***Abstract*—**

I. Introduction

Middlewares in IoT aim to bridge the gap between the data producers and the data consumers. With the rise of the Internet of Things, comes the need for different applications, and different services with different requirements. Each of these applications will ideally want its own middleware and sensor network so that they can be suited perfectly to their needs. In practical terms this is not possible, as it would be a waste of resources. Therefore, applications will need to work with existing sensor networks and will either develop its own middleware, or choose an existing one that better suits their needs. The question then becomes: how to choose the best one for the task at hand? There are a great number [1] of available middlewares to choose from, which makes the selection process very time-consuming. A comparison must be made between them to evaluate which is better suited for which task. But then comes the problem of how to make the evaluation, as performance measuring is not trivial, and common ground must exist for the comparison to be valid. Furthermore, since we have a great number of middlewares, ensuring such common ground will not be possible across different experiments, and different researchers. From these difficulties arises the need for such common ground, a platform that enables multiple comparisons across different middlewares in an efficient manner. To solve these issues, we propose a modular architecture from which will stem a unified platform in which the benchmarks can be run. The main benefits of such a platform are twofold: to provide a common ground in which the middlewares can be benchmarked to ensure equal an playing field, and to ease the addition of other subsequent middlewares.

II. Related work

In [2] the goal was to obtain a set of qualitative and quantitative metrics that are suited for IoT middleware benchmarking and develop a test methodology around those metrics. Following this, they were able to use said methodology by making a comparison of two IoT middleware platforms, FIWARE [3] and ETSI M2M [4]. They used a smart cities scenario, fairly common in IoT, which typically use the publish-subscribe communication model. The platforms themselves were treated as black-boxes, disregarding information about internal implementation. This eases not only the process, but also the creation of a common benchmarking platform. This can cause the middleware to be used sub-optimally, but is a necessary step in an effort to generalize the process, and should also be the most common situation in real world usage. A qualitative analysis was conducted with the goal of identifying certain middleware functionalities which are relevant for IoT applications and ease their development, such as documentation availability and which communication models are supported. This type of analysis and metrics requires a case-by-case look on each of the platforms being compared, and while they cannot be implemented as part of a benchmarking platform, as they are not measurable automatically, they can still take part in the global architecture. A quantitative analysis was performed using specific metrics relevant for communications scenarios, particularly mass publication of resources using a publish-subscribe model, such as publish time, which is defined as the elapsed time since sending the HTTP request and receiving the HTTP response. Of course, the measurements are specific to this type of communication model and protocol, which we are trying to avoid in this paper, by providing general metrics that are protocol independent. For protocols that share a communication model, such as request-response, this should be relatively straightforward, but it could be more complicated in other cases, for instance with HTTP and MQTT.

In [5], the authors attempted to recreate and improve the previous experiment using a controlled environment, but using OM2M, which is an implementation of the oneM2M [6] standards. A java application was created for each of the four setups they aimed to measure: Fiware, OM2M HTTP, OM2M CoAP, OM2M MQTT. They raise a few concerns which should be had when attempting the benchmarking process, such as where and how the data is published, which will mainly concern application development. This is a point which we want to address in this paper by developing a module for a certain middleware and re-utilizing it for subsequent measurements. Another question raised is the enabling of multi-connection capable clients, which might create an uneven playing field if different clients are used. With the creation of a unified platform, this will have to be taken into account for the client development. Finally the Java configurations, which address the issue of different optimizations with different default Java Virtual Machine packages, which might vary across operating system and also depending on if they are client or server VMs.

III. Solution

A. Modular Architecture

In order to achieve the aforementioned requirements, we aim to create a modular architecture by factoring the common elements that go in the creation of any benchmarking application. The general plan for our architecture can be seen in 1.

A user must be able to simply swap instances of a block as required. To achieve this, we defined a set of inputs and outputs for each one to maintain modularity.

