

**PRACTICAL WORK REPORT**

**MIDDLEWARE FOR THE INTERNET OF THINGS**

5 ISS - INSA Toulouse – Teacher : Ghada Gharbi

Students :

Ilhame Hadouch  
Lorine Pose  
Théo Rup-Pedoussat

# Table des matières

Lab 1 – MQTT protocol for IoT .....	1
Lab 3 – Interact with the oneM2M RESTful architecture.....	12
Lab 4 – Fast application prototyping for IoT .....	15

# Lab 1 – MQTT protocol for IoT

## Objective

The goal of this lab is to explore the capabilities of the MQTT protocol for IoT. To do that you will first make a little state of art about the main characteristics of MQTT, then you will install several software on your laptop to manipulate MQTT. Finally, you will develop a simple application with an IoT device (ESP8266) using the MQTT protocol to communicate with a server on your laptop.

## I. MQTT

Based on web resources and the previous course respond to those questions:

- What is the typical architecture of an IoT system based on the MQTT protocol?

The connected devices in the MQTT (Message Queuing Telemetry Transport) protocol are known as “clients,” which communicate with a server referred to as the “broker.” The broker handles the task of data transmission between clients.

Whenever a client (known as the “publisher”) wants to distribute information, it will publish to a particular topic, the broker then sends this information to any clients that have subscribed to that topic (known as “subscribers”).

The publisher does not need any data on the number or the locations of subscribers. In turn, subscribers do not need any data about the publisher. Any client can be a publisher, subscriber, or both. The clients are typically not aware of each other, only of the broker that serves as the intermediary. This setup is popularly known as the “pub/sub model.”

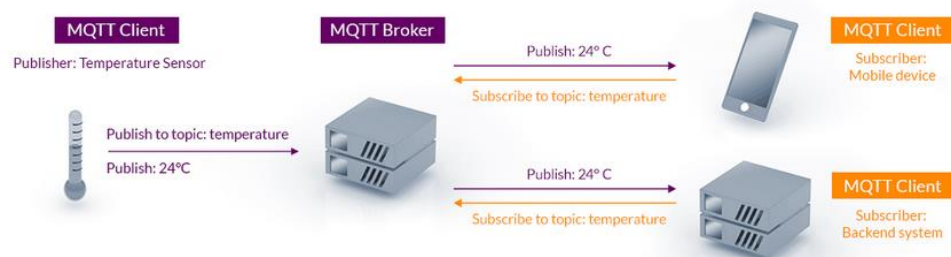


Figure 1 : Example of a MQTT architecture

- What is the IP protocol under MQTT? What does it mean in terms of bandwidth usage, type of communication, etc ?

The MQTT protocol runs over TCP/IP protocol, or other network protocols that provide ordered, lossless, bidirectional connections. It is light weight, open, simple, uses a small network bandwidth and is designed to be easy to implement. These characteristics make it ideal for use in many contexts : when a small code footprint is required, when we need to connect distant sites and the network bandwidth is limited...

MQTT is a protocol suitable for connections which include a mobility part between the client and the server which will store the data. This mode of operation allows to have clients that can have changing connections or variable IP address, such as sensors connected via mobile connections. The same goes for servers.

- What are the different versions of MQTT?

MQTT is an OASIS standard. The specification is managed by the OASIS MQTT Technical Committee. There are two different variants of MQTT and several versions :

- MQTT-SN v1.2
- MQTT v3.1
- MQTT v3.1.1
- MQTT v5.0

During this lab, we will use the classic MQTT protocol for TCP/IP networks. Contrarily, MQTT-SN (MQTT for Sensor Networks) is aimed at embedded devices on non-TCP/IP networks, such as Zigbee. MQTT-SN is a publish/subscribe messaging protocol for wireless sensor networks (WSN), with the aim of extending the MQTT protocol beyond the reach of TCP/IP infrastructure for Sensor and Actuator solutions.

- What kind of security/authentication/encryption are used in MQTT?

One of the characteristics of MQTT is the encryption of data by the TLS / SSL security protocol. It allows the management of disconnections and reconnections in a simplified way, making it useful for unstable connections. It is possible to secure a connection using a username and password to connect to the broker or for data exchange.

- Suppose you have devices that include one button, one light and light sensor. You would like to create a smart system for you house with this behaviour:
  - you would like to be able to switch on the light manually with the button
  - the light is automatically switched on when the luminosity is under a certain value

What different topics will be necessary to get this behaviour and what will the connection be in terms of publishing or subscribing?

For this application case, we would need 2 different topics :

- test/light\_sensor : this topic will receive the values coming from the light sensor (publisher 1)
- test/switch : this topic will receive the state (ON or OFF) coming from the switch button (publisher 2)

While the 2 publishers continuously send data to their own topic, the light (subscriber 1), which has subscribed to these 2 topics, continuously read the values from the light sensor and the switch. Depending on the values, the light will either switch on or switch off. The figure below describes this system architecture :

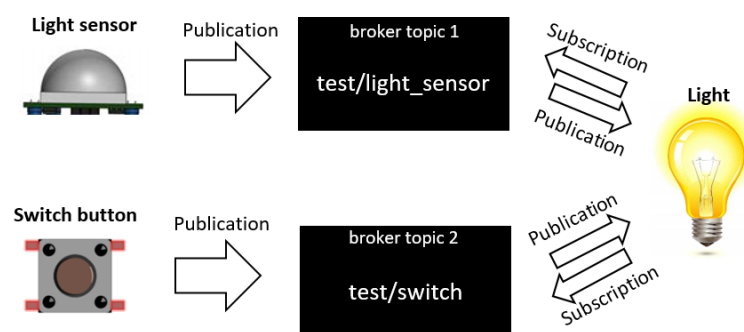


Figure 2 : Representation of our light regulation smart system architecture

## II. Install and test the broker

For this TP, we will use the mosquitto broker made by the eclipse opensource foundation (<https://mosquitto.org>). Mosquitto exists for many platforms: linux, windows, macOS, etc.

