Journal of Reliable Intelligent Environments
https://doi.org/10.1007/s40860-018-0056-3

ORIGINAL ARTICLE

Benchmarking Pub/Sub IoT middleware platforms for smart services

Carlos Pereira[1] · João Cardoso[1] · Ana Aguiar[1] · Ricardo Morla[2]

Received: 15 September 2017 / Accepted: 23 January 2018

Abstract
Middleware is being extensively used in Internet of Things (IoT) deployments and is available in a variety of flavors. Despite this extensive use and diversity, a fair comparison of the benefits, disadvantages, and performance of each middleware platform is missing. This comparison is relevant to support the decision process for IoT infrastructure. In this paper, we propose a set of qualitative and quantitative dimensions for benchmarking IoT middleware. We use the publication–subscription of a large dataset as use case inspired by a smart city scenario to compare two middleware platforms with standard ambition: FIWARE and oneM2M. We take these metrics and use case and systematically compare the two middleware platforms in the wild. We identify inefficiencies in implementations and characterize performance variations throughout the day, showing that the metrics may also be used for monitoring. Furthermore, we apply the same metrics and use case to two brokers set up in a controlled environment, providing infrastructure- and networking-independent insights. Finally, we summarize useful practical know-how acquired in the process that can speed up entrance into the topic and avoid configuration and implementation pitfalls that impact performance.

Keywords Benchmarking · Internet of things (IoT) · Machine-to-machine (M2M) communications · Middleware platforms · System performance

1 Introduction

The promises brought forth by Internet of Things (IoT) have fueled the deployment of large-scale sensing and actuating infrastructures in diverse areas of application, like smart cities, smart grids, smart home, logistics or healthcare. Middleware platforms are intermediaries between sensors, services, and applications, managing the flow of data and allowing them to interoperate. Different flavors of middleware are being used to speed up IoT deployment, by providing a set of common functionalities and allowing interoperability between devices and services/applications that consume the data and make it useful [1]. Because data flows through the middleware at all components of the system, a particular implementation of the chosen middleware may not provide the features that a given deployment requires or may have a detrimental impact on the performance of the applications.

We have failed to find a systematic study and comparison of the performance, benefits, and disadvantages of the different middleware platforms that can be used in IoT, although some efforts exist [2–4]. In [5], the authors address the performance of data ingestion rate in OpenIoT [6], but focus mainly in the resource (CPU and memory) usage effects.

Our goal is to define a set of qualitative and quantitative dimensions along which it is possible to compare middleware platforms that allow the development of IoT applications and services. To achieve this, we chose a typical smart cities [7] IoT scenario that identifies communication models and load

This work is a result of the project NanoSTIMA (NORTE-01-0145-FEDER-000016), supported by Norte Portugal Regional Operational Programme 2014/2020 (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF).

✉ Carlos Pereira
dee12014@fe.up.pt

João Cardoso
jmmesquitacardoso@gmail.com

Ana Aguiar
anaa@fe.up.pt

Ricardo Morla
rmorla@fe.up.pt

[1] Faculty of Engineering of the University of Porto, Instituto de Telecomunicações, Porto, Portugal

[2] Faculty of Engineering of the University of Porto, INESC TEC, Porto, Portugal

Published online: 02 February 2018

scenarios. A typical communication model in this context is the publish–subscribe model [8] for accessing dynamic data as it allows greater scalability and flexibility than the request–response model. Load varies greatly depending on the process and type of data being sensed. We consider the motivating use case of the periodic publication–subscription of a large dataset. The data that our application publishes is the average speed of traffic in each street of the city of Porto in an hourly basis, corresponding to 19,884 data points every hour, e.g., as would be required for a routing service. This data might be extracted from a wide range of the mobile and static sensors spread throughout the city, and mapped to the edges of the OpenStreetMaps' city graph. The qualitative and quantitative analyses proposed can provide insights on which middleware is better suited for each scenario, bringing rational arguments to the problem of choosing which middleware platform to use.

In this paper, we start by performing a qualitative and quantitative evaluation of two middleware platforms in the wild (as per the shorter version of this paper [9]). The qualitative analysis identifies middleware functionalities and characteristics relevant for an IoT application. For quantitative evaluation we define a set of performance metrics on specific communication scenarios. We apply the methodology to two different middleware platforms to which we have access for these experiments (FIWARE[1] and oneM2M/ETSI M2M[2,3]). Despite the concrete instantiation, we expect the proposed methodology to generalize to other middleware platforms, use cases and loads. The metrics proposed may be used to monitor the operation of such platforms, by offering insights on the internal behavior and may allow to uncover problems with specific implementations. We then extend our contributions by applying the same methodology in a controlled environment. This removes variability related to the network and computing infrastructure, and provides fairer comparison conditions for protocols, marshalling and also implementation options, e.g., the chosen database engine.

This paper is organized as follows: Sect. 2 presents our benchmarking dimensions, Sect. 3 describes the middleware platforms that we analyze, and Sects. 4, 5, and 6 show, respectively, the results of our qualitative, in the wild, and controlled quantitative analyses to the two middleware platforms. We present pitfalls and recommendations that result from our work in Sect. 7 and concluding remarks in Sect. 8.

[1] https://www.fiware.org.

[2] http://www.onem2m.org.

[3] http://www.etsi.org.

2 Benchmarking IoT middleware

2.1 Qualitative dimensions

To define the qualitative dimensions along which to compare middleware, we took the perspective of an IoT infrastructure operator that would like to see his data used by services/applications. In this sense, besides requirements analysis, it is also important to take into account aspects that foster the adoption of platforms by developers, like quality of documentation and support. Thus, we arrived at the following aspects:

- support for the desired communication model (pub-sub and/or request–response);
- IoT application requirements, based on the requirements defined by the IoT-A[4];
- viability and possible limitations in each scenario;
- availability and clarity of the documentation, as well as available tutorials, and the quality of the support and livelihood of developer communities, e.g., Stack Overflow.[5]

2.2 Quantitative dimensions

To evaluate their performance, we chose speed and efficiency. As speed, we mean the time to send or retrieve data. The efficiency measures the overhead imposed by the middleware, including the increase in the size of the data sent through the network and the total number of bytes needed to send the data.

We propose the following metrics:

- publish time: elapsed time between sending the publish request and receiving the publish response; t_{PUB} in Fig. 1;
- subscribe time: elapsed time between sending the publish request and receiving the subscribe notification; t_{SUB} in Fig. 1;
- total time: elapsed time between sending the first publish request and receiving the last publish response;
- size of marshalled data: data serialization overhead measured as the content-length [11] header of the application protocol, in bytes;
- size of the publication: size in bytes of the payload of the transport protocol of the publish packet;
- total amount of data used to publish a resource: measured as the sum of the network level sizes of all packets exchanged (publisher to broker and reverse direction), in bytes;

[4] http://www.meet-iot.eu/iot-a-requirements.html.

[5] http://stackoverflow.com.

