IoT Bridge Platform

INTERNET OF THINGS

Artificial Intelligence

University of Bologna

Mariana Oliveira Ramos
Professors: Luciano Bononi, Marco Di Felice
June 15, 2022

1 Introduction

This project was developed during the course of Internet of Things. Its aim is to implement a software platform able to acquire sensor data flows using different protocols and integrate advanced functionalities of protocol bridging and of data persistence. The PONTE1 tool inspires this project. It allows the acquisition of IoT data flows in *input*, and the creation of new/processed data flows in *output*. More specifically, the framework must be able to receive data from generic IoT data sources, supporting these protocols:

- MQTT: in this case, the user must set the broker configuration (e.g. network address) and the topic of interest;
- COAP: in this case, the user must set the URL of the COAP server;
- HTTP: in this case, the user must set the URL of the HTTP server.

To test the correct functioning of the application, three sensor simulators were created to emulate the sensor data. The sensor data consists of the output of a device that detects and responds to some type of input from the physical environment. In my case, I simulated data coming from both a **temperature** and an **humidity** sensor using random values via script.

1.1 Requirements and compilation

This project was created using the programming language JavaScript with a node.js environment. To interact with it, the user can enable its functionalities through a CLI (Command Line Interface).

Several libraries and environments were used in the making of this project. The links to each library can be found in the References section. To download all the dependencies simply run the following commands in the Command Line:

```
1    npm install yargs --save
2    npm install coap --save
3    npm install mqtt --save
4    npm install axios --save
5    npm install express --save
6    npm install --save @influxdata/influxdb-client
```

Listing 1: Requirements

To see general information about the project compilation it is possible to run:

```
1    node ./bridgeIoT.js --help
```

For more details and specific examples of the requirements and compilation of the project, there is a README.md file delivered with the code.

2 Project Architecture

2.1 Protocols

As mentioned before, the framework must be able to receive data from generic IoT data sources, supporting MQTT, COAP, and HTTP protocols.

Although the data acquisition methods differ from protocol to protocol, it is possible to group them into two categories called *Request/Reply* and *Publish/-Subscribe*.

(1) Request/Reply (2) Publish/Subscribe

The MQTT protocol belongs to the Public/Subscribe category (2). A client can act as both a Publisher and a Subscriber and has no knowledge of who will receive the message it publishes or from which client it will receive the message to which it is subscribed. Furthermore, there is always a broker who takes care of sorting the messages received based on the topic. On a temporal level, the Publishing and Subscribing phases are completely separate: a client that subscribes to a topic will not know if and when it will receive the desired message.

The Request/Response paradigm (1), of which the COAP and HTTP protocols are part, on the contrary, does not presuppose the presence of a broker. Typically the roles are divided between Server and Client. The resources are indicated by URL and not by topic, moreover, the Server must satisfy each request individually.

2.2 Functionalities

This subsection describes briefly the high-level functionalities of the tool and how each of them works. The remaining implementation details are shown in the next section.

2.2.1 Data visualization

The data visualization option allows the user to print the JSON objects acquired in real-time. This JSON objects contain the **Sensor Type** that is

being read and the **Value** received.

In regards to the MQTT protocol, the display simply consists in printing the values received by the broker for a particular topic (e.g. *temperature*). The values are displayed on the screen each time they are received, in real-time.

On the other hand, for the HTTP and COAP protocols, a request is made and it is expected to receive a response. After having obtained the response the program waits for a predetermined number of seconds and a new request is sent! This way it is possible to observe different values sent by the sensor. It is important to underline that, as far as the COAP and HTTP protocols are concerned, a new request is sent only after obtaining the response to the previous request. The COAP protocol works in a "Non-confirmable" manner, not requiring an Acknowledge by the server.

2.2.2 Data aggregation

The data aggregation option enables the user to visualize the statistical features of the sensory data. This statistics consist of the **maximum**, **minimum**, **average** and **standard deviation** and they are computed every n observations, where n is a tunable parameter.

The process is very similar to the data visualization, except the values are stored temporally in an array in order to compute the statistics that are then displayed on the screen.

2.2.3 Data storage

The data storage option consists in storing the data on the *INFLUX 2.0* database. For that, the user must provide the URL of the device hosting the INFLUX instance, the access token, the name of the bucket, and additional tags. All this information should be defined in a JSON file that is then passed through the command line. As a result of this, the flow of the sensor data is automatically sent to INFLUX and stored there.

To do that, the code used for the visualization of the data is reused but instead of displaying the results obtained on the screen, they are saved in the form of pairs **«time, value»** in the *INFLUX database*.