- Download and install this broker on your laptop.
- Run the mosquitto broker
- Test publish and subscribe with the command shell programs: mosquitto\_pub and mosquitto\_sub

At first, we have download and install the version 1.6.12 of mosquitto broker which is compatible with MQTT version 5.0, 3.1.1 and 3.1.

To run the broker with windows, we have opened a command prompt on mosquitto directory and wrote the command mosquitto:

```
C:\Program Files\mosquitto > mosquitto
```

To subscribe to a topic, we have opened another command prompt which aim is to display the messages received after the subscription. We have arbitrary decided to subscribe to the topic 'test' by writing the command line :

```
C:\Program Files\mosquitto > mosquitto_sub -t 'test'
```

The, we came back to the first command prompt to publish elements on the topic 'test' so that it would be displayed in the other command prompt who subscribed to that topic. We decided to send the message 'hello' by writing the command line :

```
C:\Program Files\mosquitto > mosquitto_pub -t 'test' -m 'hello'
```

Below, you can see the 2 command prompts after that process. Note that the message 'hello' has been successfully displayed by the command prompt who subscribed to the 'test' topic.

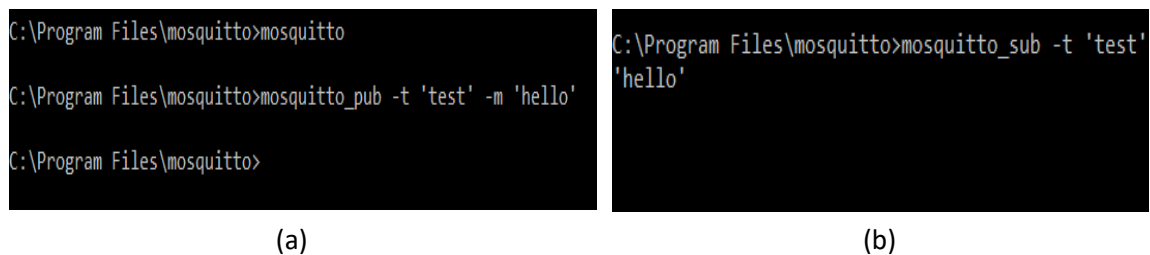


Figure 3 :

(a) Command prompt publishing the message in the topic 'test'

(b) Command prompt subscribing to the topic 'test' and waiting to display the messages sent on the topic

### III. Creation of an IoT device with the nodeMCU board that uses MQTT communication

During the TP, we will use the nodeMCU board based on ESP8266.

- a. Give the main characteristics of NodeMCU board in term of communication, programming language, Inputs/outputs capabilities

NodeMCU development board is composed of a low-cost ESP8266 WiFi chip developed with TCP/IP protocol. ESP8266 chip has GPIO pins, serial communication protocol, etc. features on it. Indeed, NodeMCU development board has Arduino like Analog (i.e. ADC0) and Digital (D0-D8) pins on its board. It supports serial communication protocols i.e. UART, SPI, I2C, etc. Using such serial protocols, we can connect it with serial devices like I2C enabled LCD display, Magnetometer HMC5883, MPU-6050 Gyro meter + Accelerometer, RTC chips, GPS modules, touch screen displays, SD cards, etc.

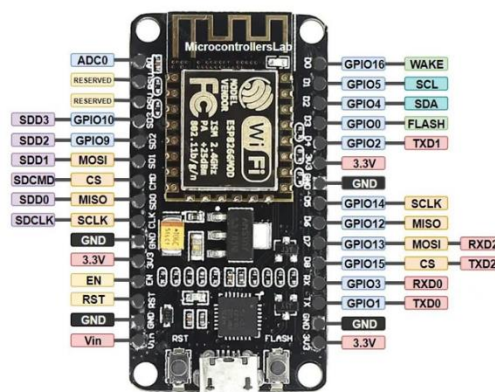


Figure 4 : NodeMCU development board with its pins

There is a programming language difference while developing an application for NodeMCU using ESPlorer IDE and Arduino IDE. We need to code in C/C++ programming language if we are using Arduino IDE for developing NodeMCU applications and Lua language if we are using ESPlorer IDE.