The load block will enable different types of IoT scenarios to be programmed and dynamically changed, so that we can attempt to mimic real world scenarios such as Smart Cities. Again, this should be independent from each of the other blocks so that the same workloads can be used throughout all

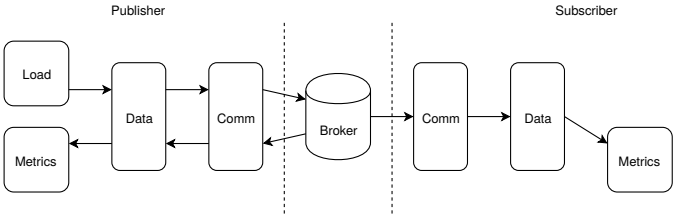

Fig. 1. Main architecture building blocks

middlewares and protocols, providing a basis for comparison and ensuring high flexibility.

The data block is where the middleware specific functions reside, and each of these is responsible for implementing its data structure and bridging the gap to the protocols. Similarly to the data block, it is designed so that each is independent so that all can use the same communication methods implemented. With a new middleware entry, one can observe how the existing functions are structured, thereby speeding up the process of implementing its methods. This entry will be added as a new instance of the data block so as to not interfere with the previous middlewares.

Next, we have the communication block where the protocols are lodged, such as HTTP or CoAP, and each has its methods implemented, e.g., POST or GET, so that they are totally platform independent and can be reused. If a new protocol is required to be added to the platform, its methods can be implemented without interfering with the remaining structure.

After the cycle is complete, a set of defined values, such as times and publish request sizes, will be saved and fed into the metrics block, which will extract information from them, such as average publish time or generated traffic. As we increase the set, more metrics can be generated, without affecting those that are already implemented.

B. OM2M Implementation

In an initial phase, we attempted to create an application to benchmark the OM2M middleware with a basis on the work conducted in [5] and [2], while keeping it as generic as possible to enable future middleware additions and follow our architecture guidelines. This resulted in the structure visible in 2.

Since the load class will be performing the actual requests, it will also call upon the metrics class to perform get the results from the measurements. The load will consist of a certain number of publishes, with a certain message, at a given throughput. All can be easily defined by the user. It's implemented by way of a loop, with each cycle corresponding to a publish request, with sleeps in between to limit the throughput.

The metrics currently implemented on are publish and subscribe time, both visible in 3, generated traffic, failed and successful publishes, size of a publish structure and size of the url. Each publish time is simply the difference between sending the request and receiving the response from the broker, easily implemented in the main class by measuring the elapsed