It is important to notice that the INFLUX interface should be used to do the database setting up phases. These setup operations are required before trying to store any data. Namely the setup of the database and bucket using the graphical interface provided with influx 2.0.

2.2.4 Data bridging

Data bridging consists of transforming an incoming data stream of a particular protocol into an outgoing data stream of another protocol. E.g. the tool is able to acquire data in *input* in any protocol and produce an *IoT* flow in *output* in any other protocol. In this case, the user must specify the destination protocol as well as the above-mentioned protocol-specific parameters in a JSON file.

Having considered three protocols MQTT, COAP and HTTP, there are six cases to be addressed:

1. COAP to MQTT bridging
2. HTTP to MQTT bridging
3. HTTP to COAP bridging
4. COAP to HTTP bridging
5. MQTT to COAP bridging
6. MQTT to HTTP bridging

Translation to MQTT (case 1. and 2.)

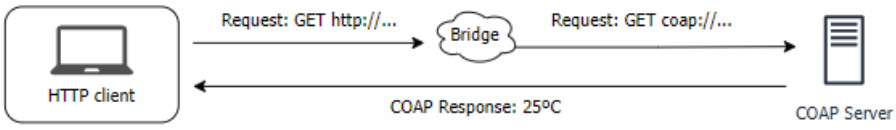
The conversion from COAP to MQTT and from HTTP to MQTT is described in the figure below.

At startup, a request is made to the COAP or HTTP server, which is replayed every preset time interval but only after receiving the response. The response obtained is published to a Mosquito broker having the topic equal to the path of the request. Essentially, the publication of the value takes place after receiving the response.

Translation between HTTP and COAP (case 3. and 4.)

The conversion from HTTP to COAP or from COAP to HTTP can be illustrated by the figure below where the conversion from COAP to HTTP is considered. This is the most simple case of the bridging protocol.

Each request to an HTTP server is forwarded to a COAP server which will send any response to the HTTP client.

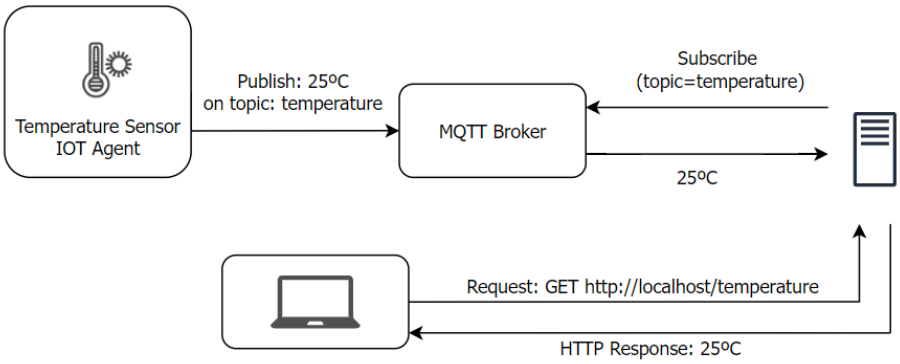


Translation from MQTT (case 5. and 6.)

The conversion from MQTT to COAP and from MQTT to HTTP is handled in the following manner:

As shown in the figure below, an execution flow subscribes and listens to receive a reply message for a particular topic. The reply is saved in a variable and when a client connects to the COAP or HTTP server where the conversion takes place (with the path equal to the topic), the stored response is returned.

For example, an ESP 32 board (in my case the MQTT simulator script) publishes the humidity and temperature values as topic 'temperature' and 'humidity'. If the conversion from MQTT to HTTP is used and you listen on topic 'temperature', the following URL will be created to which you shall connect: http://localhost:3001/temperature.



3 Implementation details

The tool was created using the programming language Javascript with a node.js environment. To interact with it, the user can enable its functionalities through a CLI (Command Line Interface).

In regards to the general organization of the code, four modules have been created corresponding to the four features described above.

1) visualization.js

The *visualization.js* file takes care of recovering the data produced by the protocols and showing it on the screen. It contains a class *getProtocol* which is responsible for connecting to every server and get their information.

The display of the values from the COAP and HTTP protocols, is made using the ***event*** functionality made available by node.js. As soon as the response is received, the result is printed on the screen. After 3 seconds a new event is "emitted" which causes a new request to be forwarded to the server.

To implement the functionality that retrieves data from the MQTT protocol, a ***read_only stream*** has been returned via a *callback*, which contains the values published by the broker as the sensor data is produced. In the same manner as before, as soon as the read_only stream as something to read, the results are displayed on the screen.

