A Modular Tool for Benchmarking IoT Publish-Subscribe Middleware

L. Zilhão*†, Ricardo Morla*‡, Ana Aguiar*†
*Faculdade de Engenharia, Universidade do Porto, Portugal
†Instituto de Telecomunicações, Portugal
‡INESC TEC, Portugal
Email: ee12150@fe.up.pt, rmorla@fe.up.pt, anaa@fe.up.pt

***Abstract*—With the rise in popularity of the Internet of Things in all kinds of different application scenarios, various middleware solutions have appeared with different use-cases and optimizations in mind. The design space for any specific deployment is thus increasingly large, but little objective support exists to help choosing the best middleware for each use-case. From this stems the need to evaluate how different IoT middleware solutions perform in different use-cases. Measuring the performance of IoT middleware in a way that 1) provides common ground between experiments, and 2) makes it easier to integrate new IoT middleware in the benchmark is not straightforward. In this paper we propose a generic architecture for evaluating the performance of publish/subscribe middleware, develop a tool that implements this architecture, and show the benefits in time and effort that can be reaped from our approach. We further validate our approach by using the architecture and tool to benchmark different middleware solutions, taking lessons from the changes we had to do to the architecture in order to support new middleware, and attempting to quantify the effort through lines of code and to qualitatively assess code structure similarity.**

I. Introduction

IoT middleware aims to bridge the gap between data producers and data consumers. Internet of Things encompasses a wide variety of different applications and services with coreespondignly different requirements. Each of these applications would ideally require a middleware and sensor network tailored perfectly to its needs. However, this would be a waste of resources. Therefore, application developers will choose an existing middleware that suits their needs. The question then becomes: how to choose the most adequate middleware for the task at hand? There is a large number [1] of available middleware solutions to choose from, which makes the selection process resource- and time-consuming. A comparison must be made between different middleware solutions to evaluate which is better suited for which task. But then the problem arises of how to make the evaluation, given that performance measuring is not trivial and that common ground must exist for the comparison to be valid. Furthermore, since we have a large number of middleware solutions to choose from, ensuring such common ground across different experiments is very cumbersome and error prone.

To address these difficulties, we propose a tool that enables multiple comparisons across different middleware solutions in an efficient manner. We extend our previous work and propose a generic architecture from which will stem a modular tool in which the benchmarks can be run. The main benefits of such a tool are twofold: to provide a common ground in which the different middleware solutions can be benchmarked to ensure equal an playing field, and to reduce the cost of trying out subsequent middleware solutions.

In one of our previous works [2], we set off to obtain a set of qualitative and quantitative metrics suited for benchmarking IoT middleware and to apply a test methodology using those metrics to FIWARE [3] and ETSI M2M [4]. We used a smart cities scenario with the publish-subscribe communication model, which is fairly common in IoT. The IoT middleware to benchmark was treated as a black-box, disregarding information about internal implementation. This eases the benchmarking process and the creation of a common benchmarking tool, but can cause the middleware to be used sub-optimally. This is a necessary step in an effort to generalize the process, and reflects to a large extent the typical user's approach. 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 qualitative analysis and metrics requires a case-by-case study of each broker, and although it cannot be automated within a benchmarking platform it should be considered globally when comparing IoT middleware. A quantitative analysis was performed using metrics relevant for communication 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. In that work, the measurements were specific to the communication model and protocol, something we try to avoid in this paper by providing generic metrics that are protocol independent. For protocols that share a communication model this should be relatively straightforward, but could be more complicated for instance when one IoT middleware uses HTTP and another

This article was carried out in the scope of the project SmartCityMules, UID/EEA/50008/2013, funded by the applicable financial framework (FCT/MCTES) (PIDDAC).

uses MQTT.

