Aguttes Lilian
Hellouin Zacharie

TP Machine to Machine

TP1: MQTT

TP3 : OneM2M REST API

TP4: Fast application prototyping for IoT

Aguttes Lilian
Hellouin Zacharie

Contents

Aguttes Lilian
Hellouin Zacharie

Introduction

In this report, we detail the various experiments we conducted in order to explore Machine to Machine (M2M) communication. We divided our work in three parts, each part being about a specific lab.

The first part is about the MQTT protocol, its main characteristics and how to deploy it using our computer and a NodeMCU. The second one explains how we used the Eclipse OM2M platform to develop a REST API capable of managing different sensors. We also used Postman to make HTTP requests. Finally, the third part explains how to use Node-RED for high-level application prototyping.

The goal of this group of labs is to explore the different possibilities offered by the Internet of Things. We also aim to present the practical knowledge we acquired about the development of an M2M architecture.

Aguttes Lilian
Hellouin Zacharie

TP1: MQTT

1. 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.

2. MQTT

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

a. *What is the typical architecture of an IoT system based on the MQTT protocol?*
b. *What is the IP protocol under MQTT? What does it mean in terms of bandwidth usage, type of communication, etc?*
c. *What are the different versions of MQTT?*
d. *What kind of security/authentication/encryption are used in MQTT?*
e. *Suppose you have devices that include one button, one light and luminosity 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?*

a) MQTT (Message Queuing Telemetry Transport) is a publish-subscribe network protocol that transports messages between devices. The MQTT protocol defines two types of network entities: a message broker and a number of clients. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients. During this practical work, we are going to use an open-source broker: Mosquitto. An MQTT client is any device (from a micro controller up to a full-fledged server) that runs an MQTT library and connects to an MQTT broker over a network. Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers.

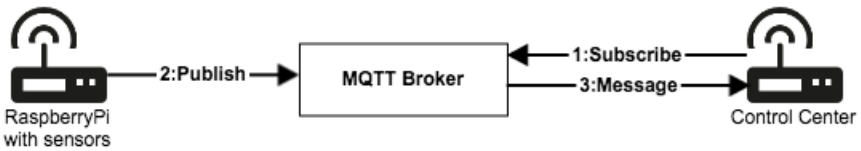


Figure 1: MQTT Architecture

b) MQTT works on top of the TCP/IP protocol, and is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. It allows bi-directional cloud-device communications and facilitates broadcasting with its publish-suscribe behaviour. The goal of MQTT is to provide a protocol which is bandwidth-efficient and uses little battery power. MQTT is also capable of being deployed on unstable networks

c) There are four different versions available currently. There is very little difference between MQTT v3.10 and MQTT v3.1.1. MQTT v5.0 is the successor of MQTT v3.1.1. MQTT v5.0 adds a significant number of new features to MQTT, as user properties, shared subscriptions, Payload format indicator... And finally, MQTT-SN who was designed to work over UDP, ZigBee and other transports.

d) Regarding security, it is necessary to be able to carry out mutual validation. That is to say that unlike the normal client server situation, here the server must be sure that not just anyone is posting any data. Mosquitto offers certificate validation and data encryption by using TLS protocol. It is also possible to restrict access to data by an acl (access list user password).

e) For this smart system, it will be necessary to create at least two topics. The first one will get the value of the light sensor. The second topic will keep in memory the button state. Each sensor (button/luminosity sensor) will publish on his specified topic, and the microcontroller which control the light will subscribe to both topics. We will find the connexion architecture below.

