applied sciences

MDPI

Article

Secure Data Distribution Architecture in IoT Using MQTT

Farag Azzedin [1,*] and Turki Alhazmi [2]

[1] Information & Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[2] Interdisciplinary Research Center for Intelligent Secure Systems, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
* Correspondence: fazzedin@kfupm.edu.sa

Abstract: Message Queuing Telemetry Transport (MQTT) is one of the standard application layer protocols for the Internet of Things. It uses a publish/subscribe mechanism which organizes a set of clients around a server called the broker, which delivers published data to its intended recipients. This article proposes an architecture that allows MQTT brokers to cooperate and share their data with other interested MQTT brokers. It is a service-oriented architecture that wraps an MQTT broker with a well defined WebSockets-based interface which allows it to offer its topic space and published data to other MQTT brokers. The wrapped MQTT broker is called a broker service, and it discovers other broker services through a discovery service. Each broker service only connects to services that have data its clients are interested. Furthermore, these services are authenticated by obtaining tokens from an authentication service that registers and issues JSON Web Tokens for them. These tokens contain the identity and claims of their owners and they can be verified without contacting the authentication service. The proposed architecture simplifies data sharing and improves the security in scenarios with multiple MQTT brokers where clients can move between them. In these scenarios, the MQTT brokers need to obtain data based on their clients interests, which are constantly changing. It does so by isolating MQTT brokers into services that can be discovered and consumed over well-defined interfaces. The architecture was implemented in javascript using MQTT 3.1.1 standard complaint library. We demonstrate the performance characteristics of our architecture using our implementation through three scenarios, which are designed to compare the delay from publisher to subscriber when they operate within the same MQTT broker and different MQTT brokers. The results show that the overhead of our architecture is around 50% in two synthetic scenarios (performed on a single machine) and around 27% in a third scenario performed on the cloud with multiple virtual machines hosting the broker services and simulated clients.

Keywords: IoT; MQTT; security; brokers

Citation: Alhazmi, T.; Azzedin, F. Secure Data Distribution Architecture in IoT Using MQTT. *Appl. Sci.* **2023**, *13*, 2515. https://doi.org/10.3390/app13042515

Academic Editors: Howon Kim and Thi-Thu-Huong Le

Received: 12 November 2022
Revised: 13 January 2023
Accepted: 14 January 2023
Published: 15 February 2023

1. Introduction

Message Queuing Telemetry Transport (MQTT) is a standard application layer protocol for Internet of Things (IoT) ecosystems [1], organizing a set of clients around a central broker that distributes messages between clients. It is based on the publish/subscribe paradigm where clients can be publishers or subscribers of topics. A topic can be viewed as a channel that clients listen or push data through. When data are put on the channel (published) by a client, all the clients listening (subscribing) on that channel receive the data. The MQTT broker keeps track of all the publishers and subscribers and handles all the details of delivering data between the clients end to end.

MQTT brokers were originally designed to work independently, where information published under a certain topic does not propagate across brokers [2]. However, broker cooperation is essential for building scalable systems based on MQTT, because as the number of clients increases beyond the capabilities of a single broker, more brokers could be introduced to handle a portion of the clients. Hence, a mechanism for cooperation is

needed that allows a set of brokers to cooperate and share data seamlessly. This is performed by popular implementations of the standard support broker cooperation through bridging which allow a pair of brokers to share all or part of their topic space (e.g., [3]). The idea of bridging has been used in the literature to build different types of architectures [4] increasing client anonymity and scalability [5].

However, bridging works by making one broker a client of the other, which places a significant burden on the latter as more brokers bridge to it. This centralizes all cross broker communication through that broker making it a single point of failure while limiting scalability in the process. Another issue associated with broker cooperation is broker discovery. It is a more fundamental issue than cooperation, because brokers need to know one another before they can cooperate. To the best of our knowledge, other architectures have tackled this problem through peer-to-peer discovery as in [5], while other architectures assume the brokers are known or a fixed topology for the network as in [6]. Therefore, we propose a new Service-Oriented Architecture (SOA) that attempts to address the problems of broker discovery and cooperation.

It uses a discovery service for broker, topic and interest tracking and peer-to-peer communication between brokers to disseminate data. Hence, new brokers communicate with the discovery service and negotiate participating in the network of brokers. Furthermore, participating brokers also provide the service of exposing their internal publications and metadata to potential consumers through a standard interface. Thus, the brokers consume the discovery service by sending updates about their interests dynamically by aggregating the interests of their local clients. Furthermore, the brokers consume the services provided by other brokers to obtain data they are interested in. For example, a certain client subscribes to topic A within a given broker, then the broker sends this information to the discovery service, which in turn notifies all brokers that manage publishers of topic A. This has the following benefits:

1. Transparency to clients: All the details of how data are collected and sent is hidden from the clients. In other words, the client does not distinguish between local and global data. This allows the client to move between brokers while maintaining access to their data.
2. Unified global access: Clients can instantly subscribe or publish new topics globally across all brokers without having to do any extra work.
3. Scalability: SOA ensures that individual services can be scaled individually without affecting the rest of the system.
4. Interoperability: Communication between the services is performed using HTTP WebSockets, which maximizes interoperability with many existing webservices and web technologies.
5. Improved security: The brokers are isolated into services which provide their internal data over well-defined interfaces that are easier to secure and maintain.