In [5], we improve upon the previous experiment by using a controlled environment and evaluating Fi-Ware and OM2M [6] with different protocol configurations – Fiware, OM2M with HTTP, OM2M with CoAP, and OM2M with MQTT. That paper identifies where and how data is published and highlights some concerns for benchmarking including different client capabilities and Java Virtual Machine (JVM) configurations. We address these concerns in this paper by developing middleware-specific modules and re-utilizing them for subsequent measurements. One client capablility concern is the possibility that multi-connection capable clients cause unfair middleware comparisons. The generic tool we develop in this paper will have to consider this in the client development. One concern with JVM configurations is that different packages implement varied optimizations, creating additional variability across operating systems and depending on whether they are client or server VMs. To avoid unfair comparisons, we must configure our application so that it requires a specific setup, such as Server mode, ensuring a level playing field.

II. Related Work

There are few studies comparing IoT middleware performance. Those that exist typically focus on the specifics of each middleware solution and thus are not generic – e.g. Kafka vs. RabbitMQ [7] for Pub/Sub and ZStreaming vs. Kafka Streaming [8] for streaming sensor data. We found two benchmarking tools for IoT middleware. However, one is focused on stream data processing [9] and the other [10], despite providing a publish/subscribe spcecific workload generator, uses a simplistic model for abstracting the middleware which still leaves much of the burden to the programmer that wants to benchmark a new middleware. Finally, we also found benchmarking tools for other, cloud based, software. In [11], the authors propose an architecture and tool for benchmarking Docker containers, detailing on the specifics of benchmarking containerized applications and extensively testing their tool. In [12], the authors discuss different issues of benchmarking Infrastructure-as-a-Service solutions for the Cloud, specifying benchmarking components, design features, and open challenges. Although these tools cannot be used directly for publish/subscribe IoT middleware, we feel they are solid references for benchmarking in general and for IoT middleware benchmarking in particular.

III. Solution

A. Modular Architecture

To comply with the aforementioned requirements, we aim to create a modular architecture by factoring the common elements in the benchmarking applications. The general plan for our architecture can be seen in 1.

A user must be able to swap instances of a block as required. To achieve this, we defined interfaces (inputs and outputs) for all blocks.

The *load block* will enable different types of IoT scenarios to be programmed and dynamically changed, so that we can attempt to mimic varied real world scenarios. Again, this should be totally independent from any other block so that the same workloads can be used throughout all middlewares and protocols, providing a basis for comparison and ensuring high flexibility.

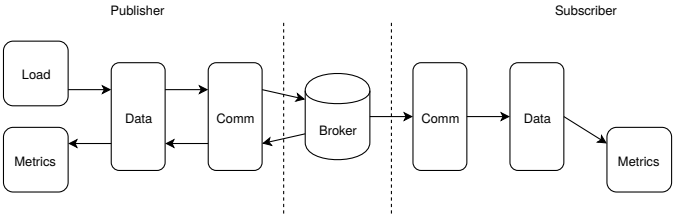

Fig. 1. Main architecture building blocks

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. When a new middleware is to be tested, one can leverage the existing functions, thereby speeding up the implementation process. A new middleware 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, such as HTTP, CoAP, MQTT, are lodged. Each has its methods implemented, e.g. POST or GET, so that they are middleware independent and can be reused. If a new protocol is required, its methods can be implemented without interfering with the remaining structure.

During the test cycle, a set of defined values, such as times and publish request sizes, will be stored and fed into the *metrics block*, which will extract metrics from them, such as average publish time or generated traffic. New metrics can be added without affecting those that are already implemented, also keeping result compatibility.

B. OM2M Implementation

In an initial step, we create an application to benchmark the OM2M middleware by abstracting our previous work [5], [2], while keeping it as generic as possible to enable future middleware additions, following 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 rate. All can be easily defined by the user. It is implemented by way of a loop, with each cycle corresponding to a publish request, with sleeps in between to meet the desired rate.

