Département du GEI
INSA Toulouse
2018-2019

# Middleware for the Internet of things


Tehema TEITI
Michael NEGATU
5th year - ISS - B1
01/12/2019

Under the direction of: Thierry Monteil

TP supervisor : KHADIR Karima

# Introduction

The aim of this report is to position the new IoT standards and software architectures adapted to connected objects. We will see through this report that the use of a middleware layer allows in particular to ignore the varied nature of objects to connect. During the practical works, we studied the Middleware OM2M which is an implementation of the oneM2M standard for Machine-to-Machine communication.

# 1. Know how to position main standards of the Internet of Thing

With the emergence of Iot in recent years, one of the main problems is the heterogeneity of the large number of devices connected. In fact, every connected device is different in terms of need it meets ( QoS requirements, rates …) and the communication protocol it uses ( Wifi, 3G, LoRa …). The challenge is to develop a system capable of analysing the information regardless of the nature of the device : thus, the name machine to machine.
OneM2M, created in 2012, has proposed horizontal platform that provides telecom operators with a standardized platform of services for their various applications. The idea is to have a layer of services that allows to access different applications in a homogenous way through a standardized API.

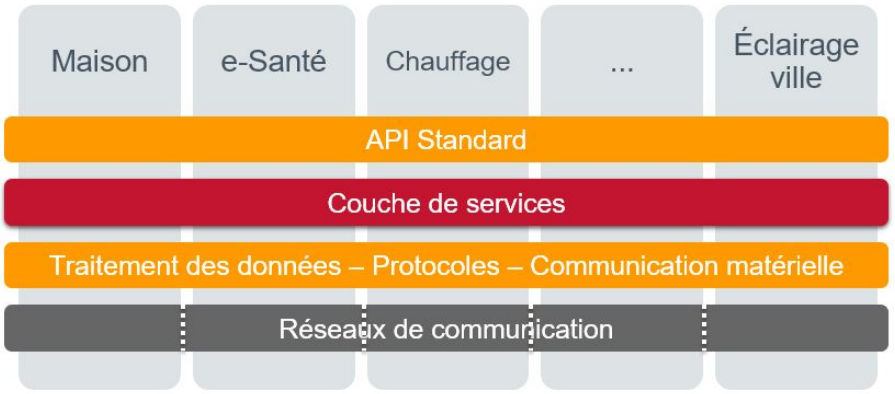



Figure 1 : Om2M service

The main advantages of OM2M are as follow :

- facilitates collaboration between industrialists
- allows interoperability between different machines
- is reliable and guarantees a good Quality of Service
- is easy to use and reduces costs for service providers

| | Standard coverage | Deployment model | Data model | Hardware platform | Security |
|---|---|---|---|---|---|
| oneM2M | Generic IoT service platform designed for multiple verticals | Centralized architecture. Multiple gateways register to a server | Resource-oriented architecture (REST) | Several types of hardware | Use HTTPS to secure communication |
| Homekit | Designed for smart-home appliances | Connected objects have to be compatible with Homekit. They are manageable through an iOS application | | Only Apple devices | Software authentication |
| LWM2M | Designed for lightweight and low power devices on a variety of networks. | Centralized architecture. Communication between a LWM2M server and a LWM2M client. | Resource-oriented architecture (REST) | Implemented in Java, C, C++ and C# | DTLS based security |
| Thread | Designed for low-bandwidth application layers and able to operate | Distributed architecture. Enable device-to-device and device-to-cloud communication. | | Thread devices use IP networks (Wi-Fi, Ethernet, …) | Only authenticated devices can join the network. The network is secured with a network key. |

Fig 2 : Positioning of oneM2M compared to other standards

# 2. Deploy a standard-compliant architecture and implement a sensor network system to services

## 2.1. Deploy and configure an IoT architecture using OM2M

OM2M is an Eclipse open source project which implements the oneM2M standard. It provides a platform based on the Eclipse framework.