The rest of the article is organized as follows. Section 2 details the problem statement and related work while Section 3 presents the proposed architecture. Section 4 discusses the details of our implementation, experimental setup, and test scenarios. The evaluation results of the proposed architecture are presented in Section 5. The section also discusses the results and draws some observations. Finally, Section 6 concludes and envisions future directions.

2. Problem Statement and Related Works

The aim of this article is to build an architecture that tackles the issues of cross broker data discovery and dissemination. These problems need to be addressed because without proper data discovery and dissemination MQTT brokers cannot be scaled properly. Therefore, many architectures were proposed in the literature to tackle these two problems. These can be abstractly categorized architectures that require client involvement, those with static topologies and fully peer-to-peer ones. In fixed topology architectures, assumptions about the structure and behavior of the network are made to increase the efficiency of resource

utilization [7]. However, such architectures are static by design, therefore, it is difficult to add/remove brokers or migrate clients across brokers at runtime. Kawaguchi et al. [8] proposed an MQTT broker architecture for the edge that improves availability while reducing the amount of broadcasting that is performed by the broker. It mainly uses a topic structure that is optimised for geographically distributed IoT applications. Furthermore, Jutadhamakorn et al. [9] explored the use of virtualization technologies to build highly scalable distributed broker clusters. Similarly, Sen et al. [10] proposed a container-based architecture that is designed to scale through decoupling the state of a given broker from its functionality.

In architectures that require client involvement, the client is the one responsible for discovering and then connecting to the broker that has the data it needs. The discovery process starts with the client asking its broker about where a certain resource exists in the network. Subsequently, the broker discovers where that resource is either in a peer-to-peer fashion or through a dedicated discovery entity. Upon discovery, the broker redirects the client to that broker which has the resource. Finally, the client connects the broker with the resource as a client to consume that resource. The architecture is illustrated in Figure 1. Park et al. proposed in [11] that creates multi-cast groups for each topic using an SDN controller which disseminates topic data between brokers through interest information obtained from an assumed root broker. Furthermore, the authors in [12] proposed a Quality of Service (QoS) aware distributed architecture based on MQTT, which is aimed at migrating clients between brokers to enhance performance and achieve desired QoS. Banno et al. [13] proposed an approach for handling the distribution of data at the edge with heterogeneous brokers using an inter-networking layer. This inter-networking layer sits between the end clients and their local brokers at the edge. The authors in [14] proposed a topic management system for MQTT to facilitate the sharing of open data in the context of smart cities. It uses the hierarchical topic model that allows publishers and subscribers to navigate from more generic data to more specific data using a URL-like structure. For example, the topic "earthquake/west/district9", would mean that the user is interested in earthquake data coming from District 9 in the west. Furthermore, the subscriber could also subscribe to the more generic topic "earthquake/west" to get earthquake data for all districts in the west.

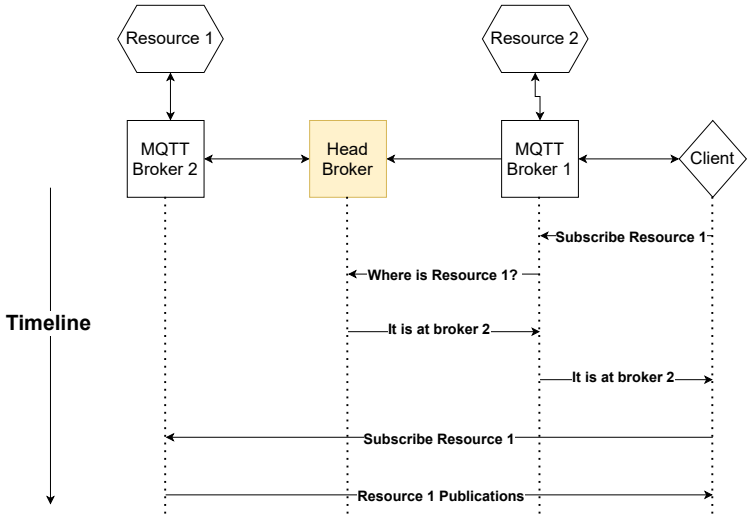

Figure 1. Illustration of architectures where the client is involved in the discovery process.

The main advantage of this type of architecture is scalability and fault tolerance, because of the following reasons:

1. Clients directly connect to the brokers that have the resources they need.
2. By design, these architectures implement constructs for client migration which allows for dynamic load balancing and fault tolerance among brokers producing the same resource.

However, this type of architecture has the following disadvantages:

1. The clients have to connect to multiple brokers if the resources they need are scattered. This depletes their resources quickly, especially if they are constrained devices.
2. The clients are less secure because they are exposed to several brokers.
3. Inefficient utilization of network resources, because when multiple clients within the same broker are interested in the same resource each client establishes a separate connection with the broker producing the resource they want.

