Journal of Computer Science

Original Research Paper

A New Approach to a Self-Organizing Federation of MQTT Brokers

Nicolas Kolling Ribas and Marco Aurélio Spohn

Department of Computer Science, Federal University of Fronteira Sul, Brazil

Article history
Received: 19-04-2022
Revised: 18-06-2022
Accepted: 04-07-2022

Corresponding Author:
Marco Aurélio Spohn
Department of Computer Science, Federal University of Fronteira Sul, Brazil
Email: marco.spohn@uffs.edu.br

Abstract: In the Publish/Subscribe (PS) communication paradigm clients producing content (i.e., publishers) send it to a broker, which relays the content to consumers (i.e., subscribers). Communication can happen asynchronously, leaving all the hurdles to the server/broker. MQ Telemetry Transport (MQTT) is the most used P/S protocol in designing IoT applications. The usual MQTT scenario comprises a single broker, making it a potential bottleneck and a single point of failure. Clustering of brokers is the typical approach for scaling MQTT, usually restricting deployment to a single administrative domain. The federation of autonomous brokers is a more flexible approach for scaling the MQTT protocol, with just one representative self-organizing protocol at the moment. We present a new variant for such protocol, providing a well-structured mechanism for building and orchestrating the federation of brokers. The main contribution of this study is to offer a feasible solution for deploying the MQTT federation without requiring critical changes to regular clients. By following the topic naming conventions, clients are unaware they rely on a federation of brokers. To provide a glimpse of our solution in action, a case study demonstrates that the protocol can easily provide flexible reliability with low complexity.

Keywords: Publish/Subscribe Communication, Federation of P/S Brokers, MQTT

Introduction

The Publish/Subscribe (P/S) communication paradigm decouples client-to-client communication from a traditional direct interaction between clients to a brokering communication model. It is also a consumer-producer representative, where the producer (publisher) sends its messages to a broker, which relays them to their consumers (subscribers). Such decoupling makes asynchronous communication easier to handle at the application layer.

MQ Telemetry Transport (MQTT) (Banks *et al*., 2019) is one of the most employed P/S protocols for implementing IoT applications. Its lower control overhead and support for small packets address one primary requirement for most IoT devices' reduced computing and communication capacity. On the other hand, the global number of IoT devices is growing exponentially, requiring more scalable MQTT systems. An MQTT system employs a single server on its standard configuration, characterizing itself as a potential bottleneck and a single point of failure.

MQTT systems can scale horizontally and vertically (Pipatsakulroj *et al*., 2017; Al-Fuqaha *et al*., 2015; Longo *et al*., 2020). When employing one physical machine, it is possible to scale it vertically by boosting processing and memory capacity. In addition to that, it is feasible to scale a single machine horizontally by running several broker instances, keeping the service running while any of the brokers stay functional. However, the machine remains a single point of failure in both scenarios.

Clustering of brokers is the de facto approach for horizontal MQTT scalability (Jutadhamakorn *et al*., 2017). Each broker can run on an independent physical machine, considering that all clustering brokers are mutually reachable throughout the network. Clients access brokers through a load balancer, which strives to spread the work evenly among brokers. Clients' peers do not need to use the same broker, leaving the proper message routing between brokers as one of the main tasks for the clustering system. Clustering usually entails a single administrative domain, even though deploying brokers over separate data centers is possible.

The federation of brokers is another way to scale the system (Spohn, 2020). The orchestration of a set of brokers can follow a centralized approach or an autonomous one, possibly spanning several administrative domains. There is not much evidence of

how the federation and clustering compare in performance; however, the federation can improve the availability and the overall fault tolerance when the solution provides multiple paths among the clients and their brokers.

To the best of our knowledge, there is just one self-organizing MQTT federation protocol (Spohn, 2021) and this study presents a new variant of such a model. Our approach provides a more straightforward scheme to orchestrate the federation, sticking to the original strategy of handling all the necessary tasks at the application layer. The main contribution of this study is to propose a viable solution for deploying the MQTT federation without requiring crucial changes to regular clients: By following the topic naming conventions, clients are not even aware they are relying on a federation of brokers.

Next, we present the related work, focusing on the current self-organizing MQTT federation protocols. After that, we deliver our approach for better organizing and orchestrating an autonomous and self-organizing federation of brokers, including a case study. Finally, we present our last thoughts on the present work.

