INNOVATIVE SMART SYSTEMS

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

MIDDLEWARE FOR THE IOT – **OM2M**

PRACTICAL WORKS REPORT

PHILIPS hue personal wireless lighting

Teacher: Nicolas Seydoux

OM2M Connecting things

Agathe Limouzy
Jonathan Malique
Sophie Rougeaux
Elie Taillardat

PTP ISS
Group A1
28/12/2018

SUMMARY

INTRODUCTION

A **middleware** is a software that acts as a **bridge** between applications and an operating system or database (it provides services to applications). During our fifth year at INSA Toulouse, we are studying the middleware in the case of the **Internet of Things**. It is about linking different connected devices and deploying an architecture to organize and store the retrieved information.

For our practical works, we are using **OM2M**, a middleware based on a **standard, OneM2M**, developed by the LAAS at Toulouse. OneM2M is a global organization providing a **framework** to support IoT applications created in 2012 and deploying interoperability. OM2M is an **Eclipse open-source** project implementing this standard; it is a platform allowing to develop services.

The goals of the different practical works were to interact with the oneM2M **RESTful** architecture using Eclipse OM2M, to develop a REST client with **XML mapping**, to connect an existing technology to a oneM2M system (Philips HUE lamps using Zigbee protocol and a REST API), to create an **interworking proxy** and finally, to develop a high level application based on real IoT architecture with OM2M and **Node-Red**.

1. POSITIONING THE MAIN STANDARDS OF THE IOT

The **OneM2M standard** is a standard supporting the deployment and management of the IoT applications in multiple industrial verticals. It is focused on the common M2M service layer communications. Its aim is to make the development of applications based on M2M applications as easy as possible. It is developed for lots of domains like healthcare, industrial automation or smart home to have easily access to these standards and easily answer to any kind of problematics. That is why this is a **horizontal platform**.

It has already **200 members** with different ways of standardisations in Europe, America and Asia like CNRS, Intel, Orange, Veolia.

OneM2M is a platform used to resolve **any type of problem in a homogeneous way**:

- API standard
- Service layer
- Mac layer
- Communication link

We can modelized this system with different entities:

- **AE** (Applications Entities): represent the different connected applications of our system.
- **CSE** (Common Services Entities): permit to give access to AE at the different services of this standard.
- **Communications link**: allow the circulations of data to other services layers.

We can quickly talk about other standards that we already know: Thread, Homekit and Fiware.

Thread is an industrials alliance based on **IPv6 and 6LOWPAN** allowing a **communication low energy**. Their policy is: “Low-energy footprint, secure and reliable, No single point of failure, interoperable, scalable and based on proven standards”. This standard is used by ARM, LG and NXP for example.

Homekit is a standard created by **Apple** to link all the objects of this licence to build simply a connected house. That allows two Apple products to communicate between each other easily and to control them **with Siri or with some Homekit applications** that you can download when you buy some accessories like a Fibaro sensor or a connected lamp Eve Flare.

Fiware is a European alliance proposing a **horizontal alliance** like oneM2M, it is able to **connect to any equipment.** Their policy is: “Driving key standard for breaking the information silos, making IoT simpler, transforming Big Data into knowledge, unleashing the potential of right-time Open Data, enabling the Data Economy, Ensuring sovereignty on your data”. Atos, Smart Cities Lab and Antel are some members of this alliance.

2. Deploying a standardized architecture & Setting up a Sensors network system

2.1. Deploying and setting up an IoT architecture using OM2M

During these three practical sessions, we have deployed an **IoT architecture** using OM2M. We used two main nodes containing a Common Service Layer (CSE):

- An **I**nfrastructure **N**ode, called **IN-CSE**. It is the base **Common Service Entity**. It acts like a server.
- A **M**iddle **N**ode, called **MN-CSE**. It is used as a **gateway** for devices to connect. The MN-CSE automatically identifies itself on the specified IN-CSE, and the requests are automatically redirected from the IN-CSE to the MN-CSE. The IN-CSE will act as a **proxy** for the MN-CSE.

After developing the Hue **IPE** for the Phillips Lamps, we could use another node dedicated to the device, this one containing only the Application Layer:

- An **A**pplication **D**edicated **N**ode, called **ADN**. This node contains one or more **A**pplication **E**ntities (**AE**) that represent the **devices**, but it does not implement the Common Service Layer. This node will typically connect to an MN-CSE, to communicate on the OM2M network.