In a fully peer-to-peer architecture, the brokers dynamically organize themselves in a topology according to a distributed algorithm running on all of them [15]. Subsequently, data discovery and dissemination are performed in a cooperative manner, where brokers route other brokers messages from source to destination. Their peer-to-peer nature is a good indication of how well they scale; however, it is still heavily influenced by their final topology which also affects other aspects such as fault tolerance and security [16]. Another interesting aspect of these architectures is that the clients only need to connect to one broker to get resources produced across all brokers in the network. This overcomes most of the limitations of the previous type. However, there is still an issue of latency, where data might have to propagate through several brokers to reach its final destination. The architecture is illustrated in Figure 2. Longo et al. [5] presented another architecture that aims to improve scalability by organizing the brokers in a spanning tree using the spanning tree protocol.

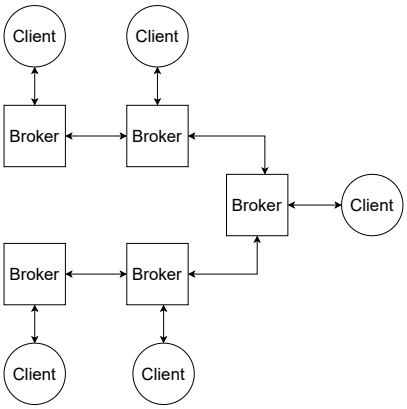

Figure 2. Illustration of fully peer-to-peer architectures.

Table 1 summarizes related works and compares them to our proposed architecture in terms of autonomy, client involvement and topology. Autonomy refers to the brokers ability to choose other brokers to communicate with. Client involvement refers to whether the clients are aware of the existence of multiple brokers. The topology refers to the brokers' arrangement and whether that can change during the runtime. Our proposed architecture improves upon existing architectures by allowing the brokers to be fully autonomous while hiding the complexity of obtaining data from multiple brokers from the clients. This leads to the broker's communicating in an ad hoc fashion to form a topology that best serves their clients at any given time.

Table 1. Related work summary and comparison with the proposed solution.

Reference	Autonomy	Client Involvement	Topology
[8–10]	No autonomy	No involvement	Fixed
[11–14]	Full autonomy	Involved	Dynamic
[5]	No autonomy	No involvement	Dynamic
Proposed architecture	Full autonomy	No involvement	Dynamic

3. Distributed MQTT Architecture

Distributed MQTT Architecture (DMQTTA) is an architecture that allows a set of MQTT brokers to cooperate and share their resources. It is modelled after SoA [17], where each broker provides a service that exposes resources of its domain. Furthermore, all the brokers in the system utilize a discovery service to aid them in the discovery of other brokers and resources. All the services in the system are exposed over well-defined interfaces which allow them to be easily discovered and consumed. The only assumption that is made about the system is that the discovery service is known by all the brokers in the system. There are several key advantages, as follows:

1. Scalability: Since DMQTTA is a service-oriented architecture, it inherits all the scalability characteristics associated with SoAs. One of those is being able to scale each service independently of the others because they are loosely coupled [18]. For example, clients of a single broker could be distributed across a set of cooperating brokers that share their resources.
2. Interoperability: Any service provider/consumer can connect and provide/consume services as long as they adhere to the interfaces of the network.
3. Security: Since the MQTT broker's domain is exposed over a well-defined interface, arbitrary security checks could be applied to consumers before allowing them access.

The architecture defines three main types of services, namely, discovery service, broker service and authentication service. We detail them in Sections 3.2, 3.3, and 3.4, respectively. We use a running example defined in Section 3.1 to facilitate our discussion.

3.1. Running Example

Suppose there are two brokers A and B and the discovery service D in the system. Furthermore, A is managing a publisher of topic T, while B is managing a subscriber of the same topic T. Therefore, we can say that A is publishing topic T and B is subscribing to topic T. This example is illustrated in Figure 3.

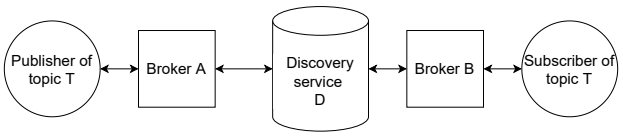

Figure 3. Running example illustration.

The goal is to have each broker aggregate the interests of their clients and ultimately propagate them to the other brokers in the system through the discovery service. Then, use consume other brokers services to obtain publications pertaining to those interests.

3.2. Discovery Service

The discovery service is an HTTP service that provides real-time updates to all its passed interests of consumers between them. The service uses topic names as grouping criteria for different brokers such that all brokers managing publishers of a certain topic always receive updates about new subscriptions of that topic. In our running example, the interactions with the discovery service are as follows:

1. A and B connect and authenticate to *discovery service*, where sockets are maintained at all times.
2. A advertises interest in publishing to topic T to the *discovery service*, which adds A to a group called T.
3. B advertises interest in subscribing to topic T to the *discovery service*.
4. Upon receiving the subscription request the *discovery service* pushes a notification of a new subscription to all the brokers in the group called T, which only includes A at the moment.
5. A can decide independently if it wants to disseminate data to B or not.