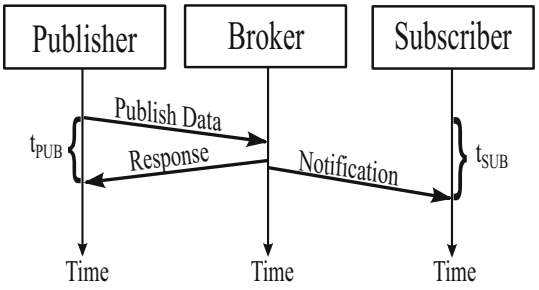

Fig. 1 Sequence diagram exemplifying the publish and subscribe times

– goodput: the useful number of bytes sent in a given period, measured as the size of the serialized data divided by the publish time.

During analysis of performance results, we defined an additional set of metrics which is useful to validate results and provide support for explaining behaviors observed in the main metrics. These auxiliary metrics are:

– number of request retries in cases where the publish request fails due to a problem with the broker. E.g., errors in the 500 range status code for HTTP [12], RST flags for TCP, etc.;
– round trip time to broker, obtained through the difference between the timestamp of the TCP SYN packet and its response [13] for TCP connections, or measured at application-level for UDP connections through the difference between the timestamp of a publish request and the respective acknowledge;
– number of re-transmissions of transport and application protocol packets and the delay verified for the re-transmission.

By combining main and auxiliary metrics, we can infer whether:

– an increase in the publish or subscribe times is related to an increase in the round trip time (network congestion), or to reduced broker performance;
– an increase in the total time to publish all data is related to an increase in the number of retries, packet re-transmissions, or both.

3 IoT middleware platforms

3.1 FIWARE

The FIWARE middleware has a series of components known as Generic Enablers. These components aim to ease the development of complex applications in areas including security and data management. One of the data management components is the Orion Context Broker. Orion uses the publish–subscribe model and generic data structures, known as Context Elements, to represent information. This information can be, for example, the temperature of a given room. These context elements are represented using JSON, and they have a predefined structure (NGSIv2). This platform is RESTful, and therefore, all operations use one of the CRUD methods. The broker has two APIs, v1 and v2.[6] The APIs differ in the data structure and in the fact that only the second version allows string or geographical filters to be applied to entity queries. The latter is a useful feature for location-based services, as often used in smart city applications and in our use case.

Access control can be provided when Orion is combined with the Steelskin PEP[7] component, which is an authentication mechanism independent of the broker that verifies whether the client has permissions to access a resource by intercepting the request before it reaches the broker. The information verified to allow access is the following:

– an OAuth[8] token generated by the authentication server, which is generated once and then included in the requests via x-auth-token header;
– a ServiceId, obtained through the Fiware-Service header and that identifies the component protected by this mechanism;
– a SubServiceId, obtained through the Fiware-Service Path header, that identifies futures sub-divisions of the service;
– the desired action.

An open source reference implementation of each of the FIWARE components is publicly available.

3.2 OneM2M/ETSI M2M

Machine-to-Machine (M2M) communications allow wireless and wired devices and services to exchange or control information without the need for human intervention [14]. M2M communications are a key enabler of IoT by, for example, making data from several sensors available publicly or connecting devices and sensors to the Internet (IoT) to process the data collected by these.

The oneM2M standard is the reference for global and end-to-end M2M communications in terms of service level. Although oneM2M cannot be considered an extension of

[6] https://github.com/telefonicaid/fiware-orion.
[7] https://github.com/telefonicaid/fiware-pep-steelskin.
[8] http://oauth.net.

ETSI M2M, both share the same functions and capabilities at the service layer as most of oneM2M current specifications are based on the ETSI M2M service layer. OneM2M/ETSI M2M settles on an M2M architecture with a series of generic capabilities for M2M services, and it defines a resource model, easing the device's integration and interoperability, as well as the development effort of horizontal applications. The homogeneity the standard provides makes it easier to develop M2M applications and for objects using M2M applications to interact.

OneM2M/ETSI M2M is RESTful, thus adhering to the REST principles and using CRUD methods. Information is represented in the form of resources that are structured in a tree-like way [15,16]. OneM2M/ETSI M2M allows for synchronous communications, using the request–response model, and asynchronous communications, using the publish–subscribe model.

Unlike FIWARE, oneM2M/ETSI M2M does not have a reference implementation and we had to choose a middleware implementation for the broker and API library. We use the ETSI M2M-compliant broker and client library described in [17] for the measurements in the wild. We use the oneM2M-compliant OM2M broker and client library[9], called OM2M, initially described in [18] for compliance with ETSI M2M, for the controlled measurements.

4 Qualitative analysis

4.1 Communication model

Both FIWARE Orion Context Broker and oneM2M/ETSI M2M brokers implement the publish–subscribe model.

4.2 IoT-A requirements

It would be too exhausting to go over all IoT-A requirements. However, we identified the following as relevant for an architectural analysis: UNI.001, UNI.002, UNI.005, UNI.008, UNI.016, UNI.022, UNI.023, UNI.030, UNI.036, UNI.047, UNI.048, UNI.067, UNI.071, UNI.073, UNI.092, UNI.094, UNI.240, UNI.245, UNI.405, UNI.406, UNI.426, UNI.607, UNI.608. These requisites concern interoperability at protocol and data level, security, access control and anonymity, name-based access and self-description, support for queries (semantic, location). Of these, FIWARE does not fulfill UNI.405 (support for multiple coordinate systems), and oneM2M/ETSI M2M does not fulfill UNI.405 and UNI.406 (support for geographic queries). Although we found no support for geographic queries in oneM2M, we believe they will be included in the standard in the near future.

[9] http://www.eclipse.org/om2m/.

4.3 Viability and limitations

FIWARE Orion Context Broker imposes a 1 Mbyte limit on the publish request size and 8 Mbyte limit on the subscription/notification size. Our example dataset contains 19884 data points. It is not feasible, or desirable, to publish all this data as a single resource. The first reason for this is that the notification size sent to subscribers would exceed the limit. The second and more important reason is that filter queries would not be possible, each subscriber only having the possibility to receive the whole dataset. This might not be desired, e.g., for mobile applications. We thus decided to publish a single resource per street (graph edge). We define the following structure for the resource name: *average_speed_edgeId*. This way we can apply regular expressions to select all the resources whose name starts with 'average_speed' (*average_speed_*.**) and then apply filters that, for example, return the edges where the attribute 'speed' is higher than a given value.

We also use this mapping in oneM2M/ETSI M2M. Different containers are created to store the different edges, and inside each container the contentInstance resources, which are the resources where data is meant to be stored, store the hourly data. Please note that, oneM2M/ETSI M2M defines the CRUD methods possible in each resource type, and contentInstance resources shall not be updated via API according to oneM2M (Clause 7.4.7.2.3 [15]).

4.4 Documentation and support

FIWARE is an open standard with a reference implementation and thus it has a variety of online documentation. The documentation includes an API walkthrough that describes each version of the API, detailing each operation that the Orion Context Broker component supports, as well as explaining how to set up an instance of the broker and how to run it. Furthermore, each version of the API has a specific web page directed at detailing each operation and also offering code snippets of these operations in several programming languages. As the second version is still in beta state, there are two web pages for this version, one with the latest changes and the other with a stable version of the API.