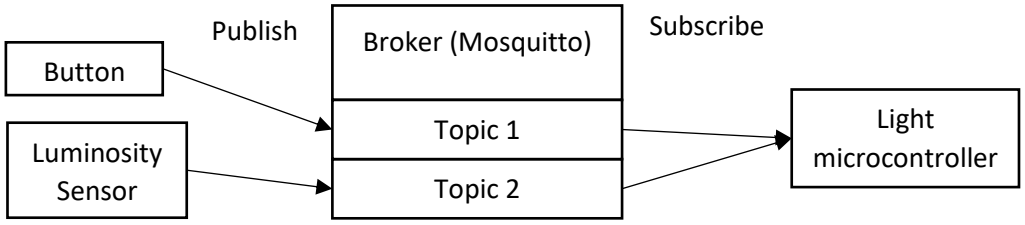


Figure 2: Smart System connexion architecture

3. 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 mosquito broker*
- *Test publish and subscribe with the command shell programs: mosquitto_pub and mosquitto_sub*

4. Creation of an IoT device with the node MCU 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.
The NodeMCU board, based on ESP8266 provides a compact standalone Wi-Fi networking solution. It also offers USB, UART, SPI, I2C connectivity. 10 GPIOs are PWM ready and can be used to control external devices.

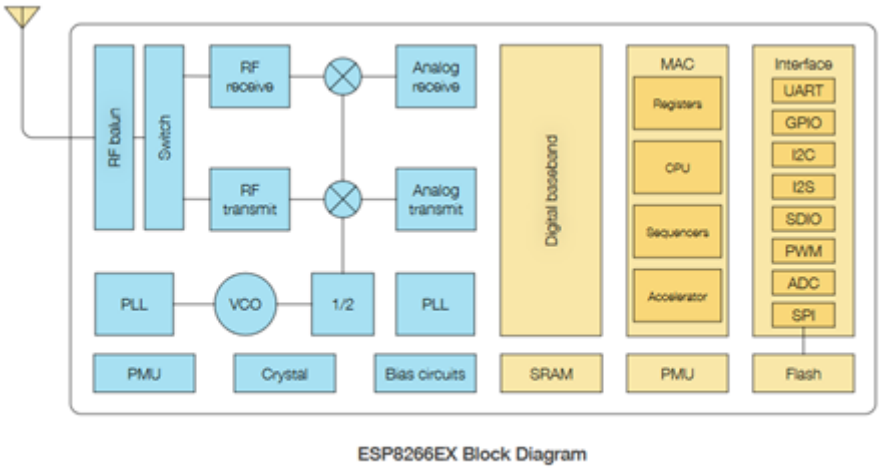


Figure 3 ESP8266EX Block Diagram

It initially included firmware which runs on the ESP8266 Wi-Fi SoC from Espressif Systems, and hardware which was based on the ESP-12 module. The firmware uses the Lua scripting language but is also programmable using Arduino 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: exemples/arduinoMqtt/connectESP8266wificlient and have a look at the different parts of the code. Explain those different parts*
- *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: exemples/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*