In an OM2M network, there are 2 types of entities. CSE (Common Services Entity) offering services and Application Entity (AE) applications offered by the CSE. Inside the CSE we can find two nodes :

- IN (Infrastructure Node): Data collection server (points to MN or ASN)
- MN (Middle Node): relay node between DNA and IN (Gateway)

In the same we can find different devices under the AE :

- DNA (Application Dedicated Node): application requiring a gateway to communicate with the M2M network
- ASN (Application Service Node): application that can communicate directly with the M2M network

In the Eclipse OM2M project, we deployed two components :

- the MN-CSE, containing resources,
- the IN-CSE can also contain resources but especially often pointing to a remote-CSE type MN or ASN.

An example of a resource architecture, used in the lab is the following:

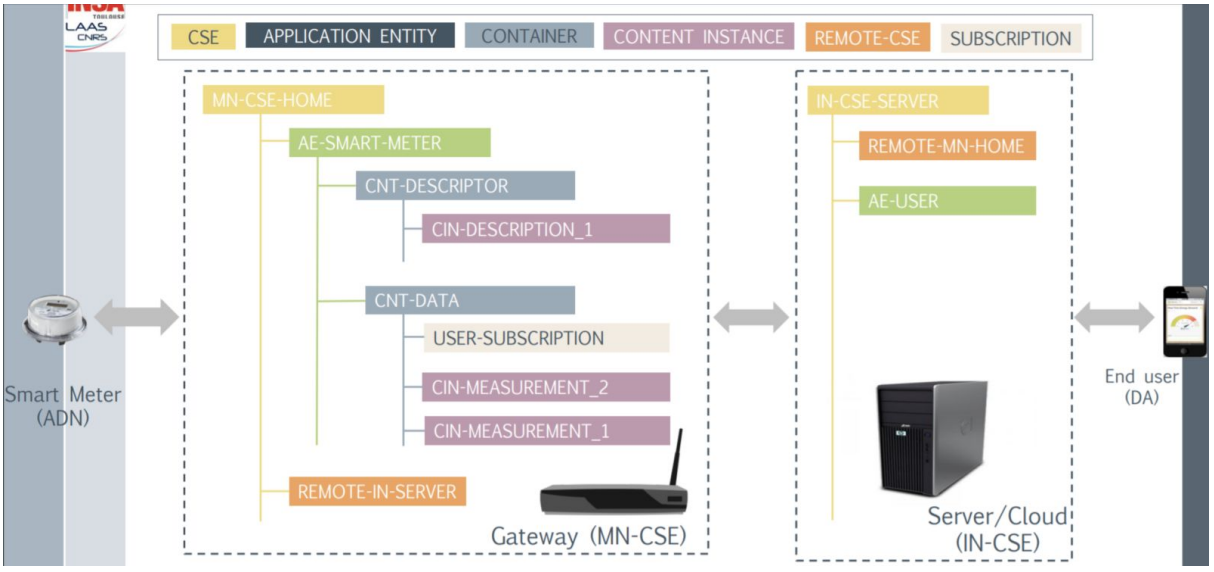



Fig 3 : OM2M resource architecture

## 2.2. Interact with objects using a REST architecture

Now that we have explained the architecture, we are going to discuss about interacting with different devices and visualising the data sent. First, we are going to use the REST API.