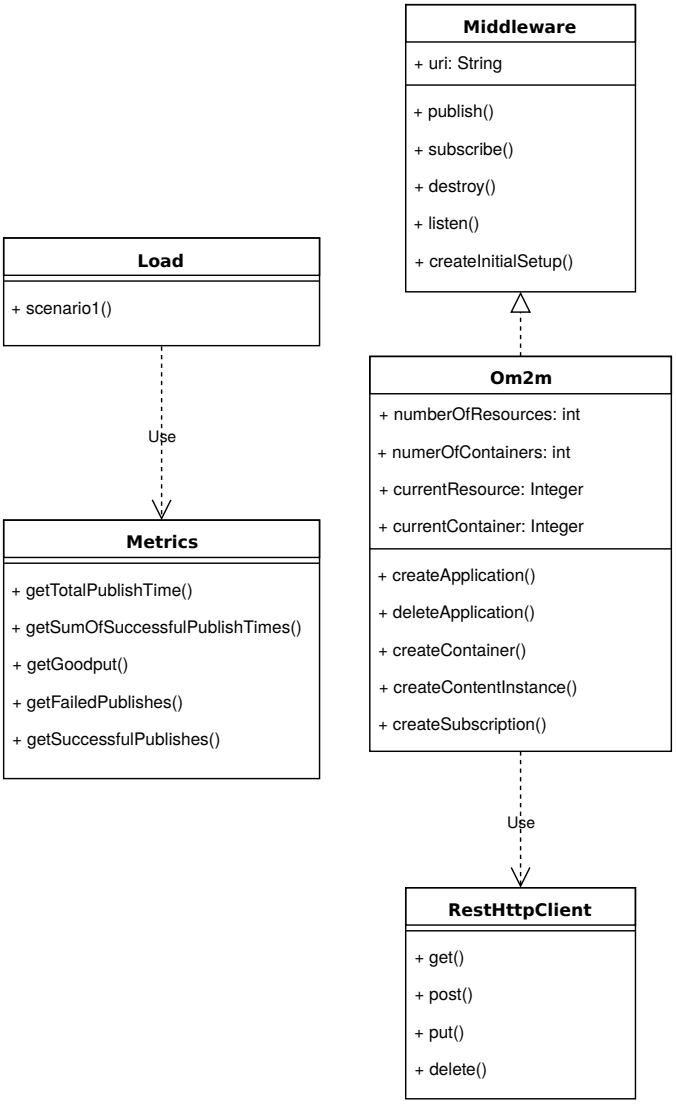

Fig. 2. Class diagram for the initial platform stage

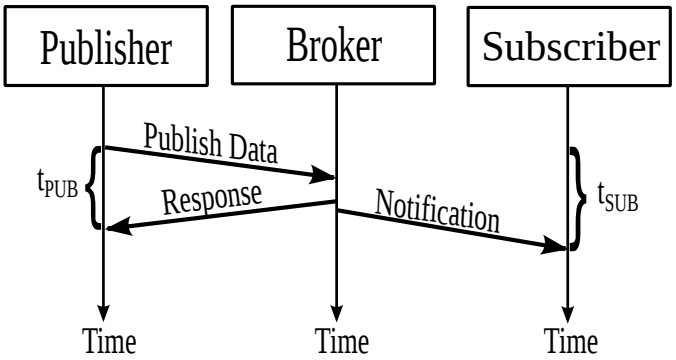

Fig. 3. Publish and subscribe times [2]

time of a single cycle in the load loop. Generated traffic is measured by dividing the publish structure of each message by each publish time. The publish structure corresponds to the full message that is assembled by each middleware. Let's take the Om2m class as an example. Here, a publish request corresponds to the creation of a content instance of the container where we wish to publish. Therefore, the **createContentInstance()** method will take the message as input and create the appropriate data structure, such as in 1 for creating an application, to be sent as payload for a certain protocol, e.g., HTTP.

```
{
   "m2m:ae": {
     "api": "app-sensor",
     "rr": "false",
     "lbl": ["Type/sensor", "Category/temperature", "
          Location/home"],
     "rn": "MY_SENSOR"
   }
}
```

Listing 1. JSON payload for application creation

This will be returned to the calling publish method, in order for it to know the payload size for that particular middleware publish request. Since this class extends the **middleware** superclass, this method is always present and always has the same return values, providing generic metrics. Following this, we have the failed and successful publishes. In order to determine the whether a certain request was successful or not, some level of analysis must be conducted to the response provided by the broker. Naturally, this is protocol dependent, so in order to create a layer of abstraction, the basic communication methods of the used protocol, such as **POST** or **PUT** must return the broker response, in order for the Om2m class to be able to interpret if a publish was successful or not. This way, it will then return to the main class a generic indicator, independent of protocol, indicating its success or failure. Then, we have subscribe time which is implemented differently, as it potentially relies on times registered at different machines. In order for the subscriber to register the times, a listener must be created for the protocol it is expecting to receive. This listener will be in charge of registering the times at which the notification arrive, meaning this metric is implemented at the protocol level. Lastly, we have the sizes for the publish structures and for the URL. These aim to enable a user see how much data needs to be added onto the message that they want to send, as it can be quite larger than the message itself, meaning a bigger impact.