A typical example of architecture is represented on **Figure 1**.

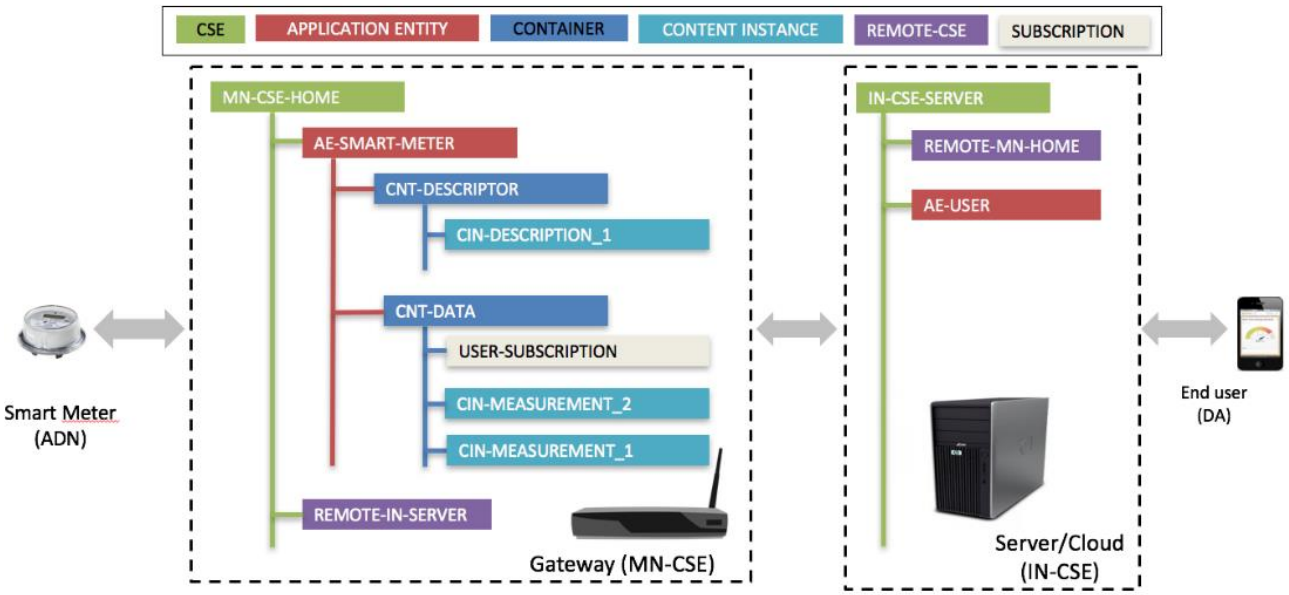

Figure 1: OM2M architecture behavior

An **end user**, external to the middleware, will be for example a monitoring smartphone application. It will interact with the system using the **HTTP REST API**. This app can use the **subscription**/notification mechanisms of OM2M to get each new observation pushed on the middleware.

The different elements interact at **different** layers: The Network Layer, Service Layer and Application Layer drawn on Figure 2.

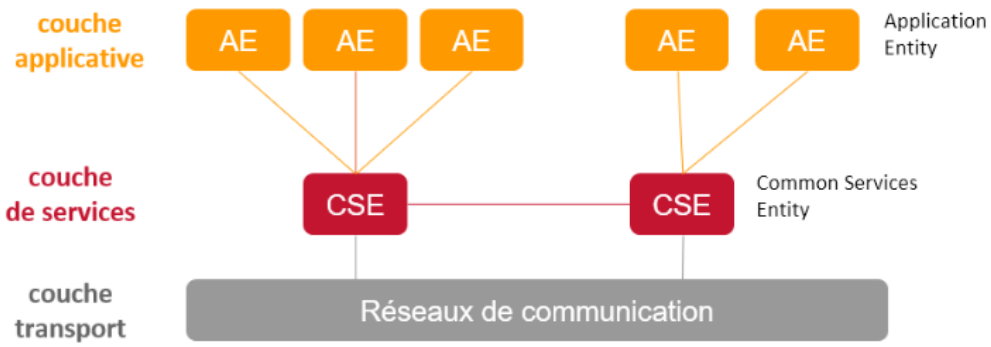

Figure 2: Interaction scheme between entities. Grey: Network Layer. Red: Service Layer. Orange: Application Layer