The metrics currently implemented 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 load class by measuring the elapsed time of a single cycle in the load loop. Generated traffic is measured by dividing the publish structure of each message by each publish

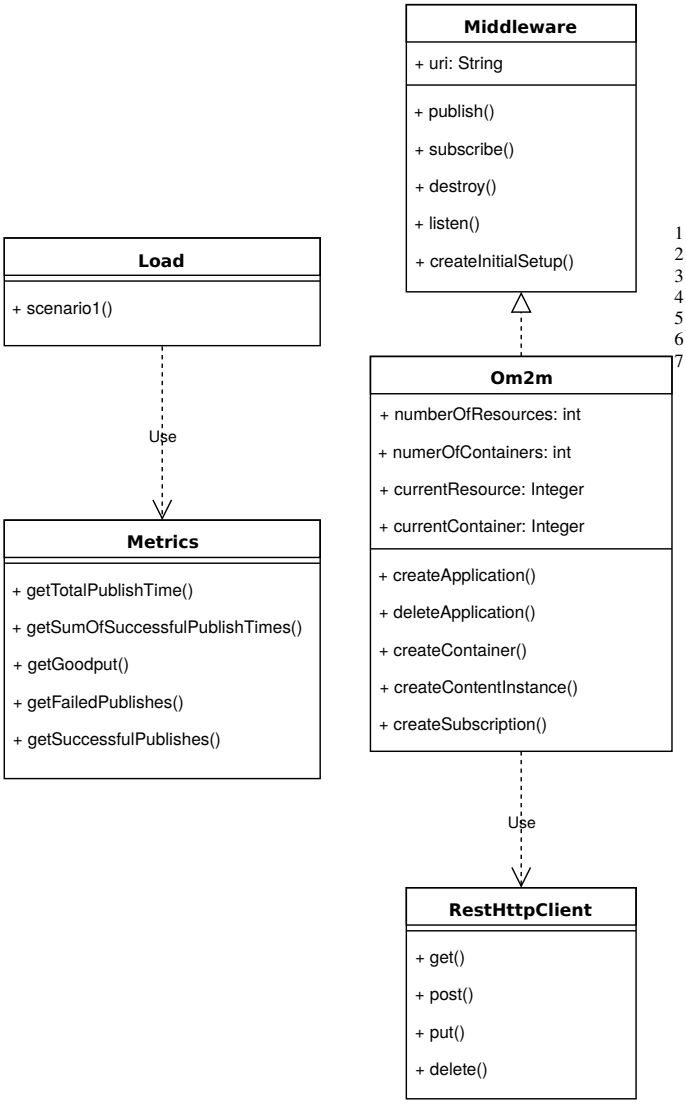

Fig. 2. Class diagram for the initial platform stage

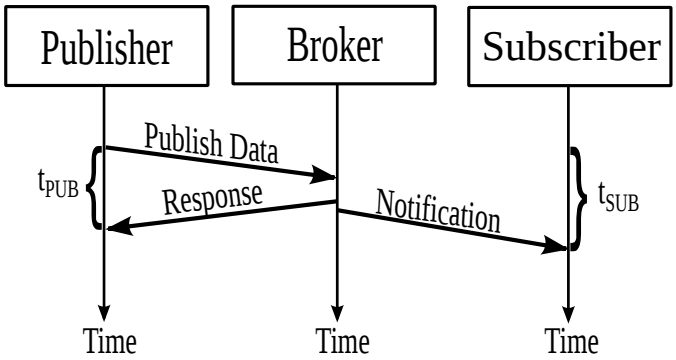

Fig. 3. Publish and subscribe times [2]

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 to create a content instance, to be sent as payload for a certain protocol, e.g., HTTP.

```
{
        "m2m:cin": {
                "con": "my_message-00000000000",
                "cnf": "application/json",
                "rn": "time_1520895952979"
        }
}
```

Listing 1. JSON payload for content instance 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 whether a certain request was successful or not, the response provided by the broker must be analysed. 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 return to the main class a generic success or failure indicator, independent of protocol. Then, we have subscribe time which is implemented differently, as it potentially relies on times registered at different machines. The subscriber must create a listener for the protocol it is expecting to receive in order to register the times. This listener will be in charge of registering the times at which the notifications arrive, meaning this metric is implemented at the protocol level. Lastly, we have the sizes for the publish structures and for the URL. This metric aims to enable a user to see how much data needs to be added to the message that they want to send, as it can be quite larger than the message itself, imposing a bigger overhead.