- b. Install Arduino IDE on your laptop
- c. Add the board NodeMCU 09 (ESP-12 module)
- d. Add the library (and necessary dependency) ArduinoMqtt by Oleg Kovalenko
- e. Based on examples in the library we will build our own application
  - Open the file in the menu: examples/arduinoMqtt/connectESP8266wificlient and have a look at the different parts of the code.
  - Modify the network characteristics to connect to the local wifi network(cisco38658 or you own wifi network with your smartphone)
  - Modify the address of the MQTT server to be able to connect to the broker on your laptop (remember that your laptop should be on the same wifi network)
  - Open the serial monitor on Arduino IDE and run your Arduino code, validate that the MQTT connection is done between your device and the broker
- f. Add a publish/subscribe behaviour in your device

- Open the file in menu: `examples/arduinoMqtt/PubSub`
- Extract the specific part: publish and subscribe and necessary declaration and put that in the previous example
- Open the serial monitor on Arduino IDE and run your Arduino code, using the command `mosquitto_pub` and `mosquitto_sub` validate that your device publishes values and can receive the result of subscription

Our device both publish and subscribe to 2 different topics :

Figure 5 : Publication of the message *Hello* each 30s (red arrows) on the topic “test/TEST-ID/pub”

Figure 6 : Reception of messages (red arrows) coming from the subscription to the topic "test/TEST-ID/sub"

## IV. Creation of the application

By using what you did on MQTT and the necessary sensors and actuators, program the application's light management behaviour through MQTT exchanges.

The objective here is to design an application as defined in the system architecture seen on Figure 1 with a NodeMCU board.

To do so, we have at our disposition an external light sensor and an external switch button. The light will be the LED pin of the NodeMCU board. The following wiring diagram shows the hardware part.

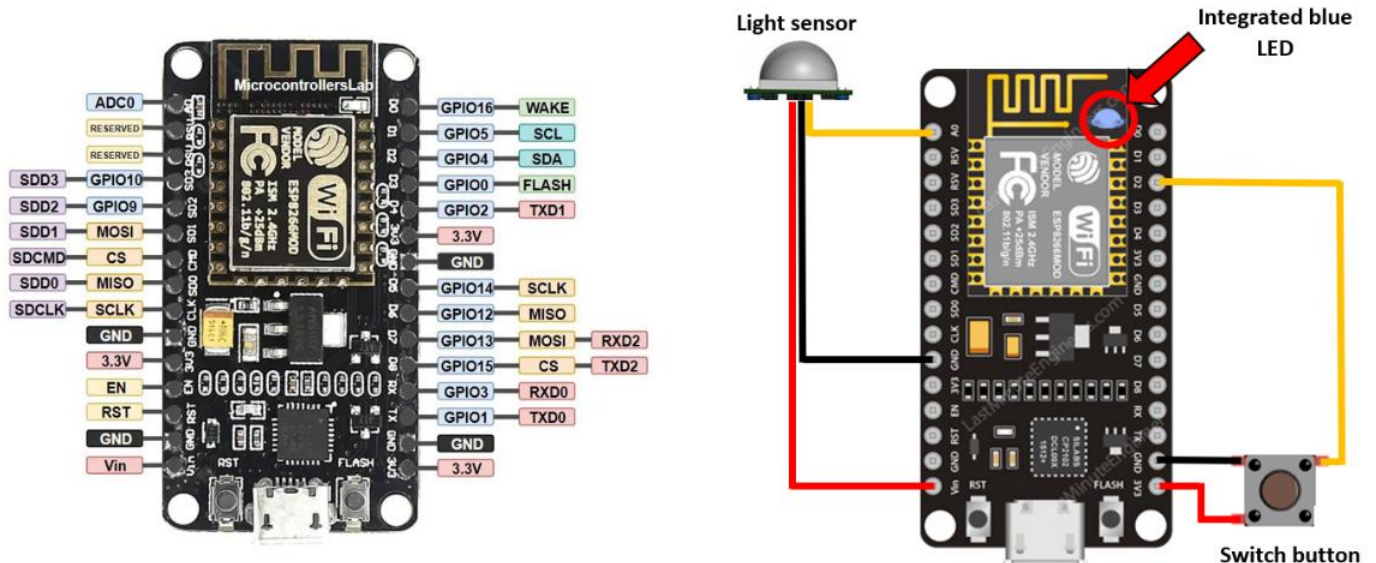


Figure 7 : Wiring diagram of our sensors on the NodeMCU board

For the Arduino program<sup>1</sup>, we have at first setup the private WiFi network and the MQTT client. In the loop part, we check the connection status at the beginning. For the first loop when the TCP connection with the MQTT broker mosquitto is not established, the program makes the MQTT connection and subscribe to both test/light\_sensor and test/switch topics.

When the connection has been established, the other loops will publish every 0.5s the values reported by the light sensor and the switch button respectively on the topics test/light\_sensor and test/switch.

Once the values are published on their topics, they are directly received by the card (because it also subscribes to these topics). With these informations about the light level and the switch button state, it is possible to know whether the LED needs to be switched on or switched off.

The following figure shows the values published on the 2 topics and the messages displayed in the Arduino serial monitor. The coloured arrows emphasise in the Arduino serial monitor the publication of data on the topic (arrow 1 and 2) and the reception of these data (arrows 3 and 4) from the topic. The history is shown on the 2 command prompts who subscribed to the 2 topics.

<sup>1</sup> Available on the appendices of that lab report





Figure 8 : Arduino serial monitor and history of the 2 topics when the program is running

With that program, we successfully managed to obtain the desired behaviour for our application. The tests of the different cases are described in the table below.





Switch button Light sensor	ON	
	OFF	
Luminosity above the limit level		
Luminosity under the limit level		

Figure 9 : Table resuming the behaviour of the system in the 4 situations

## Resources

[1] What is MQTT in IoT ?