2.2. INTERACTING WITH OBJECTS USING A REST ARCHITECTURE

To interact with objects in OM2M, we used oneM2M resources and the REST architecture. In the case of the IPE Sample, we used the following resources:

- **CSE-BASE**: It is the root of the Common Service Entities, it contains the child resources, typically our IN-CSE.
- **REMOTE-CSE**: It is a CSE node linked to the CSE-BASE, typically our MN-CSE.
- **AE**: Application Entity, it can be a sensor, actuator or an end-user application.
- **CNT**: Container, it is like a directory. We used DESCRIPTOR and DATA containers in the AEs.
- **CIN**: Content Instance, it is an instance, generally placed in a container. For example, in a DATA container we put content instances that are the values of this data.
- **SUB**: Subscription, it is a resource that allows a node to subscribe to events on another resource. For example, when a new content instance is pushed on the subscribed resource, the subscribing node will receive a «Notify ».

We had two virtual lamps communicating with the REMOTE-CSE (MN-CSE), represented by two AEs in OM2M. The resources trees are:

OM2M CSE Resource Tree

http://127.0.0.1:8080/~/in-cse/csr-646761466

- in-name
  - acp_admin
  - mn-name

OM2M CSE Resource Tree

http://127.0.0.1:8080/~/mn-cse/cnt-678702496

- mn-name
  - acp_admin
  - LAMP_0
    - DESCRIPTOR
    - DATA
      - cin_559500351
      - cin_31938377
      - cin_704397527
  - LAMP_1
  - LAMP_ALL
  - in-name

Each lamp had a **DESCRIPTOR** container, with a description content instance, and a **DATA** container. The new states of the lamp are pushed as content instances in the DATA container.

To interact with the resources, we used **RESTful methods** (Retrieve, Create, Update and Delete) that we implemented in a class *Client.java*. These methods are HTML requests (GET, POST, PUT, DELETE), with a header and a body, and the structure is specified by the OM2M documentation[1]. You can see these methods in **[Annexe 1]**.

We first used **Postman, an HTTP client** to manually send HTML requests. Then, we implemented a monitoring application using an HTTP server. This monitoring application uses the RESTful methods. It registers itself as an AE on the OM2M architecture (see **Figure 3**) using the URI of the HTTP server for its POA.

It subscribes to a lamp resource (a resource SUB is created in the LAMP_0 container) to get notifications. Then, at each notification, the app decodes the notification and prints the read value.

To decode the XML object returned by OM2M, we used the **unmarshalling process** (and the marshalling process to map objects to XML). We used **JAXB** mapping to serialize and deserialize, and the **oBIX** library to customize our objects.