There are two interesting observations about this scheme. First, connections are maintained with the discovery service at all times, which adds the following benefits: (a) More efficient because it reduces the amount of data sent per transmission because it cuts most of the application layer headers and (b) Enables real-time bidirectional communication with all brokers. The second interesting aspect is that the subscribers are informed of new publishers not the other way around, which is performed because of the following reasons: (a) The number of publishers is usually less than the number of subscribers in a given system, (b) List of publishers is usually more stable than the list of subscribers in a given system, (c) Most of the time, subscriptions come after the publications which means we can present a subscriber with all available publishers and gradually stream new publishers to existing subscribers and (d) Give the subscribers more privacy, in that the subscribers can choose which publisher to connect to and consume its services. It should be noted that all communication is performed via the opened sockets in JSON format to send different types of messages, as follows:

1. New publisher advertisement(NPA): Sent to the discovery service by a new publisher that wants to publish data on topic *T*. The message structure is as follows:

```
{
type: ‘‘newpub’’,
address:‘‘broker address’’,
topic: ‘‘T’’
}
```

2. New subscriber advertisement(NSA): Sent to the discovery service by a new subscriber that wants to subscribe to the publication of topic *T*. The message structure is as follows:

```
{
type: ‘‘newsub’’,
address:‘‘broker address’’,
topic: ‘‘T’’
}
```

The server responds with a list of all existing publishers of topic *T*. Then, it streams new publishers gradually using *NPA* messages to that subscriber. The discovery service acts as a streaming server that keeps track of all publishers and subscribers in memory and streams new publishers to interested subscribers as soon as they come. Figure 4 illustrates a more generic process when there is more than one subscriber and more than one publisher.

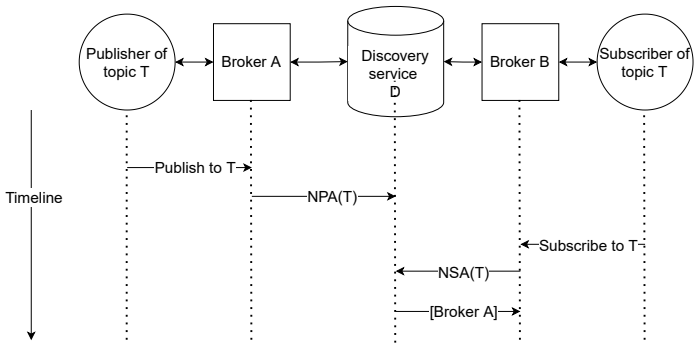

Figure 4. Discovery process illustration.

3.3. Broker Service

The broker service wraps an MQTT broker to expose its internal resources and allow it to publish and subscribe to topics outside its jurisdiction. The service is mainly concerned

with interactions with the discovery service and other broker services, leaving interactions between MQTT clients and the broker intact. This ensures backward compatibility of the service with the basic MQTT protocol, while having all that functionality associated with the discovery and consumption of external data, in addition to exposing internal data. We detail the three main responsibilities of a broker service as follows:

1. Discover global topics for local clients: The broker service advertises its interests in new topics to the discovery service as they come. The entire process is event-driven, where if a subscription for a new topic has not been seen before by the service, it advertises that topic to the discovery server. In response to that, the discovery service sends subscription notification messages to the publisher of that topic, as illustrated in Section 3.2.
2. Deliver local publications to global clients: The broker service that is managing a publisher provides the service that allows brokers managing subscribers to connect and join groups pertaining to the topics they are interested in. Subsequently, the broker managing a publisher is responsible to push new publications to all new publications to the respective groups.
3. Deliver global publications to local clients: This is the last mile delivery of data coming from global publishers, which has to be published to local clients. The broker service in this case publishes incoming data to its intended topic locally.

An illustration of the end-to-end delivery of publications is shown in Figure 5. In the figure, it is assumed the subscriber already knows about the publisher following the process described in Figure 4. Immediately following the discovery, Broker B connects to Broker A's Websockets server and asks to join a group call *T* about the topic. Hence, whenever a new publication of topic *T* is generated at Broker A's end, it is pushed through all the sockets in the group called *T*. This greatly simplifies the design of architecture and improves the reliability of the implementation as all the socket management functionality can be delegated to a robust socket management library, such as Socket. IO. Furthermore, the use of WebSockets as transport between brokers means that the delivery of messages across brokers is reliable by default.

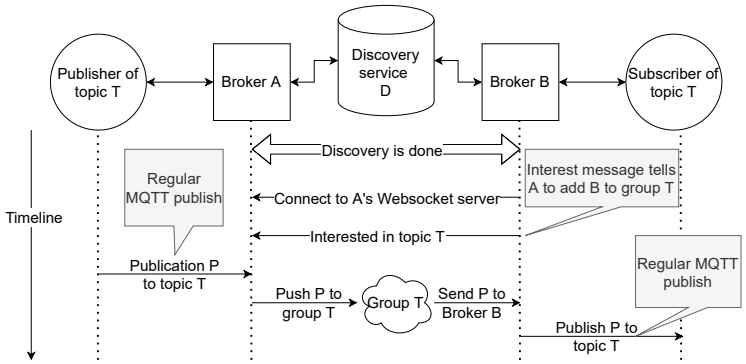

Figure 5. Illustration of the end to end delivery process.

3.4. Authentication Service

The authentication service issues JSON Web Tokens (JWTs) to other services which authenticates. It has a unique public key that is known to all the other services, which is used to verify the tokens it issues to other services.

