Journal of Computer Science

Original Research Paper

Self-Managed Federation of MQTT Brokers with Dynamic Topology Control

Bruno Bevilaqua and Marco Aurélio Spohn

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

Article history
Received: 17-03-2023
Revised: 30-05-2023
Accepted: 07-08-2023

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

Abstract: The Message Queuing Telemetry Transport (MQTT) protocol is most used in Internet of Things (IoT) applications. The protocol implements the Publish/Subscribe (P/S) communication model. Publishers are entities providing data to a server (broker), and subscribers are those showing interest in such data. The standard MQTT scenario relies on a single broker, a potential bottleneck, and a single point of failure. The best way to scale MQTT systems is through horizontal approaches like clustering and federation. In particular, this study focuses on improving the capabilities of a self-managed federation of brokers. We present the first solution to address the dynamic management of an overlay network for the federation of autonomous brokers. The system provides the primary mechanisms for building and self-healing the federation network. We develop a new variant for the original federation protocol integrating the dynamic topology management. We present a case study as a proof of concept, showing that all designed features work as expected.

Keywords: Publish/Subscribe Communication, MQTT, Federation of MQTT Brokers, Network Topology Management

Introduction

The term Internet of Things (IoT) often refers to scenarios where network, connectivity, and computing capacity extend to objects, sensors, and everyday items not considered computers, allowing these devices to manage, exchange, and consume data with minimal human intervention (Al-Fuqaha *et al*., 2015). This technology is available in a broad spectrum of networked products, systems, and sensors, which leverage advances in computing power, electronics miniaturization, and network interconnections to deliver new features that were impossible a few years ago (Rose *et al*., 2015).

These technologies' application fields are diverse and increasingly expand to all areas of day-to-day life. The most prominent application areas include, for example, homes or buildings, which incorporate systems for monitoring electricity, gas, or water expenditures and even security systems for the environment in general. The health area has a high application in tracking patients' chronic diseases through sensors, usually part of wearable devices. One can also find IoT-based intelligent city projects focused on vehicle traffic control, lighting, and parking space control, among others (Wortmann and Fluchter, 2015).

The development of IoT-oriented applications brings the need to employ efficient communication protocols since the capacity of connected devices is often scarce both in terms of processing power and network availability. Therefore, asynchronous approaches such as the Message Queuing Telemetry Transport (MQTT) protocol (MQTT, 2023), which makes use of the Publish/Subscribe (P/S) mechanism, where publishing devices send data to a server known as a broker, which in turn distributes received messages to devices interested in receiving these messages (consumers), end up becoming critical parts in producing these applications.

As application demand grows, the need arises to scale the infrastructure. In its most straightforward implementation, MQTT uses only one broker; hence, ensuring high performance and availability become constraints since there is a single point for failures. To achieve large scale and availability, techniques such as clustering, where a load balancer works to direct requests to a set of servers, are typical. Emerging as an option for clustering, the federation of brokers, initially proposed by Spohn (2020), aims to be a self-organizing model where subscribers in different brokers interconnect employing meshes so that it is possible to do the routing of messages from publishers.

Science Publications

© 2023 Bruno Bevilaqua and Marco Aurelio Spohn. This open-access article is distributed under a Creative Commons Attribution (CC-BY) 4.0 license.

There are two recent new federation variants, the first conceived by Spohn (2021) and the second by Ribas and Spohn (2022). These solutions introduce new strategies to circumvent requirements imposed by the original approach, including an entity, the federator, that cooperates with the broker and provides all the necessary mechanisms for the self-managed federation without requiring changes in the broker.

So far, federation solutions have been built on static virtual topologies. It is possible to define any topology; however, it does not allow changes once specified, besides not handling connectivity failures from federators. This study aims to present the first solution for the dynamic treatment of the federation virtual topology. The topology management service is treated separately as a microservice to provide a scalable solution, and its coupling to the federator is minimally intrusive.

The current approach for the self-managed federation of autonomous brokers has the potential to assist its introduction to other P/S protocols, not only MQTT. The main potential federation advantage is its increased reliability when compared to clustering. Depending on the application/client, it can still be functional, even when facing node and communication failures. To this end, we can summarize our main contributions as follows:

- Self-managed virtual federation topology: It works as a microservice and can address the properties of any virtual network topology. The service provides mechanisms for creating and maintaining the virtual topology, providing means for detecting and correcting malfunctioning nodes and connections
- New federation variant: We changed the original protocol to adhere to the topology service with minimal modifications, keeping the protocol's essence unmodified

Next, we present the background needed to understand the fundamentals behind our main contribution better. After that, we offer a glimpse into the related work. Then, we present our solution for the federation of MQTT brokers with the support of dynamic topology. Finally, we give our last thoughts on the present work.

Background

MQTT Protocol

Designed to be an extremely lightweight and easy-to-implement message transport protocol, Message Queuing Telemetry Transport (MQTT) is ideal for IoT applications (MQTT, 2023).

It uses the Publish/Subscribe (P/S) mechanism, in which a publisher member sends messages to a specific topic. Another member, called the consumer, indicates its intention to receive messages from this topic when it subscribes to the same topic (Soni and Makwana, 2017). Figure 1, publishers and subscribers are unaware of each other. They use a broker mediator, which acts as a bridge connecting both. Its function is to filter the incoming messages, organize them into topics, and distribute them to their subscribers (Soni and Makwana, 2017).

The MQTT protocol provides three Quality-of-Service (QoS) levels, which act as an arrangement between the two parties (producers and consumers) concerning message delivery guarantees. The levels are:

- QoS 0: Sends every message at most once, with no delivery guarantee (i.e., best effort). It is also known as "At most once"
- QoS 1: Sends every message at least once, and duplicate deliveries are possible. It is also known as "At least once"
- QoS 2: Known as Exactly Once, uses a four-way handshake to send a message exactly once

When data traffic between publishers and subscribers is noteworthy and increasing, a simple deployment containing one broker may configure a bottleneck besides being a single point of failure. Vertical and horizontal scalability can ensure system operation within the minimum quality standards.