Moving on we have the **Middleware** superclass. Here the goal is to provide the methods that all middlewares are expected to implement and any attributes that are common as well. We therefore chose to have an **uri** to identify where it will be located on the network. The **publish()** and **subscribe()** methods are evident as we are dealing with publish/subscribe scenarios. The **destroy()** method provides a way to clear any created resources so that the experiment may be conducted again on a clean broker. Next, we have **listen()**, which is for the subscriber to call so that it may receive and parse notifications as needed, and register their arrival times. Finally, **createInitialSetup()** is for registering resources, such as applications in the case of OM2M, the number of which is defined by the user.

Then, we come to the protocol classes. Currently, only HTTP is implemented in the **RestHttpClient()**, but others may be added in the future, such as CoAP or MQTT. The four methods are ubiquitous across several applications, and typically most middelwares which rely on HTTP will make use of these.

IV. Evaluation

A. Adding new middlewares

The platform development was always conducted taking into account the addition of new middlewares, therefore it was made to be as generic as possible and to facilitate any new addition. However, during such an addition, there can always be details which were not accounted for and can force the platform to change to be able to accommodate this new addition. In this section, we will see the changes to the architecture, and the differences and similarities between the both implementations, how much was reusable in terms of code and overall structure, and attempt to quantify the changes through the number of lines of code.

1) FIWARE: We decided to add the FIWARE middleware as a result of previous work and greater familiarity with it. The class diagram is similar to that in figure 2, with the difference being the new Fiware class, visible in 4.

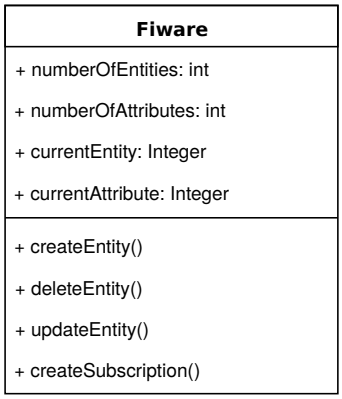

Fig. 4. Fiware class diagram

Since it is also an extension of the **Middleware** superclass, it shares the five methods and attributes with **Om2m**, with the rest being specific the this class, with a similar structure. The superclass methods will call the specific functions to bridge the gap to the communication protocol, in this case HTTP. Starting with **publish()**, the structure is basically identical, with only 4 lines of code being different between both middlewares out of 22, which merely correspond to differing function calls and variable names. A Om2m publish calls upon the **createContentInstance()** function to create a new instance in a previously created container in an application, receiving the intended message and constructing a JSON payload in

accordance to its standards, and sending it via an HTTP POST. Similarly, Fiware uses the **updateEntity()** method to update the current status of an entity, also constructing an appropriate JSON, but sending it with an HTTP PATCH. A key detail to note here, is that using this method, Fiware does not retain memory of previous status, as it is overwritten. Both of these methods return the JSON payloads and the HTTP response, and register the times at which the publishes were sent. The **subscribe()** method in both implementations is similar to this, but Fiware also uses HTTP POST as opposed to a PATCH request.

The **listen()** function is the same in both, as it only creates an HTTP server for the subscriber to listen and parse the notifications.

Next, we have **destroy()** which aims to delete all the created resources so that it is easy to start from scratch. For this, Om2m calls **deleteApplication()** which takes the name of the resource to be deleted, constructs the JSON payload and POSTs it to the broker. For Fiware its **deleteEntity()**, and it works much the same, the only change being the JSON created.

Finally we have **createInitialSetup()**, where resources are registered for the first time. For Om2m, a nested **for** loop is used to create the desired number of applications with **createApplication()**, and for each of these the number of containers with **createContainer()**. For Fiware, the number of attributes must be defined upon entity creation, so these are handled on a lower level at the **createEntity()** method, so only a simple **for** loop is used to create the necessary entities.

Almost all functions used share the same structure: create the JSON to encapsulate the message, create the appropriate HTTP headers and make the HTTP request to the broker. This greatly eases the process of adding middlewares.

2) Ponte: In 5 we can see the class diagram for Ponte. Since it is another implementation of the middleware superclass, it shares the same methods, so they are not referenced in the image.

