MIDDLEWARE FOR THE INTERNET OF THINGS

TP REPORT – 5 ISS INSA TOULOUSE

Last name 1: Kim

First name 1: Isabella Mi Hyun

Last name 2: Pajanissamy

First name 2: Poonkuzhali

Group: A2

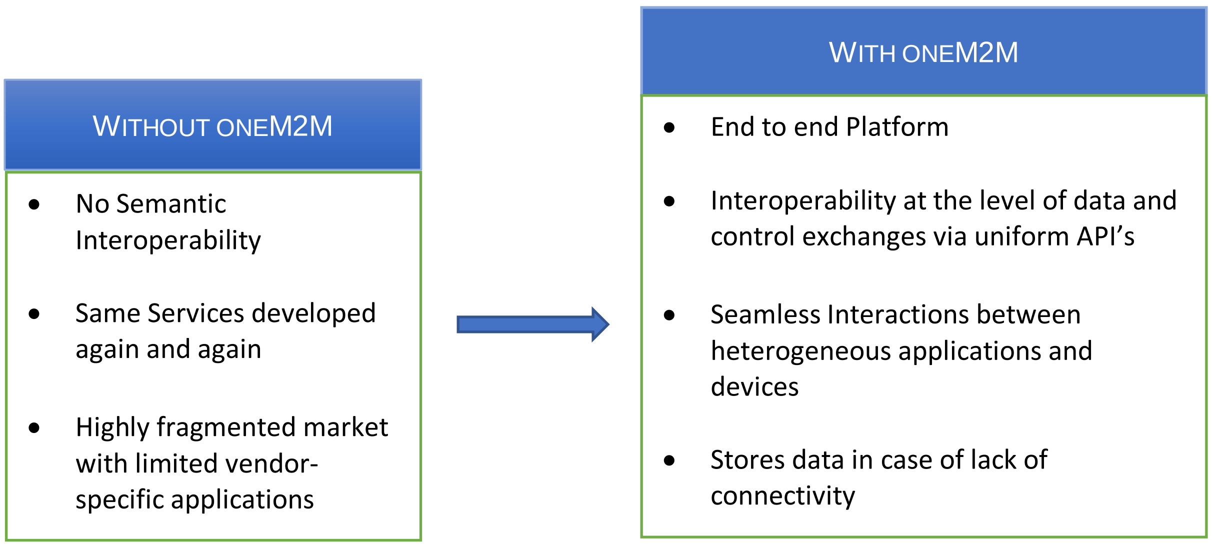
TP Teacher : Guillaume Garzone

Github repository: https://github.com/mihyunki/MiddlewareIoT-TP-INSAT

1. KNOW HOW TO POSITION THE MAIN STANDARDS OF THE INTERNET OF THINGS

Smart objects produce large volumes of data that needs to be managed, processed, transferred and stored safely. In this context, **standardization** becomes a key element to achieve universally accepted specifications and protocols for true **interoperability** between devices and applications. A wide range of technologies work together to connect things in the Internet of Things (IoT) and there are lots of existing wired and wireless protocols such as Ethernet, Rs-232, Wi-Fi, Bluetooth, Cellular, Zig Bee, Z-Wave, Thread, 802.15.4, 6LoWPAN, Sig fox, LoRaWAN, etc. In the world of fragmented market in the IoT world, there are less vendor specific application and they do not interact with each other.

OneM2M is a **service layer** that is situated between the hardware/software communication and the IoT application. It provides many functions needed for IoT applications, such as data transport, security/encryption, scheduling, event notifications, etc., as well as a standardization of the system architecture. The OneM2m provides a **horizontal layer** enhancing interoperability between heterogeneous devices.



WITHOUT ONEM2M

- No Semantic Interoperability
- Same Services developed again and again
- Highly fragmented market with limited vendor-specific applications

WITH ONEM2M

- End to end Platform
- Interoperability at the level of data and control exchanges via uniform API's
- Seamless Interactions between heterogeneous applications and devices
- Stores data in case of lack of connectivity

The M2M Service Capabilities are both at network level (M2M Service Capabilities in the Network Domain) and at local level (M2M Service Capabilities in the M2M Gateway and in the M2M Device). These Service Capabilities are the set of functionalities defined in the specification and are used to put in communication applications among them; both network applications, and gateway and device applications. The architecture of oneM2M follows the REST client for handling the resources by sending the HTTP requests through the AJAX web interface for monitoring sensors and controlling actuators.