Vertical scalability resorts to increasing a server's computing resources, allowing more simultaneous clients. In addition to boosting machine resources, multiple brokers can be started on the same machine to increase message transmission flow.

This scalability does not provide high availability because it relies on a single machine, still setting itself up as a single point of failure. To overcome such limitations, there is horizontal scalability. This strategy has as its central rationale spreading client demand over multiple machines. Its main advantage is elastic availability since it no longer has a single point of failure.

To scale MQTT horizontally, the concept known as clustering is an option (Fig. 2). A load balancer works as an access point for brokers and is responsible for choosing which server will handle new clients. Such a decision results from applying metrics to distribute the demand evenly among the servers.

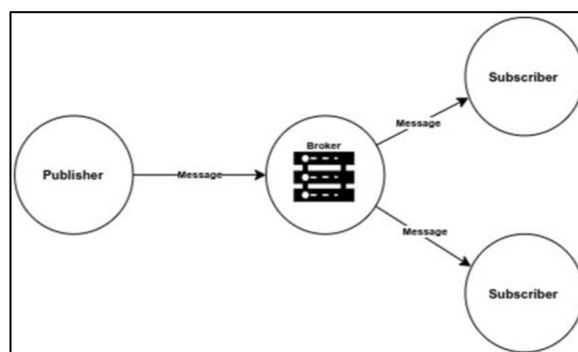

Fig. 1: MQTT protocol

DOI: 10.3844/jcssp.2023.1398.1409

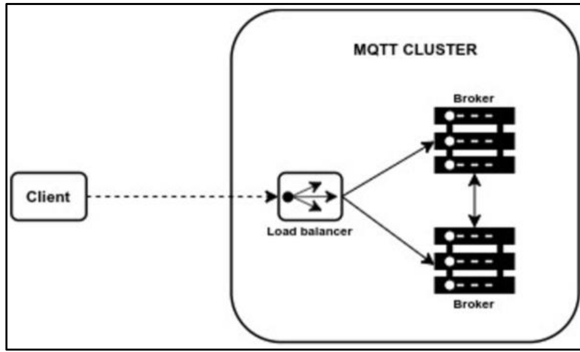

Fig. 2: MQTT cluster

In cluster formation, brokers have to communicate with each other to keep any crucial data synchronization. Publishers and subscribers may connect to different brokers, requiring the proper routing of topic messages between brokers. Despite being more effective than vertical scaling, clustering can present two main areas for improvement: Dependence on the load balancer and a possible stutter in communication between brokers.

Spohn (2021) proposed a self-organizing federation for brokers connecting through an overlay network, building meshes for linking subscribers. Creating meshes starts when the first subscriber for a particular topic connects to a federated broker. If there is yet to be a mesh for this topic, the broker advertises itself as the core for the new mesh. Core announcements are broadcast throughout the overlay network so that all federated brokers learn how to reach the core. This process entitles keeping the necessary routing information (i.e., next hop and distance to the core) and compliance with the required mesh redundancy (i.e., multiple paths to the core, if the overlay topology allows that). In case numerous nodes simultaneously announce themselves as cores, the process converges by electing the core with the smallest (or largest) ID.

While there is just the core itself, there has yet to be a proper mesh. The mesh starts building when new subscribers for the same topic connect to other nodes. Joining the mesh happens by sending a mesh membership toward the core. The membership message travels toward the core by making intermediate nodes mesh members or until it reaches a mesh member. Figure 3 depicts an example of the mesh creation process: A subscriber at node one makes it advertise itself as the core for the related topic; in a second moment, a subscriber at node five requires it to join the corresponding topic mesh.

As for the routing of messages sent by publishers, there are two possible cases. In the first one, the publisher's broker is in the mesh, resulting in the broker sending the message to all neighboring mesh members. In the second case, the publisher's broker sends the message to the next hop toward the related topic's core. Upon reaching a mesh member, the message spreads throughout mesh member nodes, as for the first case. Nodes avoid looping in the mesh by keeping a local cache for recently forwarded messages. Figure 4 depicts an example of the routing process in the federation: In the first situation, a client publishes at node four, which is a mesh member, making routing straightforward through the mesh; in the second case, a client starts publishing from a node outside the mesh (node zero), which requires first routing the publications towards the core (node one).

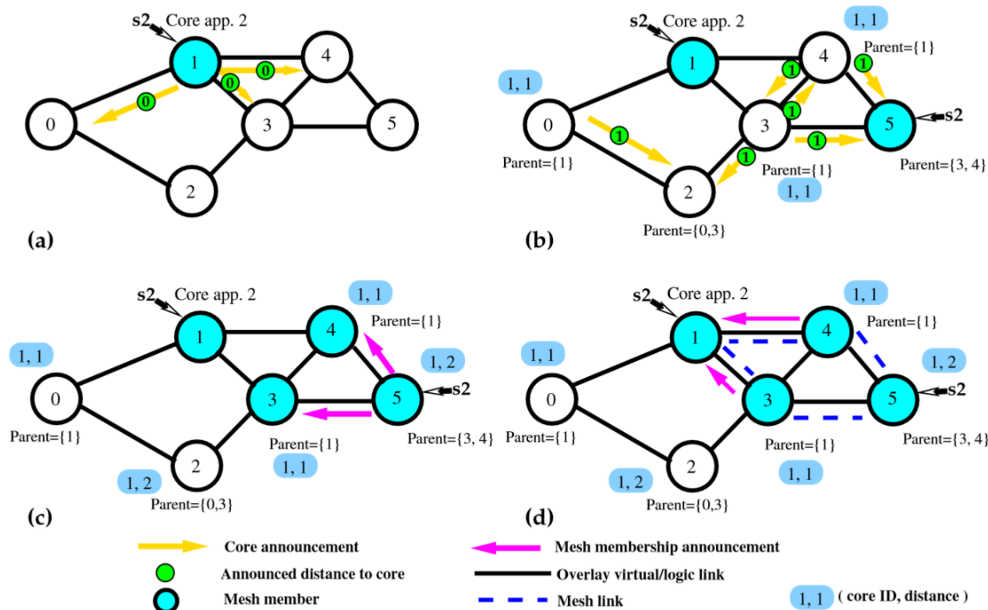

Fig. 3: Mesh creation process (Spohn, 2020)