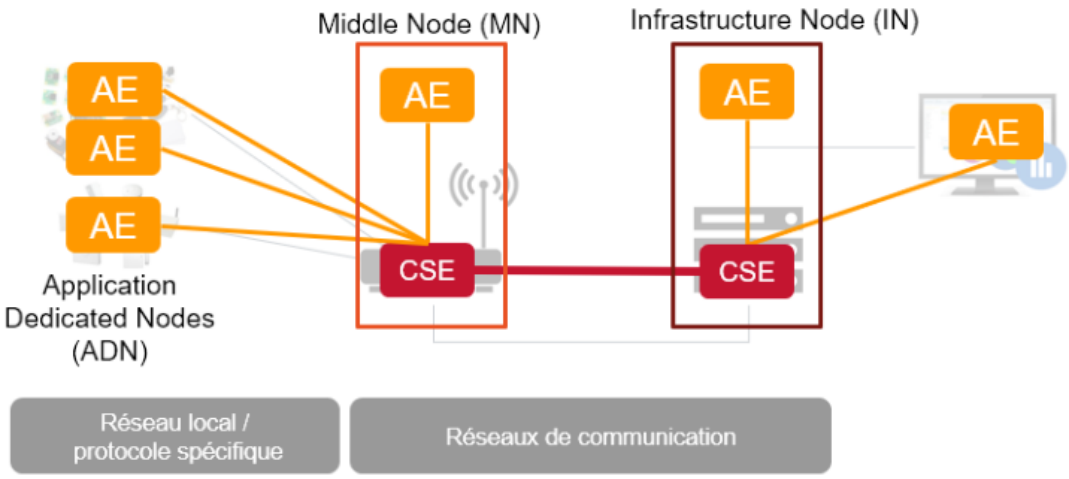

Figure 3: Architecture deployed with the IPE Sample

[1] http://wiki.eclipse.org/OM2M/one/REST_API

2.3. IMPLEMENTING A NEW/OTHER TECHNOLOGY IN AN IOT ARCHITECTURE

In general, IoT technologies are specific, **proprietary**, and they do not implement the oneM2M standard. To be able to integrate a new technology in a oneM2M-based architecture, the standard defined plugins called "**interworking proxy**", dedicated to the translation between specific technologies and oneM2M.

In our case, to integrate Philips Hue lamps to our system, we created an **IPE Server** that connects to the HUE bridge, listens to it in order to write the lamps state in OM2M, and sends the requests from OM2M. The methods to connect to the bridge, get and set the states of the lamp are declared in the class *HueUtils.java*. We use the **SDK** furnished by Philips.

The architecture deployed is the following one:

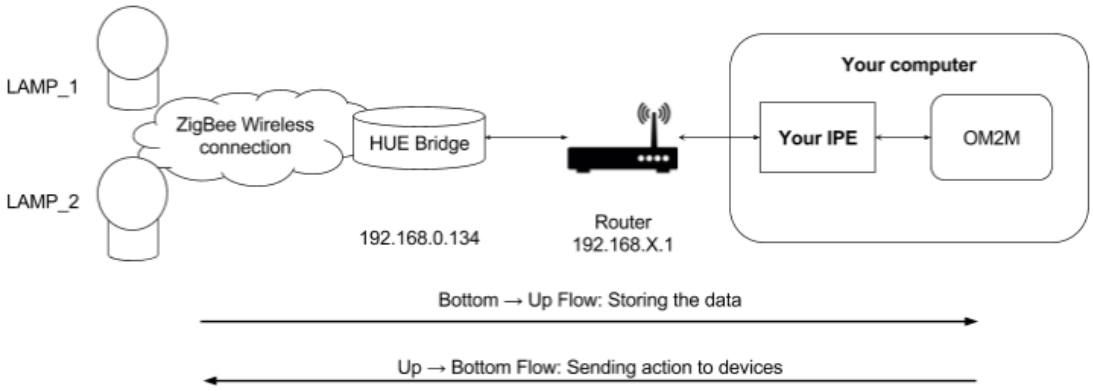

Figure 4: Architecture implemented for our IPE

To make the link between OM2M and the Philips Hue technology, our IPE works this way:

- It creates an **AE** for each lamp with a **DESCRIPTOR** container and a **DATA** container where the observations are stored. It is made by the *Monitor* class of our IPE server. The **POA**s of our AEs are the **URL** of the server.
- The bottom-to-up communication is controlled by a *Monitor* class: when a new lamp state happens, the *Monitor* creates a **new** Content Instance in the DATA container of OM2M. It is a ***POST*** sent by the method:
  ```
  pushState(String lampId, String state, String color, int hue)
  ```
- The up-to-bottom communication is controlled by a *Controller* class on the IPE server: when a button on the Web Interface of OM2M is clicked (meaning a **POST request is sent to the server**), the *Controller* class makes the corresponding state change on the device with the method:
  ```
  execute(HashMap<String, String> queryStrings)
  ```

In the end, we have created a new node on the network: an ADN that interacts with OM2M through an HTTP REST API. The communication has been made totally transparent **from and to** the device.

3. Deploying a Composite Application between several Technologies using Node-RED, based on a standardized middleware

The last part of this TP was to deploy a high-level application using **Node-RED**, a powerful tool to develop **faster** apps for the IoT. We had a previous little experience with this tool during the **LoRa/Arduino project** about the Gas sensor connected system using TTN (The Thing Network) and Node-RED.