There are various documents specifying all aspects of the oneM2M/ETSI M2M standard, such as the binding with protocols like HTTP, CoAP, and MQTT, or its functional architecture [16]. Because oneM2M/ETSI M2M only defines the standard, these documents do not cover implementation issues. Nonetheless, specific implementations do offer their own documentation and wiki.

In terms of support, FIWARE has a community on the Stack Overflow web site with over 1800 questions tagged at the moment of writing this paper. There is also a FAQ (Frequently Asked Questions) available about the general

FIWARE platform.[10] Additionally, there are mailing lists directed at each kind of problem, such as technical issues with some of the Generic Enablers, that were used to clarify some doubts in the obtained results.

The oneM2M/ETSI M2M community on Stack Overflow is less active, with fewer than 40 questions tagged with “OneM2M" or “ETSI M2M" or “OM2M". Nonetheless, OM2M wiki and forum are available.[11]

5 Quantitative analysis—in the wild

5.1 Setup

These experiments were done on a DigitalOcean virtual machine located on their London data center and running Ubuntu 14.04 x64, on a two core *Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz* with 2 Gbytes of RAM, and a 1 Gbit/s Ethernet card shared between the virtual machines running on the same physical machine.

We used a FIWARE account on the public Barcelona FIWARE broker. This broker was chosen as it is similar to a production one and no production brokers were available to us. This broker implements HTTP.

The ETSI M2M broker is available to our group in the Faculty of Engineering in Porto. The broker implements HTTPS; due to this, the payload TCP size of the TLS Application Data packet is evaluated instead of the payload TCP size of the HTTP protocol.

The publisher and subscriber clients are implemented in Node.js for FIWARE, and in Java for ETSI M2M using the library available from [17]. Additionally, a packet capture process runs the tcpdump[12] tool in the background to capture network level metrics.

Taking into consideration the limitations of each middleware, we used the following mapping to comply with our use case scenario. For ETSI M2M, each edge was published by creating a new contentInstance resource in an individual container resource already existent in the broker. For FIWARE, each edge was published in an individual entity resource already existent in the broker, that is, in each publish request we updated the value of an existing, but different for each edge, entity resource.

The data that is sent in each publication contains the average speed of traffic in the last hour of the street where the edge is, the average speed in the last week, and a combined average of the speed, as well as the edge identification, and the GPS location of the edge. For example, the following JSON string contains the data for a specific edge: {“*last_hour_average*" : 8.942157, “*weekly_average*" : 9.942157, “*combined_average*" : 7.942157, “*edge_id*" : 146212295, “*lon*" : −8.449180921568626, “*lat*" : 41.51657045098038}.

5.2 Methodology

Publishing a large dataset as individual data elements can be done in two ways: sequentially, or in parallel. For sequential publication, we performed 4 publish cycles as the amount of time required to perform each one limited the total number of measurements. For the parallel publication, each experiment consisted of 10 publish cycles with a number of parallel requests ranging from 50 to 500, in increments of 50. We limited the number of parallel requests to 500 because brokers were not able to cope with more parallel requests. This variation on the number of parallel requests allowed us to observe differences in the behavior of brokers, as the load on the broker increases, including the number of retries, the publish–subscribe times, and the number of packet re-transmissions. FIWARE experiments were done in the morning, afternoon, and night. In each part of the day, we repeated each experiment 3 times. In the case of ETSI M2M, only one experiment was done in the whole day, since the ETSI M2M’s excessive publish time did not allow for more than 4 publish cycles in each part of the day. No synchronization was necessary between publisher and subscriber as they were located in the same machine.

5.3 Measurement results

5.3.1 Parallel publication

ETSI M2M publish, subscribe, and total publish times were approximately 850% greater than FIWARE times. We believe it was caused by a problem with the ETSI M2M library we used that did not allow for more than 2 network connections at the same time. As such, it was not possible to publish more than 2 resources at the same time, no matter what number of parallel requests was chosen. This also led to the absence of retries.

Each boxplot, in the figures presented throughout this work, show the distribution of the measured data. The top and bottom of each “box” represent, respectively, the 25th and 75th percentiles of the measured samples, and the distance between these two is the interquartile range. The outliers, observations outside the whisker length, are the values that are more than 1.5 times the interquartile range away from the top or bottom of the box. The line in the middle of each box represents the median.

Publish times for both middleware platforms can be observed in Figs. 2 and 3. The ETSI M2M average publish

[10] http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE_Frequently_Asked_Questions_(FAQ).

[11] http://www.eclipse.org/om2m/.

[12] http://www.tcpdump.org.

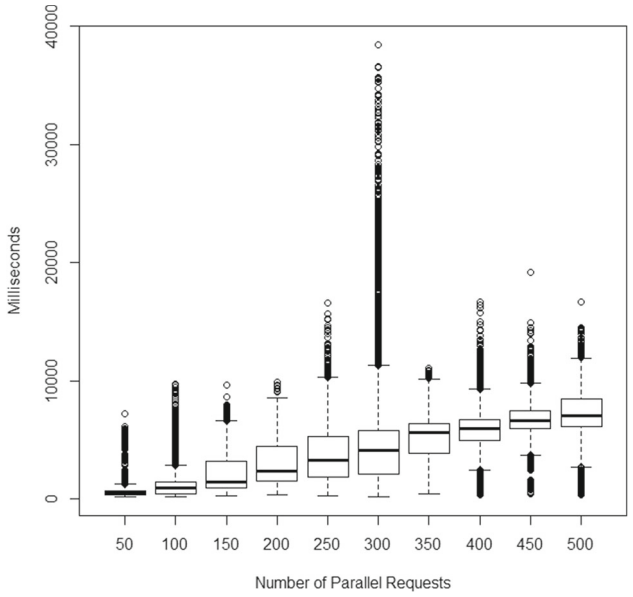

Fig. 2 FIWARE publish times for different number of parallel requests

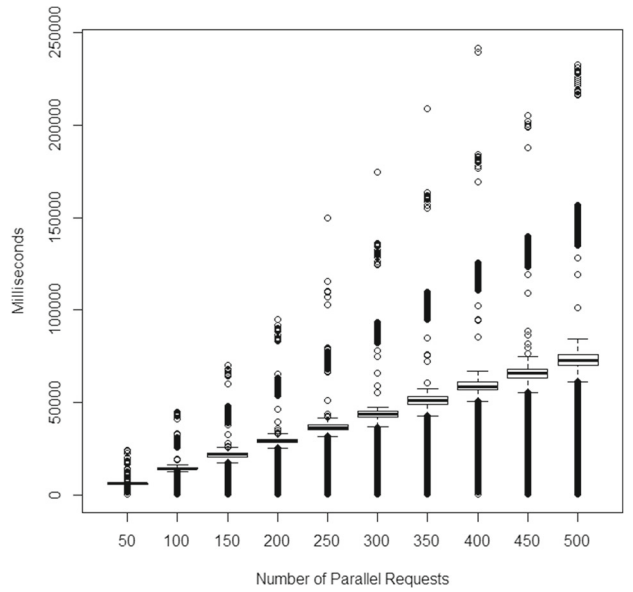

Fig. 3 ETSI M2M publish times for different number of parallel requests

times for a single resource publication range from 5992 to 728,58 ms for 50 and 500 requests in parallel, respectively. This is much larger than FIWARE publish times ranging from 630 to 7400 ms, again for 50 and 500 requests in parallel, respectively. The subscribe times are very similar to the publish times.

