***Abstract*—**

I. Introduction

II. Related work

III. Solution

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. This is achieved by factoring the common elements that go in the creation of any benchmarking application. The main building blocks of our architecture are visible in figure 1.

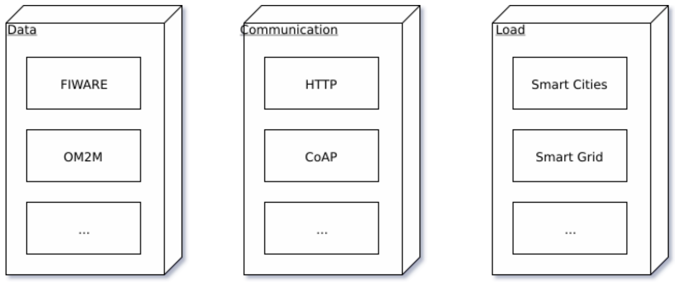

Fig. 1. Main architecture building blocks

The goal is to only implement what is necessary for a new entry in the benchmarking platform instead of using a similar implementation for each one. We propose a communication block where the communication protocols are lodged, such as HTTP or CoAP, and each has the methods implemented, such as 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.

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 in the communication block. With a new entry, one can observe how the existing ones are structured, thereby speeding up the process and keeping it isolated without interfering with the existing setup.

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.

Each building block will have several sub-blocks, which will correspond to a given class. In an initial phase, we attempted to create an application to benchmark the OM2M middleware with a basis on the work conducted in [1] 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.

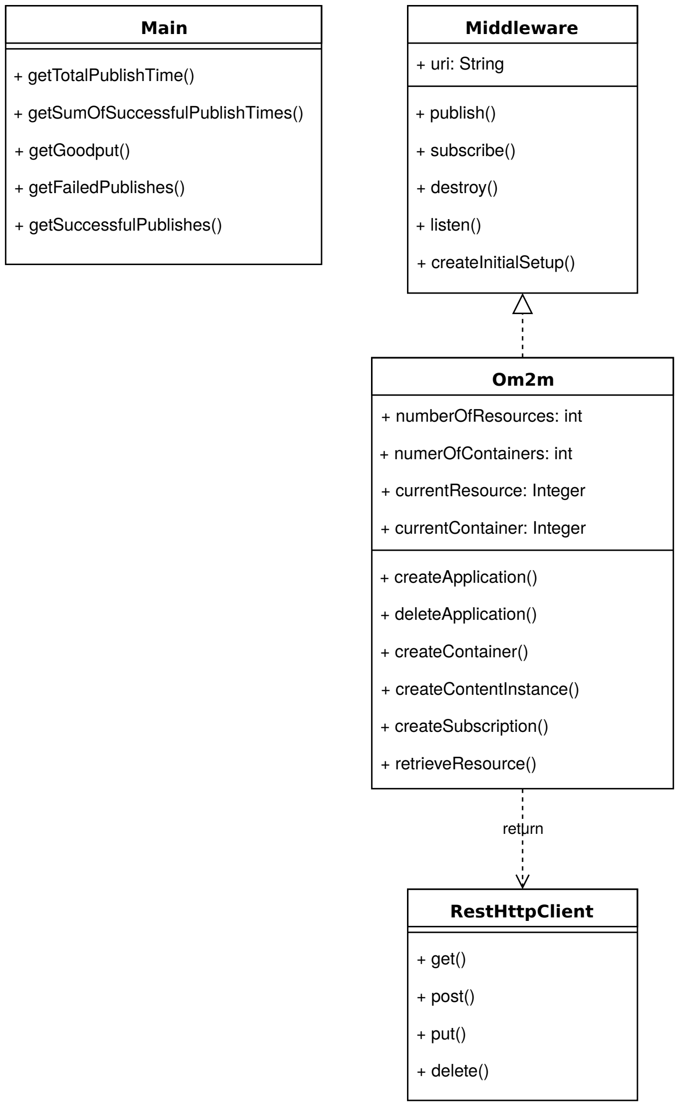

Fig. 2. Class diagram for the initial platform stage

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

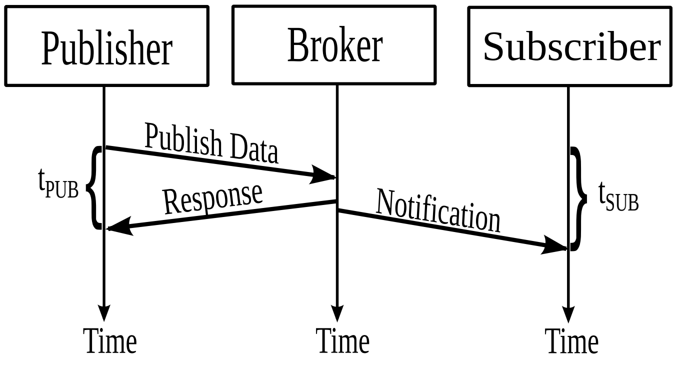

Fig. 3. Publish and subscribe times [2]

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 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

V. Conclusion

References

[1] 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.

[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.