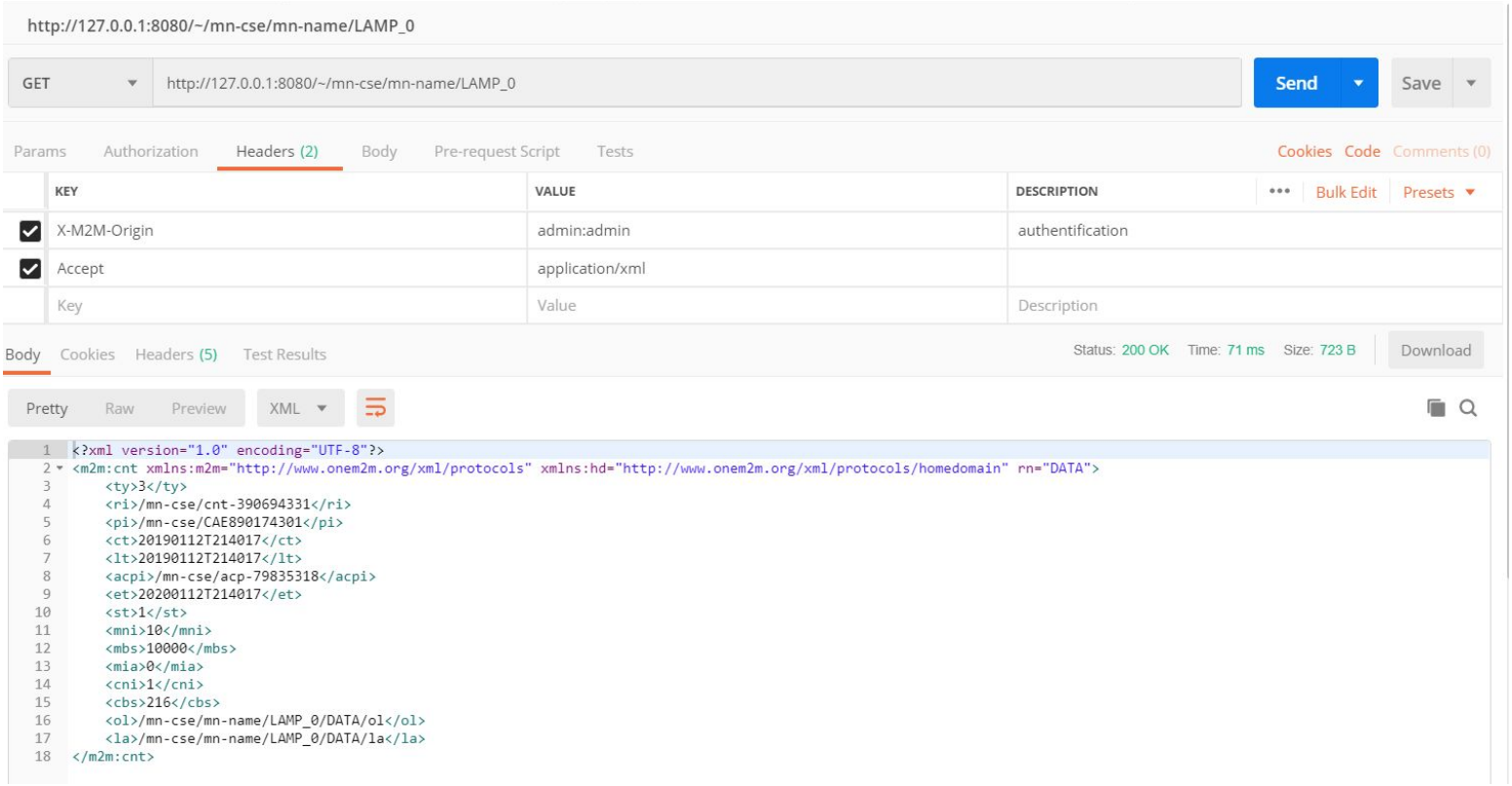
**Rest API**

Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating web services. It is resource oriented and the exchange between the client and the user can have different representation : XML, JSON, BSON. Due to the fact that it is based on HTTP operations, the rest API can interact with distant resources using these 4 requests :

**HTTP GET** : Retrieve a resource
**HTTP POST** : Create new resource
**HTTP PUT** : Update a resource
**HTTP DELETE** : Delete a resource

Using these 4 requests, The REST API is able to identify machines, get ressources, save applications, mange the containers, create container instances, authorize access rights ...

In order to test the REST API, we first used the POSTMAN software that allows the sending / receiving of requests using the REST standard. In the example below, we used the HTTP GET request to retrieve a resource that represents a lamp (simulated)




We have also defined access rights to resources via ACP rules (Access Control Policy). The operations allowed by the originator in the Access Control Rule is defined by an integer. This integer is the sum of the Operation value. This value is associated with two types of attributes: privilege (pv) and self privilege (pvs). PV is tied to the control of access to resources and the pvs to access to privilege modification functions.

