Access Time Improvement Framework for Standardized IoT Gateways

Asad Javed*, Narges Yousefnezhad*, Jérémy Robert†, Keijo Heljanko‡§ and Kary Främling*
*Department of Computer Science, Aalto University, Espoo, Finland
†University of Luxembourg - Interdisciplinary Centre for Security, Reliability and Trust, Luxembourg
‡Department of Computer Science, University of Helsinki, Helsinki, Finland
§Helsinki Institute for Information Technology HIIT
{firstname.lastname}*@aalto.fi / †@uni.lu / ‡@helsinki.fi

***Abstract*—Internet of Things (IoT) is a computing infrastructure underlying powerful systems and applications, enabling autonomous interconnection of people, vehicles, devices, and information systems. Many IoT sectors such as smart grid or smart mobility will benefit from the recent evolutions of the smart city initiatives for building more advanced IoT services, from the collection of human- and machine-generated data to their storage and analysis. It is therefore of utmost importance to manage the volume, velocity, and variety of the data, in particular at the IoT gateways level, where data are published and consumed. This paper proposes an access time improvement framework to optimize the publication and consumption steps, the storage and retrieval of data at the gateways level to be more precise. This new distributed framework relies on a consistent hashing mechanism and modular characteristics of microservices to ensure a flexible and scalable solution. Applied and assessed on a real case study, experimental results show how the proposed framework improves data access time for standardized IoT gateways.**

***Index Terms*—Internet of Things, microservices, consistent hashing, distributed system, gateway, big data**

I. Introduction

The Internet of Things (IoT) technology is becoming a promising opportunity for developing powerful systems and applications, enabling autonomous interconnection of people, vehicles, devices, and information systems [1]. Various IoT sectors (e.g. industries, smart grid, smart mobility, ...) will benefit from the recent evolutions of the smart city initiatives, by leveraging ubiquitous connectivity, system interoperability, and analytics, for building more advanced IoT services. Such services can be shaped by the interaction and cooperation of Cyber-Physical Systems or Industrial Internet (of Things) technologies allowing the collection of human- and machine-generated data as well as their storage and analysis.

Whether in smart cities or industrial environments, these services have to be offered in real-time. As a consequence, it is of utmost importance to manage the volume, velocity, and variety of the data from different sources to be able to make timely decisions based on them. These data are usually made available through IoT gateways that implement open and standardized API, while enabling horizontal interoperability across vertically-oriented closed systems. Two challenges need therefore to be tackled: (i) collecting, storing, and making available data provided by different stakeholders and systems in a near real-time and in a standardized way, also referred to as the *publication* step in this paper; (ii) retrieving and sending back data as fast as possible whenever an end-user requests for them, also referred to as the *consumption* step. Both contribute to ensure optimal data quality, particularly in terms of availability and freshness.

To address these challenges, this paper proposes a distributed access time improvement framework for optimizing the publication and the consumption steps, including the storage and retrieval of data at the IoT gateways level. To reach near real-time performance, the framework relies on a consistent hashing mechanism [2] and a distributed storage system. It speeds up access to a batch of data and can also be considered a load balancing system when the number of users increases. We exploit the modular characteristics of microservices [3] to implement consistent hashing, as they provide an independent environment to run the application code. With microservices, a single application can be developed as a suite of small autonomous services, each running on its own domain. Our framework is assessed on a real city pilot use case of the ongoing bIoTope[1] H2020 project. The core component of bIoTope is the IoT gateway, referred to as O-MI node[2], relying on the Open Messaging Interface (O-MI) [4] and Open Data Format (O-DF) [5] standards. This study allows us to (i) show the improvement in terms of data access time at the gateways level, and (ii) draw conclusions on the impact of our framework on the access time.

The rest of the article is organized as follows. Section II explains the related work on IoT gateways, data management systems, and microservices architecture. Section III proposes a distributed framework for optimizing data storage and retrieval. In Section IV, our proposed framework is assessed on the Brussels city use case along with the experimental results. Finally, Section V concludes this paper.

II. Related Work

Existing approaches and tools can be found in literature concerning IoT applications, data management, and the design of IoT gateways. Many standard organizations (e.g. ETSI, W3C, 3GPP, or IETF) provide several specifications for the

[1]Building an IoT OPen innovation Ecosystem for connected smart objects
[2]O-MI node implemented by Aalto University: https://github.com/AaltoAsia/O-MI, accessed in December 2018

978-1-5386-9151-9/19/$31.00 ©2019 IEEE

IoT architecture, including service requirements, functional design, communication interfaces, and data standardization. These specifications can be adopted at the gateway level to offer interoperable and scalable solutions.