DOI: 10.3844/jcssp.2023.1398.1409

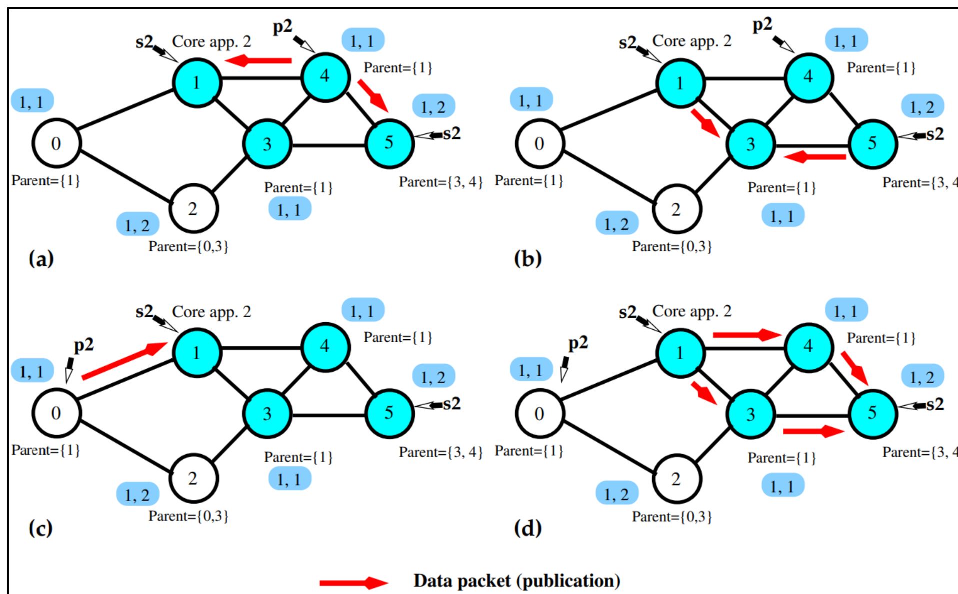

Fig. 4: Message routing process (Spohn, 2020)

The advantages of adopting the autonomous federation, according to the author, are:

- There is no single point of failure: Clients can choose to join any federated brokers
- Load balancing: There is always the possibility to choose from a set of available brokers and get what one needs
- Exploration of virtualized topologies or network capabilities: The complete virtualized deployment is achievable through agents instantiated in virtual machines or containers

System Architecture

The architecture of a system defines its structure and behavior, allowing systems to evolve while supplying a particular level of service throughout their lifecycle. In software engineering, architecture is concerned with providing a bridge between system functionality and the quality attribute requirements that the system must meet (Alshuqayran *et al*., 2016).

The monolithic architecture standard is distinguished as an application model in which the modules cannot execute independently. Although more typical, as an application grows, it is more difficult to maintain and evolve due to its complexity. Tracking bugs requires long reads through the code base, and any external dependency makes it a cumbersome task when adding or updating libraries (Dragoni *et al*., 2017). For large projects, rebooting may result in considerable downtime, making project development, testing, and maintenance difficult. In addition, monolithic applications present a technological lock-in for developers, who must use the same programming language and structures defined at the beginning of development (Dragoni *et al*., 2017).

As the monoliths grow, the demand for machine resources tends to grow together, thus requiring the application of scalable solutions that often become unfeasible due to the high complexity of the software.

Microservices were first introduced in 2011 at a software architecture workshop to describe participants' common ideas on architectural patterns. More recently, leading software consulting and product design firms have discovered that the microservices approach is a compelling architecture that enables teams and organizations to be more productive overall and often create more successful software products (Alshuqayran *et al*., 2016). Companies like Amazon, Netflix, eBay, and LinkedIn have used this architecture to deploy their extensive services through small components.

Figure 5 illustrates, compared to the monolithic architecture, microservices must be independent components conceptually deployed in isolation and equipped with dedicated memory persistence tools (e.g., databases), resulting in a distributed application. As all components of a microservices architecture are microservices, their differential derives from the composition and coordination of their components through messages (Dragoni *et al*., 2017).

Resulting from a combination of Service-Oriented Computing (SOC) and Service-Oriented Architectures (SOA), the microservices architecture was developed by abstracting complexity levels essential for its predecessors so that developers can focus only on programming simple and effective systems.