The ETSI M2M broker we use always sends the subscription notification before sending the HTTP response to the publish request. This results in a scalability problem when there are several subscriptions to a given resource, since the broker will send all the subscription notifications first before sending the HTTP response to the publish requests, and, ultimately, leading to larger than necessary publish times. On the contrary, FIWARE sends the subscription notifications asynchronously, and therefore, sometimes the subscription notification arrives before the HTTP response to the publish request and at other times it arrives after.

The average content-length for ETSI M2M is 390 bytes, 55% greater than that of FIWARE which is 251 bytes. This is unexpected as FIWARE data structure format is JSON which tends to be lengthy. We believe the reason behind this larger than expected ETSI M2M content-length is an implementation inefficiency of the library: we verified that the ETSI M2M library sends unnecessary attributes in the payload, such as the content-type which should only be in the headers of the HTTP packet. The data itself only accounts for 146 bytes, which is lower than in FIWARE that requires the JSON overhead.

An average of 511 bytes of TCP payload size was observed for FIWARE, whereas for ETSI M2M this value is 728 bytes; we do not include the 40 bytes average TLS overhead in this value. This difference in size is due to the increase in size of the content-length discussed above.

ETSI M2M registers an average of 1118 bytes for the HTTP response to the publish request, whereas FIWARE registers an average of 417 bytes. There is quite a difference in size between these two middleware platforms, and that again is due to the specific implementation we use, since the response to the publish request of the ETSI M2M library is not a simple OK with no payload, but instead contains a copy of the published data which only adds inefficiency to the communication.

The goodput measured in ETSI M2M is on average 2662 bytes/s, while in FIWARE the goodput ranges from 351 bytes/s (lowest value registered) to 1700 bytes/s (highest value registered). This is due to the fact that only 2 network connections are active at the same time, and therefore, the goodput is higher because the lower the number of active requests there is, the higher the rate at which they are processed. These values are depicted in Figs. 4 and 5.

Figure 6 shows that the number of retries in the FIWARE experiments grows throughout the day, indicating a possible a memory leak in the broker. To verify this hypothesis, additional measurements were conducted on two consecutive days. We observed that the number of retries grew only throughout the day and did not pick up from the last value registered on the previous day, as seen in Fig. 7. We updated our hypothesis and assumed that the broker had been restarted somewhere in between the period where no measurements were taken. To confirm this hypothesis, we contacted the FIWARE team via email and were told that the retries were due to the authentication proxy—Steelskin PEP, mentioned in Sect. 3—and not due to the broker load itself, as was

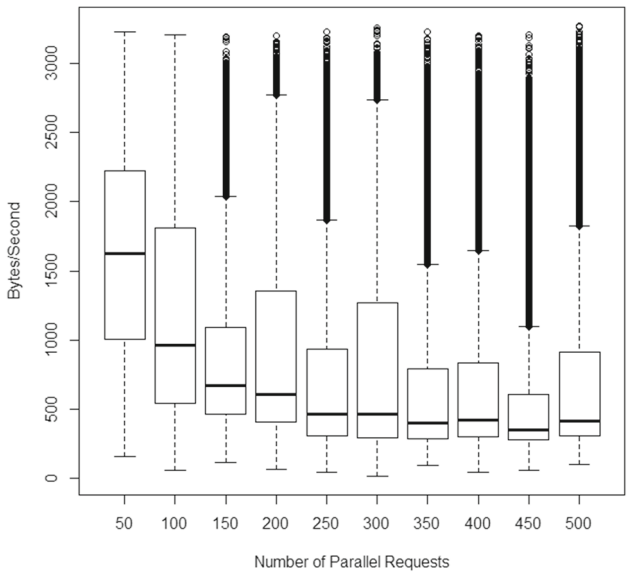

Fig. 4 FIWARE goodput for different number of parallel requests

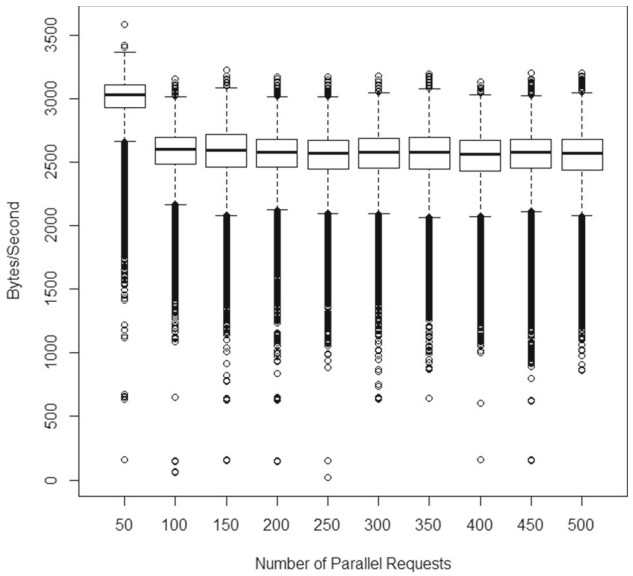

Fig. 5 ETSI M2M goodput for different number of parallel requests

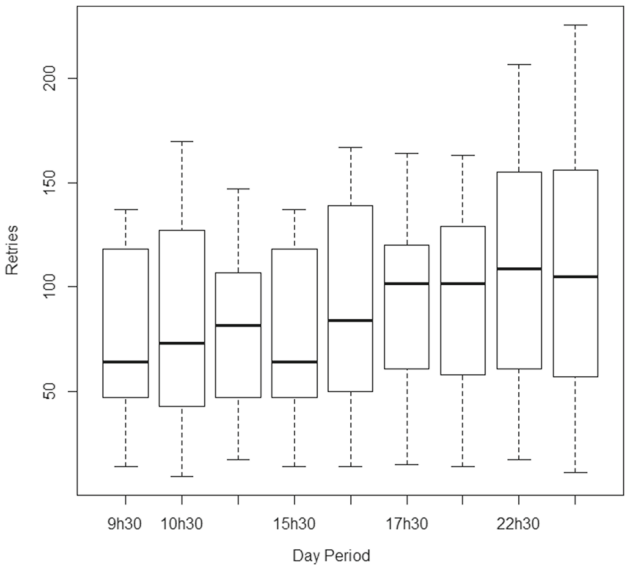

Fig. 6 Number of retries in FIWARE throughout a day

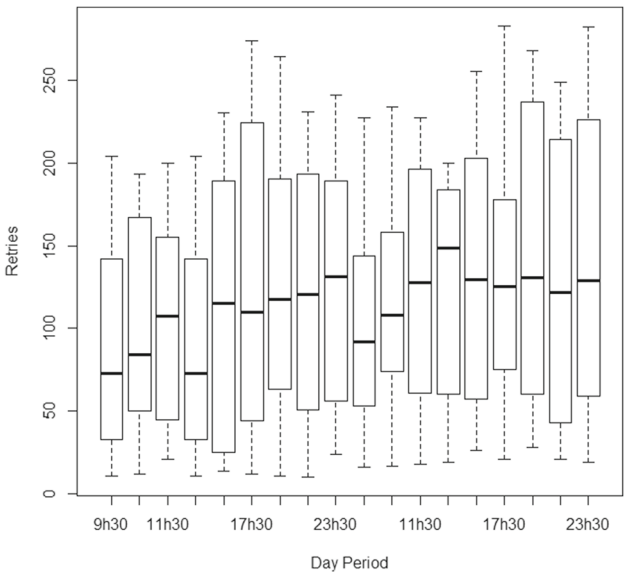

Fig. 7 Number of retries in FIWARE throughout two consecutive days

suspected. The FIWARE team also told us that they were not aware of any broker malfunction, restart, or memory leak.