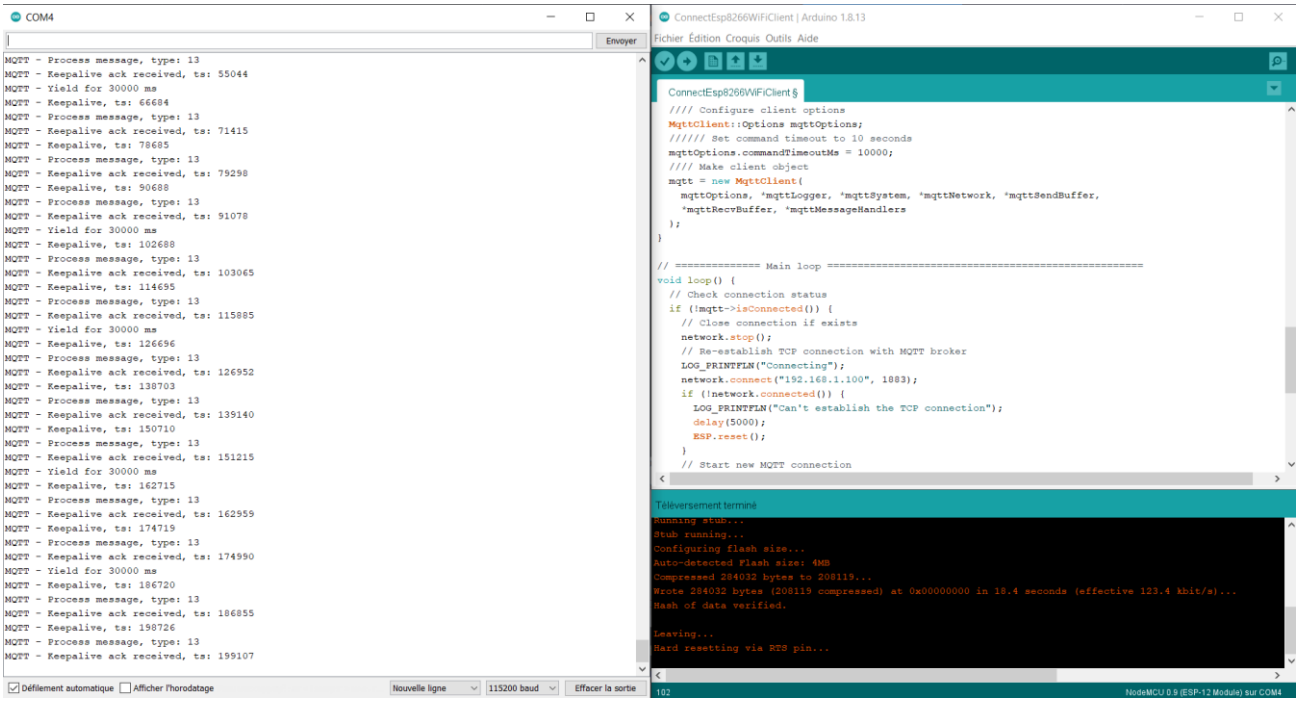


Figure 4: Arduino Connexion Test

5. 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.

```
#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_BUTTON = "Light/button"; //Creation d'un TOPIC obtenir l'état de bouton
const char* MQTT_TOPIC_LUM = "Light/luminosity"; //Creation d'un TOPIC obtenir la mesure de luminosité
static MqttClient *mqtt = NULL;
static WiFiClient network;

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

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

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

// ============== Setup all objects ================================================
#define BUTTON D5
#define SENSOR D3
#define LED D0
void setup() {

  // Set up port pin
  pinMode(BUTTON, INPUT);
  pinMode(SENSOR, INPUT);
  pinMode(LED, OUTPUT);
  digitalWrite(LED, HIGH);

  // 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(".");
  }
  LOG_PRINTFLN("Connected to WiFi");
```

```
  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 processMessageLum(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
  );
}
  );
}
void processMessageBut(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
  );
  char test[10];
  test[0] = '1';
  if (payload[0]=='1') {
    Serial.println("coucou jallume");
    digitalWrite(LED, LOW);
  } else {
    digitalWrite(LED, HIGH);
  }
}
// ============== 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("192.168.1.100", 1883);
    if (!network.connected()) {
      LOG_PRINTFLN("Can't establish the TCP connection");
      delay(5000);
      ESP.reset();
    }
    // Start new MQTT connection
```

```
  // 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;
    }
  }
  // Subscribe
  {
    MqttClient::Error::type rcLum = mqtt->subscribe(MQTT_TOPIC_LUM, MqttClient::QOS0, processMessageLum);
    MqttClient::Error::type rcButton = mqtt->subscribe(MQTT_TOPIC_BUTTON, MqttClient::QOS0, processMessageBut);
    if (rcLum != MqttClient::Error::SUCCESS || rcButton != MqttClient::Error::SUCCESS) {
      LOG_PRINTFLN("Subscribe error: %i %i", rcLum, rcButton);
      LOG_PRINTFLN("Drop connection");
      mqtt->disconnect();
      return;
    }
  }
} else {
  // Publish
  {
    char bufLum[10];
    int state = analogRead(SENSOR);
    itoa(state, bufLum, 10);
    Serial.println(state);
    MqttClient::Message messageLum;
    messageLum.qos = MqttClient::QOS0;
          char bufLum[10];
          int state = analogRead(SENSOR);
          itoa(state, bufLum, 10);
          Serial.println(state);
          MqttClient::Message messageLum;
          messageLum.qos = MqttClient::QOS0;
          messageLum.retained = false;
          messageLum.dup = false;
          messageLum.payload = (void*) bufLum;
          messageLum.payloadLen = strlen(bufLum);
          mqtt->publish(MQTT_TOPIC_LUM, messageLum);

          char bufBut[10];
          itoa(digitalRead(BUTTON), bufBut, 2);
          MqttClient::Message messageBut;
          messageBut.qos = MqttClient::QOS0;
          messageBut.retained = false;
          messageBut.dup = false;
          messageBut.payload = (void*) bufBut;
          messageBut.payloadLen = strlen( bufBut);
          mqtt->publish(MQTT_TOPIC_BUTTON, messageBut);
        }
        // Idle for 30 seconds
        mqtt->yield(5000L);
      }
    }
```