A. IoT Gateways

In many IoT systems and applications, the gateway provides an interface between backbone and sensor networks, and acts as a communication layer through which the data are being published and consumed. Chen et al. [6] mention the IoT gateway as a proxy for the sensing and network domains to communicate efficiently with other systems. The Open Group consortium has defined universal messaging standards, namely O-MI [4] and O-DF [5] (previously known as Quantum Lifecycle Messaging QLM [7]) to provide peer-to-peer communication and real-time interaction between heterogeneous systems. These standards reside independently at the communication and format levels of the OSI Application layer, thus overcoming horizontal interoperability while operating as an IoT gateway. O-MI provides a generic open API for transporting data payloads in nearly any format. The complementary O-DF standard is currently the most common text-based payload format due to its flexibility [8]. O-DF is defined as a simple ontology, specified using XML schema, which is generic enough for representing *any object* in the IoT. The four basic operations for sending O-MI requests are read, write, cancel, and subscribe (a specific read operation) [8] [9]. Since the gateway is one of the essential components in IoT, Morabito et al. [10] propose a LEGIoT gateway architecture. It optimizes resource management, improves latency, and offers an interoperable solution by relying on container-based virtualization. Similarly, the Agile[3] project has implemented an adaptive IoT gateway, which supports protocol interoperability, cloud communication, and device and data management. In addition, Kang et al. [11] propose a self-configurable gateway featuring dynamic discovery, real-time detection, and configuration of smart devices over the wireless networks.

B. IoT Data Management

Since a large number of devices is connected to the gateways, the data generated from them are vast in volume, which forces IoT to adopt flexible approaches for IoT data management. Li et al. [12] propose IoTMDB, a centralized data storage management system, based on NoSQL database to store massive IoT data. Similarly, an open-source NoSQL distributed database called Apache Cassandra[4] is designed to handle enormous amounts of structured, semi-structured, and unstructured data. It applies a consistent hashing mechanism to distribute data across a cluster. Although Cassandra provides frequent read and write capabilities, there is no functionality of data subscription for automatic data updates, which is central to other mechanisms, e.g. O-MI and O-DF. Thus, Cassandra is not a directly suitable solution. Another data storage framework is implemented by Jiang et al. [13] to manage unstructured and structured IoT data, exploiting Hadoop and other databases (NoSQL and relational). Fazio et al. [14] propose a Cloud storage solution to optimize data storage, querying and retrieval for huge amounts of heterogeneous data. Although this solution manages massive IoT data, it may suffer from bandwidth overhead and high latency to retrieve immediate data for processing. Furthermore, Tian et al. [15] propose a load-balancing algorithm (DAIRS) for cloud data centers, which ensures the allocation of both physical and virtual servers by considering CPU, memory, and network bandwidth. Similarly, another load-balancing algorithm called VectorDot is implemented by Singh et al. [16] to manage the hierarchical and multi-dimensional load distribution in agile data centers. However, there is no functionality of balancing standardized data within the servers.

Additionally, one of the solutions for addressing big data challenges in various IoT areas is employing microservices. Recent literature claims that the unified (monolithic) framework for IoT applications is an unrealistic approach; instead, microservices are the natural fit for IoT development [3]. A microservice[5] is a small, cohesive, and autonomous service that deploys independently and interacts using lightweight messaging protocols. Microservices architecture is an architectural style enabling the concept of modular independence to structure an application as a collection of loosely coupled services [3]. In this context, Krylovskiy et al. [17] exploit microservices architecture to propose a decentralized data management approach for smart cities. Vresk and Cavrak [18] propose a microservices-based middleware to connect different devices, services, and protocols ensuring system scalability and interoperability. Microservices can also be applied to other e.g. location- and context-based applications [19].

III. Data Publication and Consumption Framework

In recent IoT literature, data management is mainly performed at the cloud level without any standardized computing technologies. However, to the best of our knowledge, this work is the first to optimize the storage and retrieval of standardized IoT data by leveraging state-of-the-art computing technologies at the gateways level. The proposed framework tends to improve access time by relying on a consistent hashing mechanism, which is implemented in the form of wrappers. This new distributed framework is designed to overcome the problems with collecting, storing, and accessing a large volume of data in near real-time by offering a generic mechanism mainly running at the IoT gateways level. Fig. 1 provides a high-level overview of the proposed framework. Data providers grant access to their data often by using proprietary APIs. Wrappers enable publishing these data on an IoT gateway in a standardized way. To optimize the storage, data are assigned and distributed to multiple node/gateway instances (as well as the retrieval of these data) with the

[3] http://agile-iot.eu/about/, accessed in September 2018

[4] http://cassandra.apache.org/doc/latest/, accessed in September 2018

[5] https://www.martinfowler.com/articles/microservices.html, accessed in September 2018