5.3.2 Sequential publication

We observe differences between the sequential and the parallel publication scenarios for both middleware platforms. This is expected as the sequential mechanism has an implication in the publish time, subscribe time, total time to publish all data, and goodput. Publish times are smaller in the sequential publication for both middleware platforms. FIWARE's average publish time is 175 ms, while ETSI M2M's is 282 ms which is almost 61% greater than FIWARE's. Subscribe times were also smaller for both middleware platforms in the sequential publishing. FIWARE's average subscribe time was 227 ms with a performance similar to ETSI M2M with 237 ms.

ETSI M2M performs considerably better for sequential publication, as its implementation limits the performance in parallel publication. This scenario allows a fair comparison between both middleware platforms; however, the results show that FIWARE performs better overall.

The observed total time to publish all data sequentially is 48% greater in ETSI M2M than in FIWARE. FIWARE took nearly 64 minutes to publish all data sequentially, an increase

of 1180% from parallel publication. ETSI M2M took nearly 98 minutes publish all data sequentially, an increase of 98% from parallel publication. As expected, the parallel publication is the recommended choice for the timely dissemination of data.

The average goodput for ETSI M2M is 1444 bytes/sec, while for FIWARE it is 2660 bytes/s. Both middleware platforms show different behaviors in terms of goodput when compared to the previous scenario which can be once again due to differences in the implementation.

6 Quantitative analysis—controlled environment

6.1 Setup

These experiments were performed in our lab, with the brokers installed on a dedicated server running CentOS 6.9, on a *Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz* with 8 Gbytes of RAM, and with a connection of 100 Mbit/s with our client machine running Debian 8.9, on a *Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz* with 2 Gbytes of RAM. We followed the same approach as previously by deploying publisher and subscriber in the same client machine.

We installed an Orion Context Broker instance to evaluate FIWARE's performance and the OM2M broker to evaluate oneM2M's performance. FIWARE uses HTTP as application protocol. The OM2M contains bindings for HTTP, CoAP, and MQTT application protocols. The use of MQTT, however, is only possible with the use of an external MQTT broker as intermediary. For that, we used the Mosquitto broker.[13]

The publisher and subscriber clients were implemented in Java for all cases. For FIWARE and OM2M HTTP clients, we used our own REST clients, based in the HTTPClient Apache implementation. We used the Eclipse Paho open-source client implementation of MQTT as base for our OM2M MQTT clients, and CoAP Californium as base for our OM2M CoAP clients. We used MQTT with QoS 1 and CoAP with confirmable messages to get publish responses.

We followed the same approach in all clients to create publish requests to have the same payload, a JSON format payload, except for MQTT as the client requires additional data. No security was added to any of the brokers, and only a short token was added to both OM2M (11 Bytes) and FIWARE (5 Bytes) as access control.

We use the same mapping as in the previous section: for OM2M, each edge is published by creating a new contentInstance resource; contentInstances are located under different container resources representing different edges. For FIWARE, each edge is published in an individual entity resource that is created in the broker beforehand, and in each publish request we update the value of the entity resource of the corresponding edge.

The sensor data that is sent in each publication is the same as presented in the previous section.

6.2 Methodology

We performed sequential as well as parallel publication measurements. We varied the number of parallel requests from 1 (i.e., sequential) to 5. We limited the number of parallel requests to 5 because we observed that because of some configurations of the local implementation of the OM2M broker it was not able to cope with more parallel requests. Nevertheless, this methodology allows us to perceive not only differences in the behavior of the brokers but differences of the same broker for different application protocols. We repeated each measurement 3 times.

There is only one broker running during each measurement. Between measurements the broker is reset—including its database—to ensure the same conditions for all measurements. In the case of updating values in FIWARE we only start the measurements after the entity resources are created.

6.3 Measurement results

6.3.1 Publish and subscribe times

OM2M HTTP and OM2M CoAP have much better average publish times than FIWARE. FIWARE's publish times range from 21251 to 98,635 ms for 1 and 5 requests, respectively. The average publish times of OM2M CoAP are the best, ranging from 5873 to 8696 ms. The use of CoAP represents an average decrease of 1.1% of the publish time when compared with the use of HTTP in OM2M, which shows small benefits of using UDP in addition to the smaller, binary base header format.

The use of MQTT as application protocol—including the Mosquitto broker as proxy between the client and the OM2M broker—represents an increase of the publish time of nearly 9.6 times when compared with the use of HTTP and an increase of approximately 9.7 times when compared with the use of CoAP. The publish times for the middleware platforms with the different configurations can be observed in Fig. 8.[14]

Figure 9 shows the difference between publish and subscribe times for the different configurations of the two middleware platforms. Subscription notifications are sent asynchronously, and therefore, sometimes the subscription notification arrives at the subscriber before the response to

[13] https://mosquitto.org/.

[14] Y axis are truncated for visualization purposes in boxplots throughout this section.

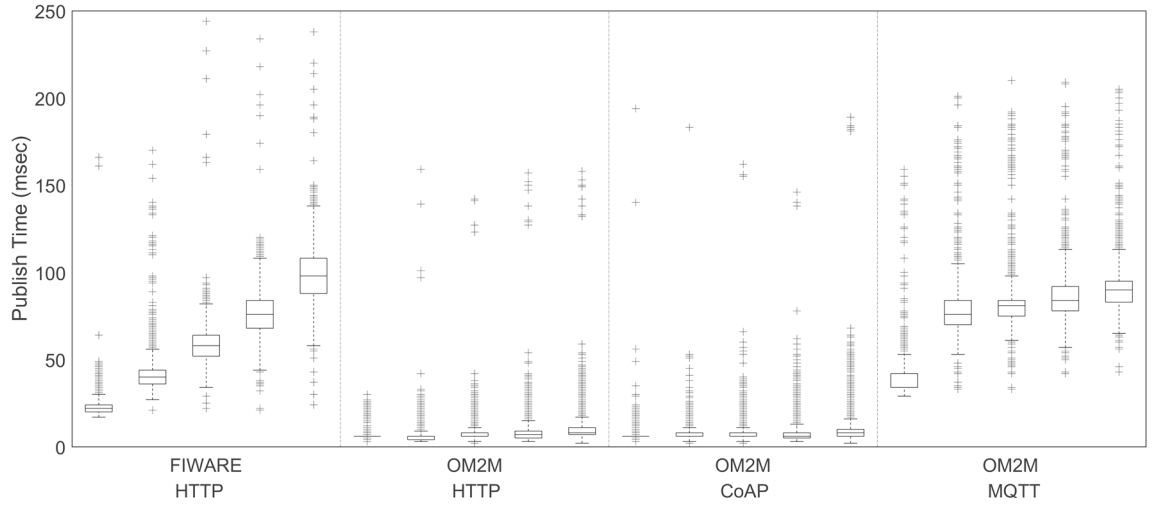

Fig. 8 Publish times for different broker configurations and different number of parallel requests (Left: 1; Right: 5)