Aguttes Lilian
Hellouin Zacharie

TP 3 – Middleware for the IoT

During this lab, we are going to create an architecture OM2M that is able of storing and retrieving data from different sensor. To proceed, we are going to do some HTTP requests using Postman. In the various parts below, we detail how we created each application, container, subscription…

ACP (Control Access Policy)

First, we created an ACP resource. This resource's is to control the access of application that we are going to create later. To create this resource, we use the HTTP request below.

URL	http://127.0.0.1:8282/~/mn-cse/mn-name
Method	POST
Headers	X-M2M-Origin: admin: admin Content-type : application/xml; ty = 1
Body	<m2m:acp xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="ACP_1"> <pv> <acr> <acor>user</acor> <acop>2</acop> </acr> </pv> <pvs> <acr> <acor>admin</acor> <acop>63</acop> </acr> </pvs> </m2m:acp>

In the request body, we only declare the retrieve right for users who wanted access to the targeted application. Only the administrator has the full right on this ACP.

Temperature Sensor Application

Now, we can create an AE for each sensor (Temperature, luminosity, …). In the table below, we are showing how to create a temperature sensor AE with an HTTP request.

URL	http://127.0.0.1:8282/~/mn-cse/mn-name
Method	POST
Headers	X-M2M-Origin: admin: admin Content-type : application / xml; ty = 2
Body	<m2m:ae xmlns:m2m = "http://www.onem2m.org/xml/protocols" rn = "TemperatureSensor"> <acpi>/mn-cse/mn-name/ACP_1</acpi> <api> app-sensor </api> <lbl> Type/sensor Category/temperature Location/home </lbl> <rr> false </rr> </m2m:ae>

Do not forget in the request body to declare the ACP which this AE belong to (ACP_1 in our case).

Aguttes Lilian
Hellouin Zacharie

Container DATA in TemperatureSensor

Under each sensor application, we create a container Data, where values from each sensor will be stored. For example, we used the HTTP request below for the Temperature Sensor application.

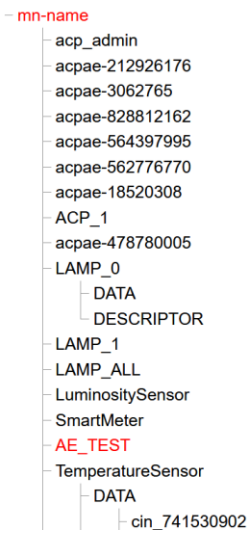
URL	http://127.0.0.1:8282/~/mn-cse/mn-name/TemperatureSensor
Method	POST
Headers	X-M2M-Origin: admin: admin Content-type : application / xml; ty = 3
Body	<m2m:cnt xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="DATA"> </m2m:cnt>

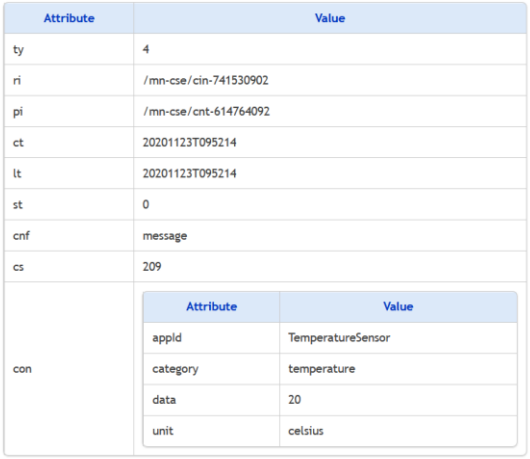
Content Instance data

We are trying to store a value that is coming from one of our sensor (temperature).