Related Work

Spohn (2020) proposed the first solution for brokers' federation. The orchestration centers on the subscribers, which self-organize to build a mesh structure that interconnects them. A mesh has a central broker (i.e., core) elected to build and maintain the mesh. As soon as a broker receives the subscription of a new topic, it starts announcing itself as the core of a new mesh. Core announcements propagate throughout the federation network, allowing all participating brokers to learn how to reach the core of any mesh. Brokers with local subscribers send a mesh membership announcement through the shortest paths to the core to join a mesh. Depending on the federation network topology, one can define the mesh redundancy to achieve multiple paths between mesh members. All brokers with local subscribers or those that interconnect them to the core participate in a mesh.

Publishers must send their publications toward the mesh with the core as a first target. Upon reaching any mesh member, a publication then propagates throughout the mesh. Therefore, it allows any publication to reach all subscribers regardless of which broker they select.

Figure 1 illustrates the process of building a mesh in a virtual network topology comprised of six brokers and the redundancy of two parents (if the topology allows). When broker one receives a local subscription, it triggers the mesh creation process for the respective topic. It sends a core announcement to neighboring brokers, which will forward the announcement until it reaches all brokers in the federation. This way, all brokers learn of any core and how to reach it. So far, only broker one is taking part in the mesh. Once a new subscriber starts at broker five, it sends a mesh membership announcement to its parents (i.e., brokers three and four). When brokers three and four receive the membership announcement, they become part of the mesh, also passing the member announcement to their parents, which in this case is the mesh core.

Figure 2 displays the routing process for two publications in the newly constructed mesh. The first publication takes place at broker four and, as it belongs to the corresponding mesh, the broker routes the message to all its mesh neighbors (brokers one and five). The forwarding process continues until all mesh members receive at least one message for the corresponding publication (notice that message duplicates can happen depending on the mesh redundancy). The second publication takes place at broker zero and, as it is not a mesh member, the broker forwards the publication towards the corresponding mesh core (i.e., broker one). In this example, once the publication reaches the core, the publication is flooded throughout the mesh, reaching all corresponding subscribers.

In its original solution, brokers must implement the federation protocol, which might hinder its adoption.

Spohn (2021) proposed an endogenous self-organizing federation approach, working at the application layer without any modification to standard brokers. For this purpose, the concept of a federator surfaces an application associated with the broker, responsible for performing both the creation and management of meshes and the routing of control messages and publications. The whole system works based solely on the P/S mechanism and it is composed of two entities:

- Pub_Fed: Maintains direct connection with all neighboring brokers, responsible for sending control messages and routing regular messages
- Sub_Fed: Handles control and regular messages received from neighbors

Figure 3 exhibits the architecture of the proposed federator. Sub_Fed receives core and mesh member announcements through publications from a neighboring federated broker through two control topics: CORE_ANN and MESH_MEMB_ANN, respectively. Sub_Fed also subscribes to all regular federated topics, so if there is a new publication, it can forward them to Pub_Fed, which in turn will relay these posts to neighbors by encapsulating the original messages in a control topic data message. Although there is no need for changes in the broker, the solution mandates that clients meet a requirement: They must report through a control topic (i.e., NEW_REGULAR_TOPIC) of every new subscription or first post to a regular topic. It is necessary so that the federator learns what regular topics it should handle.