So depending on the rights you want to give to a guest certain rights we do it using the ACP. To demonstrate this, we created an ACP for the data container from the previous exercise with the possibility to only access the data. On the right we have the default ACP and on the

left we have he ACP we created with the access right value of 2 to only give the right to retrieve the data.

| Attribute | Value |
|---|---|
| rn | acp_monitor3 |
| ty | 1 |
| ri | /mn-cse/acp-278203091 |
| pi | /mn-cse |
| ct | 20190118T205811 |
| lt | 20190118T205811 |
| pv | AccessControlOriginator: guest:guest; AccessControlOperation: 2 |
| pvs | AccessControlOriginator: admin:admin; AccessControlOperation: 63 |

| Attribute | Value |
|---|---|
| rn | acp_admin |
| ty | 1 |
| ri | /mn-cse/acp-622819999 |
| pi | /mn-cse |
| ct | 20190118T203239 |
| lt | 20190118T203239 |
| pv | AccessControlOriginator: admin:admin, /mn-cse; AccessControlOperation: 63<br>AccessControlOriginator: guest:guest, *; AccessControlOperation: 34 |
| pvs | AccessControlOriginator: admin:admin; AccessControlOperation: 63 |

After manipulating the API Rest using POSTMAN, we designed our own RESTful oneM2M client in Java based on the HTTPClientAPI from Apache. This client has to contain four methods representing the HTTP operations : GET, POST, UPDATE and DELETE. Each methods wait at least the URL of the remote resource and the authentication parameters. For the POST and the PUT methods, it is also necessary to specify the type of the resource (AE, CNT, CIN, …) and the body of the request in XML to set the values of the resource to create or to update. The following code shows an example of the using of the retrieve method :

```
public static void main(String[] args) throws IOException {
    // setup
    Client client = new Client();
    String url = "http://localhost:8080/~/in-cse";
    String originator = "admin:admin";

    // GET request
    Response response = client.retrieve(url, originator);

    // show response
    System.out.println(response.getRepresentation());
}
```

Here, we use the retrieve method to get information about the CSE representing the server and we obtain the following response :

```
<?xml version="1.0" encoding="UTF-8"?>
<m2m:cb xmlns:m2m="http://www.onem2m.org/xml/protocols" xmlns:hd="http://www.onem2m.org/xml/protocols/homedomain" rn="in-name">
   <ty>5</ty>
   <ri>/in-cse</ri>
   <ct>20190112T215008</ct>
   <lt>20190112T215008</lt>
   <acpi>/in-cse/acp-230473069</acpi>
   <cst>1</cst>
   <csi>in-cse</csi>
   <srt>1 2 3 4 5 9 14 15 16 17 23 28</srt>
   <poa>http://127.0.0.1:8080/</poa>
</m2m:cb>
```

The response is in XML and represents the resource of the Infrastructure Node.

In order to facilitate the creation or the modification of a resource, two tools provides methods to interact with Java objects and XML representations :

- the JABX tool that enables the marshalling and the unmarshalling of a resource
- the oBIX library that enables the encoding of an oBIX object to an XML string and the reverse operation

For instance, the following code shows the marshalling and the unmarshalling operations on a simple AE using JABX tool.

```
public static void main(String[] args) {
    MapperInterface mapper = new Mapper();

    // marshalling of an AE object
    AE ae = new AE();
    ae.setRequestReachability(false);
    String marshalled = mapper.marshal(ae);
    System.out.println("Marshalling:");
    System.out.println(marshalled);

    // unmarshalling
    AE unmarshalled = (AE) mapper.unmarshal(marshalled);
    unmarshalled.setRequestReachability(true);
    System.out.println("Marshalling after unmarshalling:");
    System.out.println(mapper.marshal(unmarshalled));
}
```

```
Marshalling:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<m2m:ae xmlns:m2m="http://www.onem2m.org/xml/protocols">
    <rr>false</rr>
</m2m:ae>

Marshalling after unmarshalling:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<m2m:ae xmlns:m2m="http://www.onem2m.org/xml/protocols">
    <rr>true</rr>
</m2m:ae>
```

Here, the marshalling operation consists of creating an AE Java object and transforms it into an XML string. The unmarshalling operation convert the previous XML string into an AE Java object, toggles its request reachability parameter to true and marshals this object into an XML string. We can see that we were able to have an XML representation of the AE Java object, to retrieve the object representation from this XML string and to modify the object.