The important factors of oneM2M are:

Communication binding, when one or more than one interfaces are combined between different kinds of protocols, which require a translation protocol to translate the interfaces of different kinds of protocols (example: HTTP, CoAP, MQTT, etc). The interaction is implemented during the TP1 and TP2 using HTTP and REST client using the web browser.

Interworking, allows non-ETSI-compliant devices such as Zig bee, Phidgets, and many other technologies to interwork with the ETSI M2M standard by performing mapping/converting operations. During the practical session (TP3), we have implemented this concept where we made connection between Philips HUE lamps that use ZigBee technology and the oneM2M.

Device management, is essentially a specific aspect of oneM2M to provide unified & simplified management APIs for M2M applications.

ETSI (European Telecommunications Standards Institute): It is one of the founding partners in oneM2M the global standards initiative that covers requirements, architecture, Application Programming Interface (API) specifications, security solutions and interoperability for M2M and IoT technologies.

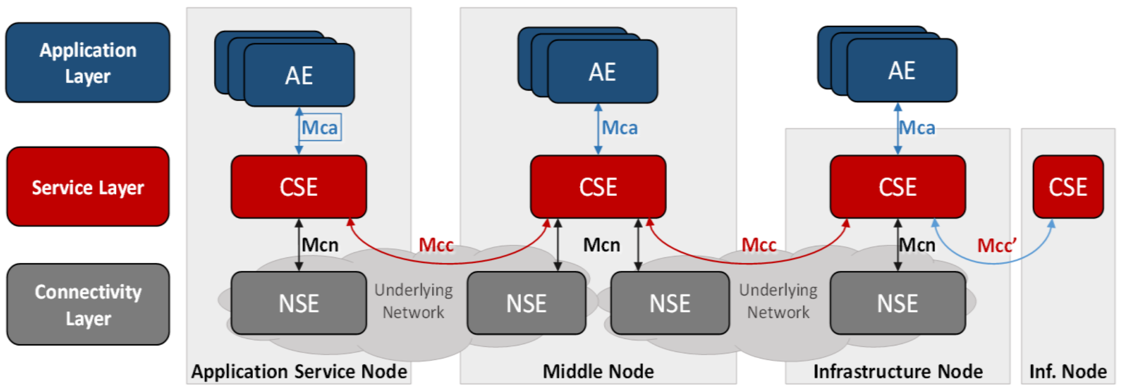


Figure 1.1 - Functional Architecture of OneM2M

The oneM2M standard organization has defined the standard of application/ service entities that has been used in common in M2M/IoT environment.

In the **CSE (Common Service Entity)**, twelve services have been defined and each service has been provided through the CSF (Common Service Function). Currently, the 12 kinds of CSF are device management, group management, register, discovery, subscription and alert, application/service management, security, communication management and charging, etc.

The oneM2M is using a **REST**ful resource-based architecture, which is a very important characteristic. In other words, every service and data defined in the oneM2M are expressed in the resource tree. Each service and data are being controlled through the **Create, Read, Update, Delete, and Notify (: CRUDN)** method of the CSFs.

The CSF is called through the API reference point – **Mcc** (CSE to CSE), **Mca** (CSE to AE) and **Mcn** (CSE to NSE)- defined in the oneM2M entity. The API request/response is rendered in the oneM2M Primitive and finally transmitted as a message.

Mca: the interface point between **AE (Application Entity)** and CSE which provides the applications access to the common services.

Mcc: the reference point between two CSEs.

Mcn: the reference point between CSE and underlying **NSE (Network Service Entity)** like transport and connectivity services.

Mcc': the interface between two M2M service providers.

2. DEPLOY A STANDARD-COMPLIANT ARCHITECTURE AND IMPLEMENT A NETWORK OF SENSORS

2.1. DEPLOY AND CONFIGURE AN IOT ARCHITECTURE USING OM2M

During our practical work sessions, we deployed an IoT architecture with OM2M, which had:

- **IN** for **Infrastructure Node**: a server
- **MN** for **Middle Node**: a gateway

For each of these nodes, we could create three entities:

- **Application Entity (AE)**: to represent the different connected applications
- **Containers (CNT)**: grouping instances of the AE, such as data, descriptor, etc
- **Content Instance (CIN)**: which are attributes of the application. These attributes can include both sensor readings and the control of actuators