<https://www.verypossible.com/insights/what-is-mqtt-in-iot>

[2] Retrouvez-vous dans la jungle des standards

<https://openclassrooms.com/fr/courses/5079046-mettez-en-place-une-architecture-pour-objets-connectes-avec-le-standard-onem2m/5079156-retrouvez-vous-dans-la-jungle-des-standards>

[3] MQTT Version 3.1.1 Plus Errata 01

<https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>

[4] MQTT specifications

<https://mqtt.org/mqtt-specification/>

[5] Introduction to NodeMCU

<https://www.electronicwings.com/nodemcu/introduction-to-nodemcu>

# Appendices

## Appendices 1 : Arduino code

```
#include <Arduino.h>
#include <ESP8266WiFi.h>

// Enable MqttClient logs
#define MQTT_LOG_ENABLED 1
// Include library
#include <MqttClient.h>

#define LOG_PRINTFLN(fmt, ...) logfln(fmt, ##__VA_ARGS__)
#define LOG_SIZE_MAX 128
void logfln(const char *fmt, ...) {
    char buf[LOG_SIZE_MAX];
    va_list ap;
    va_start(ap, fmt);
    vsnprintf(buf, LOG_SIZE_MAX, fmt, ap);
    va_end(ap);
    Serial.println(buf);
}

#define HW_UART_SPEED          115200L
#define MQTT_ID                "TEST-ID"
const char* MQTT_TOPIC_SWITCH = "test/switch";
const char* MQTT_TOPIC_LIGHT_SENSOR = "test/light_sensor";

static MqttClient *mqtt = NULL;
static WiFiClient network;

int light_sensor_pin = A0;
int led_pin = 2; //LED incorporated to the ESP8266 board
int switch_pin = 4;

int limit_level = 800;
int light_level = 0; //value given by the light sensor
const char* switch_state = "OFF";

..
// ===== Object to supply system functions =====
class System: public MqttClient::System {
public:

    unsigned long millis() const {
        return ::millis();
    }

    void yield(void) {
        ::yield();
    }
};

// ===== Setup all objects =====
void setup() {

    //attribution of the pins
    pinMode(led_pin, OUTPUT);
    digitalWrite(led_pin, LOW);
    pinMode(switch_pin, INPUT);

    // 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("Honor 9 Lite", "9b246ad538d8");
    LOG_PRINTFLN("\n");
    LOG_PRINTFLN("Connecting to WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        LOG_PRINTFLN(".");
    }
    LOG_PRINTFLN("Connected to WiFi");
    LOG_PRINTFLN("IP: %s", WiFi.localIP().toString().c_str());
}
```

```

// Setup MqttClient
MqttClient::System *mqttSystem = new System;
MqttClient::Logger *mqttLogger = new MqttClient::LoggerImpl<HardwareSerial>(Serial);
MqttClient::Network *mqttNetwork = new MqttClient::NetworkClientImpl<WiFiClient>(network, *mqttSystem);
//// Make 128 bytes send buffer
MqttClient::Buffer *mqttSendBuffer = new MqttClient::ArrayBuffer<128>();
//// Make 128 bytes receive buffer
MqttClient::Buffer *mqttRecvBuffer = new MqttClient::ArrayBuffer<128>();
//// Allow up to 2 subscriptions simultaneously
MqttClient::MessageHandlers *mqttMessageHandlers = new MqttClient::MessageHandlersImpl<2>();
//// Configure client options
MqttClient::Options mqttOptions;
//// Set command timeout to 10 seconds
mqttOptions.commandTimeoutMs = 10000;
//// Make client object
mqtt = new MqttClient(
    mqttOptions, *mqttLogger, *mqttSystem, *mqttNetwork, *mqttSendBuffer,
    *mqttRecvBuffer, *mqttMessageHandlers
);
}

// ===== Subscription callback =====
void processMessageLightSensor(MqttClient::MessageData& md) {
    const MqttClient::Message& msg = md.message;
    char payload[msg.payloadLen + 1];
    memcpy(payload, msg.payload, msg.payloadLen);
    payload[msg.payloadLen] = '\0';
    LOG_PRINTFLN(
        "Message arrived: qos %d, retained %d, dup %d, packetid %d, payload:[%s]",
        msg.qos, msg.retained, msg.dup, msg.id, payload
    );
    light_level = atoi((char*) payload);
}

void processMessageSwitch(MqttClient::MessageData& md) {
    const MqttClient::Message& msg = md.message;
    char payload[msg.payloadLen + 1];
    memcpy(payload, msg.payload, msg.payloadLen);
    payload[msg.payloadLen] = '\0';
    LOG_PRINTFLN(
        "Message arrived: qos %d, retained %d, dup %d, packetid %d, payload:[%s]",
        msg.qos, msg.retained, msg.dup, msg.id, payload
    );
    if ((String)payload == "OFF" && light_level >= limit_level) {
        digitalWrite(led_pin, HIGH); //LED switch off
    }
    else {
        digitalWrite(led_pin, LOW); //LED switch on
    }
}

// ===== Main loop =====
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("192.168.43.169", 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;
            }
        }
    }
}