Here the objective was to make a complete application which has to interact with the Hue Philips lamps, depending on **Fibaro sensors** (motion/luminosity/temperature/alarm) linked with **ZWave** technology and the **OM2M/IPE** architecture we started to develop earlier in the previous TP, as you can see in the **Figure 5** below:

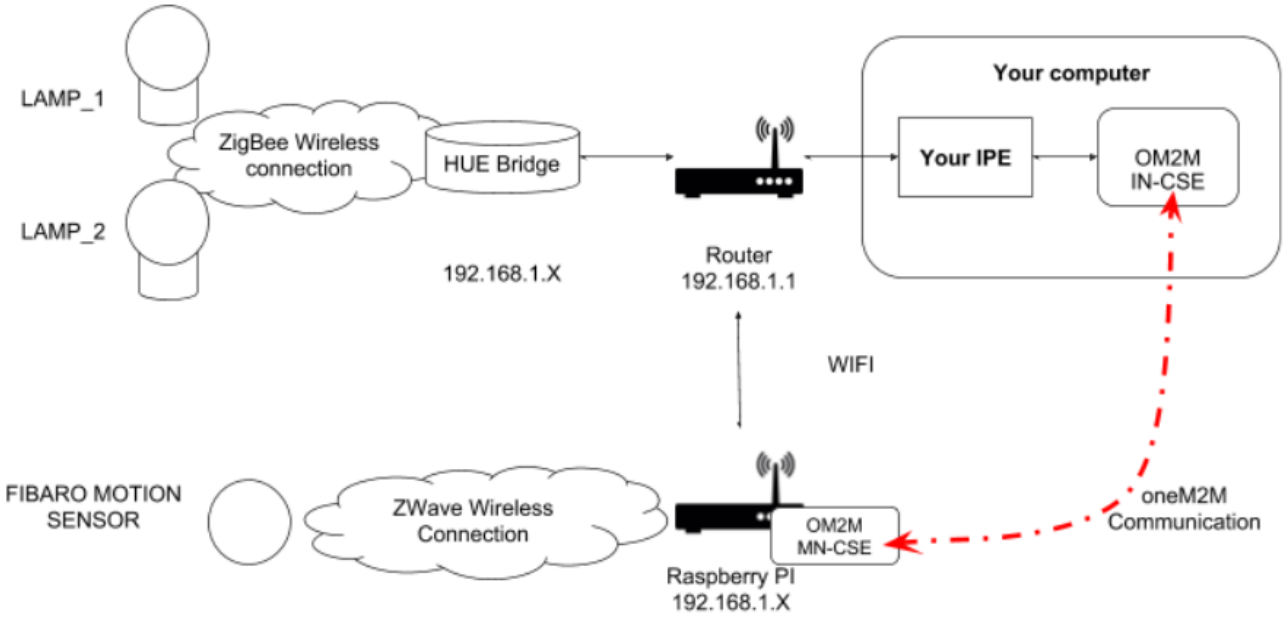

Figure 5: System architecture

We first had to connect to the WIFI router of Philips, and then configure the MN-CSE-PI on the **Raspberry Pi** to be able to start the **ZWave network** with the good IP address and retrieve the sensors data in OM2M. We now had to configure Node-RED, which provides a browser-based editor to wire easily flows together using a large range of nodes.

In order to do that, we installed the special package '***node-red-contrib-ide-iot***' to be able to use nodes to join our OM2M architecture. The first thing we wanted to achieve was to get temperature and light values sent by the Fibaro sensor.

We designed a simple flow in the first time, composed of various nodes in our possession, as you can see in **Figure 6**: **inject**, **NamedSensor**, **DataExtractor** and **Debug**.

- The **inject** node was configured in **timestamp** mode, to trigger the action of retrieving data, here every 3 seconds.