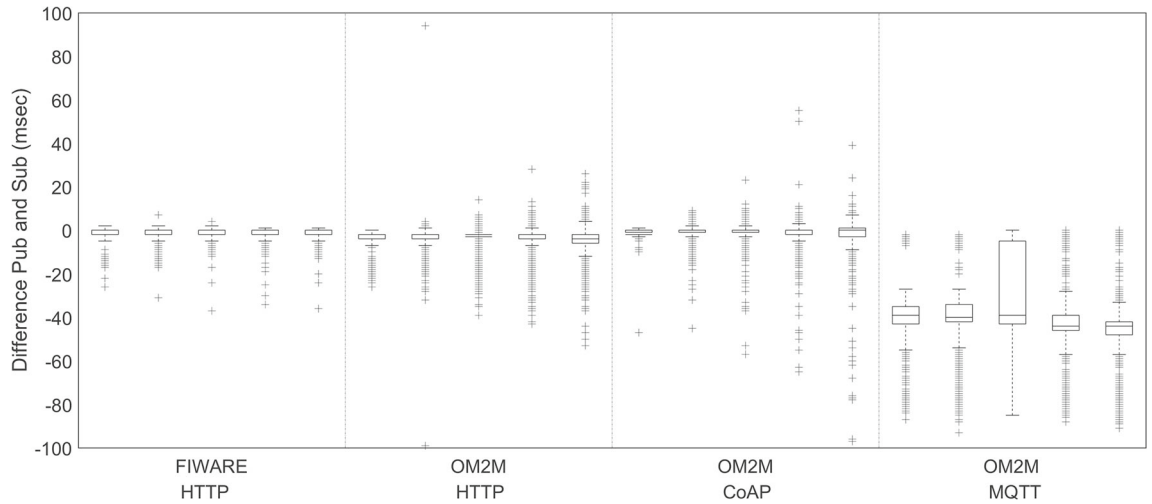

Fig. 9 Observed difference between publish and subscribe times for different broker configurations and different number of parallel requests (Left: 1; Right: 5)

the publish request arrives at the publisher. This produces both positive and negative values for the publish–subscribe difference that are independent of the number of parallel requests and practically the same—with the exception of OM2M MQTT. Because the Mosquitto broker is also used as intermediary in the notifications, the publish–subscribe difference in OM2M MQTT is much larger than in the other cases.

We detected a problem with the use of OM2M CoAP. After almost 1000 publish requests, a router internal error occurs in the broker. This error originates from a DatabaseException and causes the broker to stop delivering notifications. Therefore, the OM2M CoAP's subscribe time and publish–subscribe difference are based on this number of publications/notifications.

Figure 10 shows the total time to publish all data for all scenarios. The average total time to publish all data was nearly 395 (423 to 392) s for FIWARE, 58 (118 to 37) s for OM2M HTTP, 58 (117 to 35) s for OM2M CoAP, and 567 (755 to 358) s for OM2M MQTT. The latter represents an increase of nearly 8.8 times to the fastest (OM2M HTTP and OM2M CoAP). We observe that although the average publish time increases with the number of parallel requests (as per Fig. 8.), the total time to publish all data decreases—except for FIWARE. This shows that the parallel publication is thus the recommended choice for sending large datasets; in the end of this section we discuss the reasons for this not happening with FIWARE. We further observe that the use of Mosquitto considerably decreases the performance.

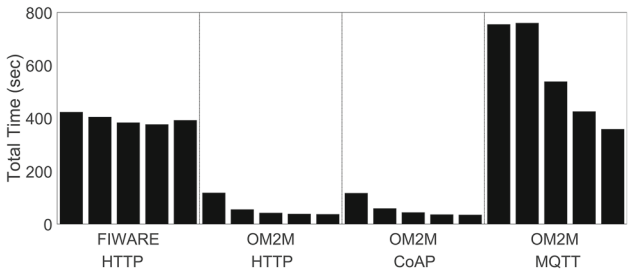

Fig. 10 Total time to publish all data for different broker configurations and different number of parallel requests (Left: 1; Right: 5)

6.3.2 Goodput and publish subscribe sizes

The goodput values measured for each packet and shown in Fig. 11 are a clear indicator of the performance of the different configurations. FIWARE's average goodput is 5 Kbyte/s, similar to the 6 Kbyte/s measured with OM2M MQTT. OM2M CoAP and OM2M HTTP have an average of 40 Kbyte/s. We did not observe publish request retries on any of FIWARE's or OM2M MQTT's measurements. We observed a negligible number of retries on OM2M HTTP and OM2M CoAP.

Given that in a controlled environment we have extended flexibility over both FIWARE and OM2M clients, we aim to create payload contents with similar sizes for publishing data. As such, the average content-length is 235 bytes for OM2M HTTP and 241 bytes for OM2M CoAP. As FIWARE does not require the use of specific JSON fields in the payload, the average content-length is slightly smaller at 220 bytes. The average content-length for OM2M MQTT is 349 bytes— larger than for the others and as expected given that the payload must contain the necessary data for the OM2M broker to decode the client's request. This payload includes the identification of the operation to be performed and a request identifier. FIWARE uses 296 bytes for the HTTP protocol headers (Token and URI path are different) and 52 bytes for TCP. The use of MQTT specifically in OM2M MQTT only adds 36 bytes of application protocol headers including the topic filter and in addition to the 52 bytes of the TCP header. OM2M CoAP uses 71 bytes for application protocol overhead and 20 bytes for UDP. OM2M HTTP uses 301 bytes for the HTTP headers and 52 bytes for TCP. We used shorter URIs with oneM2M than previously with ETSI M2M, as in oneM2M there is no need to send literal strings preceding each resource like in ETSI M2M is – such as applications/<app>, containers/<container>, or contentInstances/<contentInstance>.

For response to the publish request we register an average of 164 bytes for the HTTP protocol headers with no payload in FIWARE. FIWARE does not include a copy of the information published, while OM2M responds every time with the resource being created plus a series of additional M2M verbosity. The publish responses of the Mosquitto broker also do not have payload, because identifying the topic is not needed in the response. As such only 4 bytes are used for MQTT protocol headers. The average response length for OM2M HTTP and OM2M CoAP is 465 bytes. The OM2M HTTP application protocol headers are an average of 210 bytes long; application protocol headers are 50 bytes long in OM2M CoAP. The oneM2M-compliant broker adds significant inefficiency to the communication by responding with the published data—a fact we observe also in our on the wild experiment in Sect. 5. This behavior is allowed, but not mandatory by standard.

