UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOSÉ FERNANDO DE LACERDA MACHADO JR.

Client-Transparent and Self-Managed MQTT Broker Federation at the Application Layer

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Lisandro Zambenedetti
Granville
Coadvisor: Prof. Dr. Marco Aurélio Spohn

Porto Alegre
June 2023

CIP — CATALOGING-IN-PUBLICATION

de Lacerda Machado Jr., José Fernando

Client-Transparent and Self-Managed MQTT Broker Federation at the Application Layer / José Fernando de Lacerda Machado Jr.. – Porto Alegre: PPGC da UFRGS, 2023.

58 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Lisandro Zambenedetti Granville; Coadvisor: Marco Aurélio Spohn.

1. MQTT. 2. Self-managed networks. 3. IoT. I. Granville, Lisandro Zambenedetti. II. Spohn, Marco Aurélio. III. Título.

“The saddest aspect of life right now is that science gathers knowledge faster than society gathers wisdom.”

— ISAAC ASIMOV

ACKNOWLEDGMENTS

To my Grandmother, who passed away during the development of this work and dedicated 62 years to teaching.

To my family, without who this work would have been finished a year (or more) earlier.

To Prof. Granville and Prof. Spohn, for being patient with all the ups and downs, for the ear pulls, and for all the knowledge shared.

ABSTRACT

Scalability in messaging systems remains an open topic. Regular solutions present clusterized approaches, which can be very scalable for high-throughput systems but still rely on a unique orchestrator which is a single point of failure. On the other hand, a few solutions, mainly commercial products, provide scalability based on federation approaches, which means the solution's robustness relies on being distributed and highly fault-tolerant. Spohn (Spohn, 2020) presented an innovative solution based on a federation approach and being self-managed, which became the foundation of this work. On that, it is presented a Python-written wrapper for Mosquitto MQTT brokers providing federation capabilities with self-managed characteristics. The wrapper is client-transparent and self-managed, being capable of attaching to the MQTT Mosquitto broker without any significant customization, only with a minor tune on the configuration to allow the log output to be diverted, and can also deal with topology changes without supervision.

Keywords: MQTT. self-managed networks. IoT.

Federação de Brokers MQTT Transparentes ao Cliente e Auto Gerenciada em Nível de Aplicação

RESUMO

Escalabilidade em sistemas de mensageria ainda é um tópico a ser explorado. As soluções existentes utilizam, principalmente, estratégias de clusterização, o que torna essas soluções adequadas para sistema com grandes fluxos de dados mas continuam dependendo unicamente de um orquestrador, que caracteriza um ponto único de falha. Por outro lado, algumas soluções, normalmente produtos comerciais, oferecem escalabilidade baseada em federação, sendo as principais características de robutez dessas soluções o fato de serem distribuídas e altamente tolerante a falhas. Spohn (2020) apresentou um solução inovadora baseada em federação, também sendo auto-gerenciada, que é o principal fundamento deste trabalho. Então, apresenta-se aqui um wrapper escrito em Python utilizado em conjunto com brokers MQTT Mosquitto, possibilitando que estes trabalhem de maneira federada e auto-organizada. Este wrapper é totalmente transparente ao cliente e pode ser anexado ao broker MQTT sem necessidade customização, apenas com pequenos ajustes de configuração de saída dos logs da aplicação.

Palavras-chave: MQTT, agentes federados, rede auto-gerenciada.

LIST OF ABBREVIATIONS AND ACRONYMS

ACL	Access Control List
FBR	Filter-Based Routing
GB	Gigabyte
IBM	International Business Machines
IoT	Internet of Things
IP	Internet Protocol
ISO	International Standardization Organization
M2M	Machine-to-Machine
MANET	Mobile Ad Hoc Network
MQTT	MQ Telemetry Transport (former Message Queue)
OSI	Open Systems Interconnection
P/S	Publish/Subscribe
PIP	Packet Installer for Python
PUMA	Protocol for Unified Multicast Routing
RADIUS	Remote Authentication Dial-In User Service
RAM	Random Access Memory
RMTP	Reliable Multicast Transport Protocol
RFID	Radio Frequency Identification
SSL	Secure Socket Layer
TCP	Transport Control Protocol
TINA	Telecommunications Information Networking Architecture
UDP	User Datagram Protocol