The process starts with a new broker or discovery service that communicates with the authentication service requesting an authentication token passing in their identity. A typical token is represented as follows:

```
{
header:{
‘‘alg’’: ‘‘HS256’’,
```

```
‘‘typ’’: ‘‘JWT’’
}
payload:{
username: ‘‘Broker service 1’’
claims: {
blocked_publish_topics:[],
blocked_subscribe_topics:[],
blocked_brokers:[],
...
}
}
}
```

The requesting broker service can use this signed token to authenticate to all other broker services because the token is signed by the authentication service. The verification process does not require the verifier to contact the authentication service. Furthermore, the authentication service can also add additional parameters to the token such as authorization and expiration date [19]. The claims field above illustrates how a given broker service can be blocked from publishing certain topics or subscribing to topics published by certain brokers. This blocking is performed at two levels:

1. Discovery service level: Discovery service will not announce to a blocked broker service any new publishers that they are blocked from. Furthermore, it does not advertise NPAs to topics a broker service is blocked from.
2. Broker service level: A publishing broker service does not assign a blocked broker service to a group of a topic that they are blocked from.

Finally, It is worth noting that the authentication service is a RESTful web service that exposes an endpoint called “/auth”, where the consumer can send a POST request with their credentials to obtain a JWT. The authentication process is illustrated in Figure 6.

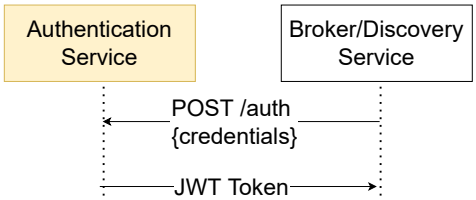

Figure 6. Communication with authentication service illustration.

4. Performance Evaluation

In this section, we discuss the details of our implementation and performance evaluation setup. The implementation of the authentication service was performed in NodeJS using express and the JSON Web Tokens library. The broker service wraps a javascript implmentation of the MQTT protocol for the server, and uses websockets to communicate with other broker services and the discovery service. Finally, the discovery service is fully implemented using websockets in NodeJS.

We use the following metrics to evaluate our architecture:

- Global delay (*GD*): Time it takes a message published within one broker to reach a single subscriber that is managed by a different broker.
- Local delay(*LD*): Time it takes a message published within a broker to reach a single subscriber within the same broker.
- Difference between global delay and local delay (*GD-LD*): Gives the overhead in terms of difference of end-to-end delays in milliseconds.
- Ratio of global delay to local delay (*GD/LD*): Gives the overhead in terms of ratio of end to end delays.

We implemented our architecture in javascript and set up three scenarios in order to evaluate it. These scenarios compute the overhead of migrating the data across broker services by comparing the time to reach local clients and a global ones. The first two scenarios are simulated on a single machine, as follows:

1. Single publisher and two subscribers residing within two different brokers. One subscriber is local to the publisher and the other subscriber is on the other broker. In this scenario, the publisher publishes messages of sizes 1, 10, 100, 1000 and 10,000 bytes to see the effects of payload size on the overhead of migrating the messages to the other broker in terms of end-to-end delay. For each size, 10 messages are sent at a rate of 1 message per second and the our metrics are computed. Then, descriptive statistics are computed for the collected values, which include the average, maximum and minimum for each metric. Finally, the results are compared across message sizes. Message Sizes of less than 100 bytes are fairly common in many IoT scenarios such as messages sent by temperature sensors. At around 1000 bytes is a common message size for video streaming applications over MQTT, because that is around the maximum payload size of an MQTT message. At 10,000 bytes which is around 10 maximum payload MQTT messages, gives a good approximation of the delay when multiple messages are bundled for the same destination.
2. One publisher and ten subscribers on one broker and ten other subscribers on another broker. In this scenario, the message size is fixed at 100 bytes and the goal is to understand the effect of adding more subscribers globally in terms of overhead and delay. Similar to the previous scenario, we compute our metrics and the same descriptive statistics for the local subscribers and the global ones and compare them.

The last scenario was performed on EC2 virtual machines on Amazon Web Services cloud. Five standard EC2 instances running Ubuntu 22.04 with 1GB of RAM are located in Tokyo and are used to host two broker services, discovery service, authentication service and an instance for client simulation. All instances where running in the same city (Tokyo). The scenario emulates a simple IoT application for collecting temperature data from two sensors running within the same MQTT broker. The simulated sensors publish their data to their MQTT broker on topic "temp". Then, two subscribers are setup where one is connected to the same broker and another one connected to a different MQTT broker. The publishers and subscribers are simulated using javascript code running from the fifth EC2 instance. Each publication is time stamped at the publisher, so that the total delay can be calculated which is used to compute descriptive statistics and the ratios of global client to local client. The number of messages sent are 10 per publisher and they are sent without any delay.

The messages sent by each of the publishers is a JSON object structured as follows:

```
{
sensor: ‘‘sensor 1’’
temp: 24.8,
timestamp: 1669816556136
}
```

The timestamp is used to compute delay which is the difference between the sent and arrival times. We compute the average delay for each publisher using 10 messages.