```

```

// Subscription 1 : Light Sensor
{
    LOG_PRINTFLN("Subscription to test/light_sensor");
    MqttClient::Error::type rc = mqtt->subscribe(MQTT_TOPIC_LIGHT_SENSOR, MqttClient::QOS0, processMessageLightSensor);
    if (rc != MqttClient::Error::SUCCESS) {
        LOG_PRINTFLN("Subscribe error: %i", rc);
        LOG_PRINTFLN("Drop connection");
        mqtt->disconnect();
        return;
    }
}
//Subscription 2 : Switch
{
    LOG_PRINTFLN("Subscription to test/switch");
    MqttClient::Error::type rc = mqtt->subscribe(MQTT_TOPIC_SWITCH, MqttClient::QOS0, processMessageSwitch);
    if (rc != MqttClient::Error::SUCCESS) {
        LOG_PRINTFLN("Subscribe error: %i", rc);
        LOG_PRINTFLN("Drop connection");
        mqtt->disconnect();
        return;
    }
}
} else {
{
    //Publication 1 : Light Sensor
    LOG_PRINTFLN("Publication on test/light_sensor");
    light_level = analogRead(light_sensor_pin);
    LOG_PRINTFLN("Value : %i", light_level);
    String light_level_str = String(light_level, DEC);
    const char* bufLightLevel = light_level_str.c_str();
    MqttClient::Message messageLightLevel;
    messageLightLevel.qos = MqttClient::QOS0;
    messageLightLevel.retained = false;
    messageLightLevel.dup = false;
    messageLightLevel.payload = (void*) bufLightLevel;
    messageLightLevel.payloadLen = strlen(bufLightLevel);
    mqtt->publish(MQTT_TOPIC_LIGHT_SENSOR, messageLightLevel);

    //Publication 2 : Switch
    LOG_PRINTFLN("Publication on test/switch");
    if (digitalRead(switch_pin) == 1) {
        switch_state = "ON";
    }
    else {
        switch_state = "OFF";
    }
    LOG_PRINTFLN("Value : %s", switch_state);
    const char* bufSwitch = switch_state;
    MqttClient::Message messageSwitch;
    messageSwitch.qos = MqttClient::QOS0;
    messageSwitch.retained = false;
    messageSwitch.dup = false;
    messageSwitch.payload = (void*) bufSwitch;
    messageSwitch.payloadLen = strlen(bufSwitch);
    mqtt->publish(MQTT_TOPIC_SWITCH, messageSwitch);
}
// Idle for 0.5 seconds
mqtt->yield(500L);
}
}

```

## Lab 3 – Interact with the oneM2M RESTful architecture

### Objective

This lab will explain how to interact with the Eclipse OM2M platform

- configure and launch the platform
- handle the resource tree through a REST client

### I. oneM2M presentation

The oneM2M consortium emerged in 2013 thanks to 8 of the world's leading ICT standards development organizations such as ATIS (United States) and ETSI (Europe). OneM2M is the global standards initiative that covers requirements, architecture, API specifications, security solutions and interoperability for Machine-to-Machine and IoT technologies. One of the particularities of oneM2M is its architecture which is generic and horizontal which means that it allows a homogeneous vision of the system regardless of the field of application. Therefore, oneM2M has been thought as an interoperability enabler for the entire machine to machine and IoT ecosystem.

The oneM2M architecture is based on three different entities : Application Entities (AE), the Common Services Entities (CSE) and the Network Services Entities (NSE). The AEs represent the different applications connected to the system. The CSEs will make it possible for the applications to be registered in their system and provide the set of "service functions" that are common to the M2M environments. Finally, the NSEs will provide services to the CSEs besides the pure data transport.

### II. Implementation of the standard with OM2M

For this lab we used the oneM2M standard through OM2M which provides an open source service platform for M2M interoperability based on this standard. OM2M follows a RESTful approach with open interfaces to enable developing services and applications independently of the underlying network.

In order to deploy the previously seen architecture on a real system, we will have a set of objects (Application Dedicated Nodes -ADN-) connected to a gateway (Middle Node -MN-) which will allow the exchange of data with the server in which will run the Infrastructure Node (IN). The IN handles the hierarchy between the different resources and the architecture.

In our particular case, we will use the HTTP protocol to communicate from the server with Postman. After starting the Infrastructure Node and the Middle Node, we can create (POST requests) our entire Application Dedicated Node, in this case, composed of three sensors (LuminositySensor, TemperatureSensor and SmartMeter). Each of these Application Entities consisting on two containers (CNT : DATA and DESCRIPTOR). Finally, we will be able to store data in the DATA container as Constant Instances (CIN). We can easily see the resource tree created on the MN-CSE during this lab with the url <http://localhost:<IN-Port>/webpage>.

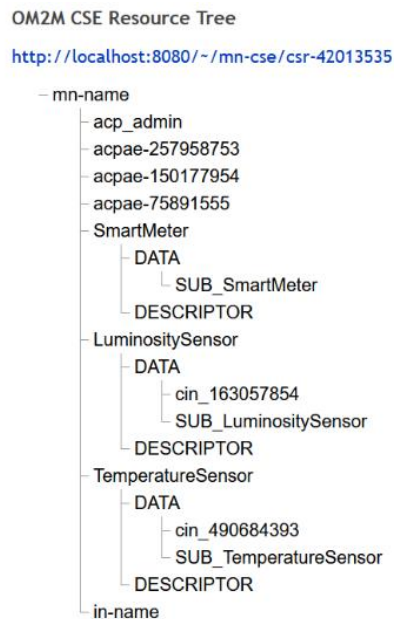


Figure 1 : Resource tree showing our architecture on the MN-CSE

From Postman we will also be able to easily retrieve the data with GET requests.

To be a little more specific, Postman makes HTTP requests such as POST (create), GET, UPDATE (modify) and DELETE. The requests require some headers such as the origin, or the type. A body containing the necessary information will also be required depending on the type of request (POST, UPDATE...).