LIST OF FIGURES

Figure 4.1 Main Functions 33
Figure 4.2 Support Functions 36

Figure 5.1 Validation Topology, extracted from [Spohn 2020] 41
Figure 5.2 Console image of federated brokers 42
Figure 5.3 Six-node topology 43
Figure 5.4 Fifteen-node topology 44
Figure 5.5 Twenty-five-node node topology 45

LIST OF TABLES

Table 4.1 Message types 37

CONTENTS

1 INTRODUCTION **11**
2 BACKGROUND **14**
2.1 The Internet of Things **14**
2.1.1 Definition 14
2.1.2 Architecture and Reference Models 16
2.2 Protocols & MQTT **18**
2.2.1 MQTT 19
2.3 Scalability **20**
3 RELATED WORK **22**
3.1 General Works and First Efforts Towards Scalability **22**
3.1.1 InfoBus Repeater: A Secure and Distributed Publish/Subscribe Middleware 22
3.1.2 The Federation of Critical Infrastructure Information via Publish-Subscribe Enabled Multisensor Data Fusion 23
3.1.3 Towards Scalable Publish/Subscribe Systems 24
3.2 Foundation Works **25**
3.2.1 Publish, Subscribe, Federate! 26
3.2.2 Federação de Brokers do Protocolo MQTT - Implementação e Análise de Desempenho 27
4 PROPOSAL AND IMPLEMENTATION **29**
4.1 Proposal **29**
4.2 Resources Overview **30**
4.2.1 Mosquitto 30
4.2.2 Python 31
4.2.3 Paho 31
4.3 Model and Implementation **32**
4.3.1 Model 32
4.3.1.1 Main Group Functions 33
4.3.1.2 Support Group Functions 35
4.3.1.3 Message Types 36
4.3.2 Implementation 38
5 VALIDATION AND PERFORMANCE TESTS **40**
5.1 Validation **40**
5.2 Performance Tests **43**
6 CONCLUSION **46**
6.1 Open Topics and Future Work **47**
REFERENCES **48**
APPENDIX A — RESUMO EXPANDIDO **51**
APPENDIX B — PUBLISHED PAPER - ICNC 2023 - QUALIS A2 .... **53**

1 INTRODUCTION

Modern automation and telemetry solutions became an essential part of modern life. Public safety, transportation, wearable gadgets, home, commercial and industrial automation, and a myriad of possible solutions based on sensors and small equipment populate the ecosystem of connected devices. Initially, those environments were called *Connected Devices (CO)* and later were named *Internet Of Things (IoT)* [Zouganeli e Svinnset 2009]. IoT devices gather and generate, mainly, small data that can be used to integrate information systems, delivering great value to information systems.

The data generated and gathered in IoT environments require an optimized infrastructure to be stored, commuted, and processed. As the complexity of the environment grows, there is a need for middleware to manage the data. Distributed solutions and specific protocols ease the integration and provide additional resources. However, another variable in this equation also needs to be addressed: the limitation of individual computational resources of the equipment involved [Castellani et al. 2010].

Based on those points, addressing scalability is strategic. Adding more devices to a given environment grows the amount of data collected and, consequently, the need for computational power to deal with it. Transporting, storing, and processing requires additional computational resources that a single device may not deliver, so there is a need to use differentiated strategies for allowing data to be distributed and delivered. Scaling computational resources in these complex environments is a challenge that needs to be addressed wisely.

Scalability is a delicate topic because it comprehends various possible solutions and approaches, each focused on a different part of the environment. On regular deployments, devices gather data transferred to some storage device, which can be an intermediary or final storage. Thus, the data can be integrated and processed into an information system. On distributed systems, there is a need to use intermediate elements, which allows scalability and availability of data to be converged. Focusing on a strategy that delivers the most significant value based on its uniqueness is necessary to scale such environments.