DOI: 10.3844/jcssp.2022.687.694

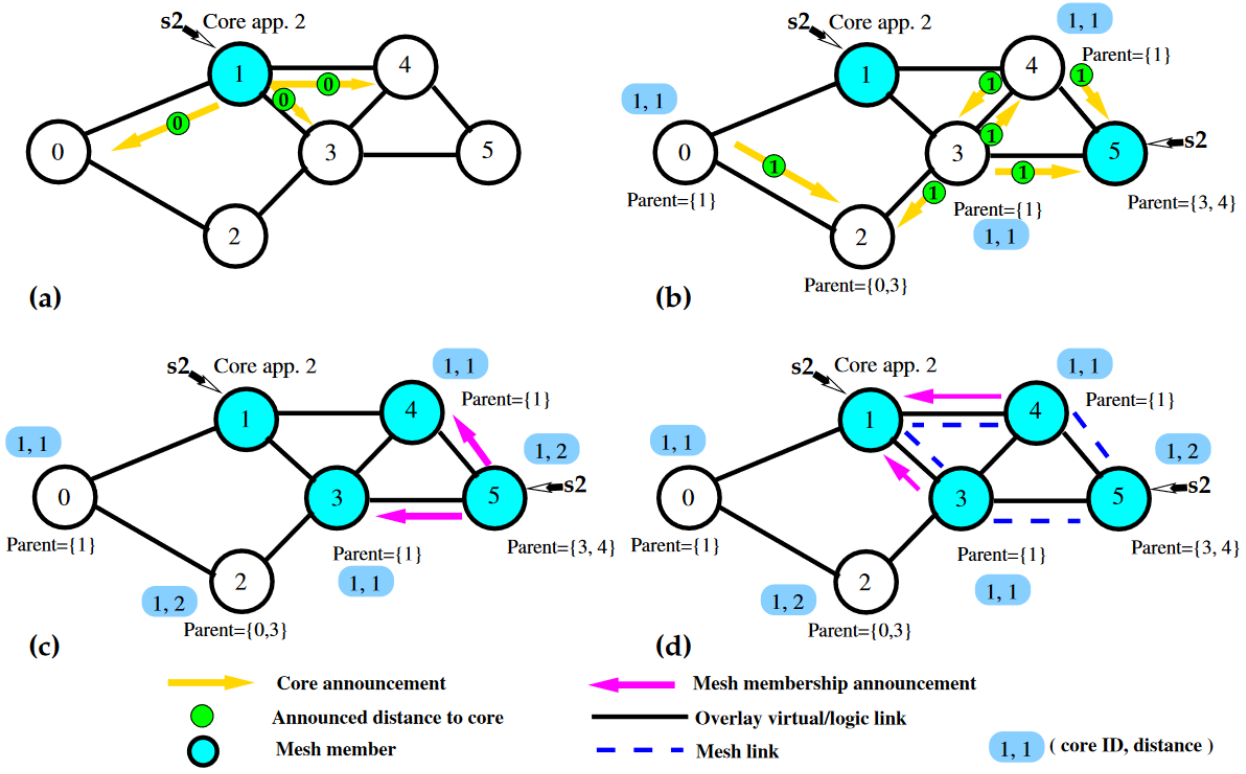

Fig. 1: Mesh construction process (Spohn, 2020)