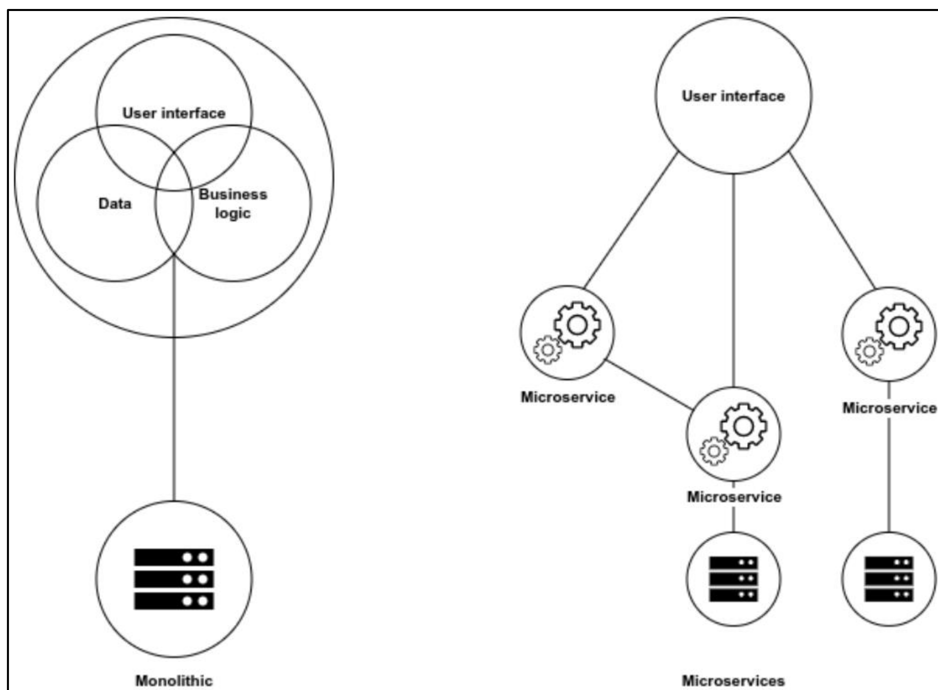

Fig. 5: Monolithic architecture versus microservices architecture

According to Alshuqayran *et al*. (2016); Dragoni *et al*. (2017), the microservice architecture increases the team's productivity since developers do not need to act on all system parts, allowing each team to focus only on specific modules. The application of automated tests and monitoring and tracking failures can also have their processes streamlined. The fact that the environments are isolated and independent enables continuous integrations and deliveries, making the system as a whole more reliable, in addition to making it highly scalable and easy to install and maintain. Using communication via messages, developers can utilize multiple technologies, employing the ones that best suit each microservice.

Adopting this architecture, on the other hand, maybe a complex task. Decomposing a monolithic application and identifying parts that can be modularized, especially in applications with legacy parts, can be a significant challenge, as it is the definition of communication standards.

Communicating parts of a monolithic system strictly can happen through database relationships. In contrast, in the microservices architecture, where all modules have their database, it is necessary to connect them by operating appropriate communication standards that can be classified as synchronous and asynchronous (Aksakalli *et al*., 2021).

Using models such as REST based on HTTP, in the synchronous approach, when a client requests the service, it blocks the client until it gets a reply. This model requires that the service be active; otherwise, if the client does not receive a response, the same should be notified (Aksakalli *et al*., 2021).

In asynchronous communication, message queues are usually the option. In this model, based on event-oriented architecture, the client sends a message that can be a request and only blocks its processing once it receives the answer because, most of the time, the client does not need this response. There are still cases in which both methods coexist, resulting in a hybrid model.

Although it is easy to deploy an application in the monolithic approach, deploying systems based on the microservices architecture can become a challenge, especially when thousands of modules make up a system, not to mention cases where the same service is scaled numerous times due to high demand.

Cloud computing emerges as an ally to microservices, enabling it to scale applications to virtual servers as it can dynamically adjust its computing resources (Aksakalli *et al*., 2021). Platforms such as Amazon Web Services, Microsoft Azure, and Google Cloud provide their resources by charging on demand, allowing the infrastructure to scale dynamically along with the growth of the application, thus avoiding initial costs with proprietary infrastructures and data centers.

According to Aksakalli *et al*. (2021), there are numerous deployment patterns, among which there is a service instance per Virtual Machine (VM), where each service is packaged as a VM image, allowing the creation of isolated environments. Another pattern concerns service instances per container. Containers are virtualization mechanisms that run at the operating system level and can be limited to consuming only selected

resources. Each server, whether physical or virtual, can run numerous containers. This model also allows orchestrators like Kubernetes or Docker Swarm to automate containerized applications' deployment, improving scalability and management.

Related Works

The original version (Spohn, 2020) for the Federation of Autonomous Brokers requires changing the MQTT broker. Intending to overcome this need, (Spohn, 2021) proposed a new federation model introducing the entity concept of federator.

In this model, an application (i.e., federator) leveraging the P/S communication model works with a broker to build and maintain topic meshes. Making the changes directly in the broker could result in better performance; however, with the possibility of virtualizing computing and communication resources, deploying brokers in containers connecting with each federation would be a strategy to attain the performance needs.

Figure 6 depicts the two main layers in the federation application. The Pub_Fed layer is responsible for publishing control topic messages and routing messages to neighboring brokers. In contrast, the Sub_Fed layer is responsible for receiving and processing incoming publications of neighboring brokers. The topic control messages are:

- CORE_ANN: Include information regarding core announcements
- MESH_MEMB_ANN: Used for joining a mesh (i.e., mesh membership announcement)
- NEW_REGULAR_TOPIC: Used to inform the federation regarding a new subscription or first publication to a regular topic
- DATA: Include regular topic messages for routing between neighboring brokers as payload

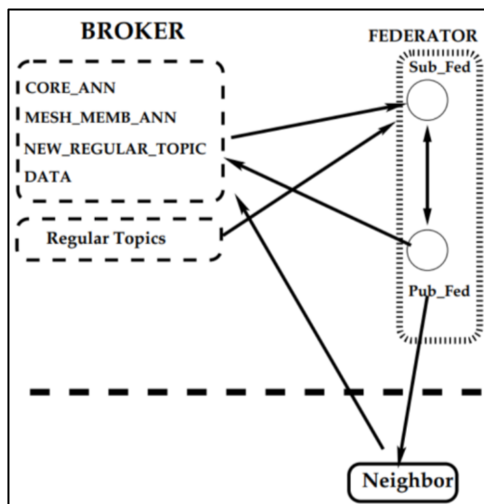

Fig. 6: MQTT federator (Spohn, 2021)

The federation requires application developers to explicitly communicate the topic details through the control topic NEW_REGULAR_TOPIC to get aware of regular subscribers and publishers. The message contains all the topic metadata as defined by the MQTT protocol.