2) aggregation.js

The *aggregation.js* file, similarly to the visualization file takes care of recovering the data produced by the protocols. However this time, the data is temporally stored in the array *storage*, which is later used to compute the statistical data. The statistics Maxim, Minimum, Average, and Standard deviation are computed in real-time with basic math operations and presented on the screen after n values have been received.

3) storage.js

The save.js file is responsible for saving data to an Influx 2.0 database. The information required to save the data within the database (specified in the previous section), is defined in a JSON file that needs to be passed as a parameter. The application starts to recover the information from the JSON file and proceeds to create a writing API, using the function ***getWriteApi*** by the "javascript client for InfluxDB 2" library. Consequently, at each value

received, the API is used to save the data into the database using the function ***writePoint***.

4) bridge.js

The conversion part between the protocols is managed by the *bridge.js* file and holds step by step the steps described in the previous section.

Regarding the translations from the MQTT protocol, since the MQTT can publish on several topics, namely ”temperature” and ”humidity”, an object is created for each MQTT topic. These objects are then inserted into an array *”topicsMQTT”*. Each object has the form: topic_name, topin_value. Whenever a COAP or HTTP request is received, a conditional statement checks that the path corresponds to one of the topics present in the object array. If not, an error message with status 404 is displayed.

5) Extra: «protocol»_simulator.js scripts

Three scripts were created in order to simulate the data coming in different protocols.

For the COAP protocol, a server is created that listens on port 5683 of the local machine. When a request is received a response is sent with a JSON object containing the Sensor Type and the Value read (in this case, randomly generated). The requests can be made with paths equal to ”/temperature” or ”/humidity”

In regards to the HTTP protocol, the *Express* framework was used. The server listens on port 3001 and when a request is received a response is sent with the JSON object specified before. Similar to before, the paths can be ”/temperature” or ”/humidity”.

Finally, the MQTT simulator connects to port 1883, and every two seconds publishes on both topics ”temperature” and ”humidity” randomly generated sensor data.

4 Results

As stated before, to test the correct functioning of the tool for the visualization, saving and transformation phases, sensors that produce random data were emulated via script. To further test the tool, the data was displayed through the dashboard made available by influx and through the Grafana. The images and code bellow show the outputs of each feature implemented in this project.

```
1
2 > node ./mqtt_simulator.js
3
4 Published on topic temp.:{sensorType:'temperature',value: '21.11'}
5 Published on topic humidity:{sensorType:'humidity',value:'64.70'}
6 Published on topic temp.:{sensorType:'temperature',value: '23.00'}
7 Published on topic humidity:{sensorType:'humidity',value:'61.40'}
8 ...
```

Listing 2: MQTT Sensor Simulator

```
1
2 > node ./bridgeIoT.js -p "mqtt" -h "localhost" -t "temperature" -c
      visualize
3
4 Starting MQTT visualization ...
5 Connecting to protocol MQTT
6 Topic subscribed!
7 Topic - temperature, Value : 21.11
8 Topic - temperature, Value : 23.00
9 ...
```