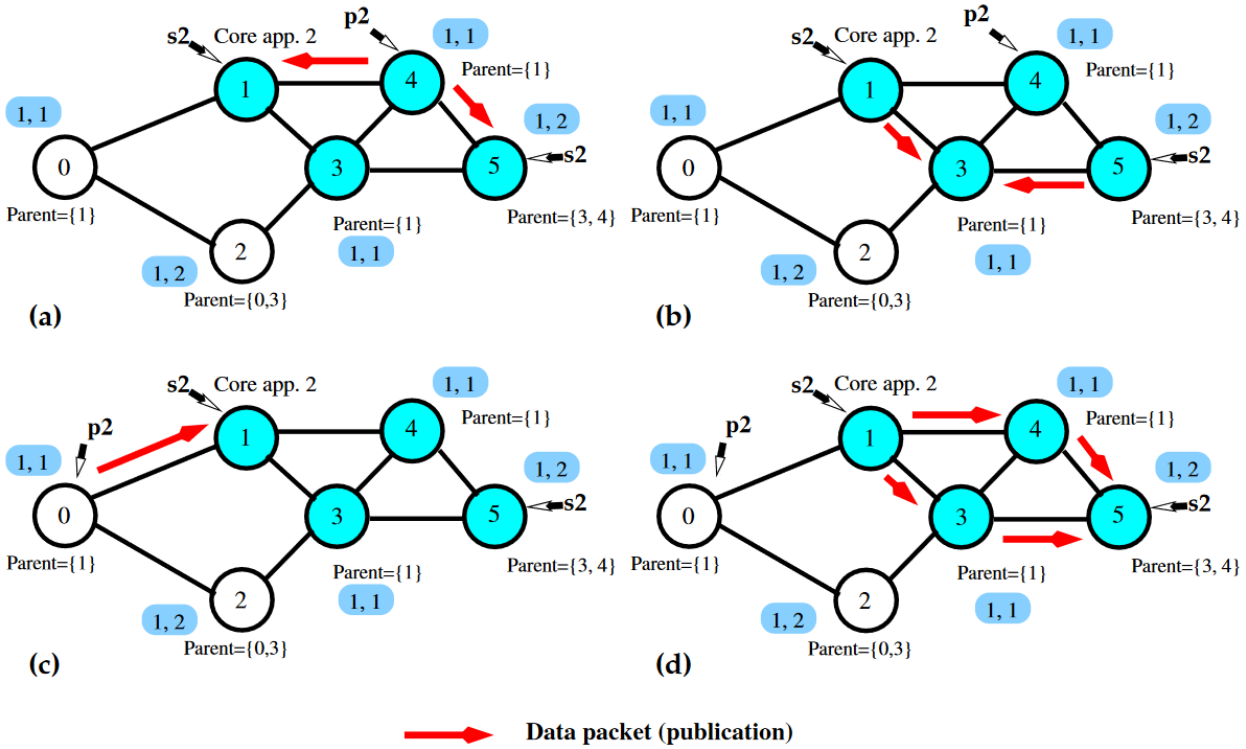

Fig. 2: Publication routing process (Spohn, 2020)

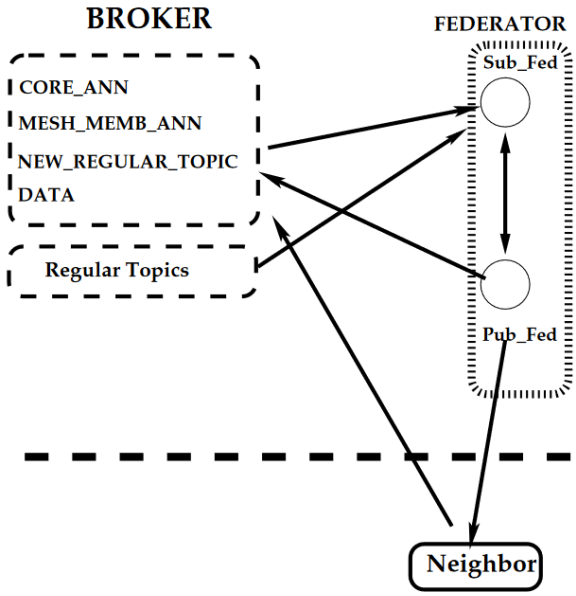

Fig. 3: Main federator entities (Spohn, 2021)

A New Federation Approach

To provide brokers with federation capabilities, we propose a federation approach evolved from the protocol presented by Spohn (2021), which uses only the native MQTT publication and subscription mechanisms. Changes were made to the protocol, effectively introducing a new variant. We describe the new application framework, changes to the original protocol, and implementation details below.

Application Framework

We define as federated publications (i.e., publications that need routing to other brokers) those targeting topics with the first level equal to "federated"; that is, federated topics follow the structure "federated/X", where X is indeed a federated topic. The federator subscribes to all federated topics at its host broker using a multilevel wildcard (i.e., "federated/#").

In addition to the federated topics, the federator also uses four control topics that observe a similar structure: The first topic level identifies which message type it carries and the last level specifies which federated topic the message takes. The control topics to which the federator subscribes are as follows:

- **federator/core_ann/#**: Intended for core announcements
- **federator/memb_ann/#**: Intended for mesh membership announcements
- **federator/routing/#**: Intended for routing of federated publications
- **federator/beacon/#**: To receive beacons reporting local subscribers

When using a multilevel wildcard replacing the last level of control topics, the federator starts to receive control messages referring to all federated topics. For example, a message for the topic "federator/core_ann/sensor" implies a core announcement referring to a federated topic "sensor".

For the federator to perform its role, it needs to discover new publications and subscriptions happening at the host broker. For this purpose, in the original protocol, there is the particular topic NEW_REGULAR_TOPIC: A topic where clients must publish informing every first publication or subscription to a federated topic. In our approach, the federator subscribes to all federated topics without the need to be informed of new publications, as it already receives them from the broker.