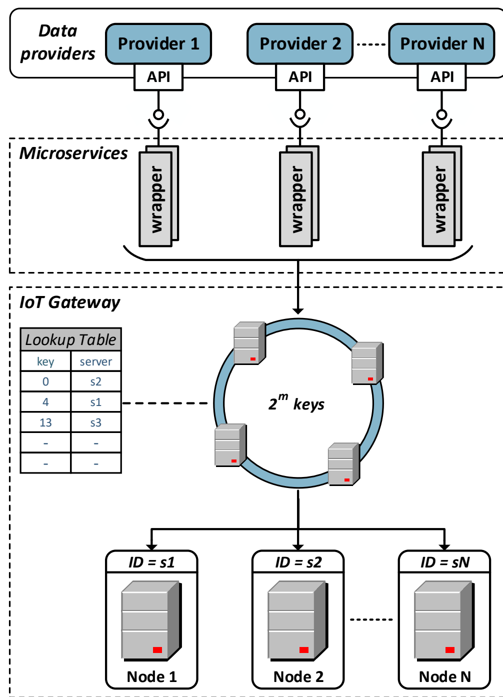


Fig. 1: High-level architecture of our framework, where *m* defines the size of the hashing mechanism

consistent hashing mechanism, as depicted in Fig. 1 with a hashing circle and lookup table. Section III-A presents the methodology for data translation in a standardized format and ways in which these data are then stored and later retrieved on the IoT gateway. In addition, Section III-B describes an implementation viewpoint.

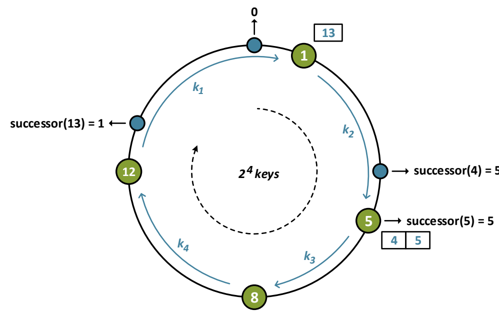
A. Data Storage Management with Consistent Hashing

In the proposed framework, raw data need to be processed and converted into standardized API format in real-time. Data standardization is essential as it enables us to make the data consistent, ensuring that the related data can easily be understood, identified, and managed through common terminology and format [20]. In data translation, the wrapper (containing multiple microservices) acquires data items from various data sources in raw format such as JSON, YAML, or XML. It then parses the data, converts them into standardized format (e.g. O-DF, JSON-LD, or RDF), and stores the translated data with distributed storage system. Our mechanism for data management is consistent hashing as it enables us to evenly distribute data items onto multiple servers with a design that is similar to Apache Cassandra. However, unlike Cassandra, our design supports microservices and offers publish/subscribe messaging. Consistent hashing tends to balance the node loads, since each node receives roughly the same number of keys and requires a minimal amount of data location changes. In this mechanism, each server node is assigned to administer several unique key ranges. Given a sufficient number (10+) of key ranges per database server guarantees near optimal load balancing behavior [2]. Unlike other load balancing mechanisms such as Round Robin [21], consistent hashing is tailored for higher accuracy as the addition and removal of servers are rather dynamic with minimal data changes.

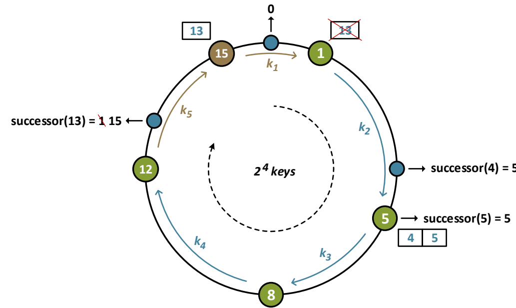
Consistent hashing provides a key space to map keys (e.g. data or server key) that can be drawn as an *identifier circle* in which the mapping starts in a clockwise manner. Data and server IDs are mapped to the same key space. If there are K keys mapped to N servers, adding or removing a server will only have a $\frac{K}{N}$ number of changes [2]. The mapping is performed by using the hash function (e.g. SHA-2), which assigns an *m-bit* key identifier to each ID. The m must be large enough so that no two IDs hash to the same key identifier. Furthermore, keys are ordered in an *identifier circle* using modulo 2^m. Key *k* is assigned to the first server in a clockwise manner whose identifier is equal to *k*. If the server with key *k* is unavailable, the next server key ahead of key *k* is used. Let us cite an example of a hashing circle shown in Fig. 2 where the key space has 16 keys ($m = 4$) with a range from 0 to 15. As can be seen in Fig. 2(a), the circle has four servers that are mapped to keys $1, 5, 8$, and 12 (shown in small green circles). In addition, each server is responsible for at least one key range (out of four key ranges k_1, k_2, k_3, and k_4). The blue circles represent the data items that are mapped to the corresponding keys. In the mapping, the successor of key identifier 4 is 5 as the key belongs to the key range k_2, which maps to the server with key 5. Similarly, the successor of key 5 is 5, hence the data reside at server key 5. From the implementation viewpoint, these key ranges are stored in a lookup table where each row specifies the mapping of the key identifier to the corresponding server node. Moreover, Fig. 2(b) demonstrates a scenario where another server with key 15 joins the system. In this case, some of the keys (in k_1) are now assigned to server 15 (with key range k_5).