URL	http://127.0.0.1:8282/~/mn-cse/mn-name/TemperatureSensor/DATA
Method	POST
Headers	X-M2M-Origin: admin: admin Content-type : application / xml; ty = 4
Body	<m2m:cin xmlns:m2m="http://www.onem2m.org/xml/protocols"> <cnf>message</cnf> <con> <obj> <str name="appId" val="TemperatureSensor"/> <str name="category" val="temperature "/> <int name="data" val="20"/> <int name="unit" val="celsius"/> </obj> </con> </m2m:cin>

We declare in our request a temperature value equal to 20°c. We can check on our OM2M webpage if this value is well stored.





Attribute	Value
ty	4
ri	/mn-cse/cin-741530902
pi	/mn-cse/cnt-614764092
ct	20201123T095214
lt	20201123T095214
st	0
cnf	message
cs	209
con	appId: TemperatureSensor category: temperature data: 20 unit: celsius

We retrieve our useful sensor value (20) which is well declared in Celsius. Our storage has worked, we can now create a descriptor container.

Aguttes Lilian
Hellouin Zacharie

Container DESCRIPTOR in TemperatureSensor

We are now creating a new container called DESCRIPTOR with the HTTP request below.

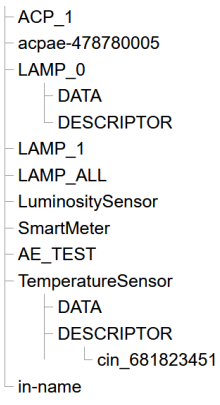
URL	http://127.0.0.1:8282/~/mn-cse/mn-name/TemperatureSensor
Method	POST
Headers	X-M2M-Origin: admin: admin Content-type : application / xml; ty = 3
Body	<m2m:cnt xmlns:m2m = "http://www.onem2m.org/xml/protocols" rn = "DESCRIPTOR"> </m2m:cnt>

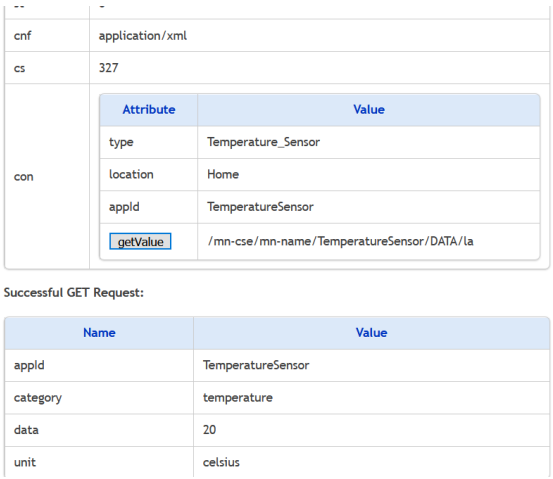
Content Instance descriptor

In this container, we are creating a content instance which can get the value of the last stored data in the DATA container.

URL	http://127.0.0.1:8282/~/mn-cse/mn-name/TemperatureSensor/DESCRIPTOR
Method	POST
Headers	X-M2M-Origin: admin: admin Content-type : application / xml; ty = 4
Body	<m2m:cin xmlns:m2m="http://www.onem2m.org/xml/protocols"> <cnf>application/xml</cnf> <con> <obj> <str name="type" val="Temperature_Sensor"/> <str name="location" val="Home"/> <str name="appId" val="TemperatureSensor"/> <op name="getValue" href="/mn-cse/mn-name/TemperatureSensor/DATA/la" in="obix:Nil" out="obix:Nil" is="retrieve"/> </obj> </con> </m2m:cin>

Do not forget to define the path and the command of the data container targeted.



