and their corresponding values. On the right, the XML notification received by the subscriber is displayed.

Attribute	Value										
rn	cin_292651817										
ty	4										
ri	/in-cse/cin-292651817										
pi	/in-cse/cin-845206916										
ct	20201119T164152										
lt	20201119T164152										
st	0										
cnf	message										
cs	201										
con	<table border="1"> <thead> <tr> <th>Attribute</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Category</td> <td>Temperature</td> </tr> <tr> <td>Data</td> <td>30</td> </tr> <tr> <td>Unit</td> <td>C</td> </tr> <tr> <td>Location</td> <td>Home</td> </tr> </tbody> </table>	Attribute	Value	Category	Temperature	Data	30	Unit	C	Location	Home
Attribute	Value										
Category	Temperature										
Data	30										
Unit	C										
Location	Home										

```

Received notification:
<?xml version="1.0" encoding="UTF-8"?>
<m2m:sgn xmlns:m2m="http://www.onem2m.org/xml/protocols" xmlns:hd="http://www.onem2m.org/xml/protocols/homedomain">
  <nev>
    <rep>
      <m2m:cin rn="cin_292651817">
        <ty>4</ty>
        <ri>/in-cse/cin-292651817</ri>
        <pi>/in-cse/cin-845206916</pi>
        <ct>20201119T164152</ct>
        <lt>20201119T164152</lt>
        <st>0</st>
        <cnf>message</cnf>
        <cs>201</cs>
        <con>
          <obj>
            <str name="Category" val="Temperature"/>
            <str name="Data" val="30"/>
            <int name="Unit" val="C"/>
            <int name="Location" val="Home"/>
          </obj>
        </con>
      </m2m:cin>
    </rep>
  </nev>
</m2m:sgn>

```

Figure 11 : Publication of data and notification on the monitoring application

Finally this, practical was extremely useful to learn how to use the platform and to understand how sensors, and monitor car interact together in order to make real smart systems.



## Practical Work 3 – IoT application prototyping with Node-RED

During this session, the main objective was to learn how to use Node-Red so we could develop an application faster, while using everything we learnt so far. During this session, we used the same configuration that we had in the previous practical work. Two lamps from the oneM2M interface were used during this session.

The first step was to display the state of the two lamps. As you can see in the figure below, the states displayed in the debug window match the real state of the lamps.

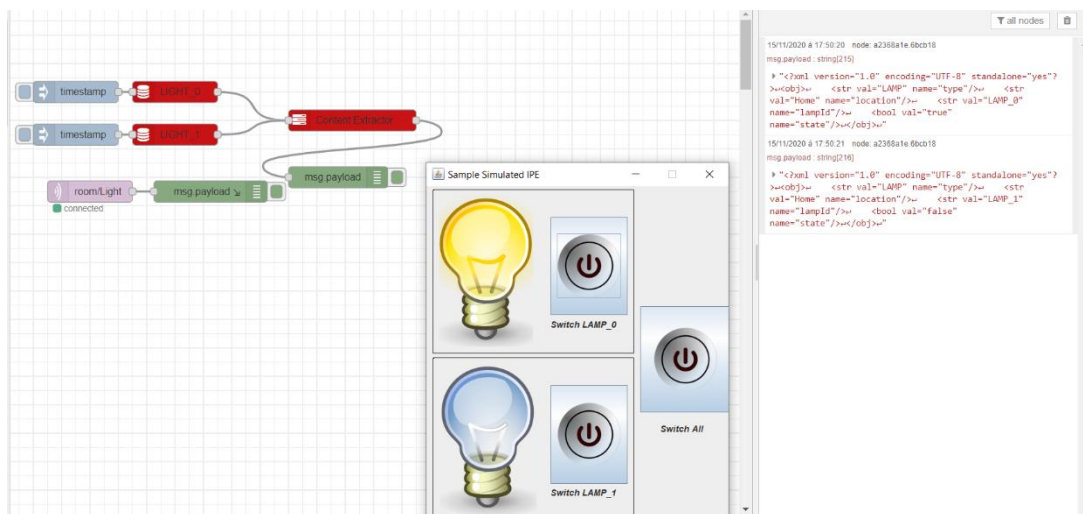
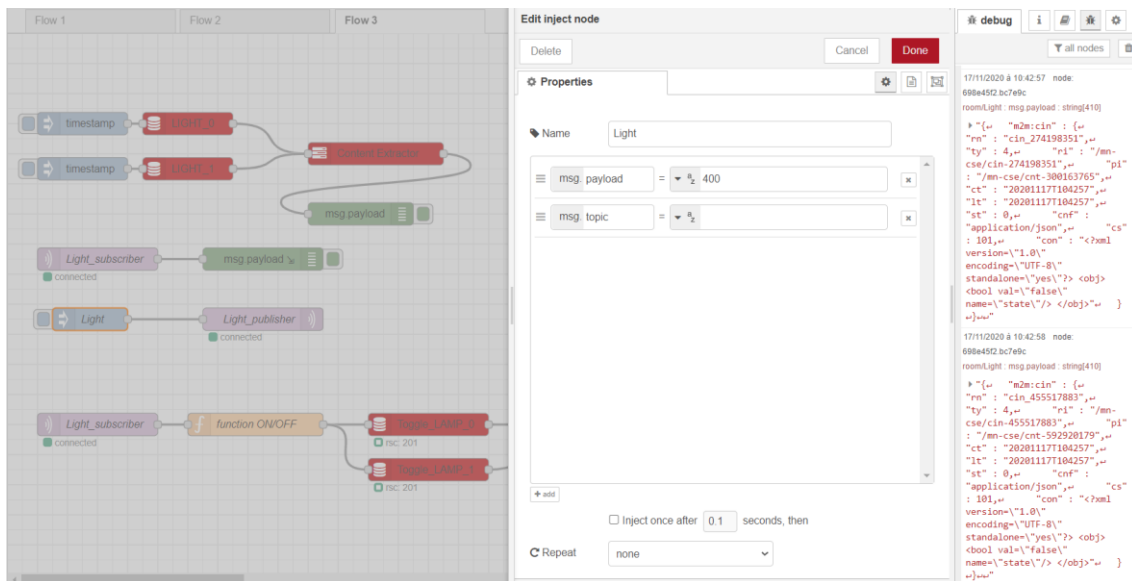