A case study shows how the solution works, being more merely a proof of concept than a performance evaluation. The overlay network connecting the federation is static and part of the federation configuration. Therefore, the federation does not handle node failures or disconnections once deployed.

Based on the federation proposed by Spohn (2021); Ribas and Spohn (2022) proposed changes to the original architecture, giving rise to a new federation variant. The federation was developed using the Rust language, the MQTT Paho client library, and the Tokio runtime (Lerche, 2022), which manages tasks as asynchronous processing units. Such units are much lighter to handle than system threads.

In their implementation, (Ribas and Spohn, 2022) introduced the concept of topic workers. Each topic worker is responsible for managing the mesh for a particular topic; therefore, each federated topic has an associated worker. Thus, the federation creates workers dynamically as it learns about new federated topics.

Figure 7 displays the federator architecture. The federation forwards federated or control topics to a dispatcher component, which identifies and delivers the message to the corresponding topic worker.

To exploit all the available redundancy, publications from a broker not participating in the mesh are forwarded to all available parents toward the core, unlike the original federation approach based on unicasting the message towards the core/mesh.

Federated topics receive a federated prefix as standard terminology. By employing the MQTT multilevel wildcard feature, the federator needs to subscribe to the "federated" wildcard topic to intercept all topics in the federation context. The new variant supplies the following control topics:

- *Federator/core_ann/*: To receive core announcements
- *Federator/memb_ann/*: To obtain mesh join announcements
- *Federator/routing/*: For routing federated publications
- *Federator/beacon/*: For receiving beacons reporting the existence of local subscribers

Another significant improvement of this variant is that it employs a more subtle way to notify the federation about new regular topic subscribers. The control topic *federator/beacon/* is the channel for such notifications by appending the federated topic identifier to the root topic name. For instance, the subscriber regularly publishes the topic *federator/beacon/door_sensor* for a regular topic named *door_sensor*. As the federator subscribes to the control topic, it gets all the beacons regularly, having supervision of all active subscribers.

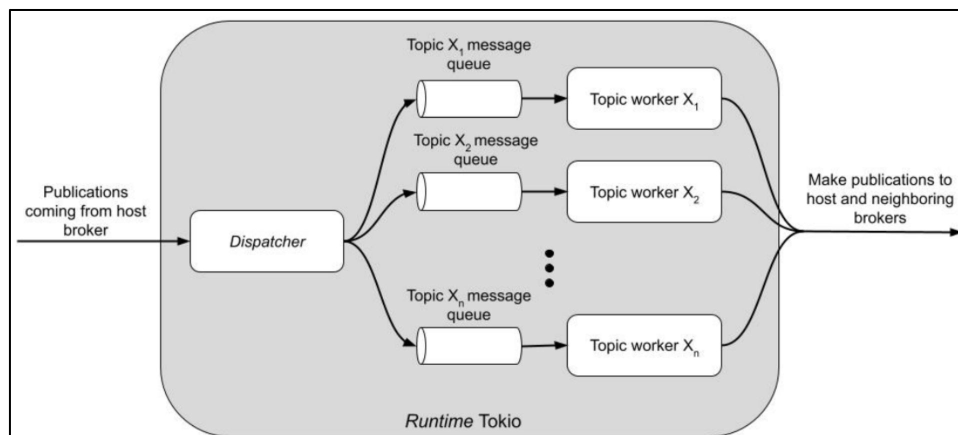

Fig. 7: MQTT federation architecture (Ribas and Spohn, 2022)

A more recent variant of the original federation approach is presented by de Lacerda Machado *et al.* (2023). The work, implemented in Python, provides a federation that monitors the log system of a Mosquitto broker (Eclipse Foundation, 2023a). Upon detecting a new client (publisher or subscriber), it proceeds to the original federation protocol by Spohn (2020). The federator works as a wrapper to the broker, communicating directly to neighboring federators through UDP. The authors present a case study showing the federation deployment based on LXC containers. One limitation of their proposal is that it relies on Mosquitto's log system, making its use with other brokers an open issue.

Materials and Methods

This section presents the main tools and the system's architecture regarding the microservice and the federation. The microservice, responsible for creating and maintaining the topology of the brokers' federation, employs algorithms for finding neighbors for an incoming node, monitors the topology at runtime, and stores topology information in a database. The federation, in turn, provides the mechanisms for performing the broker federation protocol.

Tools and Dependencies

In part, the realization of our solution resorts to existing tools and dependencies. We summarize them as follows:

- Container: Docker (2023) is an operating system virtualization capability providing fully isolated environments called containers. A container groups all the software and its dependencies, speeding up the development and deployment processes. Meanwhile, containers provide a safe execution environment on different machines
- Database: MongoDB (2023) is a document-oriented database software classified as NoSQL. Its document model fosters its use, saving development and maintenance time
- MQTT broker: Developed by the Eclipse
- Foundation, Mosquito (Eclipse Foundation, 2023a) is a broker that implements the MQTT protocol. It is light, reliable, and fit for low-performance boards and large servers
- MQTT development library: Eclipse Paho (Eclipse Foundation, 2023b) is a multi-language open-source library that provides the mechanisms for clients connecting to MQTT brokers and taking advantage of its functionalities
- MQTT benchmark tool: With the primary function of running benchmarks on brokers, the MQTT broker latency measure tool (Jianhui and Xiang, 2023) has various configurations for gathering performance metrics. The tool is implemented in Go, making adding it as a package to our system easier

System's Architecture

Both microservice and federation applications were developed using the Go programming language and are available as Docker images.

Microservice

We employ the hybrid communication model to implement the microservice, combining synchronous and asynchronous communication. We use the REST API's model to support synchronous communication. In contrast, we use MQTT clients for asynchronous communication that publish messages directly to the broker federator host through the control topic *federated_topology_ann*. We use a MongoDB database to store topology information in the data layer.

