***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

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.

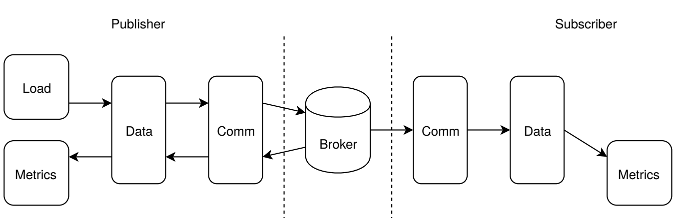

Fig. 1. Main architecture building blocks

A user must be able to simply swap instances of the 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 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.

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 [2] and [3], 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.

The main class is responsible for the load and most of 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, goodput, failed and successful publishes. 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 time of a single cycle in the load loop. Goodput is measured by dividing the useful bytes of each message by each publish time. The useful bytes correspond 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

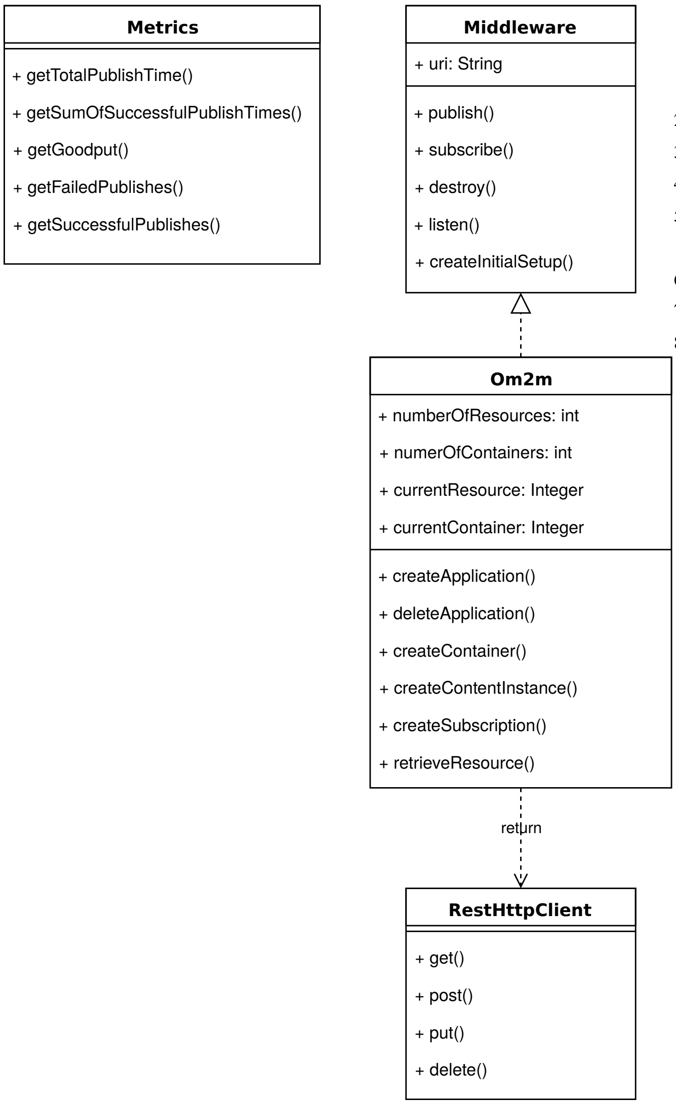

Fig. 2. Class diagram for the initial platform stage

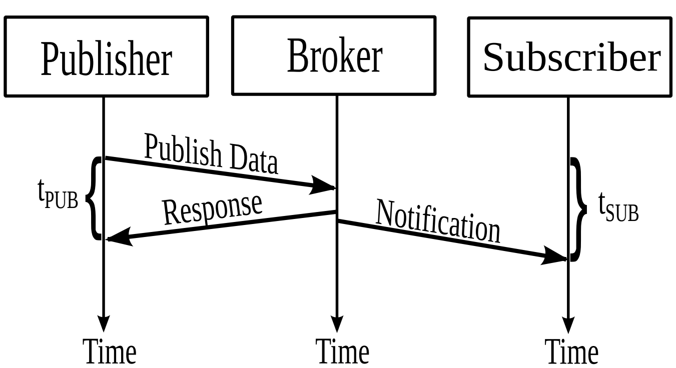

Fig. 3. Publish and subscribe times [3]

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. Lastly, 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.

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. Benchmarking results

B. Adding a new middleware

V. 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] 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.

[3] 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.