Then we used the oBIX library to create oBIX objects and to encode them into XML representations (and the reverse operation). The following code shows the encoding of a simple oBIX object into an XML string and the decoding of this string into an oBIX object.

```java
public static void main(String[] args) {
    // oBIX object
    Bool value = new Bool(true);
    Str location = new Str("home");
    Obj object = new Obj();
    object.add("value",value);
    object.add("location",location);

    // encoding
    String result = ObixEncoder.toString(object);
    System.out.println(result);

    // decoding
    Obj decoded = ObixDecoder.fromString(result);
    decoded.add("temperature",  new Int(20));
    decoded.dump();
}
```

```xml
<obj>
  <bool name="value" val="true"/>
  <str name="location" val="home"/>
</obj>

<?xml version="1.0" encoding="UTF-8"?>
<obj>
  <bool name="value" val="true"/>
  <str name="location" val="home"/>
  <int name="temperature" val="20"/>
</obj>
```

Here, we create an oBIX object with two elements and we encode it into an XML string. With this string, we decode it into an oBIX object and we add a new element to check if we can manipulate the decoded object.

Thus, we can use these tools to create easily XML representations of OM2M resources (AE, CNT, CIN) with Java objects without writing by hands this XML representation. Then, the methods implementing the HTTP operations can use these XML representations to create or update a resource on the platform. In this way, we used JABX to create an ACP, two AE and containers and content instances for each AE. JABX objects are marshalled using a mapper, and xml representations obtained are used into HTTP POST functions to create ressources on the MN. On the following code, we can see that we create an ACP that allows only the retrieve operation for guests and a temperature sensor AE and humidity sensor AE with their containers and content instance.

- mn-name
  - acp_admin
  - acp_monitor3
  - TEMPERATURE_SENSOR
    - DESCRIPTOR
    - DATA
      - cin_1
  - HUMIDITY_SENSOR
    - DESCRIPTOR
    - DATA
      - cin_2
  - in-name

```java
Mapper mapper = new Mapper();
String temperature = "TEMPERATURE_SENSOR";
String humidity = "HUMIDITY_SENSOR";

// create ACP
createAcp(mapper);

// Create AE temperature sensor
createAe(mapper, temperature);
createCnt(mapper, "DESCRIPTOR", URL_MN + "/" + temperature);
createCnt(mapper, "DATA", URL_MN + "/" + temperature);
createCin(mapper, "cin_1", "15", URL_MN + "/" + temperature + "/DATA"

// create AE humidity sensor
createAe(mapper, humidity);
createCnt(mapper, "DESCRIPTOR", URL_MN + "/" + humidity);
createCnt(mapper, "DATA", URL_MN + "/" + humidity);
createCin(mapper, "cin_2", "20", URL_MN + "/" + humidity + "/DATA");
```

## 2.3. Integrate a new technology into an IoT architecture

Now that we have seen how to interact with devices using the REST API, we are going to try to integrate an a new technology in an existing architecture. To do so, we will use some Philips HUE lamps that use the ZigBee technology between the bridge and the lamps and provides a REST API at the bridge level. However the OM2M server can't be directly linked to the HUE bridge. This is the reason why we need to develop an interworking proxy entity. The role of an IPE is to map the values coming from the devices and to enable the execution of operations on the device directly from a oneM2M system. In our case, with the use of an IPE, it is possible to interact with a client connected to Philips lamps from an OM2M webpage. It is actually an http server listening to OM2M. With an IPE we will be able to store the data sent by the Philips HUE lamps ( Bottom -> up) and send actions to the device ( Up -> bottom).