5. Results and Discussion

The results of the first scenario, illustrate the capability of the system to dynamically register brokers based on the interest of their clients. Thus, when a client publishes the first message within a broker, then the broker is automatically registered with the tracking service as a publisher of that topic, which is subsequently propagated to all registered subscribers. Furthermore, the results of the experiment shows that increasing the size of the payload up to 10,000 bytes does not affect the overhead significantly. The average ratio *GD/LD* is between 1.5 and 1.6, which translates roughly to around 2.5 ms difference

between GD and LD on average, as shown in Figures 7 and 8, respectively. The same Figure shows maximum and minimum values which gives an idea about the best and worst possible delays for local and global clients.

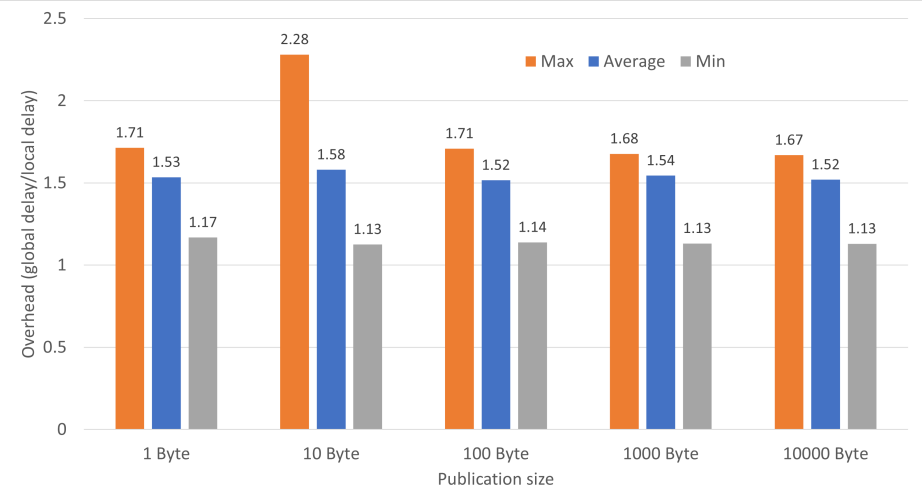

Figure 7. Ratio of delay for global client to local client as publication size increases (10 trials each).

The results from running the second scenario illustrate that a given broker does not issue multiple subscription requests to the same broker when multiple clients subscribe to the same topic. Only the first subscription to a topic acts as a trigger for global subscriptions to other brokers managing publishers of that topic. Furthermore, the migration of the 100 bytes message adds around 2.5 ms of delay to the global clients, in the best, worst and average cases, as shown in Figures 9 and 10, respectively.

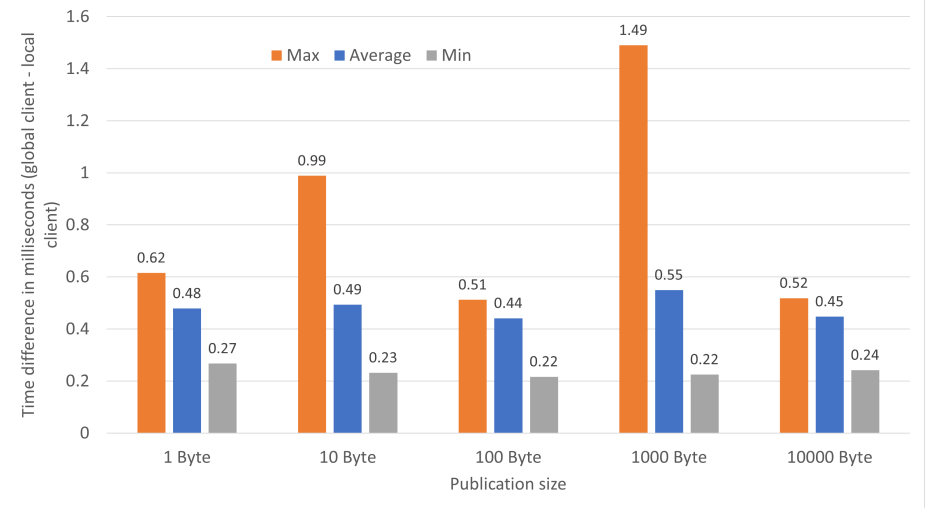

Figure 8. Difference in milliseconds between arrival time for global client and a local client as publication size increases (10 trials each).

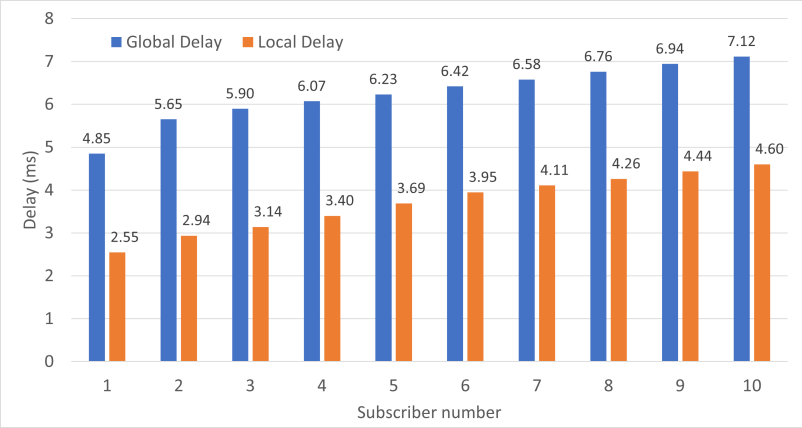

Figure 9. Comparison of delay for 10 local subscribers with 10 global subscribers for a single buffer of size 100 bytes.