We register an average of 741 bytes for each subscription notification in FIWARE—including 310 bytes for the HTTP protocol headers and 365 bytes for content. OM2M HTTP

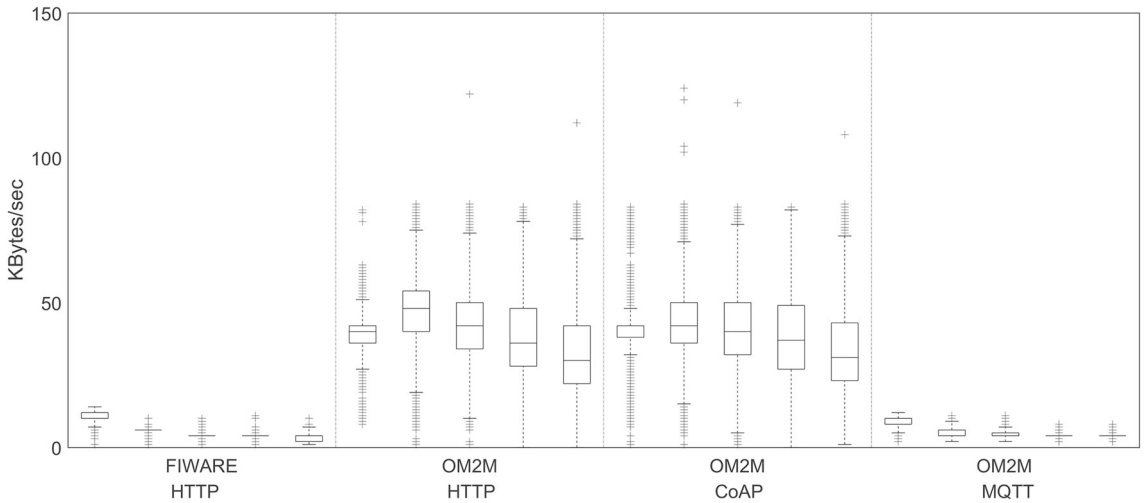

Fig. 11 Goodput for different broker configurations and different number of parallel requests (Left: 1; Right: 5)

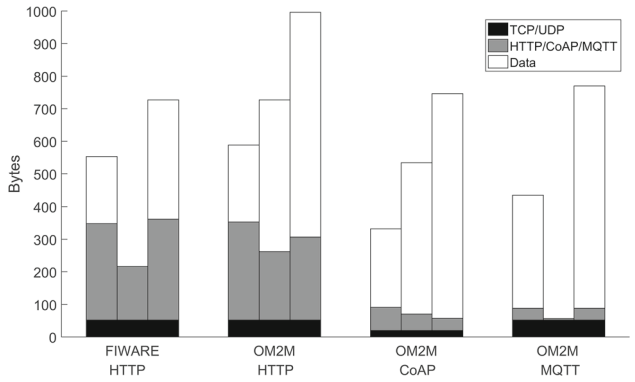

Fig. 12 Publish request (left bar), response (center bar), and notification (right bar) sizes for the different configurations

and OM2M CoAP each have an average content-length of 677 bytes per notification. CoAP protocol headers are on average only 37 bytes as they contain less header options than in publishing, and HTTP protocol headers use 255 bytes on average. Notifications of OM2M MQTT have an average content-length of 682 bytes, and MQTT protocol headers have an average of 36 bytes once again. Fig. 12 shows these values.

6.3.3 FIWARE and the database

OM2M complies with the oneM2M technical standards that strictly define how and where on the resource structure should be published. The flexibility in FIWARE allows us to compare three different approaches of publishing data using its Orion Context Broker—that we could not compare in the previous section on the wild. The first approach is the one we have been using up to here in this section: we previously create an entity for each edge and upon incoming edge data we update the values of all entities. In the second approach, upon incoming edge data we create a new set of entities—each entity with the new value for its edge. This requires additional mechanisms for entity cleanup and subscription. In the third approach, a single entity is used to publish new data for all edges. We update the edge id attribute together with edge data in the entity, once for every edge and upon incoming edge data.

Figures 13 and 14 show the publish times and goodput for the three different approaches. There is a clear decrease in performance for updating the values of several entities (first approach) and for creating entities (second approach) when compared to updating the attributes of a single entity (third approach). The average total time to publish all data in the second approach is nearly 176 (223–167) s and only 40 (65–30 s) in the third approach. The average publish time is 58 s for the first approach, 25 s for the second, and 5 s for the third.

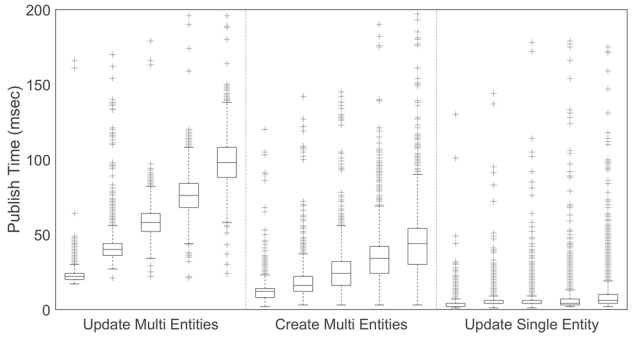

Fig. 13 Publish times for FIWARE with three different ways to publish data and for different number of parallel requests (Left: 1; Right: 5)

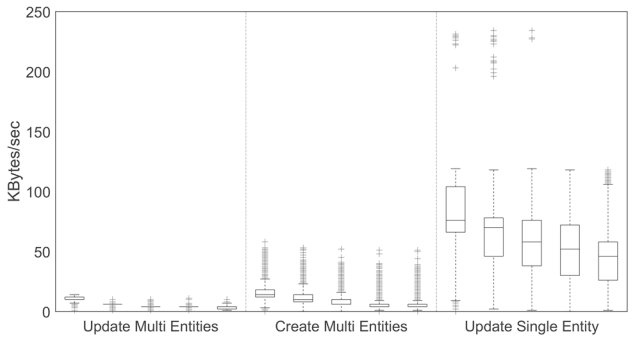

Fig. 14 Goodput for FIWARE with three different ways to publish data and for different number of parallel requests (Left: 1; Right: 5)

Publishing data using the third approach results in FIWARE having slightly better performance than OM2M HTTP and OM2M CoAP. We relate this variation in FIWARE's performance to the database. We observe that in the boxplots of the first approach there are no publish requests with small publish times. As we are doing updates to entities in a database with a rather large number of entities, each update takes longer, and more parallel requests mean larger times due to concurrency. In the second approach, we see that the boxplots have measurement points spread across the entire Y axis. The database locks writing updates to the different entities. The requests that can access the lock to create a new entity will have low publish times; the rest of the requests must wait and experience larger publish times. As a final argument for the impact of the database on the FIWARE broker performance, we stress the fact that all reported measurements were done in a clear broker and empty database. To have a better sense of the effect of the database on performance, we ran the third approach in a database that we did not reset after running the first approach. This means we create entities for every edge first—but measure and update incoming edge data on a single entity only. We observe much worse performance with this approach than with the third approach in which a single entity exists in the database and thus confirm the effect of database size on broker performance.