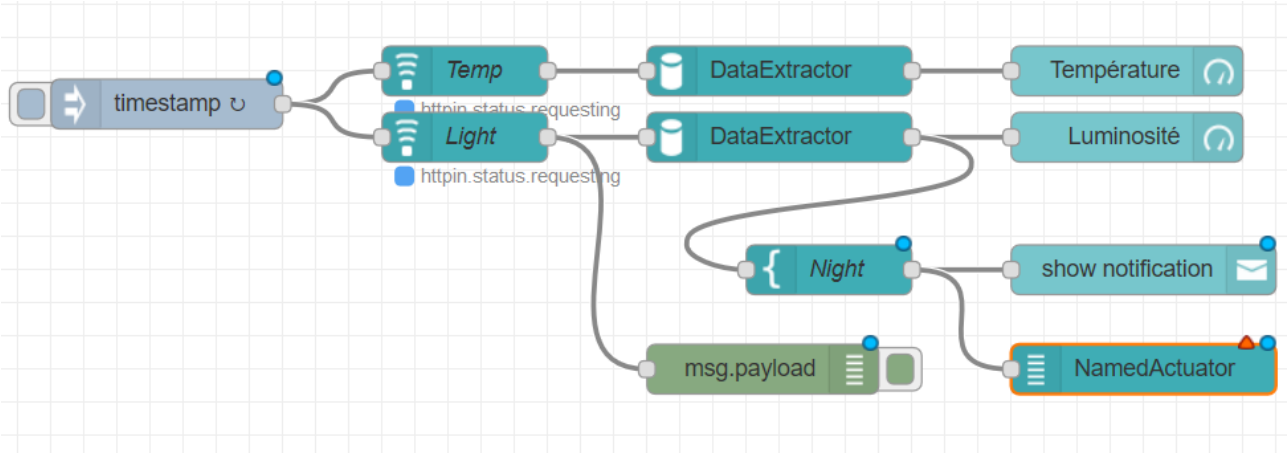

Figure 6: Node-RED flow for Fibaro sensor data retrieving

- The **NamedSensor** nodes – renamed *Temp* and *Light* – being the key ones, needed to be well configured. First, the *Platform* field has to be linked with our OM2M entity, with the specific URL and Username/Password, as presented below in **Figure 7**.

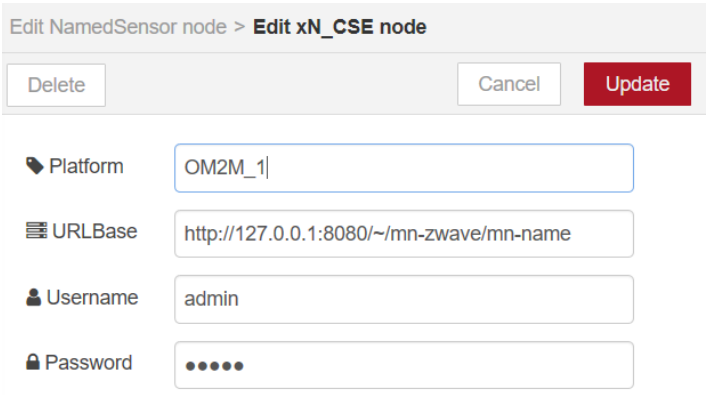

Figure 7: Platform configuration

The next thing to do was to get the *SensorName* or sensor id, given by the OM2M platform. We achieve that by creating the following GET request using **Postman**, and adding the suffix '**?rcn=4'** to get a developed answer:

GET	http://127.0.0.1:8080/~/mn-zwave/mn-name/?rcn=4

We got a positive XML answer, with, in particular, the name we desired to find:

Zw_FIBARO_MOTION_SENSOR_3259532986-3

As presented on the next **Figure 8**, we just needed to complete the field *Container*, with one of the data type given by our sensor. We made another simple request, adding the name of our sensor:

GET	http://127.0.0.1:8080/~/mn-zwave/mn-name/Zw_FIBARO_MOTION_SENSOR_3259532986-3/?rcn=4

We observed several types: TEMPERATURE, ILLUMINANCE, ALARM... We chose here to use **TEMPERATURE** and **ILLUMINANCE** for our flow. The final configuration is given in the **Figure 8**.