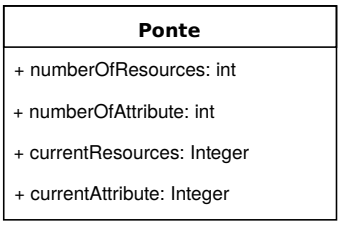

Fig. 5. Class diagram for Ponte

As before, there is an URI identifier and two variables that indicate how many resources and attributes per resource this middleware instance will take. With Ponte, only the **publish()** and createInitialSetup methods from the **Middleware** superclass was implemented. The reason for this is that there was no need to create an additional abstraction layer between the the middleware specific methods and the publish method as before with other implementations, since a simple HTTP PUT is required with the target resource and attribute in the URL, and the value as payload, without any need for JSON or XML assembly.

subscribe() and **listen()** were not implemented as there is no way for the broker to notify a subscriber through HTTP, only through GETs originating from the subscriber, which would require periodic queries in order to keep the status updated and differs from a publish/subscribe scenario.

There is no obvious way of deleting a resource so **destroy** is also not implemented, with the most obvious way to do so being simply to restart the broker.

B. Benchmarking results

The goal here is not to use the results to evaluate the performance of the platform, or to make a comparison between other middlewares. Rather, we want to use the results to validate the platform itself, and show what types of information we can extract from these tests, while still keeping the platform generic.

The test consisted of 20000 publishes of 22 byte messages, at a maximum rate of 100 publishes per second. The subscriber and the broker were located on the same machine, while the publisher was seperate. They were connected through 100 MB/s Ethernet connections. In 6 we can see a comparison between the publish times of all three implemented middlewares. In 7 only FIWARE and OM2M are present, as it is not possible to measure the subscribe times with Ponte using HTTP.

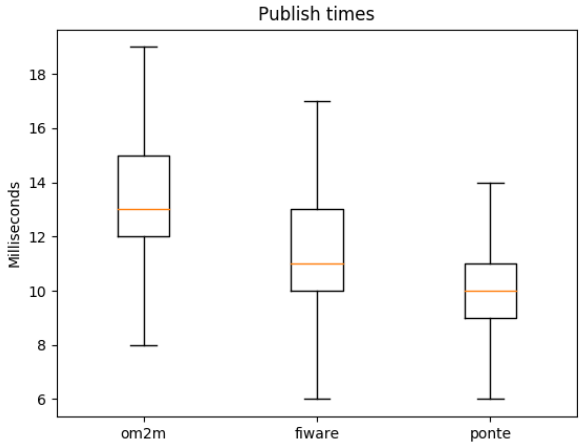

Fig. 6. OM2M publish times

A few extra metrics are available for each of them, being those the generated traffic, total publish time, number of failed and successful publishes, size of a publish structure, and size of the URL.

1) FIWARE:

- Generated traffic: 5.42 KB/s
- Failed publishes: 0
- Successful Publishes: 20000

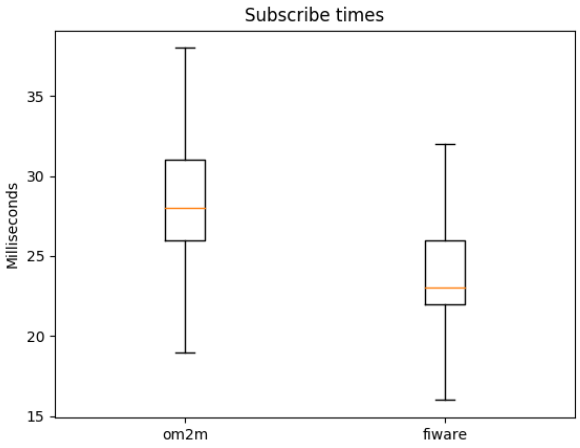

Fig. 7. OM2M subscribe times

- Total publish time: 259 seconds
- Size of a publish structure: 70 bytes
- Size of the URL: 51 bytes

2) OM2M:

- Generated traffic: 6.75 KB/s
- Failed publishes: 0
- Successful Publishes: 20000
- Total publish time: 283 seconds
- Size of a publish structure: 95 bytes
- Size of the URL: 58 bytes

3) Ponte:

- Generated traffic: 2.12 KB/s
- Failed publishes: 0
- Successful Publishes: 20000
- Total publish time: 208 seconds
- Size of a publish structure: 22 bytes
- Size of the URL: 56 bytes

In order to better show the similarities between implementations, let's look at the publish methods themselves. In figure 8 we can see a comparison between both implementations for each middleware. The highlighted lines correspond to the differences.