B. Wrap-up: An Implementation Viewpoint

This section presents the proposed algorithms implemented in our framework. **Algorithm 1** provides a mechanism for data translation, mapping, and distribution to multiple server instances. It takes *data-id* as an input and produces a *lookupTable* in which the mapping of data hash keys with the server IDs are stored. To begin, the *lookupTable* is initialized with the number of servers and the corresponding random SHA-2 keys. For instance, if there are three servers in the framework, three hash keys are generated and assigned to these servers. Once the keys have been mapped, the lookup table is sorted from the smallest to the largest keys. The algorithm proceeds by generating the hash key of the *data-id*, finding the server in the lookup table, and writing the data item on the corresponding server. In addition, the table is updated and sorted with every new key entry. We make the lookup table persistent using SQLite database management system. Similarly, **Algorithm 2** enables us to read data items stored in the servers. It takes *data-id* and *lookupTable* as inputs and returns *standardized-data* as an output. This algorithm first calculates SHA-2 of the *data-id*, it then searches the lookup



(a) An example scenario with four servers



(b) A scenario when another server joins the circle

Fig. 2: An identifier circle with m = 4; $k_1 \ldots k_5$ describes the key ranges managed by servers 1, 5, 8, 12, and 15

Algorithm 1: Store standardized data on servers

```
Function store.dataItem()
    Input: data-id
    Output: lookupTable[hash,server]
    hash ← SHA-2(data-id)
    foreach element k of the lookupTable[hash] do
        hash′ ← lookupTable[k]
        if hash ≤ hash′ then
            server ← lookupTable(hash′)
            break
        else
            server ← lookupTable(min(hash′))
            break
        end
    end
    standardized-data ← dataItem(data-id)
    write(server, standardized-data)
    lookupTable ← [hash,server]
```

table to locate the server where the data are stored, and finally returns the requested data back to the user.

IV. Case Study: An access time analysis

The data publication and consumption capabilities of our proposed framework are applied to a real use case, which relates to Brussels city with its smart mobility scenarios in the ongoing bIoTope H2020 project. The first scenario addresses safer mobility of children traveling to and from school in the Belgian capital region. The second scenario highlights smart mobility for waterbus service boats. The third scenario describes smart parking guidance based on the availability of parking places, occurrence of road congestions, or planned

Algorithm 2: Read standardized data from servers

```
Function read.dataItem()
    Input: data-id, lookupTable[hash,server]
    Output: standardized-data
    hash ← SHA-2(data-id)
    if hash ∈ lookupTable then
        server ← lookupTable(hash)
        data-item ← read(server, data-id)
        return data-item
    else
        data-item ∉ server
    end
```

events in the city. In all the aforementioned scenarios, the objective is to publish road traffic data through an O-MI node[6], *an IoT gateway implementing the core bIoTope standards i.e. O-MI and O-DF*, to make timely decisions based on these data. In the Brussels city pilot, two platforms provide data through their own proprietary APIs. First, Waze[7] is a GPS navigation platform that provides real-time traffic alerts and turn-by-turn navigation information. Through Waze, users can report traffic updates such as route details, accidents and traffic jams. Second, UrbIS[8] comprises a set of cartographic and alphanumeric data for the administration of the capital region including equipment, infrastructure, and mobility issues.

A. Wrapper Implementation

The wrapper enabling data translation, storage, and retrieval relies on the consistent hashing mechanism with microservices. Our implementation is based on three microservices functions (written in Java) as shown in Fig. 3.

- *Function 1 – Write:* This function retrieves 22,780 road segments from Brussels mobility API. Each segment has an associated segment ID and many other key-value attributes. Based on these IDs, the road segments are converted into a standardized O-DF format and mapped to the O-DF hierarchy, *where each segment is considered an O-DF object*, as depicted in Fig. 3. Besides, the objects are distributed and stored on the O-MI node instances using the hashing mechanism, as described in Section III-A. In our implementation, data are categorized into groups and hashing is applied to a group level (with group ID) instead of individual data segments, thus placing adjacent data segments closer to each other.
- *Function 2 – Update:* This function updates the data segments onto O-DF objects hierarchy (already created by Function 1) with real-time traffic data collected from the Waze API (≈ 700 segments). Data segments are updated inside O-MI nodes based on their IDs.
- *Function 3 – Read:* This function responds to the user request for reading road segments. It searches the lookup table to locate the O-MI node (where segments are stored) and returns the O-DF data back to the users.