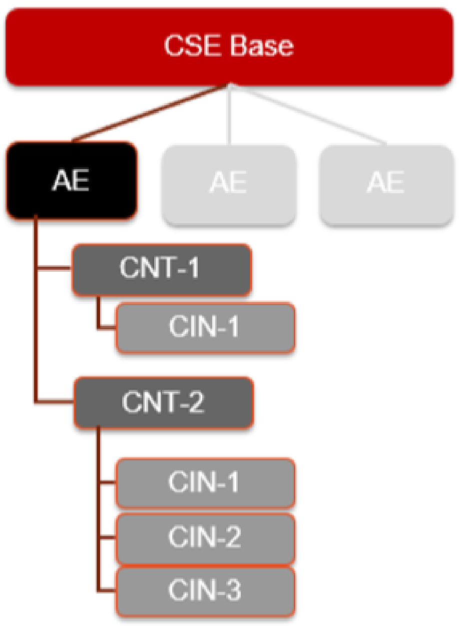
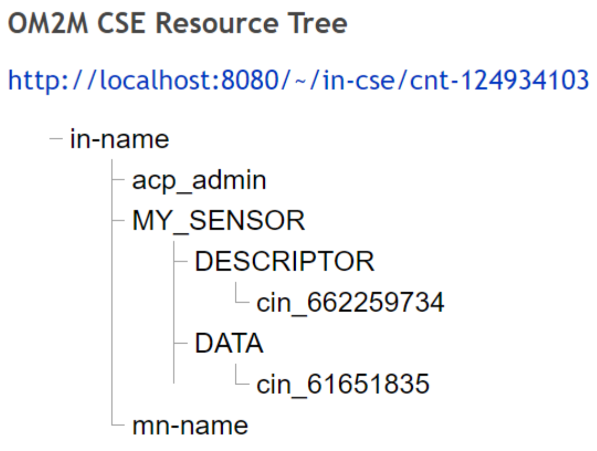


Figure 2.1.1 – OM2M architecture.

The middle node (MN) situated in the home gateway handles the data exchanged with each device. For example, in TP1, we used a simulation environment that switched on/off lamps. We could see that changing the status on the lamp in the API created modifications on the MN.

In our practical work (TP1), we have chosen to represent each component of our system (smart meter, luminosity sensor and temperature sensor) with an AE, and each AE had a:

- **DATA container**: to store information about status and values measured by the sensor
- **DESCRIPTOR container**: contains information about the AE



OM2M CSE Resource Tree

http://localhost:8080/~/in-cse/cnt-124934103

```
- in-name
    ├ acp_admin
    ├ MY_SENSOR
    │   ├ DESCRIPTOR
    │   │   └ cin_662259734
    │   ├ DATA
    │   │   └ cin_61651835
    └ mn-name
```

Figure 2.1.2 – Example of the AE sensor created in TP1 with its containers DATA and DESCRIPTOR.

OM2M ARCHITECTURE:

A content instance (CIN), as mentioned above, contains information. These are not modifiable, it is necessary to send an order back to the lamp to update its information and therefore its resulting status.

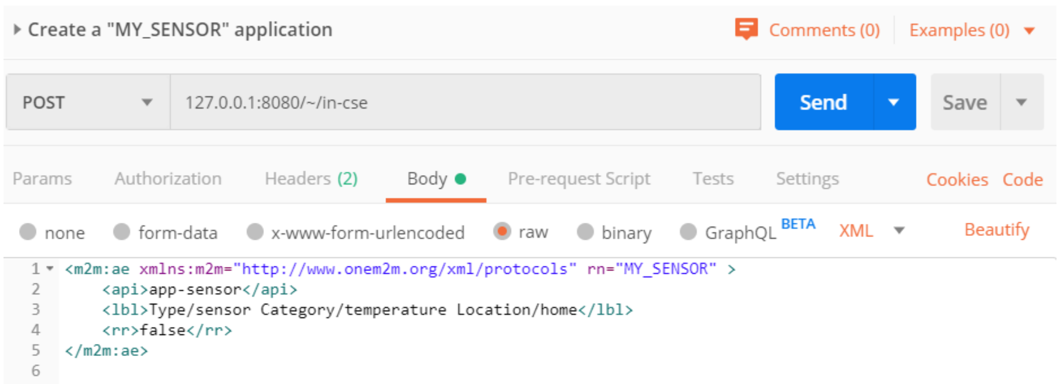
We can therefore access all the history of the lamp commands on the local server we launch, by accessing the different CINs stored in the DATA container of the lamp to which we are connected. And it is possible to put containers in containers.