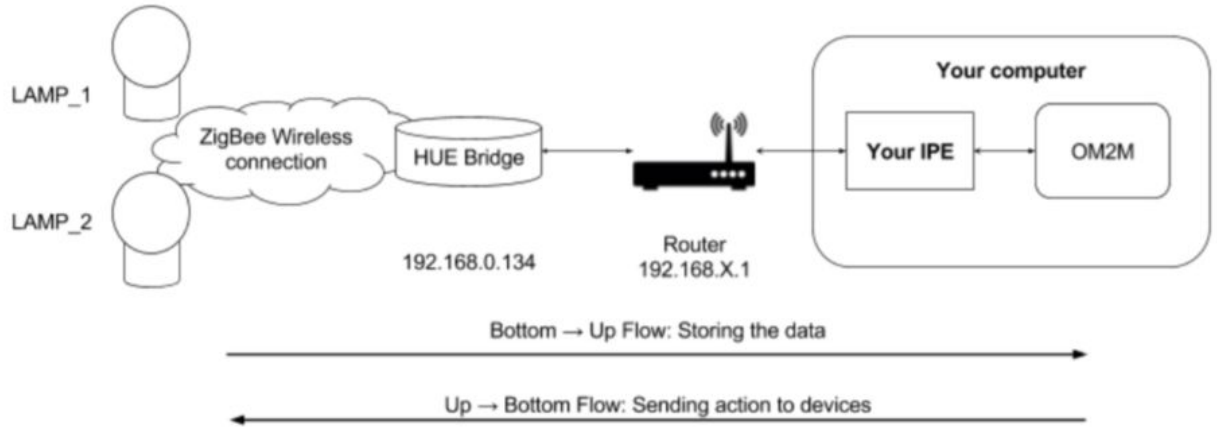



Fig 4 : Architecture to deploy to interact with Philips HUE lamps

To help us develop our IPE, philips provides an SDK for Java. The first step was to interact with the Philips HUE lamps. To do so, we used the HUE library. Once we were connected to the HUE bridge using the IP address, we used the cache from the bridge to get available Lights. Using this we were able to change the state of a Light ( code below ).

```
public static void main(String[] args) throws Exception {
    // set connection to the HUE bridge ...

    System.out.println("Retrieving lights");
    List<PHLight> lights = HueUtil.getLights();
    System.out.println("Getting last known states");
    HueUtil.getStates(lights);


    // turn on the lamp 1 with the blue color
    HueUtil.updateDevice("1", HueUtil.getHue("BLUE"), null, null, true);
```

Now that we learn how to interact with the HUE lights, we are going to add a new resource representing this light into OM2M platform. The HueSdkListener class aims to send HTTP POST request to OM2M platform in order to create a new AE resource for each HUE light connected to the HUE bridge. We had to implement the Model class to define the parameters for the content instance of a DESCRIPTOR container for each light. oBIX objects are used to initialize these parameters and are encoded into XML representation. The following code shows the parameters defined to describe HUE lights:

```
public static String getDescriptorRep(String cseId, String appId,
        String location) {
    String prefix = "/" + cseId + "/" + Main.CSE_NAME + "/" + appId;
    // Create the obix object
    Obj obj = new Obj();
    obj.add(new Str("location", location));
    obj.add(new Str("type", "AMB_LAMP"));
    obj.add(new Str("unit", ""));
    // OP GetState from SCL IPU
    Op opStateDirect = new Op();
    opStateDirect.setName("GET");
    opStateDirect.setHref(new Uri(prefix + "/DATA/la"));
    opStateDirect.setIs(new Contract("execute"));
    obj.add(opStateDirect);

    // OP set on and set off ....


    return ObixEncoder.toString(obj);
}
```

After creating a new AE, we have to push new data on OM2M platform when a lamp is updated (turned on/off, color change…). HueSdkListener implements the onCacheUpdated() method that is called when a lamp connected to the bridge has been updated. When an update is detected, the method call the pushState() method from the Monitor class to create a new content instance on OM2M platform. The following code shows the implementation of these methods.

```
@Override
public void onCacheUpdated(List cacheNotificationsList, PHBridge bridge) {
    // Here you receive notifications that the BridgeResource Cache
    // was updated. Use the PHMessageType to
    // check which cache was updated, e.g.
    if (cacheNotificationsList.contains(PHMessageType.LIGHTS_CACHE_UPDATED)) {
        System.out.println("Lights Cache Updated");

        for (PHLight light : bridge.getResourceCache().getAllLights()) {
            monitor.pushState(light.getIdentifier(),
                    light.getLastKnownLightState().isOn().toString(),
                    HueUtil.getColor(light.getLastKnownLightState().getHue()),
                    light.getLastKnownLightState().getHue());
        }
    }
```

```
public void pushState(String lampId, String state, String color, int hue) {
    ContentInstance dataInstance = new ContentInstance();
    dataInstance.setContent(Model.getDataRep("Home", state, color, hue));
    dataInstance.setContentInfo("application/obix:0");
    try {
        client.create("http://localhost:8080/~/mn-cse/mn-name/" + "LAMP_" + lampId
                + "/DATA", mapper.marshal(dataInstance), ORIGINATOR, "4");
    } catch (IOException e) {
        System.err.println("Error creating the content instance");
        e.printStackTrace();
    }
}
```