We took an interest in two other services provided by the CSE : Subscription and Notification Service and the Access Control Management.

Basically, we can subscribe to a Sensor Data. The subscription will be associated to a point of access. This point of access can be a monitor that will receive a notification each time the sensor receives a value. It can also be an AE configured with a request reachability (rr) attribute at true. As we can see, this subscription method is quite similar to the MQTT protocol seen on the first lab.

To manage the access of oneM2M resource, a specific resource exists that represents the rights given to an entity. This resource is named AccessControlPolicy (ACP). Each oneM2M resource is linked to a set of ACPs via the acpi attribute. This allows us for example to have a system that will allow the sensor to create and retrieve data, but that will only allow to retrieve data to the monitor.

## Conclusion : comparison MQTT / oneM2M

The MQTT system is simpler. There is a MQTT server and clients which will subscribe or publish data on the server. The data system is based on topics with a possibility to manage hierarchical topics.

The MQTT protocol offers mainly one service : the management and storage of data exchanged between objects.

The deployment model of oneM2M is based on three layers : application, service, and network. It is deployed on objects, gateways but also on the cloud. As regards the data model, as we saw previously, it is based on a REST architecture and a resource tree. As for the services, oneM2M is more expanded

as MQTT. It offers several services such as discovery, communication, group management, data, network... Through OM2M we can also find the publish/subscribe intrinsic aspect of the MQTT protocol with the subscription and notification services. That makes oneM2M a standard being able to cover the MQTT standard but also the need of most part of the IoT ecosystem.

But it is important to highlight that both have an important quality when it comes to interoperability since they require no specific material and can interact with several programming languages.



## Lab 4 – Fast application prototyping for IoT

### Objective

The aim of this last session is for you to fully integrate what you have done in the previous labs and have a complete application that will interact with several real and virtual devices. You will:

- Deploy a high-level application
- Deploy a concrete architecture with real devices
- Learn to use NODE RED to develop the app faster

### I. Deployment of the architecture

The idea here is to have a real-life use case where you have devices connected to several technologies. Because you use several protocols and technologies, you have to develop specific interfaces for each device. If you have used the possibility of oneM2M standard and the concept of Interworking Proxy Entity, you will have only one type of protocol to understand and to manage. oneM2M in this case will manage the interoperability between all technologies at the middleware level.

During this lab, we will consider both the architectures studied in lab 1 and lab 3. This means we will deploy the lab 3 resource tree on OM2M. We will also use the mosquitto broker associated with the nodeMCU board in which we will run a slightly different version the final program implemented during lab 1.

### II. Node RED for high level application

In order to develop a high-level application with both those architectures, we will learn to use a new tool : Node RED.

Node RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. It provides a browser-based editor that makes it easy to wire together flows, using the wide range of nodes in the palette that can be deployed to its runtime in a single-click.

### III. Configure and use Node RED

First, we installed nodeJS, then Node RED and we integrated the oneM2M nodes. We access the Node RED editor by its webpage through our web browser: <http://localhost:1880/>. From the webpage, we also installed the red-contrib-ide-iot nodes. Those nodes provide other tools to interact with the Eclipse OM2M.

## IV. Applications

### 1. Sending and receiving data from a sensor

#### a. GET and POST sensors values using OM2M

In this part we retrieve the last data instance produced by the simulated sensors in lab 3 with oneM2M nodes. To get a sensor value, we used the nodes: Named Sensor Data, Content extractor and Debug. For the Named Sensor Data node, we need to fill in the properties of the CSE and the name of the AE and container in which we will find the CIN we are looking for.

The screenshot displays the OM2M CSE Resource Tree on the left and a Node-RED flow on the right. The OM2M interface shows a tree structure under 'in-name' with nodes like 'acp\_admin', 'acpae-722412650', 'acpae-63524671', 'acpae-524766583', 'ACP\_Monitor\_Sensors', 'acpae-722719615', 'acpae-416671563', 'TEST\_AE', 'DATA', 'cin\_792035012', and 'mn-cse'. A table on the right lists attributes and their values.

Attribute	Value
ty	4
ri	/in-cse/cin-792035012
pi	/in-cse/cnt-861459365
ct	20201112T012517
lt	20201112T012517
st	0
cnf	application/xml
cs	2
con	25

The Node-RED flow, titled 'Test GET OM2M', consists of the following nodes: 'inject' (set to 'timestamp'), 'TEST\_AE DATA' (with 'cin\_792035012' in the path), 'Content Extractor', and 'msg.payload'. The debug console shows the output: '22/11/2020 à 17:05:31 node: 13e32ba3.559024 msg.payload : string[2] "25"'.

Figure 1 : Retrieving of the value “25” contained in a CIN of the container DATA of the TEST\_AE in the IN-CSE

In order to POST a sensor value, we used the nodes : Fake Sensor or Fake Actuator and a Content Instance. The Content Instance node must be filled with the CSE properties, the correct path for the CIN and the content format (we chose JSON format).

The following Node RED flow shows that we can send either on/off signal with the fake actuator or a random value with the fake sensor to the LuminositySensor located in the MN-CSE that we have created in lab 3.