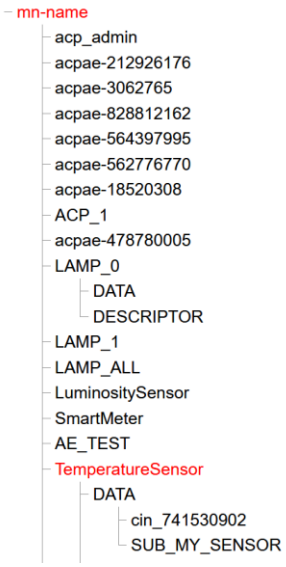


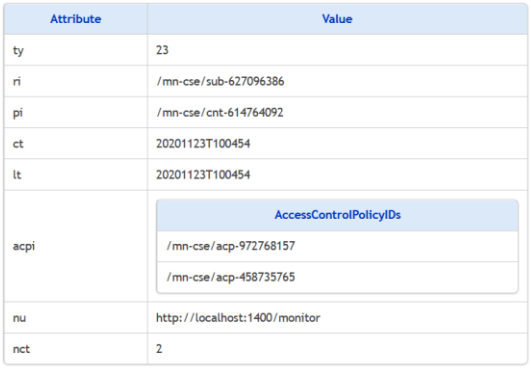
We can observe that when we click on the getValue button, our content instance sends a GET request and retrieves the previously stored data.

Subscription

Now that our OM2M is set up, we can add a subscription resource. A subscription is useful when you want to be notified when a new data is stored in the targeted container. In this example, we are keeping the DATA container of the temperature sensor.

URL	http://127.0.0.1:8282/~/mn-cse/mn-name/TemperatureSensor/DATA
Method	POST
Headers	X-M2M-Origin: admin: admin Content-type : application / xml; ty = 23
Body	<m2m:sub xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="SUB_MY_SENSOR"> <nu>http://localhost:1400/monitor</nu> <nct>2</nct> </m2m:sub>





Attribute	Value
ty	23
ri	/mn-cse/sub-627096386
pi	/mn-cse/cnt-614764092
ct	20201123T100454
lt	20201123T100454
acpi	AccessControlPolicyIDs /mn-cse/acp-972768157 /mn-cse/acp-458735765
nu	http://localhost:1400/monitor
nct	2

We are going to test our subscription with a monitor server. We tell the monitor to listen on port 1400 and we create a new content instance data as we've seen before.

C:\Windows\System32\cmd.exe - java -jar monitor.jar

```
Received notification:
<?xml version="1.0" encoding="UTF-8"?>
<m2m:sgn xmlns:m2m="http://www.onem2m.org/xml/protocols">
   <nev>
      <rep rn="cin_206626947">
         <ty>4</ty>
         <ri>/mn-cse/cin-206626947</ri>
         <pi>/mn-cse/cnt-614764092</pi>
         <ct>20201123T112654</ct>
         <lt>20201123T112654</lt>
         <st>0</st>
         <cnf>message</cnf>
         <cs>209</cs>
         <con>
      &lt;obj>
        &lt;str name=&quot;appId&quot; val=&quot;TemperatureSensor&quot;/>
        &lt;str name=&quot;category&quot; val=&quot;temperature &quot;/>
        &lt;int name=&quot;data&quot; val=&quot;22&quot;/>
        &lt;int name=&quot;unit&quot; val=&quot;celsius&quot;/>
      &lt;/obj>
    </con>
      </rep>
      <rss>1</rss>
   </nev>
   <sud>false</sud>
   <sur>/mn-cse/mn-name/TemperatureSensor/DATA/SUB_MY_SENSOR</sur>
</m2m:sgn>
```

When the content instance is created, our monitor received a notification from the server. We can see in the received notification that our data value is equal to 22, as programmed in the HTTP request.