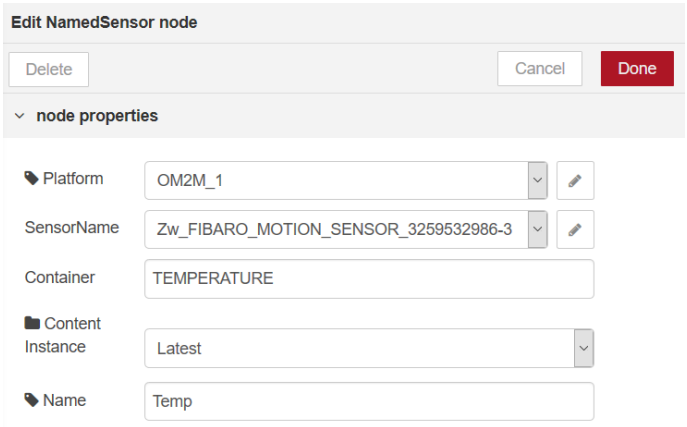

Figure 8: NamedSensor node well configured

- The two **DataExtractor** nodes were used to retrieve only the *Data* component of the payload received, so that our final **Gauges** nodes and **Debug** (*msg.playload* on the graph) node were able to display the good value. We used the Gauge nodes from the Dashboard set nodes.

We finally deploy our flow and opened our **dashboard** URL address. We put our Fibaro sensor in two different situations: **outside** the classroom (winter and night weather) and **inside** (warmer and lighted). The two results, which are coherent, are presented in the **Figure 9** below:

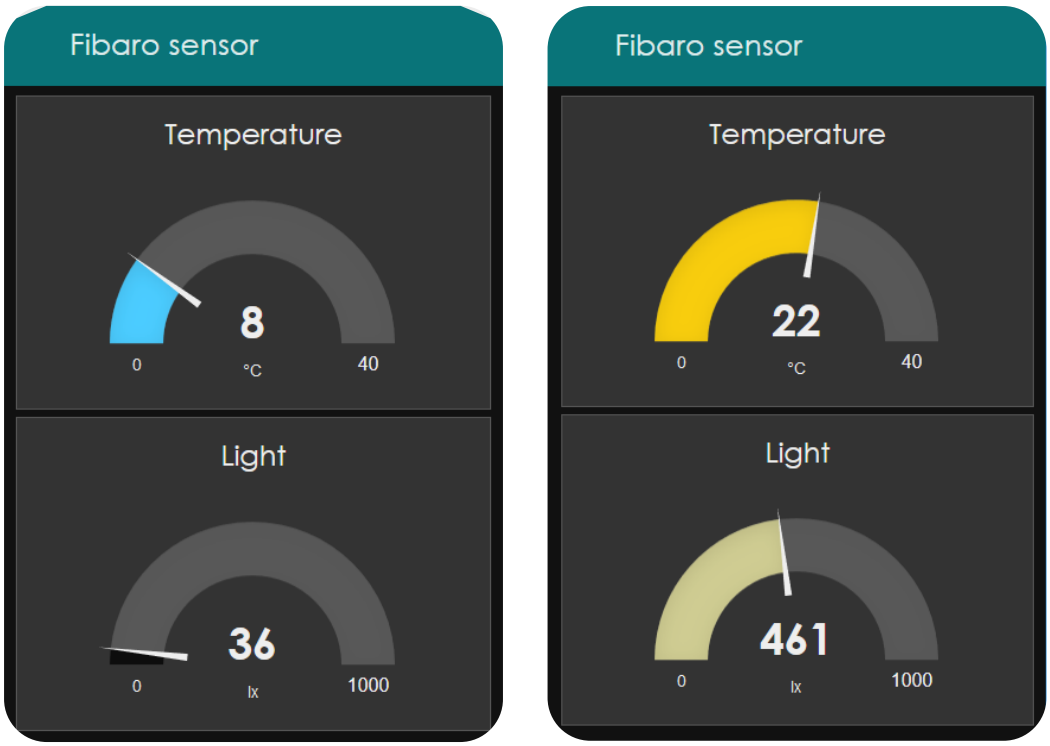

Figure 9: Node-RED dashboard view outside (left) and inside (right)

The next thing to do was to interact with the other hand of our system: the **hue** lamp. The idea was to **trigger actions depending on the value of the sensor data**. For instance, if the luminosity level is to low, we need to switch on the lamp, or if an alarm situation is triggered, the lamp has to turn in red. Unfortunately, we just had the time to implement a conditioning node, called *Night* in our flow (**Figure 6**), which allows to trigger an action with the node **NamedActuator**, we had not the time to configure properly.

CONCLUSION

To conclude, we have seen that **OM2M** is a standard for the Internet of Things providing a great platform based on **interoperability**. It is used by numerous companies and can compete with other standards like Homekit, Fiware or Thread.