The goal of the **Middleware** superclass 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 **createInitialSetup()** method is for registering resources, such as applications in the case of OM2M, the number of which is defined by the user. Next, we have **listen()**, which enables the subscriber to receive and parse notifications as needed, and register their arrival times. Finally, the **destroy()** method provides a way to clear any created resources so that the experiment may be conducted again on a clean broker.

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 in the **RestHttpClient** class 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 conducted keeping in mind the extensibility to new middlewares. However, during such an extension, needs which were not accounted for can show up, and the platform must evolve to be able to accommodate them. In this section, we will see the changes to the architecture, and the differences and similarities between the implementation of OneM2M and two additional middlewares: FIWARE and Ponte. We will also highlight 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. Since it is an extension of the **Middleware** superclass, it shares the five methods and attributes with **Om2m**; the rest is specific to 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 identical, with only 4 lines out of 22 of code being different between both middlewares, which merely correspond to differing function calls and variable names. An Om2m publish calls 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 uses HTTP POST instead of a PATCH request.

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

Next, **destroy()** deletes all 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 pretty much in the same way, 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 at 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.

But what if we wanted to use Fiware with memory? What changes would be required for it to work? This would come down to what the user interprets as a publish request. Since entities remain in memory after their creation, if we consider the creation of an entity as a request, then the only necessary change would be to use **createEntity()** in the publish method, instead of **updateEntity()**. This would also render **createInitialSetup()** unnecessary, as entities would not have to be created beforehand.

2) Ponte: Ponte is another implementation of the middleware superclass, so it shares the same methods as FIWARE and OM2M. 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 were 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. 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 GET polling originating from the subscriber. Thus, Ponte does not fully follow a publish/subscribe communication model.

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

B. Impact on Code

In order to better show the similarities between implementations, let's look at the publish methods themselves. In figure 4 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, **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.

We can attempt to quantify the changes needed for a new implementation by looking at the lines of code. The distribution of lines per class can be seen in table I.