However, clients need to explicitly notify their subscriptions, as there is no other way for the federator to identify them. For this purpose, subscribers must publish to the control beacon topic corresponding to the subscribing topic. For example, a customer subscribing to the federated topic "sensor1" must publish to the "federator/beacon/sensor1" topic. A beacon consists of an empty message and it must happen regularly. The expected interval between each beacon is configurable through the federator configuration file. Suppose no beacon message is received for a given topic for a period longer than three times the expected interval. In that case, the federator assumes that there are no more local subscribers and it will take actions such as ceasing to be a mesh member or stop advertising itself as the related topic's core.

The traffic of federated publications occurs through two distinct and already mentioned sets of topics: Federated topics (i.e., "federated/#") and routing topics (i.e., "federator/routing/#"). What differentiates these two sets is that in federated topics, posts are regular topic messages in their "raw" format. Routing topics are for the exclusive use of federators: In addition to the original message content, they carry a unique identifier, source, and forwarding broker ID.

The identifiers of federated publications are composed of two values (Fig. 4): The originating broker ID; and a sequence number, starting at 0 and incremented with each new local publication, which helps identify duplicate publications. The sender field indicates which neighboring broker has forwarded the message; therefore, we can ignore this neighbor when it requires forwarding the message to other neighbors.

When receiving a new publication on a local federated topic, the federator starts the routing process. The federator encapsulates the original message in a routing topic, assigning a unique identifier to it and itself as the sender. If the current broker belongs to the related topic mesh, the federator publishes the message to the remaining mesh neighbors. Otherwise, the federator forwards the publication toward the corresponding mesh core. Figure 5 illustrates the initial process of routing a local publication, having "sensor1" as a federated topic.

For federated publications received on a routing topic, the federator must first ensure that they are not duplicates, which is possible by keeping a record of new routed messages. In our approach, we assume an LRU cache (Froelich, 2022) (new entries replace older ones, keeping the records for the identifiers of newly routed publications) and that the cache size is configurable through the federation's configuration file.

If a publication is not a duplicate, the federator forwards the publication to the required mesh neighbors, following the same logic as routing a local publication. The federator also unwraps the original content and publishes it to the corresponding local federated topic if there are any local subscribers (Fig. 6 illustrates this process, where "sensor1" stands out as an example of a federated topic).

One could argue that publishing to the federated topics the federator subscribes to would get it into a loop due to receiving its posts and rerouting them with new identifiers, yielding duplicates. Our approach prevents loops because, when subscribing to federated topics, the federator utilizes the No Local MQTT option that informs the broker that we should not receive our publications. This option is not present in MQTT versions 3.1.1 and earlier, limiting its use to brokers supporting version 5 of the protocol.

The routing of a publication starting at a non-mesh broker operates using all available parents leading to the core, unlike the original solution based on unicasting the publication towards the mesh. Our procedure explores all available redundancy for routing, inside and outside the meshes.

Implementation

Our implementation relies on two sound MQTT open-source projects under the Eclipse Foundation: Paho (Eclipse-Foundation, 2022) and Mosquitto (Light, 2017). Paho aims to provide client implementations for the MQTT protocol. Having support available for various platforms and programming languages, the Paho client facilitates the implementation of the federator, providing the association of the application with the federated broker. In its turn, Mosquitto is a single-threaded MQTT broker implementation.

The federator implementation[1] was carried out in the Rust 2021 programming language using the Tokio runtime system (Lerche, 2022) and the MQTT Paho client library. The application's architecture, as illustrated in Fig. 7, includes as main elements the dispatcher, multiple message queues, and multiple topic workers. Both the dispatcher and the various topic workers consist of asynchronous execution units called tasks: Similar to system threads, however, being managed by the Tokio runtime system. Compared to system threads, Tokio's tasks are lighter to create, execute and destroy.

Topic workers are the main components carrying out a federation's work: They manage the fabrics and the routing of messages. For each federated topic and consequently, for each mesh, there is a companion topic worker.

A configuration file following the TOML format[2] defines the federator's settings according to the following parameters:

- *Redundancy*: Positive integer that designates the redundancy of the created meshes
- *Core_Ann_Interval*: Positive integer that designates the interval, in seconds, between core announcements
- *Beacon_Interval*: Positive integer that designates the interval, in seconds, expected between beacons coming from local subscribers
- *Cache_Size*: Positive integer specifying the maximum size for caches employed to filter topic messages duplicates
- *Host*: Defines the virtual identifier and URI for the federator's broker
- *Neighbors*: Virtual identifier and URI for each neighboring broker

Listing 1 contains an example of a federator configuration file associated with a broker with identifier 1 and whose neighbors are brokers with identifiers 2 and 4.