There are, basically, two main approaches for scalability on computational systems. One is clustering, which uses the aggregation of individual computational units to behave as a unique entity, relying on an orchestrator that distributes tasks and data, delivering great computational power in a vertical approach. On the other hand, Federating computational units allow these individuals to behave based on common roles and policies in a coordinated way, distributing computational power on a horizontal approach [Spohn 2022].

Protocols also play an essential role in those solutions. Several solutions have emerged, allowing interoperability in those environments. Still, MQTT, formerly known as MQ Telemetry Transport (sometimes referred to as Message Queue Telemetry Transport), a protocol developed by IBM (Internation Business Machines) in the '90s, delivered desired characteristics and has been adopted widely, becoming one of the most used protocols (or architectures) on IoT deployments [Naik 2017].

MQTT uses a publish/subscribe paradigm, which means data is generated by a given device that publishes its data on a storage entity - the broker - structured in topics. Systems, devices, or any integrator can easily subscribe to a topic to gather this data, which is pushed to the subscriber as soon as it is published [Firouzi et al. 2020].

So, the broker is a critical element in storing data generated and published. Here starts one of the main challenges of scaling such environments. The more publishers there are in a given environment, the more resources brokers need to gather and provide access to stored data. Brokers have limited resources to provide, and depending on the environment, using scalability strategies is imperative.

Different scaling strategies exist for MQTT deployments. The most common is scaling the environment through clustering a set of brokers, allowing a significant growth of the processing power available to interact with publishers and subscribers. The major turnabout of using a clustering approach is that it mainly relies on an orchestrator, the *load balancer*, which can be seen as a single point of failure. Having a single point of failure can be risky in critical environments, and also, the load balancer has limited resources that need to be scaled, making the solution extremely complex [Spohn 2020].

On the other hand, it is possible to scale computational systems using a federation approach, where all the members act based on specific roles and policies but autonomously, with background coordination. With this approach, the distributed system doesn't rely on a single orchestrator; consequently, there is no single point of failure [Spohn 2020].

Scaling MQTT environments using a federation approach is not a deeply explored topic and remains open. There are some solutions presented as proprietary, like HiveMQ [HiveMQ 2022] that claims to have federation capabilities, but there is no in-depth documentation presenting the implementations. Another solution that claims to be federation capable is RabbitMQ [VMWare 2022], which is open-source, but the implementation needs the subset of queues that will be federated to be previously configured. To explore a more autonomous solution, the foundation of the present work proposes a client-transparent and self-managed solution for federating MQTT brokers, written by Spohn [Spohn 2020]. Based on that, this work aims to implement the proposal using a Python-written wrapper attached to a regular Mosquitto MQTT broker, delivering autonomous federation capabilities.

The implementation proposed in the present work aims to be the least intrusive as possible and flexible enough to be adapted to any other MQTT broker that allows the same integration strategy the Mosquitto MQTT allows. This solution interacts with the broker by gathering log data and, with its federation capabilities, managing orchestration and data communication with other federation members.

The work is organized as follows: in chapter two, the bibliographic revision discusses IoT foundations, MQTT characteristics, and scalability challenges. In chapter three, the related works are presented as the theoretical proposal for this work. The implementation details are presented in chapter four, followed by chapter five, where the validation and performance tests are discussed. Chapter six brings conclusions and future work.

2 BACKGROUND

This chapter presents the background for this work, which focuses on integrating IoT devices using the MQTT protocol and scaling strategies for IoT environments. With the increasing ubiquity of IoT, it is essential to review the development of the technology and MQTT protocol and its importance to the present work.

2.1 The Internet of Things

2.1.1 Definition

The concept of IoT and its related areas have been at the forefront of computing research for almost a decade. Numerous definitions of IoT have been proposed as presented by Firouzi *et al.*, ranging from simple descriptions such as "connecting any device to the internet and other devices" to more elaborate definitions such as “the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment” [Firouzi et al. 2020]. Despite their variations, all definitions emphasize data collection from the environment by devices and their ability to interact with other systems or devices through the internet. In this study, we adopt a comprehensive definition of IoT that encompasses the various dimensions of the technology.