[6]https://github.com/AaltoAsia/O-MI, accessed in September 2018
[7]https://www.waze.com/, accessed in September 2018
[8]UrbIS solutions: https://bric.brussels/en/our-solutions/urbis-solutions, accessed in September 2018

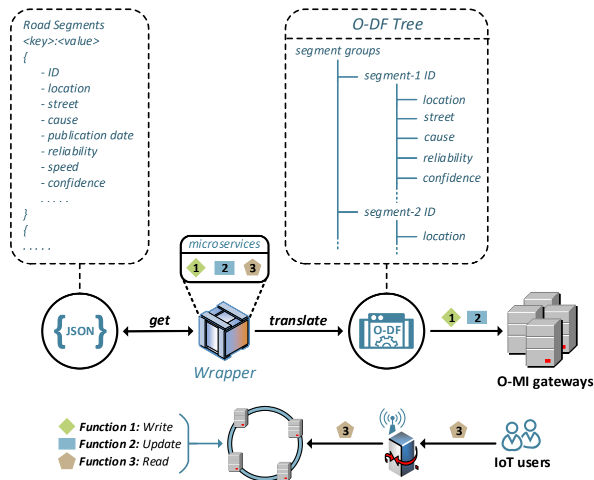


Fig. 3: High-level overview of wrapper implementation

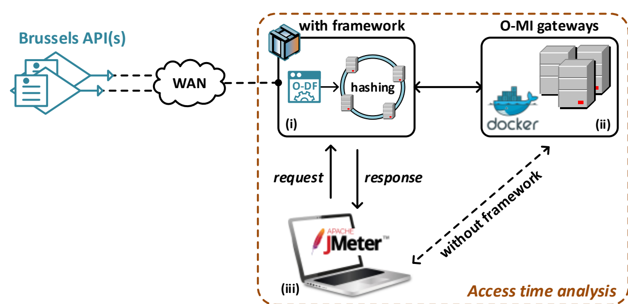


Fig. 4: Experimental setup for performance analysis

B. Performance in terms of Access Time

In order to evaluate the performance of our framework, particularly in terms of data access time, we consider the Brussels smart city use case. The experimental setup, depicted in Fig. 4, consists of three physical machines: (i) The proposed framework runs on an HP EliteBook Windows laptop with Intel Core i5 2.40GHz CPU and 8GB RAM; (ii) O-MI gateways are executed as Docker containers (v17.12) on a separate Linux machine with 16GB RAM and Intel Xeon(R) 3.20GHz 8-cores processor; (iii) The open-source software Apache JMeter[9] (v4.0) runs on another HP EliteBook Windows laptop with a memory of 8GB and Intel Core i5 2.40GHz CPU, and enables us to simulate heavy load on the O-MI gateway.

We select two cases for performance assessment. **(i) Read access time:** The time taken by the gateway to respond to a user's read request. Concurrent users send O-DF requests for reading random data items and in turn, receive Brussels road segments in the O-DF format. **(ii) Write access time:** The time taken by the gateway to respond to a user's write request. Concurrent users send O-DF write requests for updating and storing road segment data. The comparison considering the two aforementioned cases is performed between *without framework* and *with framework* setup. *Without framework* is based on O-MI reference implementation[6], which is implemented in Java and Scala, and consists of three core components: API endpoint, agent system, and the user interface (web client). API endpoint manages user requests and the database. The agent system has multiple agents which are integrated with the API endpoint to pull/push sensor data from and to the embedded version of Warp10[10] database. The user interface is a front-end service for creating read or write requests in O-DF format. On the other hand, *with framework* setup defines our consistent hashing implementation with microservices. Unlike *without framework* setup, the proposed solution allows us to distribute data onto multiple gateways. We can also use more than one gateway in *without framework* setup. However, instead of data distribution to multiple nodes, it will be considered data replication (same data in every O-MI node) which is irrelevant to this case study. Thus, *without framework* is only analyzed with single O-MI gateway.