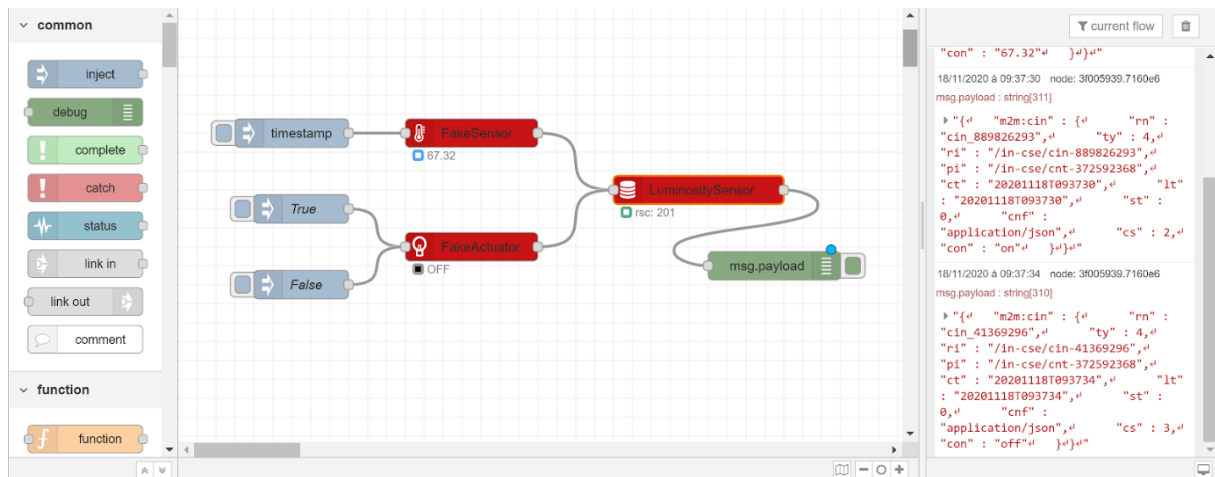


Figure 2 : Simple test POST CIN on our OM2M resource tree

### b. GET and POST sensors values using MQTT

We can also use MQTT protocol with Node-RED. To subscribe and publish data, we basically use the mqtt-in and mqtt-out nodes, in which we give the properties of the server, which hosts the mosquitto broker, the topic we want to interact with, and, in case of publishing, the input the message we want to send.

The following figure shows that a light value (contained in the payload of the inject node) is published on a specific topic by the mqtt-out node named "light\_publication". This value is then receipted with the mqtt-in node named "light\_subscription" and displayed.



Figure 3 : Sending and receiving a sensor value using MQTT protocol in Node RED

## 2. Reasoning on values and actions on equipment

Now that we have managed to send and receive data, we are going to see how to handle data treatment and how to interact with equipment.

The following flow use both MQTT and our OM2M architecture. The new thing here is the function node. This node assures the data treatment. As an input, it receives a value from the MQTT topic with the mqtt-in node. If this value is lower than a threshold value (settled here at 300), a message asking to switch on the lamp is sent to the LAMP\_0 device in the OM2M tree. It means that a CIN containing the message "on" is created in the DATA container of the AE LAMP\_0.

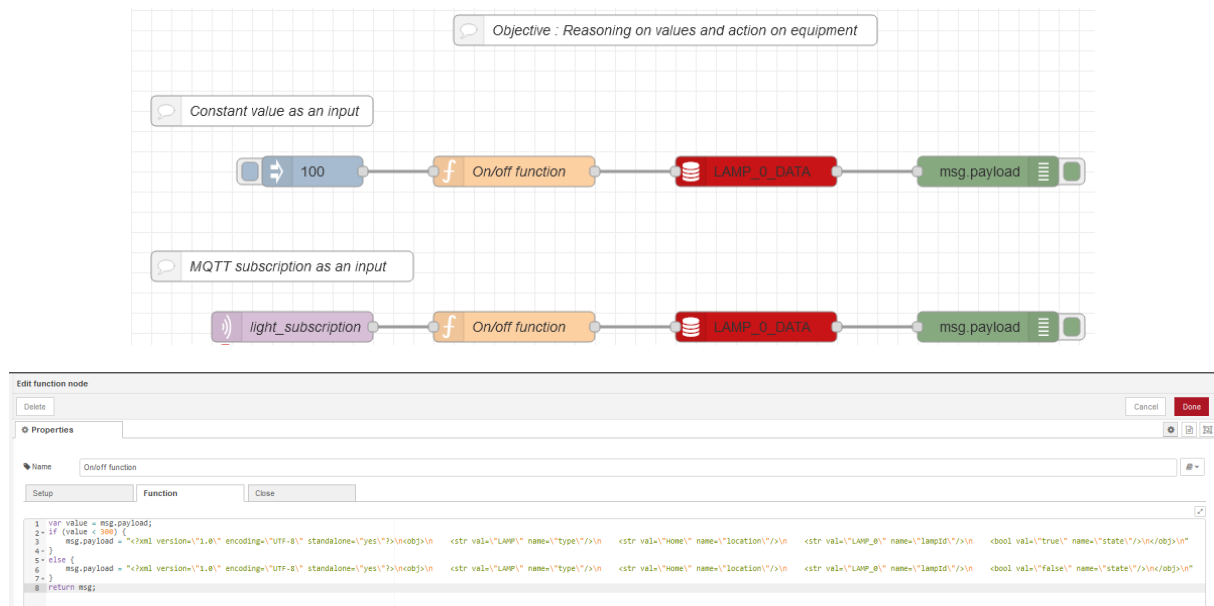


Figure 4 : Our simple reasoning circuit using a function node with the following code implemented in it

### 3. Development of a more complex application with MQTT and connect it to a new visualisation interface

Since we know how to retrieve and send data with both OM2M and MQTT nodes, we decided to develop a more complex application with only one of the two methods : MQTT. This way we can use the algorithm developed during lab 1 and see the effect on the LED of the nodeMCU board. In addition, we have decided to use the dashboard to plot the values of the light sensor with different graphs and buttons to switch on/off the LED.