7 Pitfalls and recommendations

In this section, we present important observations from designing these measurement campaigns that we believe can be useful to other researchers and developers.

- Enabling multi-connection capable clients: Default configurations for managing connections tend to limit the number of connections possible. For example, the Apache's PoolingHttpClientConnectionManager[15] creates and manages a pool of connections, where the default size of concurrent connections that can be open by the manager is 2 for each route or target host, and the total open connections is limited to 20. Furthermore, Eclipse Paho uses three threads for each client, where one is for sending, one for receiving, and one for call backs, which means that i different clients are necessary for each i number of parallel requests desired. Therefore, client-side implementations need to be tuned to allow multiple parallel connections;
- Java configurations: different OS can be packed with different defaults Java VMs packages. Although Java Server and Client VMs are rather similar, the Server VM is tuned to maximize peak operating speed. The Server VM, while taking more time to analyze and compile code and requiring higher memory footprint, executes several complex optimizations since it contains an adaptive compiler that supports several types of optimizations. Server VMs are more adequate for applications that require the faster possible operating speed, instead of, for instance, faster startup time;
- Application development: Developers need to pay close attention to how and where data is published. Publishing resources in different locations of the platform's resource tree, associated to different and/or inefficient ways to perform such requests, can cause different overall performance leading to different publishing times. Furthermore, while the execution of several parallel publications can in fact decrease the overall transmission time, in average each publication will be delayed. Therefore, if some resources need to be published with some type of quality of service requirements, then a sequential publication can be a better approach.

8 Conclusion

In this paper, we proposed a set of qualitative dimensions and quantitative metrics to compare IoT middleware. We used them together with a smart city use case with data from the Future Cities project to evaluate the performance of two well-known middleware platforms: FIWARE and OneM2M/ETSI M2M.

For publish–subscribe scenarios in which a user publishes large volumes of data either sequentially or in parallel, we measured larger publish times, subscribe times, and total times to publish all data with ETSI M2M than with FIWARE—on the wild with two specific deployments of these brokers. We conclude that the significant differences between the two middleware platforms observed in the parallel publication are mainly due to limitations of the ETSI M2M's implementation. On the other hand, we also observed performance variations throughout the day in FIWARE. We also conclude that parallel publication can significantly reduce the total time required to publish all data when compared to publish in a sequential way.

In a more controlled environment we were able to find that broker performance can depend on different components like the database and underlying communications protocol. For example, FIWARE performance with the same load degrades with single versus multiple entities in the database and the OneM2M performance is much larger with HTTP and CoAP protocols than with MQTT and its intermediate broker.

These results allow us to have a better understanding of the internal functioning of these middleware platforms, showing that the metrics proposed may be used to monitor their operation, and, by discovering implementation errors in the middleware itself or in their libraries, they allowed us to contribute for possible improvements.

Future work should focus in benchmarking and comparing more and more different middleware platforms in different use cases, eventually looking at which changes to the methodology and metrics would have to be included for different middleware platforms.

Acknowledgements The authors would like to thank Luís Zilhão for the support provided during the controlled experiments and thank the anonymous reviewers for their helpful feedback.

[15] https://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/impl/conn/PoolingHttpClientConnectionManager.html.

References

1. Bernstein PA (1996) Middleware: a model for distributed system services. Commun ACM 39(2):86–98. https://doi.org/10.1145/230798.230809
2. Konstantinos V, Vlasios T (2014) Performance evaluation of an IoT platform. In: Eighth international conference on next generation mobile apps, services and technologies, Oxford. https://doi.org/10.1109/NGMAST.2014.66
3. Martin A, Manish M, Gowtham B, Amip S, Jeff H, Ben V (2015) IoTAbench: an internet of things analytics benchmark. In: Proceedings of the 6th ACM/SPEC international conference on performance engineering (ICPE '15). https://doi.org/10.1145/2668930.2688055

4. PROBE-IT benchmarking framework (2017). http://www.probe-it.eu. Accessed 15 Dec 2017
5. Medvedev A et al (2016) Data ingestion and storage performance of IoT platforms: study of OpenIoT. In: Interoperability and open-source solutions for the internet of things (InterOSS-IoT). https://doi.org/10.1007/978-3-319-56877-5_9
6. Serrano M et al (2015) Defining the stack for service delivery models and interoperability in the internet of things: a practical case with OpenIoT-VDK. IEEE J Select Areas Commun 33(4):676–689. https://doi.org/10.1109/JSAC.2015.2393491
7. Caragliu A, Del Bo C, Nijkamp P (2011) Smart cities in Europe. J Urban Technol 18(2):65–82. https://doi.org/10.1080/10630732.2011.601117
8. Eugster PTH, Felber PA, Guerraoui R, Kermarrec A-M (2003) The many faces of publish/subscribe. ACM Comput Surv 35(2):114–131. https://doi.org/10.1145/857076.857078
9. João C, Carlos P, Ana A, Ricardo M (2017) Benchmarking IoT middleware platforms. In: IEEE 18th international symposium on a world of wireless, mobile and multimedia networks (WoWMoM). https://doi.org/10.1109/WoWMoM.2017.7974339
10. Fielding RT (2000) Architectural styles and the design of network-based software architectures. PhD thesis
11. W3C. HTTP/1.1: Header Field Definitions (2017). https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html. Accessed 15 Dec 2017
12. W3C. HTTP Status Code Definitions (2017). https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. Accessed 15 Dec 2017
13. Comer D (2000) Internetworking with TCP/IP, 4th edn. Prentice Hall, Upper Saddle River, New Jersey, USA
14. Pereira C, Aguiar A (2014) Towards efficient mobile M2M communications: survey and open challenges. Sensors 14(10):19582–19608. https://doi.org/10.3390/s141019582
15. oneM2M (2016) oneM2M TS 0001 V2.10.0 (2016-08)—oneM2M Technical specification; functional architecture. http://www.onem2m.org/images/files/deliverables/Release2/TS-0001-%20Functional_Architecture-V2_10_0.pdf. Accessed 15 Dec 2017
16. ETSI (2013) ETSI TS 102 690 V2.1.1 (2013-10)—Machine-to-Machine communications (M2M); functional architecture. http://www.etsi.org/deliver/etsi_ts/102600_102699/102690/02.01.01_60/ts_102690v020101p.pdf. Accessed 15 Dec 2017
17. Carlos P, Antonio P, Ana A, Pedro R, Fernando S, Jorge S (2016) IoT interoperability for actuating applications through standardised M2M communications. In: IEEE 17th international symposium on a world of wireless, mobile and multimedia networks (WoWMoM). https://doi.org/10.1109/WoWMoM.2016.7523564
18. Alaya MB, Banouar Y, Monteil T, Chassot C, Drira Khalil (2014) OM2M: extensible ETSI-compliant M2M service platform with self-configuration capability. Proc Comput Sci 32(1):1079–1086. https://doi.org/10.1016/j.procs.2014.05.536