Now that we are able to send data from HUE lights to OM2M, we need to implement the other way: send a command from OM2M to HUE lights. To do so, we have to implement the controller, which is aimed to execute a set of queries addressed to HUE devices. In the Controller class, the execute() method has an HashMap as parameter representing a set of queries. For each query, if it is well formatted, a Java object representing a light is initialized with commands given by the query. Finally, the state wanted for the light is sent through the updateLightState() method to the bridge.

```
public static void execute(HashMap<String, String> queryStrings) {
    PHBridge bridge = PHHueSDK.getInstance().getSelectedBridge();
    String id = queryStrings.get("id");
    id = id.split("LAMP_")[1];
    PHLight l = bridge.getResourceCache().getLights().get(id);
    if(l == null) {
        System.out.println("Lamp id not found: " + id);
        return;
    }
    String hue = queryStrings.get("hue"),
           sat = queryStrings.get("sat"),
           bri = queryStrings.get("bri"),
           on = queryStrings.get("on");

    PHLightState state = new PHLightState();
    // init state ...

    // send the requeste to the HUE vrudhr
    bridge.updateLightState(l, state);;
}
```

Thanks to HUE API provides by the HUE bridge, we were able to implement an IPE that allows the communication between an HUE light using a Zigbee technology and OM2M. The IPE aimed to communicate with HUE bridge with the API, and creates resources (POST request to OM2M) and get queries (POST request from a client) with Rest API.

# 3. Deploy a high level application between various technologies using NODE-RED based on a standardized middleware

The idea here is to have a real life use case where you have devices connected to a middleware. Then you will create a high level application that will use different sources and actuators together thanks to the interoperability provided by a oneM2M middleware. Using the REST API and the IPE explained in the previous parts, we will try to interact with real devices: a multi sensor (Fibaro motion/luminosity/temperature/alarm sensor, connected in Z Wave technology) and the HUE Lights connected thanks to your IPE.