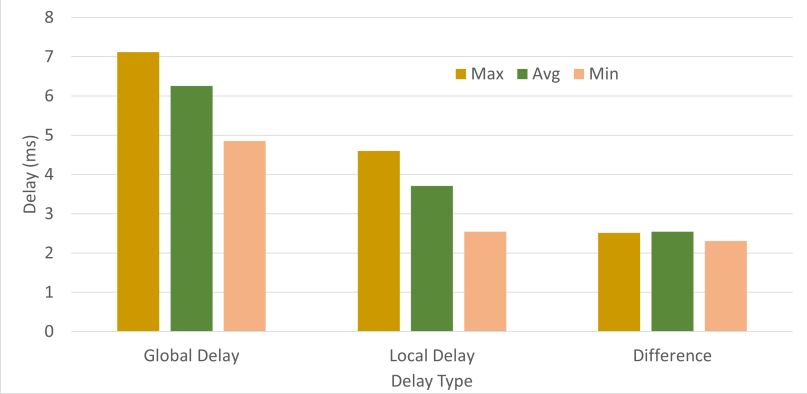

Figure 10. Descriptive statistics of global and local delays.

The results from third scenario which was done using multiple virtual machines on Amazon cloud are in line with the result from the previous two scenario. Table 2 shows the average and minimum, maximum and average delays of publication to local and global clients and their ratios. In comparing the results to the ones shown in Figures 7 and 9, we find that even in the presence of two publishers on the same topic similar results are present in terms of difference between the minimum and maximum delays. Furthermore, the ratios show a more significant difference which is likely due to precision of time in single machine tests vs. test in the cloud. In single machine tests, we used time precision up to microseconds, while in the cloud we used milliseconds.

Table 2. Average/minimum/maximum delay of messages to local and global subscribers.

Measure (ms)	Local Subscriber	Global Subscriber	Ratio (Global/Local)
Average	45.25	56.4	1.24640884
Minimum	41	51	1.243902439
Maximum	50	68	1.36

There are three main observations that can be made about the results of our experiments:

1. Payload size does not have a major impact on the delay on average: We have shown this through the results of scenario 1. We believe there are two main reasons for this: First, there is only one publisher and two subscribers in the whole system, hence, theoretically the delays are expected to be at their minimum. Second, the use of websockets as transport layer between the brokers which behave like streaming TCP

sockets, reduce the amount of overhead associated with each transmission (e.g., only 2 bytes application layer header with every transmission).

2. Increasing the number of subscribers (global and local to the publisher) introduces more delays, which is to be expected. However, an interesting observation in our system is that the difference between arrival times for global and local clients is almost the same in the best, worst and average cases. We believe there are two main reasons for this: First, the payload is delivered to the broker which publishes it locally to its clients, which greatly improves efficiency over the case where each subscriber establishes a connection with the global broker to get the data. Second, most of the delays are introduced by the the MQTT protocol itself, as evident by the end to end publishing times for localized publisher and subscriber. Furthermore, since we are technically doing two publications (one from the local publisher and another by the remote broker to its local clients) to deliver a message end to end, the results from Figure 9 show that the overhead of data migration data migration between the brokers is negligible compared to overhead of MQTT publication.
3. The synthetic nature of these tests could have minimized the actual delay times to low values; however, in our calculations we used differences and ratios which help in giving the results more context.

6. Conclusions and Future Work

We proposed a service oriented architecture for data discovery and distribution in MQTT. It has three main services for: authentication, discovery and data distribution. The data distribution services authenticate themselves to other data distribution services and the discovery service through the authentication service, which implements scalable identity and claims management scheme. Furthermore, each data distribution service manages a set of local MQTT clients, through aggregating the topics they are interested in and fetching their corresponding publications from other data distribution services. The architecture is designed to be dynamic and fault tolerant, through keeping minimal state and working on the basis MQTT clients interests. We evaluated the architecture in two main scenarios. The initial results from our evaluation indicate such an architecture is promising for environments that are highly dynamic which need to dynamically scale. As a future work, it is crucial to test the system in a simulated real scenario over a real network to get a better sense of how it performs. This would also help greatly with finding weaknesses of the architecture and fixing them.

Author Contributions: Conceptualization, T.A. and F.A.; methodology, T.A. and F.A.; software, T.A.; validation, T.A. and F.A.; formal analysis, T.A. and F.A.; investigation, T.A. and F.A.; resources, T.A. and F.A.; data curation, T.A. and F.A.; writing—original draft preparation, T.A. and F.A.; writing—review and editing, T.A. and F.A.; visualization, T.A. and F.A.; supervision, F.A.; project administration, F.A.; funding acquisition, F.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded by King Abdulaziz City for Science and Technology (KACST) under the National Science, Technology, and Innovation Plan (project number 13-INF2452-04).