Figure 12: State of the lamps

We also used a “mqtt in” and a “mqtt out” node so we could publish the value of the light and turn on or off the lamps depending on a threshold that we fixed at 300. In the figure below, you can see the test that has been done with a value over the threshold meaning that the lamps should be turned off. The function block “function ON/OFF” is the one that dictates the lamps’ behaviour.



We cannot see it here, but the value of “Light” is also displayed in the debug window.

After that, we created two different actuators: the first one is supposed to turn on the first lamp and the second one is supposed to turn off the second lamp. This is using the same inject node but this time what we inject is the value, coded in XML, that the lamp should take. In the next figures, you can see the tests showing the state changes of the lamp when we used these actuators.

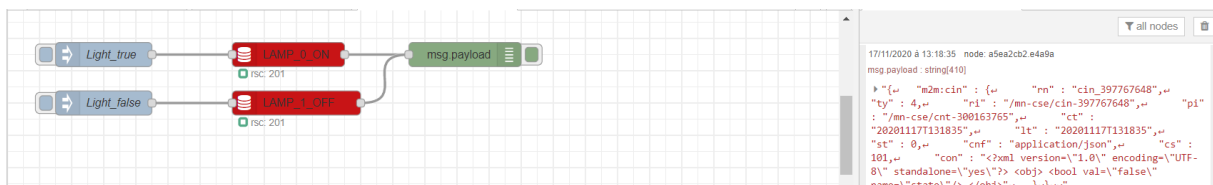


Figure 14: Turning off LAMP\_1



Figure 15: Turning on LAMP\_0

In the last part, we added the Dashboard nodes that allow us to use some components that are more graphic. For example, we could use a gauge to display the state of our lamps. Many other components can be used such as buttons, switches, or sliders.

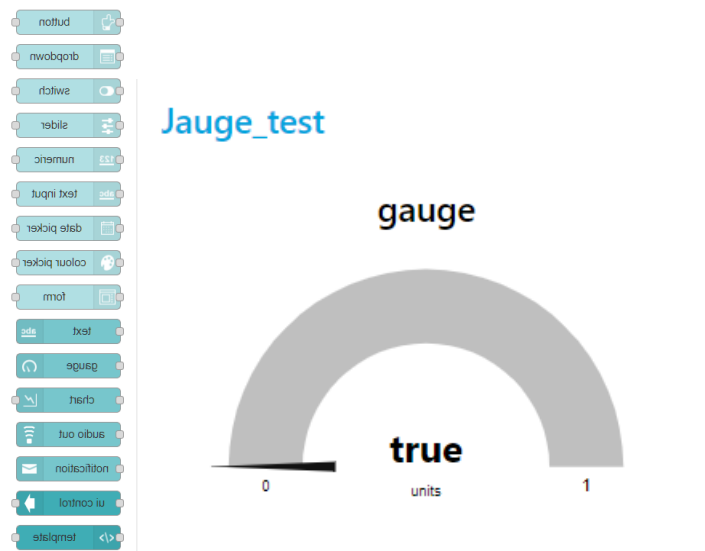


Figure 16: Dashboard nodes and gauge example

To be a little more relevant in our case, we put a switch to go along the gauge so you can see that both display the state of LAMP\_0.