Listing 3: Visualize Data From MQTT Protocol

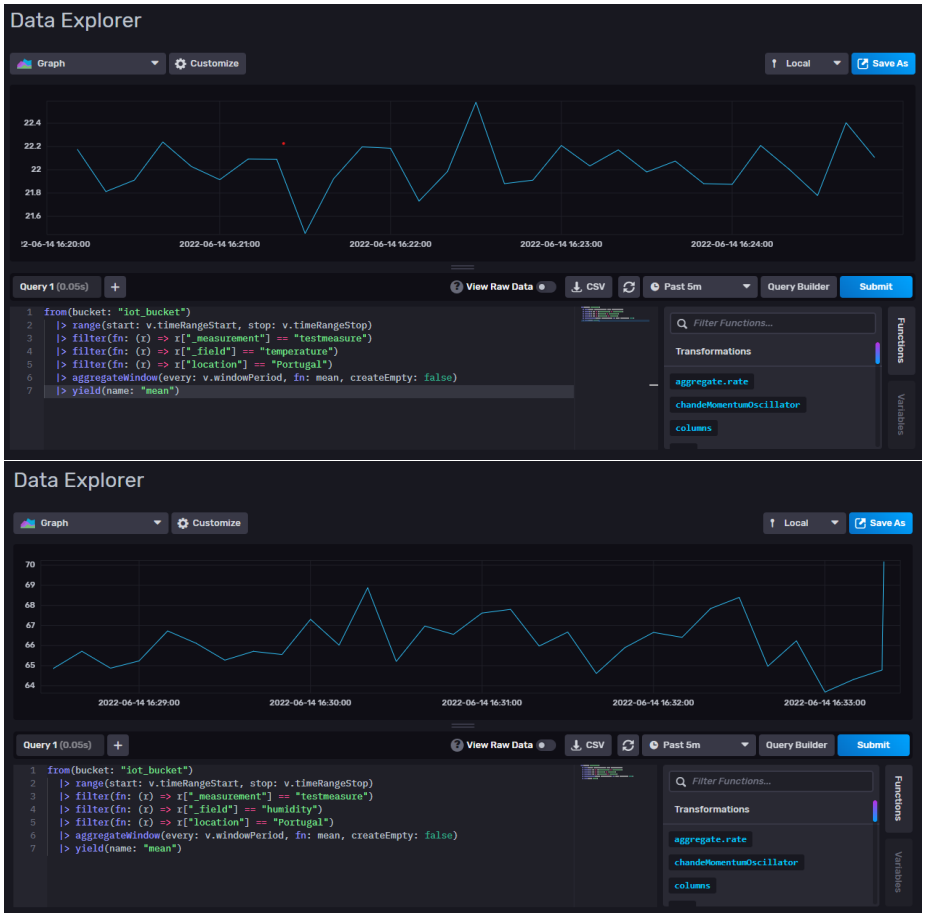
```
1
2 > node ./bridgeIoT.js -p "mqtt" -h "localhost" -t "temperature" -c
      aggregate -n 3
3
4 Starting aggregation ...
5 Connecting to protocol MQTT
6 Topic subscribed!
7 [ 21.79, 22.94, 23.08 ]
8 Maximum: 23.08
9 Minimum: 21.79
10 Average: 22.603333333333335
11 Standard Deviation: 0.5779465565449996
12 ...
```

Listing 4: Statistics From MQTT Data every 3 observations

```
> node ./bridgeIoT.js -p "mqtt" -h "mqtt://localhost" -t
    temperature -i influx_conf.json -c save

Starting storage ...
Saving MQTT protocol data...
Topic subscribed!
Point added: testmeasure,topic=temperature temperature=22.46
    1655215872540549000
Point added: testmeasure,topic=temperature temperature=22.31
    1655215874553812500
...
```

Listing 5: Save Data From MQTT Protocol on Influx Database




```
> node ./bridgeIoT.js -p mqtt -h mqtt://localhost -t temperature -c
    translate -d coap -f "protocol_conf_files/coap_conf.json"

> node ./bridgeIoT.js -p "coap" -h "coap://localhost/temperature" -
   c visualize

Starting COAP visualization ...
21.11
23.00
...
```

Listing 6: Bridging from MQTT protocol to COAP protocol

5 Conclusion and Future Work

In this paper, I introduced the main functionalities and implementation of the IoT Bridge project, made in the scope of the Internet of Things course at the University of Bologna. With this project, I was able to implement the IoT data pipeline presented during the IoT course. From data acquisition (via the HTTP/MQTT/COAP protocols), data management (via a time-series database), data visualization (via a GRAFANA dashboard), and also data processing and analytics.

Unfortunately, the data generation and processing on the edge device (for example with an ESP32 connected to a DHT11 sensor), was not achieved by lack of time. However, it is a feature that I will test in the future.

Overall, I believe it was an ambitious project to do alone, but I am proud of the final tool developed. I look forward to further improving it with new features.

6 References

node-coap: client and server library for CoAP, `https://www.npmjs.com/package/coap`

MQTT.js: client library for the MQTT protocol, written in JavaScript for node.js and the browser, `https://github.com/mqttjs/MQTT.js`

@influxdata/influxdb-client: The reference javascript client for InfluxDB 2.x. Both node and browser environments, `https://www.npmjs.com/package/@influxdata/influxdb-client`

yargs: node.js library fer hearties tryin' ter parse optstrings, `https://github.com/yargs/yargs`

Express: Fast, unopinionated, minimalist web framework for Node.js, `https://expressjs.com/`

Axios: Promise based HTTP client for the browser and node.js, `https://axios-http.com/`

InfluxDB: Get started with InfluxDB documentation, `https://docs.influxdata.com//`

Ponte: Ponte Message Broker Bridge Configuration Using MQTT and CoAP Protocol for Interoperability of IoT, `https://link.springer.com/chapter/10.1007/978-981-15-6648-6_15`