Listing 1: Federator configuration file example

```
redundancy = 2
core_ann_interval = 10
beacon_interval = 5
cache_size = 5000

[host] id = 1
uri = "tcp: //local host:1881"

[[neighbors]]
id = 2
uri = "tcp: //local host:1882"

[[neighbors]]
id = 4
uri = "tcp: // local host:1884"
```

Case Study

It is not our purpose to provide a thorough performance evaluation of our federation approach. An autonomous federation of brokers primarily targets strengthening MQTT services' overall reliability and availability. Therefore, we present a case study to describe how to build a broker federation topology while taking some performance statistics for the particular scenario.

[1]Source code available at https://github.com/ nicolaskribas/mqtt-fed

[2]https://toml.io/en/v1.0.0

Evaluation Scenarios

The evaluation scenarios are an attempt to replicate those used by Spohn (2021). Therefore, the topology has nine federated brokers arranged in a 3 × 3 grid (Fig. 8), with mesh redundancy of value two. There are two subscribers, one at node three and the other at node eight. Each federation node comprises a Mosquitto broker and a federator instance running in a Docker container. The standard settings for all federators are present in Table 1.

MQTT broker latency measure tool (Zhang Xiang, 2022), written in Go language, can measure broker latency in sending messages. The tool allows defining several configuration parameters, such as clients' QoS, message size, number of clients, and number of messages published by each client. In addition to latency, the tool can also compute other metrics such as publication rate and success rate in sending messages. This tool allows testing of the most common MQTT application based on a single broker. Therefore, to comply with a federation of MQTT clients, we had to make internal changes to the tool by adding the feature to entitle subscribers and publishers in different brokers.

There is a publisher at node seven, one hop away from node eight, and four hops away from node three. The publisher performs with two publication patterns: One with 500 messages and the other with 1000 messages, each being 64 bytes long. The publication delay is the primary performance metric under consideration. Each scenario is run ten times on a local machine with 16 GB of RAM and an AMD Ryzen™ 51500 X 3.5 GHz processor.

To evaluate a single broker scenario and to be able to make the necessary comparisons with the federated approach, we run both subscribers at the same node: First with both subscribers at node three and then with both at node eight.

Table 1: Standard federator's settings

Parameter	Value
The interval between core announcements	10 s
The expected interval between beacons	5 s
Cache size	5000