DOI: 10.3844/jcssp.2023.1398.1409

The microservice uses two algorithms for creating and maintaining the overlay topology. The first is responsible for finding neighbors for an incoming node; the other, named health check, is responsible for monitoring the topology at runtime.

The microservice provides an HTTP endpoint (*/api/v1/join*) to receive requests from federators: A request comprises a POST message containing a JSON object with the URL for the federator's broker. Listing 1 shows an example of that.

```
curl --location --request POST 'http
   ↪ ://127.0.0.1/api/v1/join' \
    --header 'Content-Type:
      ↪ application/json' \
    --data-raw '{"ip": "tcp
      ↪ ://127.0.0.1:1883"}'
```

Listing 1: A federator join request

The microservice always seeks to connect new nodes to two others; however, while there are not enough nodes to cover this requirement, the two federations joining first will be related. Each node receives an identifier based on the order of entry in the topology; this identifier, together with other data referring to the node, is stored in the database through a document with six fields:

- ID: Identifier of the federator
- IP: URL for connection to the broker's federator host
- Neighbors: Array of objects containing each neighbor's ID and IP address
- NeighborsAmount: Number of neighbors
- Latency: Metric collected by health check
- LatestHealthCheck: Timestamp of the last health check run for that node

In Listing 2, it is possible to visualize a document containing a federator's data.

```
{
    "id": 0,
    "ip": "tcp://127.0.0.1:1883"
    "neighbors": [{
        "id": 1,
        "ip": "tcp://127.0.0.1:1884"
    }],
    "neighborsAmount": 1,
    "latency": 0.10,
    "latestHealthCheck": "2023-01-01
       ↪ 00:00:00.000Z"
}
```

Listing 2: Data for a federator with one neighbor

After inserting the first node, the health check algorithm scans all nodes every five seconds by computing the latency between submitting a publication and its response. This metric works as a criterion for inserting new federators.

From the moment at least two nodes are in the federation overlay network, the microservice starts to carry out the process represented in Figure 8 when inserting new nodes. Upon receiving a join request, the microservice searches for two candidate nodes. The first chosen node will have the lowest latency and number of neighbors lower than the maximum redundancy configured for the topology. In contrast, the service includes the second one with the smallest number of neighbors among all nodes.

The health check service also maintains the topology (Fig. 9). We consider a node disconnected if the metric gathering fails twice sequentially for the same federation. On identifying the failure, using the asynchronous connection, the microservice sends a message to the node's neighbors informing them that it is no longer available. Suppose any neighbor gets disconnected from the federation due to the failing node. In that case, the microservice will obtain new neighbors for the disconnected node, relocating it to resume the connection with the federation.

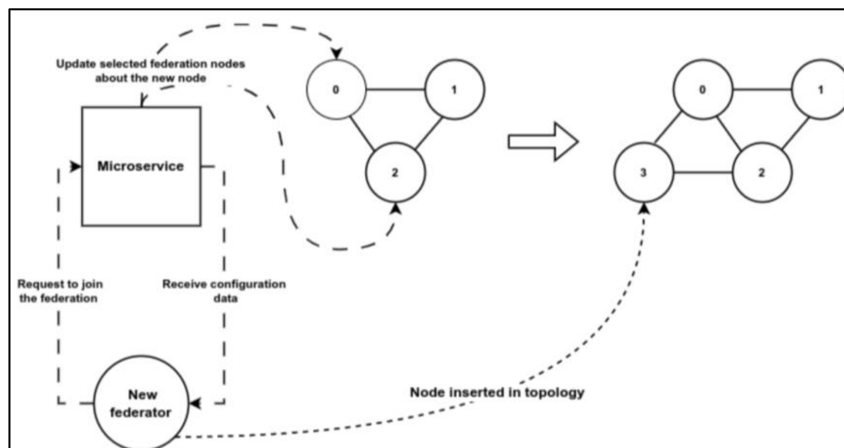

Fig. 8: Insertion of a new node in the topology

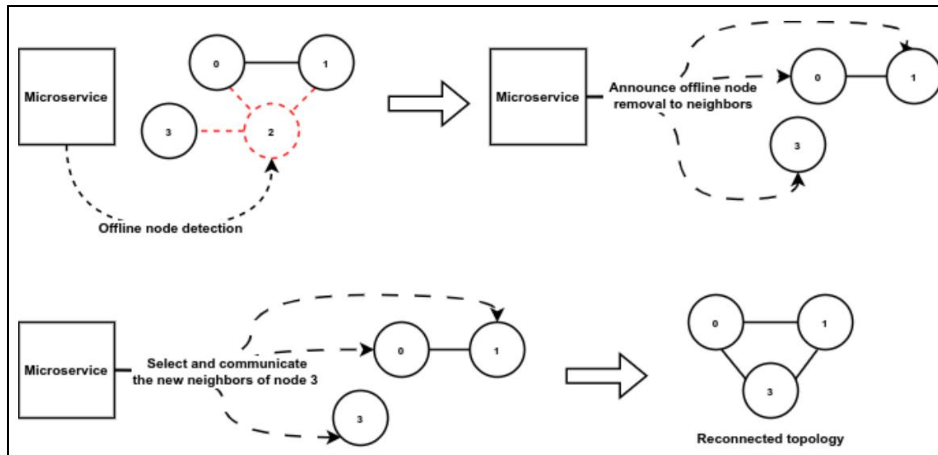

Fig. 9: Reconnection of a node via the health check mechanism

Federator

In developing the federator, we used the same structure proposed by Ribas and Spohn (2022). At this stage, the federator underwent a translation of the original code written in the Rust programming language to the Go programming language. Even so, changes were made in the federator, making it possible to integrate the functionalities contemplated by the microservice.