This brings a total of 801. At first glance, we can see that for the implementation of single middleware, the effort

```
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. 4. Difference between Fiware (top) and Ponte (bottom) publish method

Class	Lines of code
Main	51
Load	119
Middleware	16
Metrics	136
Ponte	88
Fiware	193
Om2m	198

TABLE I
LINES OF CODE FOR EACH CLASS

required can vary, with Ponte only needing 88 lines of new code. Nevertheless, there is a great deal that can be reused across all of them, greatly easing the process. As we have seen previously, not only are the structures similar, but some of the methods are similar as well, such as with the **publish**() methods between Ponte and Fiware.

V. BENCHMARKING PLATFORM VALIDATION

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 separate. They were connected through 100 MB/s Ethernet connections. In 5 we can see a comparison between the publish times of all three implemented middlewares. In 6 only FIWARE and OM2M are present, as it is not possible to measure the subscribe times with Ponte using HTTP.

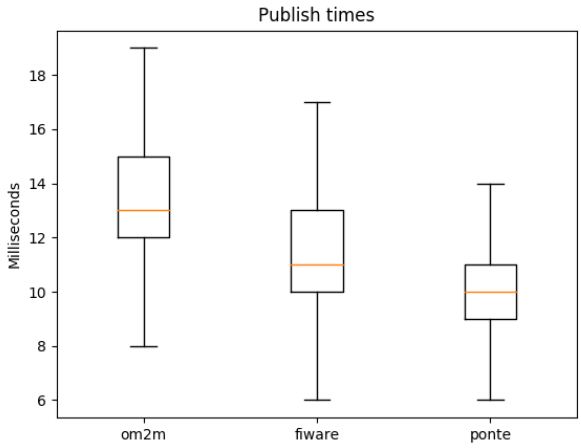

Fig. 5. Publish times for all three middlewares with 22 byte messages

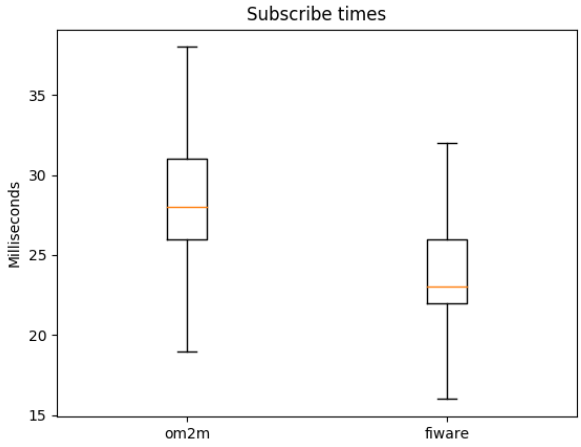

Fig. 6. Subscribe times for Om2m and Fiware with 22 byte messages

A few extra metrics are available for each of them: the generated TCP traffic, total publish time, number of failed and successful publishes, size of a publish structure, and size of the URL, all visible in table II.

Changing parameters is merely altering the value of a few variables, depending on what the user wants. A second test was performed, using the same conditions as before, but with 10012 byte messages.

The application can easily handle even large messages sizes such as in figure 7. The additional metrics can be seen in table III.

Metrics	FIWARE	OM2M	Ponte
Generated Traffic (KB/s)	5.42	6.75	2.12
Failed publishes	0	0	0
Successful publishes	20000	20000	20000
Total publish time (s)	259	283	208
Size of a publish structure	70	95	22
Size of the URL (bytes)	51	58	56

TABLE II
RESULTS WITH 22 BYTE MESSAGES

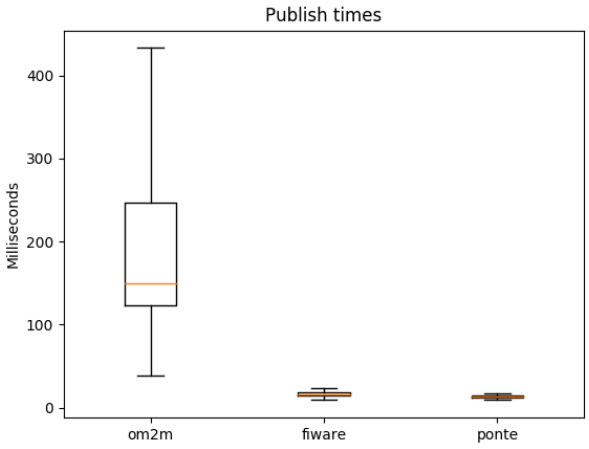

Fig. 7. Publish times for all three middlewares with 10012 byte messages

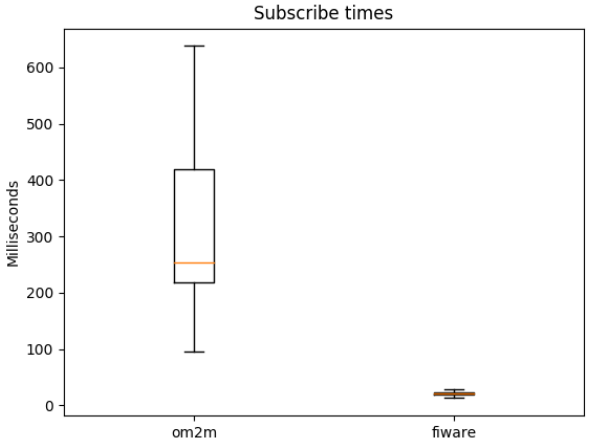

Fig. 8. Subscribe times for Om2m and Fiware with 10012 byte messages

Metrics	FIWARE	OM2M	Ponte
Generated Traffic (KB/s)	533.21	56.27	756.53
Failed publishes	0	0	0
Successful publishes	20000	20000	20000
Total publish time (s)	380	3587	267
Size of a publish structure	10060	10085	10012
Size of the URL (bytes)	51	58	56

TABLE III
RESULTS WITH 10012 BYTE MESSAGES

We can observe that all middlewares use similar amount of Bytes for the resource, but we can see significant differences in the generated traffic, especially for large payloads. A large disparity in the total publish time could also be identified for the OneM2M broker with large payloads. This is the type of issue that our benchmarking platform should enable to identify.

VI. LIMITATIONS AND FUTURE WORK

For the time being the platform is implemented so as to allow extensions to new middleware to be straighforward and low effort, but there are areas where we can modularize the structure further. Taking again the example of figure 4, we can see there are three distinct areas for a publish request: the assembly of the structure that will be sent, sending it using a certain communication protocol, and registering any relevant information from the response to implement metrics. Our goal is to cerate a further abstraction, so that only the structure assembly is required.