At the end of TP3 we therefore have a JAVA application allowing us to change the last state of a lamp via an exchange information, through the use of the Philips HUE library. We therefore interfaced HUE with OM2M through an IPE (Interworking Proxy Entity). IPEs are gateways between the OM2M standard and protocols/implementations owners.

2.2. INTERACT WITH OBJECTS USING A REST ARCHITECTURE

We used the **REST** (Representational State Transfer) architecture for accessing and manipulating resources. The REST establishes constraints for the creation of webservices on a way that we're able to manipulate resources, which are defined by their unique URIs.

The REST uses **HTTP requests** to retrieve (GET), create (POST), update (PUT) and erase (DELETE) resources. In the first practical works, we used **POSTMAN** application which provided a good interface for building HTTP requests. '



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

Figure 2.2.1 – Example of the creation of an AE with POST method in POSTMAN.

We therefore used the OM2M open source platform and provided horizontal framework to facilitate the development of services and applications. The platform was already equipped with HTTP requests using the Apache library and the methods provided Retrieve (), Create (), Update (), Delete () to perform GET, POST, PUT, DELETE respectively. These methods have as parameter the URI describing the path in the tree structure to travel to reach the entity where to perform queries.

HTTP requests need a header to specify the format (either XML or JSON) of the application, its type (the ty= 1 for credentials, 2 for an Application Entity, 3 for a Container, or 4 for a Content Instance) as well as the originator (here "admin: admin") to access the rights and execute the whole thing.

During the TP we used the **JAXB map** of the OM2M library to serialize (with the marshal () method) JAVA objects in XML string. The library gives us the three basic entities of the eponymous standard:
- The AEs (Application Entities)
- The CNT (Containers)
- The CIN (Content Instance)

Thus we were able to use the oBIX tool for representing the xml Representations and tested it.

And in a last time we were able to manipulate the subscription, used to record a hook between the OM2M server and our AE, which allows you to be notified of changes on CINs directly via OM2M on the AE where the monitor was launched.

2.3. INTEGRATING A NEW / OTHER TECHNOLOGY IN AN EXISTING IOT ARCHITECTURE

In this part of the practical work we introduced a new technology (the Philips Hue Lamp) into the deployed architecture. This practice addresses a central problem of IoT, which is the interoperability between different technology protocols. To address this problem, we used an IPE (Interworking Proxy Entity) to translate the device technology to the oneM2M standard.

The Philips Hue Lamps (Zigbee technology) integrated into our existing IoT architecture required the integration of communication protocols distributed by Philips. The IPE works as an specialized AE that allows the **communication between OM2M and non-OM2M systems** by mapping the data of the integrated device into an OM2M resource (AE, container,etc).

In the OM2M standard, HTTP requests and responses are used to communicate with the device. In this case, with the Philips lamp, the requests and responses of the device are specified by Philips. To grant the interoperability of the system, the architecture of this use case was divided into: non OM2M-nodes and oneM2M nodes.