Fig. 4: Structure of routing packet

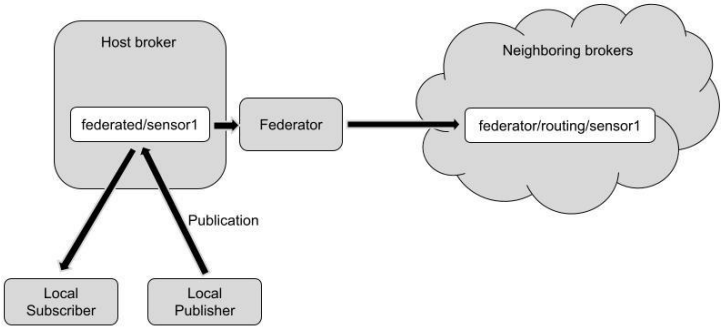

Fig. 5: Local publish routing

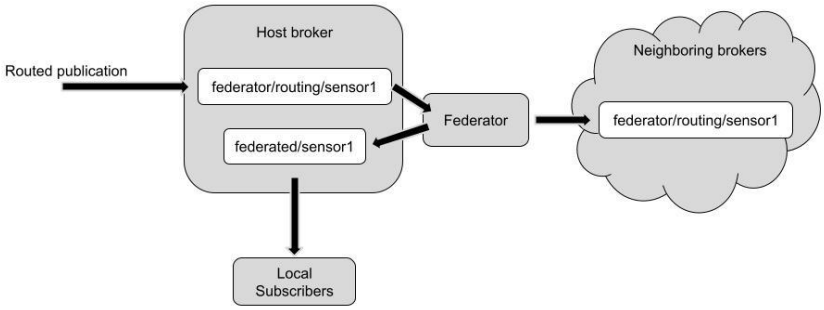

Fig. 6: Handling routed publication

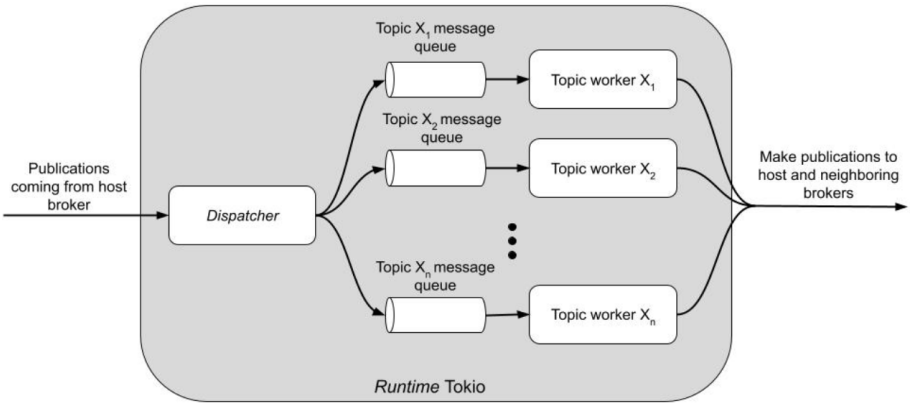

Fig. 7: Application architecture

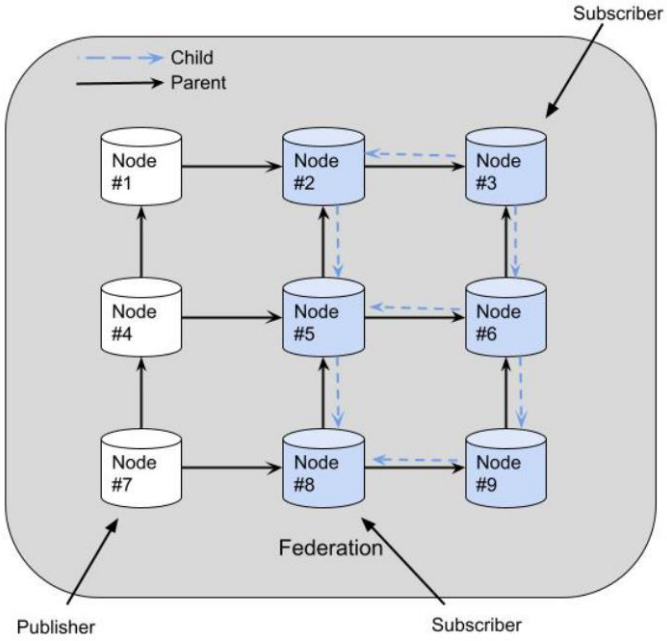

Fig. 8: Evaluation scenario

Results

Table 2 displays the results collected in the federation scenario. The subscriber at broker three, further away from the publisher, showed higher latency values than the subscriber at broker eight, closer to the publisher. We can see that the total number of publications did not quite affect the subscriber's latency at broker three. The same is not true for the subscriber at node eight. Higher latency is expected for a larger number of messages, as node eight must also handle the forwarding of messages throughout the mesh to reach node three.

Tables 3 and 4 show the results for the single broker scenarios. Again, we can observe the relation between latency results and the subscribers' distance to the publisher. In the scenario where both subscribers are on node three, aggregate latency increases compared to the federated solution, as the broker has to deal with twice the load on average. When both subscribers are on node eight, there is a substantial reduction in latency compared to the federated scenario. A possible explanation for such difference relates to the routing procedure. In both the federated and single broker scenarios, message routing happens using all the parents that the redundancy allows, incurring significant routing overhead. In the single broker scenario at node eight, routing overhead is minimal because the subscribers' broker is one hop from the publisher.