According to Firouzi *et al.* [Firouzi et al. 2020], IoT deployments typically involve several key elements. First, there are the *things* or *devices*, which are intelligent elements capable of sensing, actuating, and interacting with other objects, systems, or people. These devices typically include a processing unit, power source, sensor or actuator, network connection, and some identification method, such as a network address or tag for unique identification.

Connectivity is the second essential element in an IoT deployment, enabling devices to communicate with other elements. Connectivity means establishing a physical connection and using some protocols to support data exchange.

The third element is *data*, defined as "...the first step toward action and intelligence" [Firouzi et al. 2020]. Data collected from IoT devices range from environmental data gathered through sensors, diagnostic data, location, reports, commands for automation, and other means of interactions.

Intelligence is the element that unlocks the potentiality of an IoT environment, allowing insights to be generated from the data collected. Various approaches can be used to analyze the data, including predicting maintenance, improving services, and enhancing productivity, not limited to these. The potential applications of IoT are vast and varied, and the insights gained from analyzing the data can help businesses and organizations optimize their operations and improve decision-making.

There is no interaction without *action*. This is why intelligence and action work as pair. Automation is a keyword in IoT deployments, so interaction is a central concept.

Other concepts may be secondary, such as *ecosystem* and *heterogeneity*, meaning IoT should be analyzed through a generic optic. IoT environments usually are organized as an ecosystem of diverse equipment, which brings heterogeneity, but they must share a common reference model to interact coordinately.

Also, *dynamic changes* are expected in an organized environment to sense and interact. This is why this characteristic is also expected to be listed as a part of the IoT environment concept.

Finally, *enormous scale* and *security and privacy* are characteristics directly related to the need for scalable deployment on IoT environments and concerns about security and privacy, as data gathered, generated, and exchanged inside the ecosystem can be very sensitive.

From another angle, Firouzi *et al.* [Firouzi et al. 2020] points out that the IoT ecosystem is based on four pillars: *things, data, people,* and *processes*. This view is similar to the deployment view but focuses on a more strategic angle of the IoT existence. As one can directly relate, *things* are related to the devices and their operationality, including their ability to interact, automate, and other functionalities that can be delivered or consumed. *Data* is related directly to the data gathered and sensed through devices but also includes all the necessary treatments to filter, sort, and store the data. In this view, the role of the people is to be an agent of operation of the IoT ecosystem or a consumer of the outcomes of the data processing, playing the role of a beneficiary of these outcomes. And, at last, the final component of the IoT ecosystem is *process*, where "...the benefits of intelligent automation, informed decision-making and control, and efficient procedures are realized". This element of the IoT ecosystem includes analyzing data and delivering value-added information to consumers".

The value added by IoT is remarkable. Firouzi *et al.* [Firouzi et al. 2020] presents some differential benefits that IoT delivers. As devices can sense the external environment, delivering enhanced safety inside hazardous environments is possible, as monitoring and managing can be enormously improved. Also, operationally monotonous tasks can be improved and automated. One example is parking access (or another type of access control) automation, which needs minimal people involved today. Another topic that benefits from the IoT ecosystem is healthcare. Monitoring has become more complete and agile, and the possibility of interacting with life-support devices brings medical response to another level. Adopting IoT can be a game-changer in all areas, from smartwatches that can monitor health parameters to connected pacemakers.

Adopting IoT is very challenging. As presented by Firouzi *et al.* [Firouzi et al. 2020], significant changes need to be addressed, ranging from strategic and management changes, as new functions and roles are needed in the business, to operational and technical issues involving the deployment, administration, maintenance, and monitoring of the IoT devices and its operation. Security is another essential item to be added to this list, as IoT devices are very low on resources and can be used as a point of entry for invasions. In 2017, a security breach was explored through an aquarium monitoring device that allowed a hacker to download around ten gigabytes of data from a casino [Schiffer 2017]. Privacy concerns also are very important because IoT is now part of everyday life, leading to different means of exploitation, as presented by Stamps [Stamps 2021], who brought the story of an incident where a baby monitor was exploited, allowing the attacker to interact with the house residents.