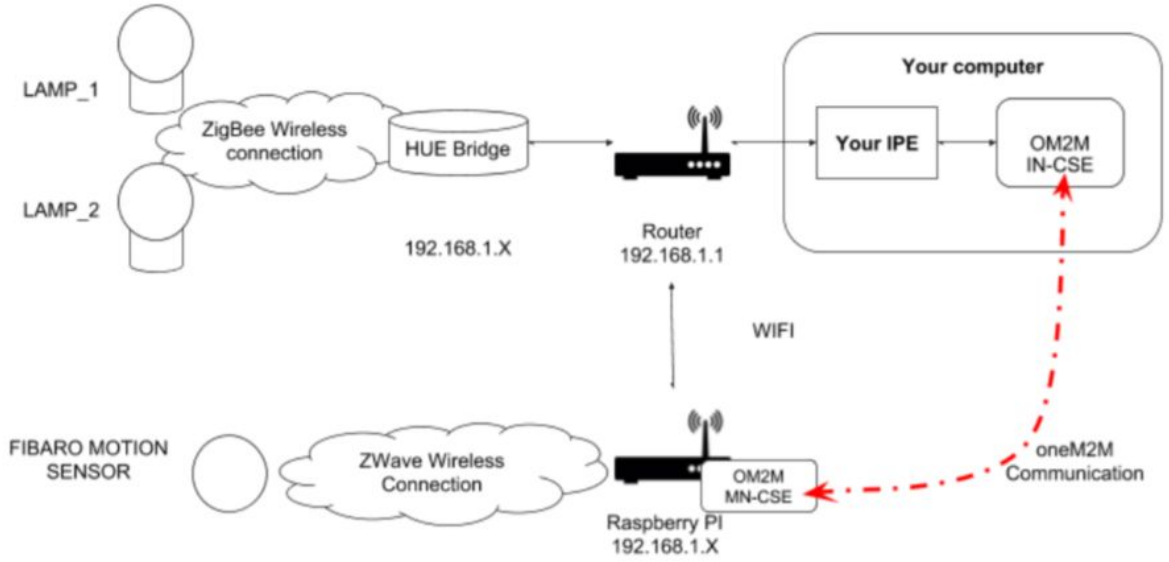
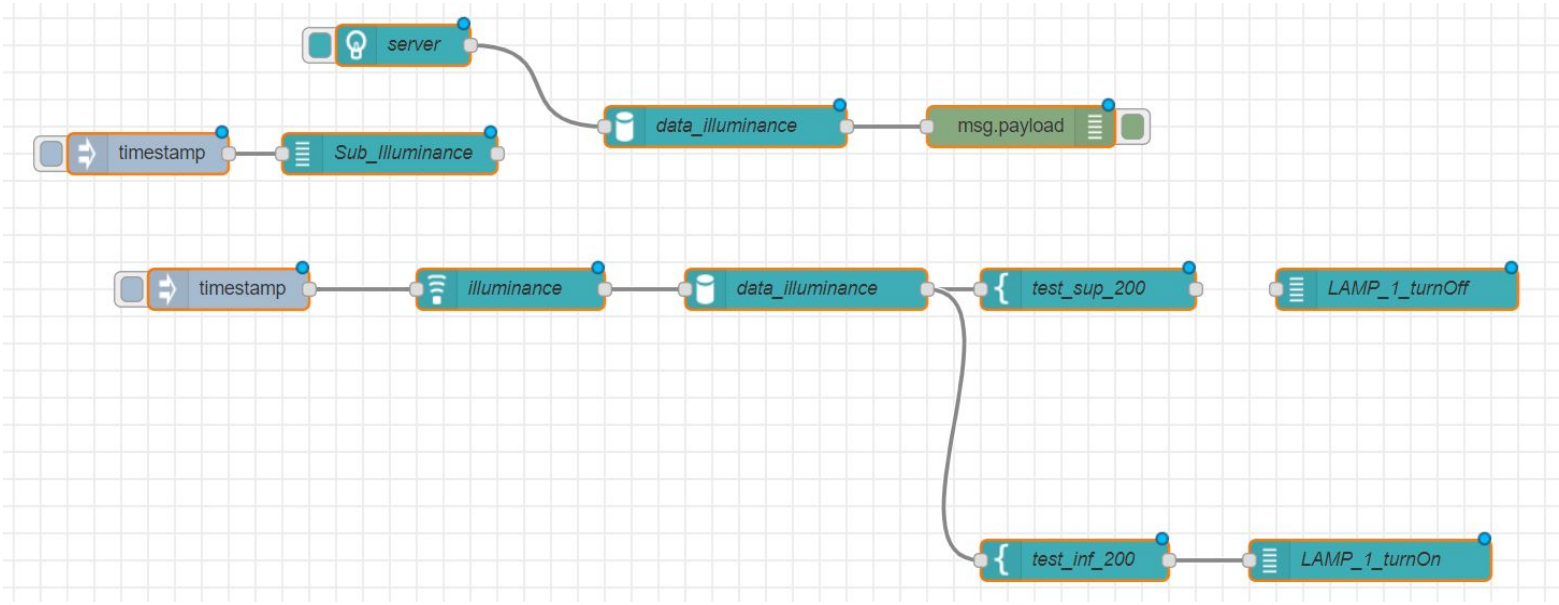



Fig 5 : Architecture to deploy

First, we have to connect to the fibaro motion sensor To do so, we would be launching the MN and the IPe on a raspberry pi and the IN on the computer. Once we are connected to the device, we can develop an high level application using NODE RED (it is possible to implement everything with your HTTP client, but it will be a bit time consuming). Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. Node-RED is built on Node.js and provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes. What makes NODE RED perfect for our application is the fact that is has a framework for developing IoT applications using the OM2M platform. This framework provides nodes that can directly interact with the Om2m platform ( retrieving data, sending commands, subscribing devices …).

The scenario we decided to implement using NODE RED is as follow :
When a new sensor value ( luminosity ) is received from the Fibaro motion sensor :

- If the luminosity value is too high, we switch off the lamp
- If the luminosity value is too low, we turn on the lamp
- Receive notifications from the sensor whenever a new data instance is produced.




## Conclusion

This Lab has allowed us to deploy a middleware adapted to the Internet Objects. The OM2M platform was used to connect smart devices to a database OM2M dynamic, editable by a client, and viewable via a web page. We have seen how Interoperability can be important in particular the use of an API for a specific device that totally disregards the physical layer. Also the use of the underwriting process and the JAVA implementation of an IPE highlighted the bidirectionality of the standard. Finally we created a high level application that uses different sources and actuators together.