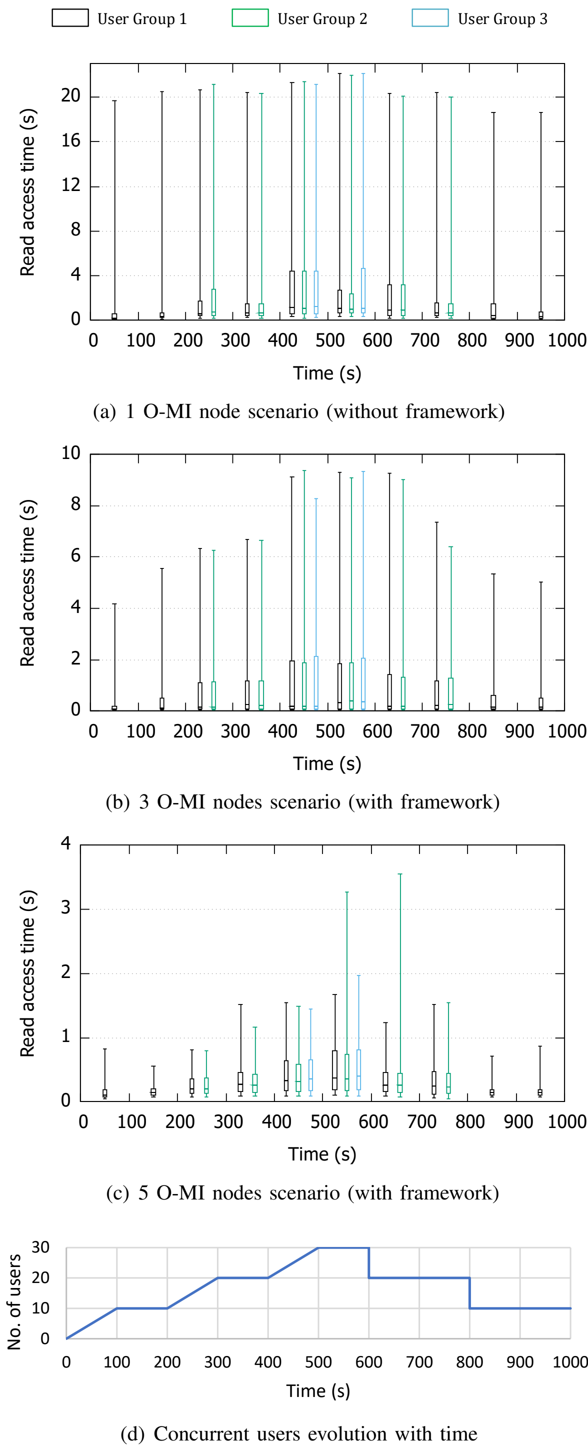
Experimental results for the read and write access time are illustrated in Fig. 5 and Fig. 6, respectively. These graphs represent a boxplot providing minimum, 10th percentile, median, 90th percentile, and maximum of the data access time for three different user groups calculated over each 100-second period. The evolution of users with time is plotted in Fig. 5(d). As can be seen, the number of users increases gradually (in a group of 10 users) up to 30 concurrent users (at t=500s) sending random requests. It then begins to decrease with a ratio of 10-by-10 until the end of the experiment (t=1000s). Fig. 5(a), Fig. 5(b), and Fig. 5(c), respectively, provide an aggregated view of the read access times for three scenarios: (i) 1 O-MI node; (ii) 3 O-MI nodes with our framework; (iii) 5 O-MI nodes with our framework. Similarly, Fig. 6(a), Fig. 6(b), and Fig. 6(c) provide an aggregated view of the write access times for the same scenarios. Each scenario is repeated three times and the observed access times do not significantly change.

The following conclusions can therefore be drawn:

1) *Read access time evolution:* As seen in Fig. 5, when the load is relatively low [0:200s], the access time is also low (less than $1s$). When the load increases [200:600s], the access time increases accordingly to around $4s$, $2s$, $1s$ in our respective scenarios. Finally, when the load decreases [600:1000s], the framework progressively adapts itself and the read access time decreases. Overall, even if in most cases (90%) the access times are still acceptable, it can be seen that our framework enables to minimize the worst case (when the load is important).
2) *Write access time evolution:* As seen in Fig. 6, when the load is comparatively low [0:200s], the access times are almost the same for all the three scenarios. However, there is a significant decrease in the maximum value of write access time from $34s$ in Fig. 6(a) to around $3s$ in Fig. 6(c). When the load increases [200:600s], the maximum value decreases accordingly to around $36s$, $18s$, $4s$ in our respective scenarios. In case of a single O-MI node, the load cannot be distributed and all the concurrent requests are handled by one gateway, which allows to

[9]http://jmeter.apache.org/, accessed in September 2018

[10]http://www.warp10.io/introduction/platform/, accessed in September 2018



(a) 1 O-MI node scenario (without framework)

(b) 3 O-MI nodes scenario (with framework)