The structure is basically identical, with the exception that Fiware calls an additional auxiliary method specific to it, **updateEntity**(), that assembles the JSON for the HTTP request that will execute the publish. Since Ponte does not need to assemble a payload in such a way, **put**() is called directly. This enabled an easy implementation for Ponte by simply analyzing what had been previously done for FIWARE, speeding up the process of adding a new middleware.

```
public long[] publish(String message, String publishSequenceNumber) {
        long[] returnArray = new long[3];
        Integer payloadSize = 0;
        long elapsedTime = 0;
        String payload = message + "-" + publishSequenceNumber;
        long start = System.currentTimeMillis();
        String[] updateEntityReturn = null;
        updateEntityReturn = updateEntity("entity" + currentEntity.toString()
        payloadSize = updateEntityReturn[0].length();
        String publishResponseStatus = updateEntityReturn[1];
        if (publishResponseStatus.contains("HTTP/1.1 2")) {
                System.out.println("Successful publish ");
                elapsedTime = System.currentTimeMillis() - start;
        }
        else {
                System.out.println("Unsuccessful publish");
                elapsedTime = -1;
        }
        setNextPublishDestinationRoundRobin();
        returnArray[0] = elapsedTime;
        returnArray[1] = payloadSize;
        returnArray[2] = start;
        return returnArray;
}
 Fiware.java     12% L44   Git:threads  (Java/l Server Fly FlyC- Wrap Abbrev)
public long[] publish(String message, String publishSequenceNumber) {
        RestHttpClient client = new RestHttpClient();
        long[] returnArray = new long[3];
        Integer payloadSize = 0;
        long elapsedTime = 0;
        String payload = message + "-" + publishSequenceNumber;
        long start = System.currentTimeMillis();
        payloadSize = payload.length();
        CloseableHttpResponse response = null;
        response = client.put(this.uri + "/resources/" + "resource" + currentR
        String publishResponseStatus = response.getStatusLine().toString();
        if (publishResponseStatus.contains("HTTP/1.1 2")) {
                System.out.println("Successful publish ");
                elapsedTime = System.currentTimeMillis() - start;
        }
        else {
                System.out.println("Unsuccessful publish");
                elapsedTime = -1;
        }
        setNextPublishDestinationRoundRobin();
        returnArray[0] = elapsedTime;
        returnArray[1] = payloadSize;
        returnArray[2] = start;
        return returnArray;
 Ponte.java      33% L39   Git:threads  (Java/l Fly FlyC- Wrap Abbrev)
```

Fig. 8. Difference between Fiware (top) and Ponte (bottom) publis method

V. Analysis for improvement

VI. Conclusion

References

[1] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(1):70–95, February 2016.

[2] J. Cardoso, C. Pereira, A. Aguiar, and R. Morla. Benchmarking IoT middleware platforms. In *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–7, June 2017.

[3] Fiware. https://www.fiware.org/, 2018. (Accessed on 01/26/2018).

[4] Etsi - welcome to the world of standards! http://www.etsi.org/, 2018. (Accessed on 01/26/2018).

[5] Carlos Pereira, João Cardoso, Ana Aguiar, and Ricardo Morla. Benchmarking Pub/Sub IoT middleware platforms for smart services. *Journal of Reliable Intelligent Environments*, pages 1–13, February 2018.

[6] onem2m - home. http://www.onem2m.org/, 2018. (Accessed on 02/02/2018).