Aguttes Lilian
Hellouin Zacharie

TP 4 - Fast application prototyping for IoT

1) Objectives

The aim of this last session is for you to fully integrate what you have done in the TP1, TP2 and TP3 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*

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.

2) Applications

Originally, the goal of this TP was to create an application able to retrieve values from a light sensor with MQTT Node, store them on OM2M and turn on/off a lamp on the ESP82666 devices. However, because of the current pandemic, we are unable to access to some hardware resources. Thus, we decided to exchange our light sensor and lamp with a fake sensor and a fake lamp.

a) Retrieve Data from a Sensor

In this part, we are going to generate a value from a fake sensor and store him in an OM2M Node. The storage will be localised in a specific directory (AE_TEST/DATA) that we have created before with a Postman HTTP request.

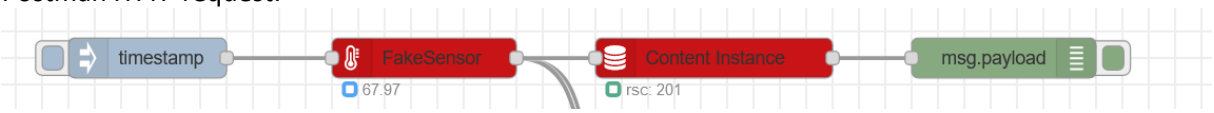


Figure 5: Retrieve data from a sensor with Node Red

As you can see, the fake sensor module creates a value (67.97) which is transmitted to a Content Instance node. This module will download the value to a defined OM2M. Before deploying this architecture, you must set up all information required in the Content Instance which are related to your targeted Node (Address, Login, …). During the application execution, Content Instance node replied to us with an HTTP code 201. It means that our storage has been well executed.

We can check OM2M webpage to be sure:

Figure 6: OM2M web page

b) Turn on/off virtual light

In this part, we are trying to turn on/off a virtual lamp created by a native application on OM2M. As you can see in the previous Figure 6, two applications (LAMP_1, LAMP_0) are presents in our OM2M tree structure. They are linked to the virtual lamp that we are going to call. We used two items for this application. The first one is a push button. The second one is named actuator node. Thank to this node, we can send a command to the right actuator Lamp_1 (if well set up previously). In our case, we will turn the lamp on or off. You can see below the node red schematic used.