During these practical works, we had to create an entire IoT architecture, using OM2M. We learned **how to access the different resources** stored in the OM2M platform and how to add some others using HTTP requests. At first, using **Postman** and, then, using an HTTP client written in **Java** (with RESTful methods).

Moreover, we have implemented a monitoring application using an **IPE** server to interact with the proprietary Philips lamps. Using an HTTP server (and the HTTP client written before), we could subscribe to the lamp resource, retrieve its state and modify it.

Finally, we developed a **composite application** to turn on a lamp regarding a sensor state. For this, we used our last application (IPE for the lamp) and **Node-Red** (more specifically the *node-red-contrib-ide-iot* package). It allowed us to retrieve OM2M resources using HTTP requests and give us a great interface to display the data.

With these practical works, we have manipulated for the first time an IoT architecture using a powerful standard. We learnt how to connect the different elements of a project (sensors, gateway, application…) and the different linked methods. The three sessions were very interesting, but we might have wished **more time** (maybe one more session), to **finish properly the last project** or to spend more time on the overall **understanding of the architecture**.

ANNEXES

ANNEXE 1 : CLASS CLIENT.JAVA

```
package fr.insat.om2m.tp2.client;

import java.io.IOException;
import java.io.UnsupportedEncodingException;

import org.apache.commons.io.IOUtils;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpPut;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;

public class Client implements ClientInterface {

	public Response retrieve(String url, String originator) throws IOException {
		Response response = new Response();
		// Instantiate a new Client
		CloseableHttpClient client = HttpClients.createDefault();
		// Instantiate the correct Http Method
		HttpGet get = new HttpGet(url);
		// add headers
		get.addHeader("X-M2M-Origin", originator);
		get.addHeader("Accept", "application/xml");
		try {
			// send request
			CloseableHttpResponse reqResp = client.execute(get);
			response.setStatusCode(reqResp.getStatusLine().getStatusCode());
			response.setRepresentation(IOUtils.toString(reqResp.getEntity()
					.getContent(), "UTF-8"));
		} catch (IOException e1) {
			e1.printStackTrace();
		} finally {
			client.close();
		}
		// return response
		return response;
	}

	public Response create(String url, String representation, String originator,
String type) throws IOException {
		Response response = new Response();
		// Instantiate a new Client
		CloseableHttpClient client = HttpClients.createDefault();
		// Instantiate the correct http Method
		HttpPost post = new HttpPost(url);
		// add headers
		post.addHeader("X-M2M-Origin", originator);
		post.addHeader("Content-type", "application/xml; ty=" + type);
		// add the body to the request
		post.setEntity(new StringEntity(representation));
		try {
			// execute the HTTP request and receive the HTTP response
			CloseableHttpResponse reqResp = client.execute(post);
```

```
                    // get the status code of the response
                    response.setStatusCode(reqResp.getStatusLine().getStatusCode());
                    // get the body of the response

      response.setRepresentation(IOUtils.toString(reqResp.getEntity().getContent(),"UTF
-8"));
              } catch (Exception e1) {
                    e1.printStackTrace();
              } finally {
                    client.close();
              }
              // return response
              return response;
      }


      public Response update(String url, String representation, String originator)
throws IOException {
              Response response = new Response();
              // Instantiate a new Client
              CloseableHttpClient client = HttpClients.createDefault();
              // Instantiate the correct http Method
              HttpPut put = new HttpPut(url);
              // add headers
              put.addHeader("X-M2M-Origin", originator);
              put.addHeader("Content-type", "application/xml");
              // add the body to the request
              put.setEntity(new StringEntity(representation));
              try {
                    // execute the HTTP request and receive the HTTP response
                    CloseableHttpResponse reqResp = client.execute(put);
                    // get the status code of the response
                    response.setStatusCode(reqResp.getStatusLine().getStatusCode());
                    // get the body of the response

      response.setRepresentation(IOUtils.toString(reqResp.getEntity().getContent(),"UTF
-8"));
              } catch (Exception e1) {
                    e1.printStackTrace();
              } finally {
                    client.close();
              }
              // return response
              return response;
      }

      public Response delete(String url, String originator) {
              // TODO Auto-generated method stub
              return null;
      }

}
```