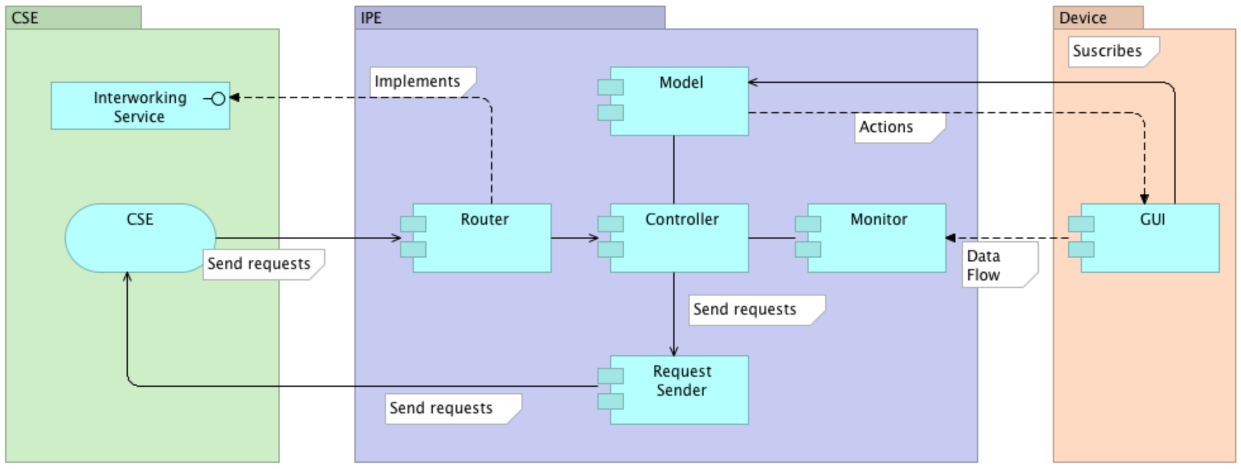


Figure 2.3.1 – Architecture of the IPE.

The architecture of the IPE used to integrate the new device in the middleware can be seen in Figure 2.3.1 above. The main components of the IPE are:

- **Controller**: receives requests from the OM2M interface and according to that implements actions on the device (such as retrieve the state, switch on a lamp, etc). This command is based on the query string provided at the creation of the DESCRIPTOR of the AE.
- **Monitor**: retrieves information from the device and pushes it into the CSE.
- **Model**: representation of the devices that we're going to monitor and control. In this case, the Hue Lamps we want to implement in our system were represented in the model as Java objects.
- **Router**: implements the InterworkingService interface and receives requests from the CSE to the IPE. It sends requests to the methods of the controller accordingly.
- **Request sender**: utility class that creates oneM2M to send to the CSE using Java objects.

In TP3 we were able to establish a communication and interact with the HUE Lamps (change on/off and change the color). Firstly we established a connection with the HUE Bridge. For that, in the *HueUtil* class, we had to set a username and the IP address to where our lamp was connected:

```
PHAccessPoint ap = new PHAccessPoint();

ap.setIpAddress("192.168.1.144");

ap.setUsername(HueProperties.getUsername());

PHHueSDK.getInstance().connect(ap);
```

After connected to the bridge, we had access to the list of lamps connected to the router so we could send commands to the lamp we wanted by using its ID. Next, to implement the IPE we had to create OM2M resources representing our devices.

After that, in *HueSdkListener* we created the representation of our device as a Java object. Using the monitor class we created a lamp object (with the ID of our device) and its resources in OM2M:

```
Monitor lamp = new Monitor();

lamp.createResources("1");
```

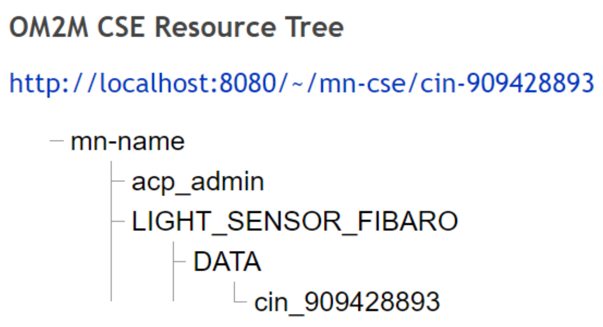
Finally, in the *HueUtil* class, we could implement changes to the color of the lamp. The code to our TP3 can be found at the github account.

3. DEPLOY A COMPOSITE APPLICATION USING VARIOUS TECHNOLOGIES THANKS TO NODE-RED,

NODE-RED is a programming interface used as a development tool to construct flux-based representations. This type of representation allows putting together APIs and services in an IoT context.

Unfortunately, due to time, we couldn't test the "Fibaro" sensor on the Raspberry Pi. But we developed in NODE-RED a system to simulate the sensor by sending random values and retrieve data from the the sensor using OM2M nodes from the IDE-OM2M framework.

We started by creating a simple AE to represent the sensor by using Application, Container, ContentInstance, as shown in Figure 3.1.



OM2M CSE Resource Tree

http://localhost:8080/~/mn-cse/cin-909428893

```
─ mn-name
    ├ acp_admin
    ├ LIGHT_SENSOR_FIBARO
    │   ├ DATA
    │   │   └ cin_909428893
```

Figure 3.1 – Example of an AE sensor created with the nodes in NODE-RED.

Using the NODE-RED stream in the Figure 3.2 below, we created an AE to represent the light sensor from the Fibaro.