To subscribe and publish data we basically use the mqtt-in and mqtt-out nodes.

Then we add a node Simple Condition, in which we inform that if the luminosity retrieved is below 300 then we publish "true" in the corresponding topic.

The goal we fixed ourselves here is to develop the same application as in the 1st lab but with Node RED. So, more concretely, an application that will turn on the light under a specific threshold (here 300). On the dashboard, we have also added two switches. One of them is for the user to decide manually if the light is on or off. The state of this switch will only be taken into consideration if the second switch is activated (see Figure 6). This second switch tells the program if the first switch overrides the Luminosity Sensor values.

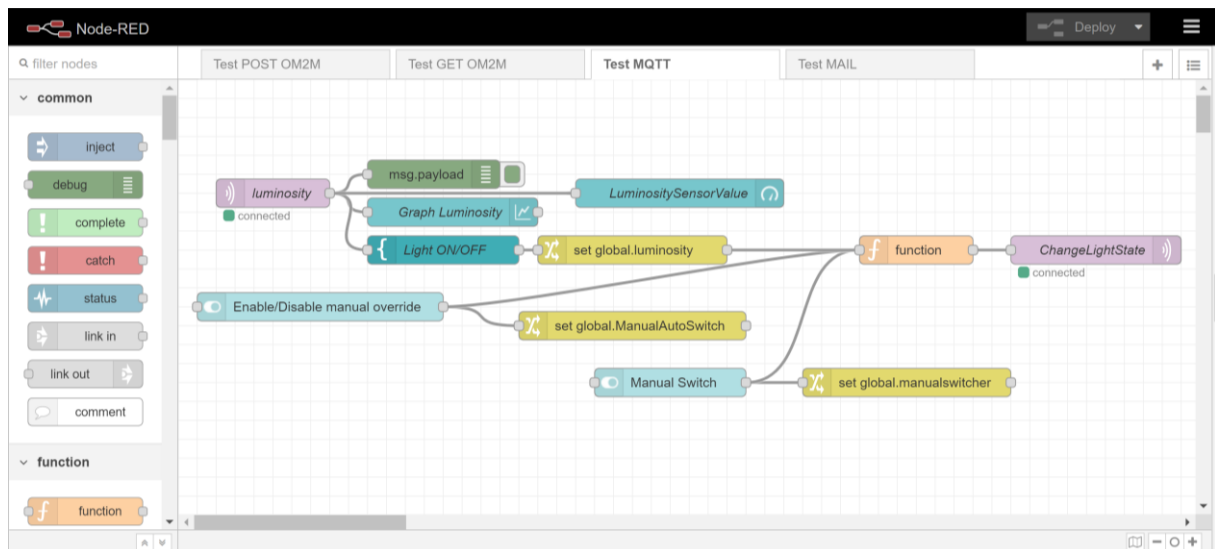


Figure 5 : Our final complex application developed with Node RED

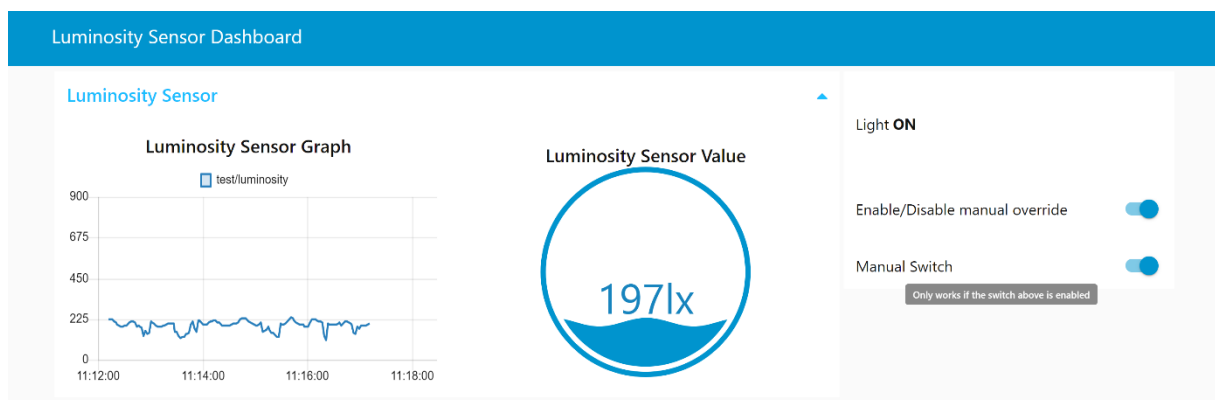


Figure 6 : Application dashboard  
2 graphs luminosity sensor value – State of light - Switches to control light

## Conclusion

As said previously , Node RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. Its intuitive interface makes it user friendly. Moreover, it makes it possible to combine several IoT technologies, with different communication protocols... As we saw, it is possible to implement both technologies (oneM2M, MQTT protocol) with Node RED. The management palette allows this tool to extend itself and to meet the interoperability need of the IoT ecosystem. Nonetheless, Node server comparing to Java server has slower performance.