Acknowledgments: The authors would like to acknowledge the support provided by the Interdisciplinary Research Center for Intelligent Secure System and the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yassein, M.B.; Shatnawi, M.Q.; Aljwarneh, S.; Al-Hatmi, R. Internet of Things: Survey and open issues of MQTT protocol. In Proceedings of the 2017 International Conference on Engineering & MIS (ICEMIS), Monastir, Tunisia, 8–10 May 2017; pp. 1–6.
2. Standard, O. MQTT Version 3.1.1. 2014; Available online: Http://docs.Oasis-Open.Org/mqtt/mqtt/v3 (accessed on 1 May 2020).
3. Light, R.A. Mosquitto: Server and client implementation of the MQTT protocol. *J. Open Source Softw.* **2017**, *2*, 265. [CrossRef]

4. Protskaya, Y.; Veltri, L. Broker Bridging Mechanism for Providing Anonymity in MQTT. In Proceedings of the 2019 10th International Conference on Networks of the Future (NoF), Rome, Italy, 1–3 October 2019; pp. 110–113.
5. Longo, E.; Redondi, A.E.; Cesana, M.; Arcia-Moret, A.; Manzoni, P. MQTT-ST: A spanning tree protocol for distributed MQTT brokers. In Proceedings of the ICC 2020-2020 IEEE International Conference on Communications (ICC), Dublin, Ireland, 7–11 June 2020; pp. 1–6.
6. Ohno, S.; Terada, K.; Yokotani, T.; Ishibashi, K. Distributed MQTT broker architecture using ring topology and its prototype. *IEICE Commun. Express* **2021**, *10*, 582–586. [CrossRef]
7. Alsbouí, T.; Hammoudeh, M.; Bandar, Z.; Nisbet, A. An overview and classification of approaches to information extraction in wireless sensor networks. In Proceedings of the 5th International Conference on Sensor Technologies and Applications (SENSORCOMM'11), Palmerston North, New Zealand, 21–27 August 2011; Volume 255.
8. Kawaguchi, R.; Bandai, M. Edge based MQTT broker architecture for geographical IoT applications. In Proceedings of the 2020 International Conference on Information Networking (ICOIN), Barcelona, Spain, 7–10 January 2020; pp. 232–235.
9. Jutadhamakorn, P.; Pillavas, T.; Visoottiviseth, V.; Takano, R.; Haga, J.; Kobayashi, D. A scalable and low-cost MQTT broker clustering system. In Proceedings of the 2017 2nd International Conference on Information Technology (INCIT), Nakhonpathom, Thailand, 2–3 November 2017; pp. 1–5.
10. Sen, S.; Balasubramanian, A. A highly resilient and scalable broker architecture for IoT applications. In Proceedings of the 2018 10th International Conference on Communication Systems & Networks (COMSNETS), Bengaluru, India, 3–7 January 2018; pp. 336–341.
11. Park, J.H.; Kim, H.S.; Kim, W.T. Dm-mqtt: An efficient MQTT based on SDN multicast for massive IoT communications. *Sensors* **2018**, *18*, 3071. [CrossRef] [PubMed]
12. Rausch, T.; Nastic, S.; Dustdar, S. Emma: Distributed QoS-aware MQTT middleware for edge computing applications. In Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, USA, 17–20 April 2018; pp. 191–197.
13. Banno, R.; Sun, J.; Fujita, M.; Takeuchi, S.; Shudo, K. Dissemination of edge-heavy data on heterogeneous MQTT brokers. In Proceedings of the 2017 IEEE 6th International Conference on Cloud Networking (CloudNet), Prague, Czech Republic, 25–27 September 2017; pp. 1–7.
14. Tantitharanukul, N.; Osathanunkul, K.; Hantrakul, K.; Pramokchon, P.; Khoenkaw, P. MQTT-topics management system for sharing of open data. In Proceedings of the 2017 International Conference on Digital Arts, Media and Technology (ICDAMT), Chiang Mai, Thailand, 1–4 March 2017; pp. 62–65.
15. Hammoudeh, M.; Newman, R.; Dennett, C.; Mount, S.; Aldabbas, O. Map as a service: A framework for visualising and maximising information return from multi-modal wireless sensor networks. *Sensors* **2015**, *15*, 22970–23003. [CrossRef] [PubMed]
16. Moffat, S.; Hammoudeh, M.; Hegarty, R. A survey on ciphertext-policy attribute-based encryption (CP-ABE) approaches to data security on mobile devices and its application to IoT. In Proceedings of the International Conference on Future Networks and Distributed Systems, Cambridge, UK, 19–20 July 2017.
17. Hammoudeh, M.; Epiphaniou, G.; Belguith, S.; Unal, D.; Adebisi, B.; Baker, T.; Kayes, A.; Watters, P. A service-oriented approach for sensing in the Internet of Things: Intelligent transportation systems and privacy use cases. *IEEE Sens. J.* **2020**, *21*, 15753–15761. [CrossRef]
18. Papazoglou, M.P.; Van Den Heuvel, W.J. Service oriented architectures: Approaches, technologies and research issues. *VLDB J.* **2007**, *16*, 389–415. [CrossRef]
19. Ahmed, S.; Mahmood, Q. An authentication based scheme for applications using JSON web token. In Proceedings of the 2019 22nd International Multitopic Conference (INMIC), Islamabad, Pakistan, 29–30 November 2019; pp. 1–6.