Also, the platform is currently restricted to run in single threaded mode, but we intend to add multi-threaded support to simulate several simultaneous publishers. Also, additional metrics may need to be added, some of them protocol specific, to better compare different middlewares with the same communication protocol.

VII. CONCLUSION

In this paper, we proposed a scalable architecture for middleware benchmarking that targets pub/sub communication scenarios. We identified the common aspects between different middlewares, and factorized them, so as to avoid repeating the same steps across different experiments. We then created a platform based on this architecture and on previous experiments conducted with OM2M. After making sure it provided good metrics and results, we proceeded to add a new middleware, FIWARE, and document the changes necessary for its implementation, both on an architecture level and on a platform level. We concluded that the architecture was adequate, and only one new class with 193 lines of code was necessary, with a lot of structural similarities between them. We continued this process with the addition of Ponte, and it proved again to be very similar to the previous two, with 88 new lines of code. On both occasions, the process was significantly quicker than it would have been if we were to create a specific application from scratch just for a single middleware. To prove the validity of our platform, we conducted two tests with different variables and obtained results with ease and, more importantly, assuring the same conditions across experiments, due to the nature of the application itself. This will allow for future users and researchers to have common ground in different tests, enabling a more direct comparison between them.

Future work should focus on better modularization for an easier implementation, multi-threaded support, and lower-level metrics on the TCP level.

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).

[7] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka Versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/-Subscribe Implementations: Industry Paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 227–238, New York, NY, USA, 2017. ACM.

[8] Jorge Y. Fernández-Rodríguez, Juan A. Álvarez García, Jesús Arias Fisteus, Miguel R. Luaces, and Victor Corcoba Magaña. Benchmarking real-time vehicle data streaming models for a smart city. *Information Systems*, 72:62 – 76, 2017.

[9] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):n/a–n/a, November 2017.

[10] Kaiwen Zhang, Tilmann Rabl, Yi Ping Sun, Rushab Kumar, Nayeem Zen, and Hans-Arno Jacobsen. PSBench: A Benchmark for Content- and Topic-based Publish/Subscribe Systems. In *Proceedings of the Posters & Demos Session*, Middleware Posters and Demos '14, pages 17–18, New York, NY, USA, 2014. ACM.

[11] B. Varghese, L. T. Subba, L. Thai, and A. Barker. DocLite: A Docker-Based Lightweight Cloud Benchmarking Tool. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 213–222, May 2016.

[12] Alexandru Iosup, Radu Prodan, and Dick Epema. IaaS Cloud Benchmarking: Approaches, Challenges, and Experience. In *Cloud Computing for Data-Intensive Applications*, pages 83–104. Springer, New York, NY, 2014.