Table 2: Federated solution: Delay in publishing messages

Publications	Subs. at broker 3	Subs. at broker 8
500	17.21±9.17 ms	9.51±6.58 ms
1000	16.71±9.53 ms	12.35±10.70 ms

Table 3: Centralized solution: Broker on node 3

Publications	Sub. 1	Sub. 2
500	19.81±16.21 ms	18.80±13.05 ms
1000	17.66±15.47 ms	17.68±15.49 ms

Table 4: Centralized solution: Broker on node 8

Publications	Sub. 1	Sub. 2
500	3.33±4.35 ms	3.34±4.35 ms
1000	5.04±7.05 ms	5.10±7.12 ms

Conclusion

The federation of MQTT brokers provides means for scaling the system while qualifying for manageable system availability through the multiple connections among brokers. Even though it is not clear how clustering and federation compare in terms of performance, it is clear that the myriad of virtual federation topologies entitles a flexible redundancy degree to the interconnected brokers. Nonetheless, while redundancy adds to processing and communication costs, the overall gain in reliability might be invaluable.

We present a new framework for building and managing an autonomous federation of MQTT brokers. Our solution builds on a previous protocol, providing a structured way for setting and managing all the federation entities.

Even though our first implementation is in the Rust language, it provides a realization of our framework and is an application architecture model for the federation. The application architecture paves the way for scaling the federation starting from the bottom up: Lightweight tasks handle all the necessary functions related to federated topics, both at the control and data planes. In future work, we intend to extend the paho client libraries so that the federated infrastructure stays transparent to the clients.

Acknowledgment

The authors feel grateful to the anonymous reviewers for their valuable suggestions and comments on improving the quality of the paper. They would like to thank the editors of the journal as well. This study was partially funded by the *Universidade Federal da Fronteira Sul* (Research Project PES-2021-0471, under call 121/GR/UFFS/2021).

Author's Contribution

Nicolas Kolling Ribas: Contribution to conception and design. Performed the implementation and the case study. Contributions to draft the article.

Marco Aurélio Spohn: Designed the research plan and supervised its execution. Contributions to conception and design. Most of the writing.

Ethics

This article is original and contains unpublished material. The authors confirm that they have read and approved this document and that no ethical issues are involved.

References

Al-Fuqaha, A., Khreishah, A., Guizani, M., Rayes, A., & Mohammadi, M. (2015). Toward better horizontal integration among IoT services. *IEEE Communications Magazine*, 53(9), 72-79. https://ieeexplore.ieee.org/abstract/document/7263375/

Banks, A., Briggs, E., Borgendale, K., & Gupta, R. (2019). MQTT Version 5.0. *OASIS Standard*, *7*, 102. http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

Eclipse-Foundation (2022). Eclipse paho. https://www.eclipse.org/paho/

Froelich, J. (2022). An implementation of a lru cache. https://github.com/jeromefroe/lru-rs

Jutadhamakorn, P., Pillavas, T., Visoottiviseth, V., Takano, R., Haga, J., & Kobayashi, D. (2017, November). A scalable and low-cost MQTT broker clustering system. In 2017 2nd *International Conference on Information Technology* (INCIT) (pp. 1-5). IEEE. https://ieeexplore.ieee.org/abstract/document/8257870

Lerche, C. (2022). Tokio api docs. https://github.com/tokio-rs/tokio

Light, R. A. (2017). Mosquitto: Server and client implementation of the MQTT protocol. *Journal of Open-Source Software*, 2(13). https://nottingham-repository.worktribe.com/index.php/outputs?Type=Journal%20Article&page=111&Year=2017

Longo, E., Redondi, A. E., Cesana, M., Arcia-Moret, A., & Manzoni, P. (2020, June). MQTT-ST: A spanning tree protocol for distributed MQTT brokers. *In ICC 2020-2020 IEEE International Conference on Communications* (*ICC*) (pp. 1-6). IEEE. https://ieeexplore.ieee.org/abstract/document/9149046

Pipatsakulroj, W., Visoottiviseth, V., & Takano, R. (2017, June). muMQ: A lightweight and scalable MQTT broker. *In 2017 IEEE International Symposium on Local and Metropolitan Area Networks* (*LANMAN*) (pp. 1-6). IEEE. https://ieeexplore.ieee.org/abstract/document/7972165/

Spohn, M. A. (2020). Publish, subscribe and federate! *Journal of Computer Science,* 16(7):863–870.

Spohn, M. A. (2021). An Endogenous and Self-organizing Approach for the Federation of Autonomous MQTT Brokers. In *ICEIS* (1) (pp. 834-841).

Zhang Xiang, J. (2022). Mqtt broker latency measure tool. https://github.com/hui6075/mqtt-bm-latency