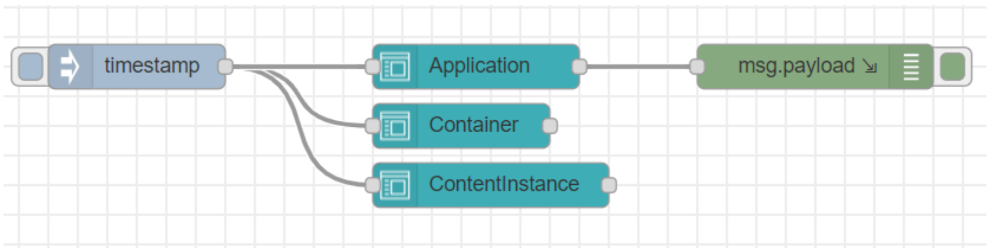


Figure 3.2 – NODE-RED stream used to create the AE with its container and content instance in OM2M.

After that, we created a stream in NODE-RED to retrieve the data from the light sensor created and switch off the lamp if the value is too high or on in the value received from the sensor is too low. For that, we used the nodes:

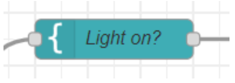
- **Inject**: to initiate the stream.

- **Named Sensor**: node to retrieve the measurements produced by a sensor using the exact name of the resource representing that sensor.

- **DataExtractor**: After receiving the data returned by the NamedSensor node, we can retrieve the value of this instance deploying this node.

With these 3 nodes we can retrieve the value from the sensor. After that we can implement some cases with conditions, as for example: switch of the lights if the value of the lights is too high or turn on if it's too low. For that, we used 2 extra nodes:

- **Simple condition**: tests if a condition is respected or not, returning true or false.



- **NamedActuator**: sends a simple command to the actuator (example: "turn on the lights") using the same name as the resource that describes it.

In the end, we created a flux to extract the data of each sensor (light, temperature, movement). And implemented a use case to turn off the lights if the value received is higher than 40 or lower than 20. This final flow can be seen in Figure 3.3 and is also in the github.

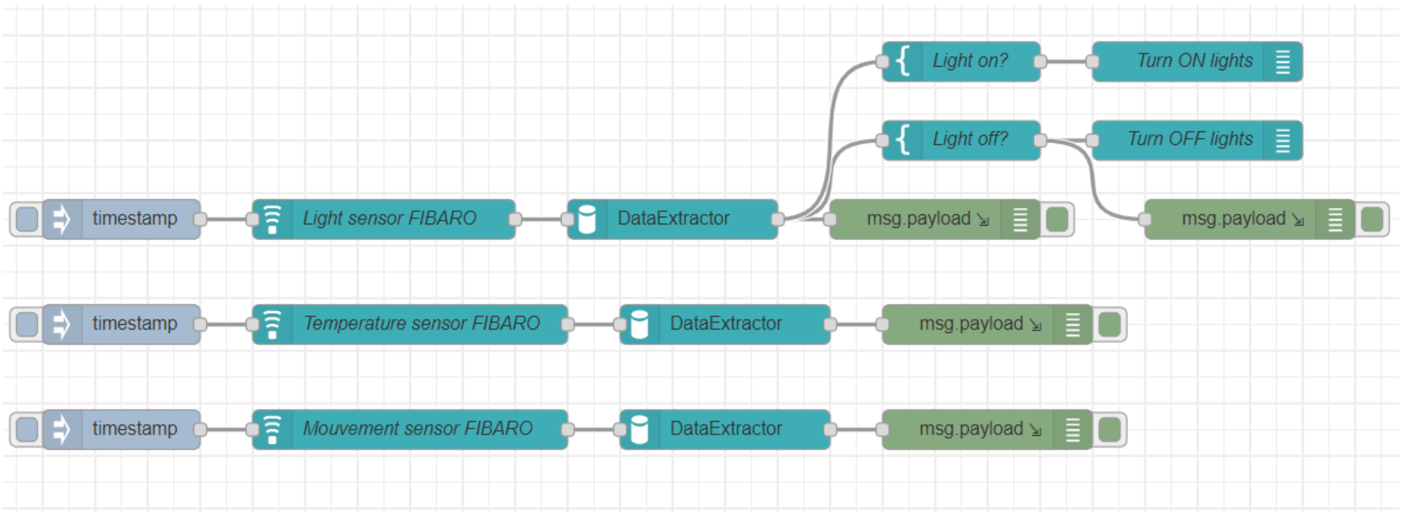


Figure 3.3 – Final flow in NODE-RED to extract data from 3 sensors and act on a sensor depending on the value received.