Initially, we added to the federator the ability to make HTTP requests to the microservice so that it can fetch the necessary information for its configuration. In addition, the federator also subscribes to the new *federated_topology_ann* control topic through which the microservice sends topology announcements. Each advertisement contains instructions for adding or removing a neighbor so the federator can connect or disconnect to another federator when required.

Configuration

The configuration of both applications happens through environment variables. For the federation, there are only two configuration parameters:

- TOPOLOGY_MANAGER_URL: URL to connect to the microservice
- ADVERTISED LISTENER: URL for external connection to federation host broker

The microservice, in addition to its settings, also centralizes the federation settings, requiring more configuration parameters:

- MONGO_URL: MongoDB database connection URL
- CORE_ANN_INTERVAL: Time interval between core announcements
- BEACON_INTERVAL: Defines the interval between beacons received by subscribers
- FED_REDUNDANCY: Defines the mesh redundancy instantiated by the federation
- TOP_MAX_REDUNDANCY: Defines the overlay topology's maximum node degree/redundancy

The example in Listing 3 refers to a file in YAML format, which serves as a template for executing a federation instance utilizing the Docker Compose tool. The file includes data for running three containers, referring to the database, microservice, and federation and their configuration parameters.

```
version: '3'
services:
  mongodb:
    image: mongo:latest
    environment:
      - MONGO_INITDB_DATABASE=root

  microservice:
    image: brunobevilaquaa/fed-topology-manager
    depends_on:
      - mongodb
    environment:
      - MONGO_URL=mongodb://mongodb:27017/root
      - CORE_ANN_INTERVAL=2s
      - BEACON_INTERVAL=2s
      - FED_REDUNDANCY=2
      - TOP_MAX_REDUNDANCY=5

  federator:
    image: brunobevilaquaa/mqtt-fed
    depends_on:
      - microservice
    ports:
      - '1883:1883'
    environment:
      - TOPOLOGY_MANAGER_URL=http://topology-
        ↪ manager:8080
      - ADVERTISED_LISTENER=tcp://mqtt-fed:1883
```

Listing 3: Configuration for the deployment of the system's containers

DOI: 10.3844/jcssp.2023.1398.1409

Results and Discussion

Two deployment plans comprise a case study, the first using cloud computing and the second using a local network. For both scenarios, the microservice is responsible for creating the federation topology.

When using cloud computing, we seek to explore environments found in real deployments, with the primary objective of building a scenario for experimenting with federation availability. With this test, it is possible to visualize how the microservice adjusts the topology when a federation is disconnected. We build the scenario using AWS, Azure, and Google Cloud computing resources. A total of 10 virtual machines run on the cloud, nine for the federation and one for the microservice, all arranged to run in different geographic regions spanning the continents of America, Europe, Africa, and Asia. For the test, we initially gathered the topology rendered by the microservice and then stopped the execution of two federations to simulate their disconnection.

The test in a local network seeks to create a scenario for performance testing. It is possible to obtain more assertive metrics because it is a more controlled and interference-free environment concerning network and computing resources. We configured the microservice to render topologies with a maximum redundancy of five, while the federation mesh has a redundancy of three.

The resulting scenario contained twelve federators, generating the topology illustrated in Fig. 10. Two subscribers were positioned in federators 0 and 5, respectively. A publisher responsible for sending 1000 messages, each with 64 bytes, was placed on node 11, one hop from federation 0 and three hops from federation 5.

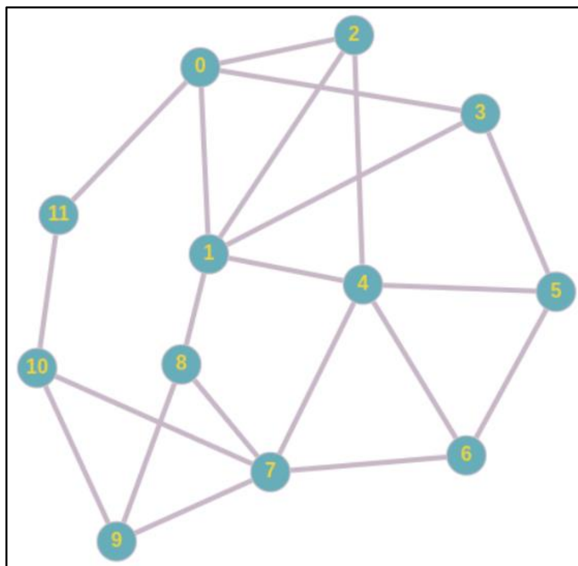

Fig. 10: Topology used in the LAN scenario

The topology initially obtained in the scenario using cloud computing is apprised in Fig. 11. It is possible to note that node 8 has the smallest number of neighbors in the topology, which makes it an easier target for isolation; therefore, when its neighbors (i.e., 0 and 7) stop running, a reallocation is necessary.

As shown in Fig. 12, the microservice chooses new neighbors for Federator 8 when reallocating. It selects Federator 3 with the highest performance, which is possible since it has the most significant number of connections. Then, it designates Federator 1 to keep the redundancy requirement.

Table 1 depicts the results for the LAN scenario (topology shown in Fig. 10). When comparing the average latency of publications for the two subscribers, it is possible to notice that the federator farthest from the publisher has a more significant latency. When analyzing these results, the overhead of publications rendered during retransmissions should be considered a factor contributing to a considerable increase in latency.