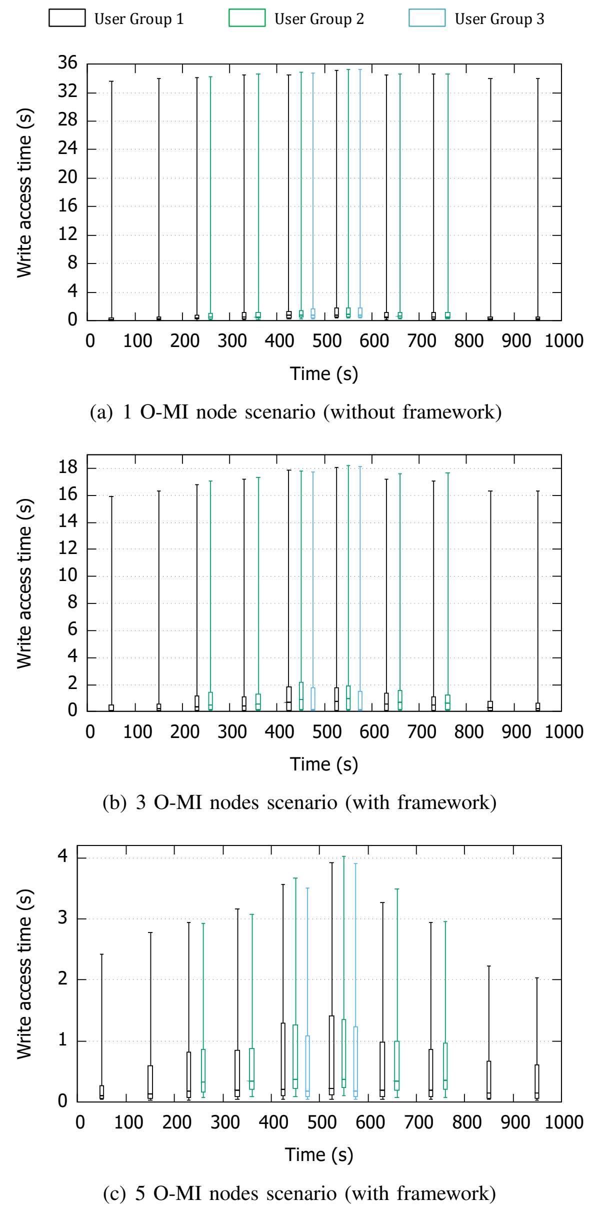
(c) 5 O-MI nodes scenario (with framework)

(d) Concurrent users evolution with time

Fig. 5: Impact on read access time

incorporate a significant amount of delay for performing successful data write. Finally, when the load decreases, the write access time also decreases with it.

3) *Number of nodes:* Compared to Fig. 5(a), Figures 5(b) and 5(c) show that adopting our framework enables decreasing read access time, even when the load is relatively high, i.e. all the concurrent users are active [400:600s].



(a) 1 O-MI node scenario (without framework)

(b) 3 O-MI nodes scenario (with framework)

(c) 5 O-MI nodes scenario (with framework)

Fig. 6: Impact on write access time

The case is the same for write access time in Fig. 6(b) and Fig. 6(c), as the load is distributed to multiple O-MI gateways. Overall, adding more nodes through our framework improves the performance as compared to a single O-MI node, thus reducing the access time.

In addition to the previous cases that consider only the read and write requests of a random data item with concurrent users, TABLE I shows the access time for three different requests. (i) ReadAll (response size: 41 Mbytes): an O-DF request is sent to read all $22,780$ road segments. (ii) Read "single" random object (response size: 85 Kbytes): a request is sent to retrieve single random road segment. (iii) Read "large" random object (response size: 422 Kbytes): a request is sent to read several road segments from multiple O-MI nodes. The

TABLE I: Access time measurements (in seconds) with and without our framework for three read requests

Request	Without Framework			With Framework					
				3 nodes			5 nodes		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
ReadAll	*	*	*	21.33	28.98	**22.89**	21.10	22	**21.51**
Read "single" random object	0.39	22.13	**1.93**	0.05	9.33	**0.75**	0.09	1.97	**0.45**
Read "large" random object	1.33	11.67	**1.66**	0.50	3.41	**0.68**	0.41	3.74	**0.55**

* *timeout*

measurements in TABLE I are taken from a period when all concurrent users were active [400:600s]. Overall, even if there is no significant difference in read access times in requests (ii) and (iii), they allow confirming the previous results. However, it is important to note that our framework can significantly improve access to all the data segments (request (i)). In case of a single O-MI node, the request is timed out (*timeout = 40s*) and the node may even become unresponsive.

V. Conclusion and Future Work

This paper proposes a distributed data publication and consumption framework to improve access time for standardized IoT gateways. It optimizes the storage and retrieval of data at the IoT gateway level and tackles the following challenges: (i) collecting, distributing, and storing data provided by different stakeholders in a near real-time and in a standardized way; (ii) retrieving and sending back data as fast as possible whenever an end-user requests for them. This new distributed framework is implemented by employing microservices, which consists of two parts: (i) It translates the raw data into a standardized API format; (ii) It relies on a consistent hashing mechanism to distribute and store real-time data onto multiple servers, thereby improving access time to a batch of data. We assessed our framework by applying it to the Brussels city use case of the ongoing bIoTope project, in which the core component is the IoT gateway, developed through the adopted messaging standards, i.e., O-MI and O-DF. The experimental results conclude that the proposed framework improves data access time and ensures a scalable solution. Our future work will address the integration of our proposed framework with the O-MI security module[11], as it provides user authentication and authorization to securely access data items. Further, the framework will be enhanced to offer fault tolerance capability in such times when either the gateways or the distributed storage mechanism fail to respond.