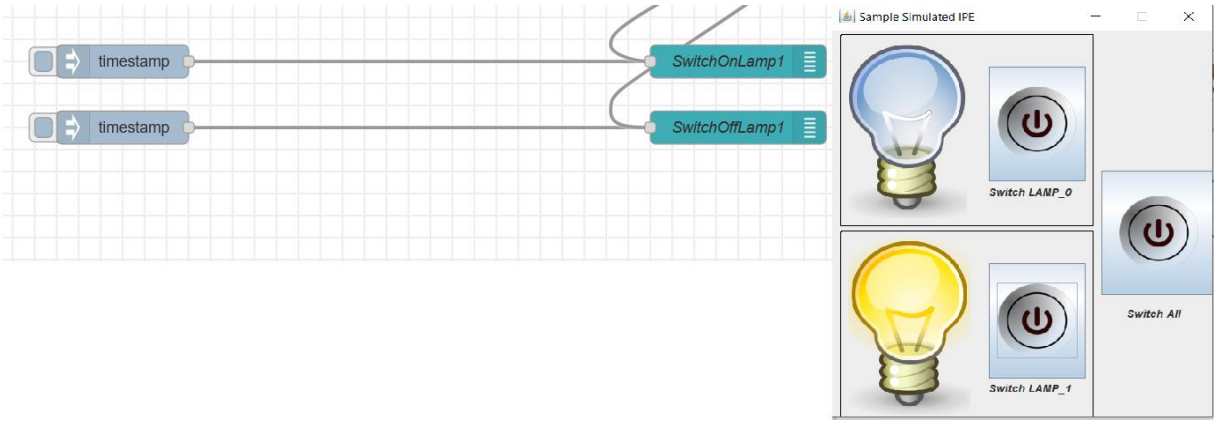


Figure 7: Turn On/Off a virtual Light with Node Red

c) Switch light with sensor value

Finally, we try to link these applications with the addition of fours nodes which allow us to retrieve data previously stored on OM2M and apply a condition on the values. To retrieve data instance, we use a named Sensor configured on the address of container desired. To keep only the sensor value, we add behind a Content extractor node. This node allows to the user the retrieve the useful value of this instance. Now we have our value, we are going to apply an operation on him. By adding a Simple Condition Node, we are testing the value sign. If it's positive, the answer will be True, and False in the other way. And finally, we add a switch node, which will decide to turn off or turn on the light in function of the response previously generated.

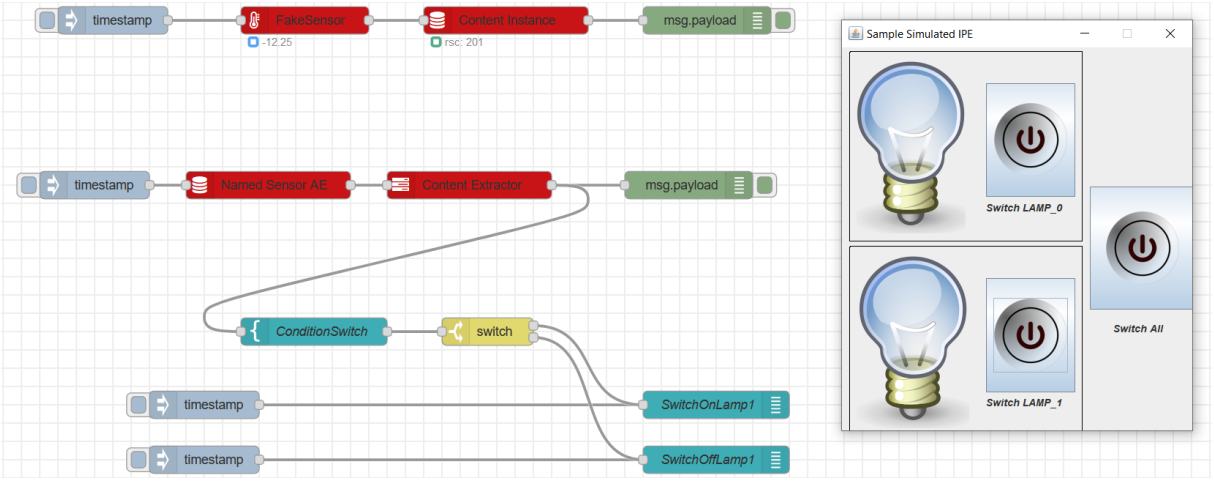


Figure 8: Final application

New generated value by the fake sensor is: -12,25. It's less than zero, so the light turns Off.

3) Report

a) Export the Node-Red flows that you have created for those applications (menu > export): Sensors and activators, dashboard and Imagine sub questions and attach them to your report.

Please find in our attached document, the Node-Red flows untitled: flows.json

b) What are the benefits and drawbacks to build an application with Node-Red?

Building an application with Node-Red comes with pros and cons.
The main advantage of Node-Red resides in its clarity. If used correctly, Node-Red offers a clear view of the program that allows fast prototyping and facilitates maintenance. This simplicity is also a great asset when you need to explain your program to someone else who's not an expert in software development. Node-Red also offers the possibility to make local servers and provides interoperability for heterogeneous environments while having low hardware requirements.
However, some nodes are not available for android, some others do not fit to each other properly and the documentation can sometimes be unsatisfactory. Another drawback is the impossibility to see the details inside the nodes when debugging.

Conclusion

In this report, we tried to give an overview of the work we did and the MQTT based M2M architecture we deployed. Due to the COVID-19 pandemic, some lab sessions took more time than expected and group work was delicate. For example, TP3 and TP4 could have been much more interesting with the possibility to test our architecture and application with real life devices.

However, we hope our report explained the knowledge we acquired about IoT architectures in a clear and cohesive way.