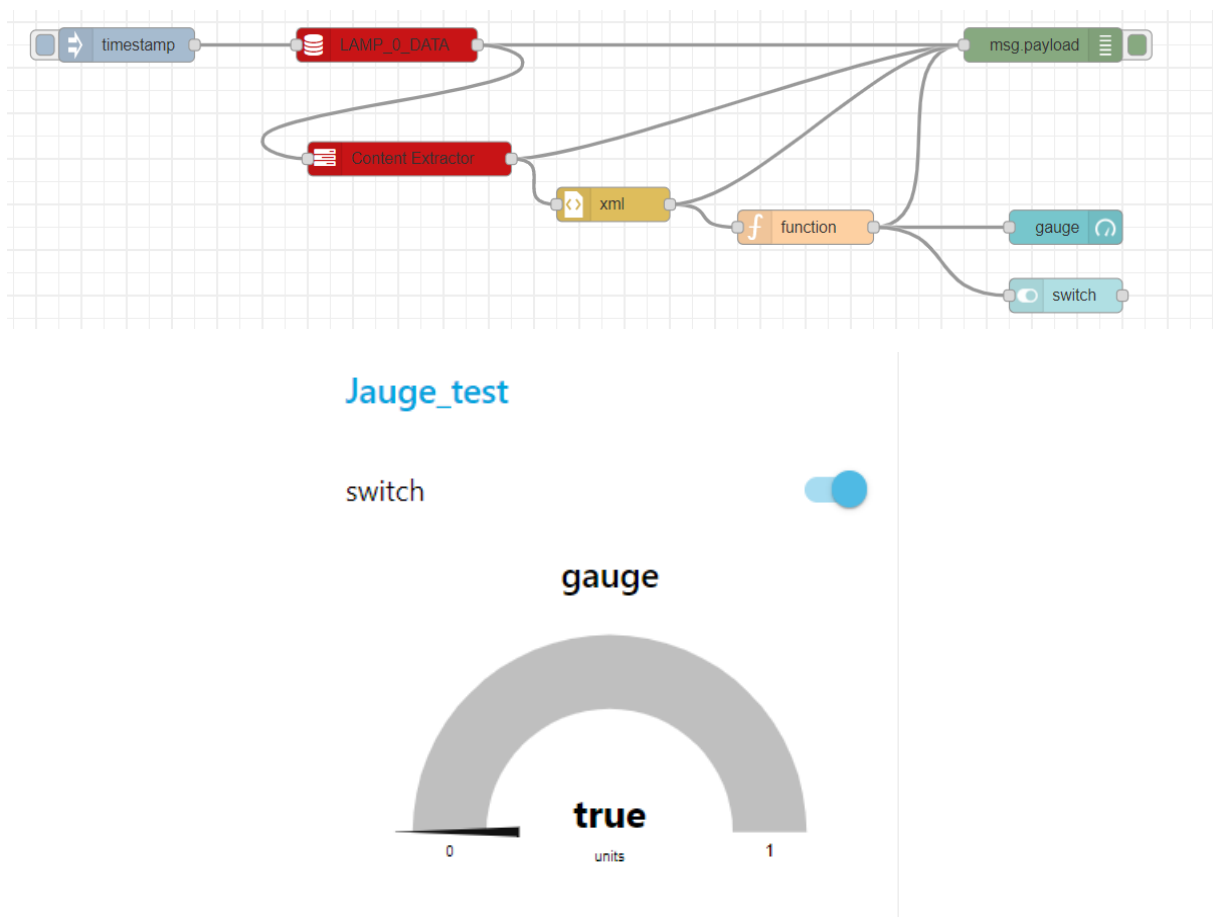


Figure 17: Switch and gauge together

## Conclusion

To conclude this project, the three practical works were highly informative and interesting. We learned a lot on each of them and we now have a better visualization of how IoT works, and how smart objects can communicate together in many fields of application like domotic, health, transports, agriculture... Due to the COVID-19 crisis, we lost a lot of time during the sessions to get the correct configuration of each of our working environment. Paul and I agreed on the fact that it would have been great to split the first practical into two sessions. For the TP1 and the TP3, we did not have enough time to go further enough in the technical points during the dedicated session, and it was sometimes hard to work this practical without tutor.