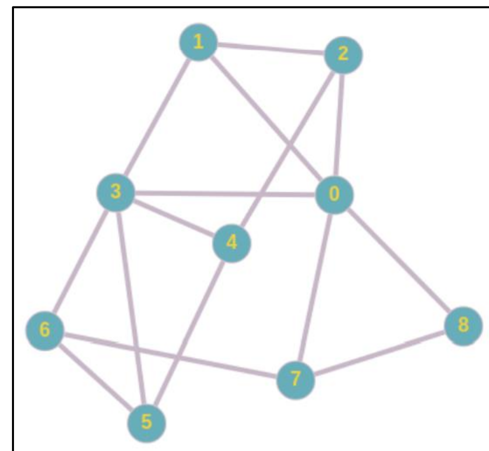

Fig. 11: Topology before relocating node 8

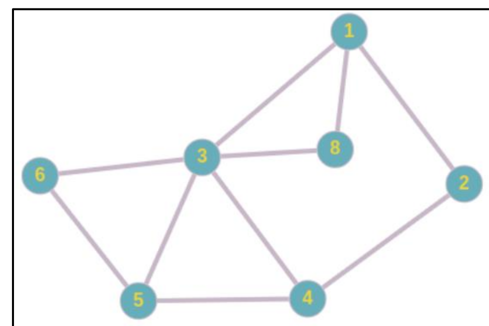

Fig. 12: Topology after relocating node 8

Table 1: Latency of Federation publications

Messages	Subs. at broker 0	Subs. at broker 5
1000	445.796 *ms*	1062.230 *ms*

Conclusion

MQTT is likely the most used protocol in the design of IoT applications. IoT is reaching many ecosystems, handling numerous devices and users. Therefore, applications require a scalable MQTT service, having clustering and federation as the best candidates to handle an elastic demand of clients. This study focuses on a self-managed federation of MQTT brokers, mainly in the dynamic creation and management of the overlay network connecting federated brokers.

We introduce the first solution for creating and maintaining a federation of autonomous brokers in a dynamic topology. The design follows the microservice model, in which the topology management occurs as an independent service. Our previous solution for the federation evolved to adhere to the new service, keeping the federation essence working as in the original proposal. However, only some adjustments are in place to allow federation nodes to join the federation overlay network.

Another improvement concerns handling failures in the overlay network. The previous solutions used a static overlay network, where nodes receive their configuration relating to neighboring peers. Once deployed, the federation network does not change, without any mechanism for handling failing or disconnected nodes. In our solution, a new federation node requests its entry via the microservice, receiving all the information necessary to start communicating with the neighboring nodes, as defined by the microservice. A health check mechanism is in place to detect any disconnected nodes and rearrange the federation topology when necessary.

We presented a case study as a proof of concept. Results show that our solution works as planned along with the evolved federation protocol. The topology management is decoupled from the federation mechanisms, requiring changes to all the topology parameters (e.g., node insertion criteria, topology degree) only in the microservice. We plan to explore how topologies can adapt to particular application/client requirements in future work.

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.

Funding Information

The *Universidade Federal da Fronteira Sul* partially funded this work (Research Project PES-2021-0471, under call 121/GR/UFFS/2021, and Research Project PES-2022-0069, under call 89/GR/UFFS/2022).

Author’s Contributions

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

Bruno Bevilaqua: Contribution to conception and design. He performed the implementation and the case study and contributed to the article's draft.

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

Aksakalli, I. K., Çelik, T., Can, A. B., & Tekinerdoğan, B. (2021). Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, *180*, 111014. https://doi.org/10.1016/j.jss.2021.111014

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols and applications. *IEEE Communications Surveys Tutorials*, *17*(4): 2347-2376. https://doi.org/10.1109/COMST.2015.2444095

Alshuqayran, N., Ali, N., & Evans, R. (2016, November). A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)* (pp. 44-51). IEEE. https://doi.org/10.1109/SOCA.2016.15

de Lacerda Machado, J. F., Spohn, M. A., and Granville, L. Z. (2023). Client-transparent and self-managed MQTT broker federation at the application layer. In *International Conference on Computing, Networking and Communications (ICNC 2023)*, Honolulu, USA. https://doi.org/10.1109/ICNC57223.2023.10074556

Docker, (2023). Docker develop faster. Run anywhere. https://www.docker.com

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today and tomorrow. *Present and Ulterior Software Engineering*, pages 195-216.

Eclipse Foundation. (2023a). Eclipse Mosquitto, an open source MQTT broker. https:// mosquitto.org

Eclipse Foundation. (2023b). Paho. https://www.eclipse.org/paho

Jianhui, & Xiang, Z. (2023). MQTT broker latency measure tool. https://github.com/hui6075/mqtt-bm-latency

Lerche, C. (2022). Tokio api docs.

DOI: 10.3844/jcssp.2023.1398.1409

https://github.com/tokio-rs/tokio

MongoDB, (2023). MongoDB. https:// www.mongodb.com

MQTT, (2023). MQTT the standard for iot messaging. *Oriented Computing and Applications (SOCA)*, pages 44-51. IEEE. https://mqtt.org/

Ribas, N. K., & Spohn, M. A. (2022). A new approach to a self-organizing federation of MQTT brokers. *Journal of Computer Science*, *18*(7): 687-694. https://doi.org/10.3844/jcssp.2022.687.694

Rose, K., Eldridge, S., & Chapin, L. (2015). The internet of things: An overview. *The Internet Society (ISOC)*, 80: 150.

Soni, D., & Makwana, A. (2017). A survey on MQTT: A protocol of Internet of Things (IoT). In *International conference on Telecommunication, Power Analysis and Computing Techniques (ICTPACT-2017)*, *20*: 173-177. https://doi.org/ 10.1109/ICTPACT.2017.8273112

Spohn, M. A. (2020). Publish, subscribe and federate! *Journal of Computer Science*, *16*(7): 863-870. https://doi.org/10.3844/jcssp.2020.863.870

Spohn, M. A. (2021). An endogenous and self-organizing approach for the federation of autonomous MQTT brokers. In *ICEIS (1)*, pages 834-841. https://doi.org/10.5220/0010408808340841

Wortmann, F., & Fluchter, K. (2015). Internet of things. *Business & Information Systems Engineering*, *57*(3): 221-224. https://doi.org/10.1007/s12599-015-0383-3