Acknowledgements

This research is supported by the EUs Horizon 2020 research and innovation program (grant 688203) and Academy of Finland (Open Messaging Interface; grant 296096, APTV; grant 277522, and SINGPRO; grant 313469). The authors would like to thank CIRB Brussels for providing a real use case based on real data.

[11]O-MI security module developed by Aalto University: https://github.com/AaltoAsia/O-MI-Security-Model, accessed in September 2018

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] I. Stoica, R. T. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[3] S. Newman, *Building Microservices - Designing Fine-grained Systems, 1st Edition*. O'Reilly, 2015, ISBN: 9781491950357.

[4] "Open Messaging Interface (O-MI), an Open Group IoT Standard," http://www.opengroup.org/iot/omi/, accessed September 2018.

[5] "Open Data Format (O-DF), an Open Group IoT Standard," http://www.opengroup.org/iot/odf/, accessed September 2018.

[6] H. Chen, X. Jia, and H. Li, "A Brief Introduction to IoT Gateway," in *Communication Technology and Application (ICCTA 2011), IET International Conference on*. IET, 2011, pp. 610–613.

[7] S. Kubler, K. Främling, and W. Derigent, "P2P Data Synchronization for Product Lifecycle Management," *Computers in Industry*, vol. 66, pp. 82–98, 2015.

[8] J. Robert, S. Kubler, Y. L. Traon, and K. Främling, "O-MI/O-DF Standards as Interoperability Enablers for Industrial Internet: A Performance Analysis," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, Florence, Italy, October 23-26, 2016*, 2016, pp. 4908–4915.

[9] M. Madhikermi, N. Yousefnezhad, and K. Främling, "Data Exchange Standard for Industrial Internet of Things," (in press) *in 3rd International Conference on System Reliability and Safety, ICSRS 2018, Barcelona, Spain*, 2018.

[10] R. Morabito, R. Petrolo, V. Loscrì, and N. Mitton, "LEGIoT: A Lightweight Edge Gateway for the Internet of Things," *Future Generation Comp. Syst.*, vol. 81, pp. 1–15, 2018.

[11] B. Kang, D. Kim, and H. Choo, "Internet of Everything: A Large-Scale Autonomic IoT Gateway," *IEEE Trans. Multi-Scale Computing Systems*, vol. 3, no. 3, pp. 206–214, 2017.

[12] T. Li, Y. Liu, Y. Tian, S. Shen, and W. Mao, "A Storage Solution for Massive IoT Data Based on NoSQL," in *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*. IEEE, 2012, pp. 50–57.

[13] L. Jiang, L. D. Xu, H. Cai, Z. Jiang, F. Bu, and B. Xu, "An IoT-Oriented Data Storage Framework in Cloud Computing Platform," *IEEE Trans. Industrial Informatics*, vol. 10, no. 2, pp. 1443–1451, 2014.

[14] M. Fazio, A. Celesti, A. Puliafito, and M. Villari, "Big Data Storage in the Cloud for Smart Environment Monitoring," in *Proceedings of the 6th International Conference on Ambient Systems, Networks and Technologies (ANT 2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015), London, UK, June 2-5, 2015*. Elsevier, 2015, pp. 500–506.

[15] W. Tian, Y. Zhao, Y. Zhong, M. Xu, and C. Jing, "A Dynamic and Integrated Load-balancing Scheduling Algorithm for Cloud Datacenters," in *IEEE International Conference on Cloud Computing and Intelligence Systems, CCIS, Beijing, China*, 2011, pp. 311–315.

[16] A. Singh, M. R. Korupolu, and D. Mohapatra, "Server-Storage Virtualization: Integration and Load Balancing in Data Centers," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, Texas, USA*, 2008, p. 53.

[17] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a Smart City Internet of Things Platform with Microservice Architecture," in *3rd International Conference on Future Internet of Things and Cloud, FiCloud 2015, Rome, Italy, August 24-26*, 2015, pp. 25–30.

[18] T. Vresk and I. Cavrak, "Architecture of an Interoperable IoT Platform Based on Microservices," in *39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016, Opatija, Croatia, May 30 - June 3*, 2016, pp. 1196–1201.

[19] P. Bak, R. Melamed, D. Moshkovich, Y. Nardi, H. J. Ship, and A. Yaeli, "Location and Context-Based Microservices for Mobile and Internet of Things Workloads," in *2015 IEEE International Conference on Mobile Services, MS 2015, New York, USA, June 27 - July 2*, 2015, pp. 1–8.

[20] L. Vallad, "Oracle Data Quality for Product Data Knowledge Studio Reference Guide, Release 5.6.2," 2011.

[21] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic Load Balancing on Web-Server Systems," *IEEE Internet Computing*, vol. 3, no. 3, pp. 28–39, 1999.