Data-related challenges can also be extended to dealing with the amount of data generated in the IoT ecosystem. Differentiated approaches are needed, as classical data storage means can not deliver the performance needed, and a structured scheme that better fits IoT data may also be imperative. Firouzi *et al.* [Firouzi et al. 2020] cites a few technologies and frameworks that appeared to propose solutions, such as NoSQL-based and time-series-based databases.

2.1.2 Architecture and Reference Models

As IoT evolved quickly, no reference model with clear guidelines existed for allowing an ordinated technology development. [Bauer et al. 2013] states that IoT is a product of the evolution of technology in different areas that started to converge. The evolution of RFID (Radio Frequency Identification) solutions was an essential enabler as this technology became widely adopted for cargo tracking, and devices (sensors) capable of reading RFID tags supported more functionalities and interconnection, allowing real-time data gathering. Also, sensors became smaller and computing power higher, enabling even more resources that could be adopted for different markets [Bauer et al. 2013].

As sensor devices and other related solutions were adopted, a reference model for supporting the coordinated evolution and development of the technology became necessary. The first result of the coordinated effort of several equipment manufacturers appeared in 2009 [Bauer et al. 2013] when the IoT- Architecture project (IoT-A) was presented. The result was the first guideline for coordinated IoT product architecture standardization. It was clear that several aspects diverge significantly from business to business, including security and privacy, which are commonly subject to different policies and laws depending on the location, as stated by Bauer [Bauer et al. 2013].

As IoT-A was the first reference model, several aspects were not covered, such as *scalability* [Bauer et al. 2013]. Also, there was a need to present more specific guidelines for developing products with different architectures, with more concrete details, focusing on functionality, performance, deployment, and security. In-depth, the reference model details design guidelines, where a *domain model* helps describe concepts of the area of interest, also defining its essential attributes; an *informational model* helps to define information structure and its relationships and how it is handled and processed; a *functional model* helps to describe the desired behavior and which functions the product shall have to act as expected; the *communication model* is aimed at defining the paradigms to allow internal and external communications, ranging from communication between processes to communicating through the internet to other devices or sensing the environment to communicate with RFID, as an example; and, finally, a *trust, security and privacy model*, supporting the development of policies, mechanisms and the adoption of solutions for a more fiducial environment.

Another reference level was proposed to address the need for more specific elements. Thus, *reference architectures* were presented to standardize interfaces and foment the adoption of best practices. The IoT Architectural Reference Model (ARM) was presented based on a deep analysis of requirements gathered between selected end-users and the project stakeholders [Bauer et al. 2013]. One of the most strategic gains was interoperability, leading to easy technology adoption.

As an architectural model, the ARM aims to address important questions in guiding technological development and adoption. Functional elements, the way they interact, the way the information is managed, which operational features a device can present, and how it can be deployed are aspects specified in the model [Bauer et al. 2013]

2.2 Protocols & MQTT

As seen in Section 2.1, communication is essential to IoT. The IoT ARM addresses communications through a structural model that "...allows the identification of homogeneous subsystems and their capabilities and constraints, identify suitable protocols stacks and network topologies to be merged in a common system view and define gateways bridging solutions" [Bauer et al. 2013].

The key to defining the communication model is the establishment of all subsystems through a complete and well-defined domain and information modeling [Bauer et al. 2013]. This allows defining "homogenous subsystems as a set of system elements sharing the same communication technology and similar hardware capability" [Bauer et al. 2013]. The communication requirements also permit identifying communication patterns, consequently identifying the interoperable stack and topologies, following three specific points: first, each stack must grow from a specific communication technology; second, interoperability shall be enforced in the lowest possible layer of the stack. Third and last, the combination of identified stacks and topologies must satisfy all the requirements [Bauer et al. 2013]. It is important to note that the second rule automatically drives the interoperability to the network stack, as devices mainly share the same ISO/OSI (International Standardization Organization/Open Systems Interconnection) network model. This simplifies interoperability and allows easy integration of devices and other systems that adhere to this model.

It can be inferred that for dealing with communication stacks, the ISO/OSI model delivers different levels of interoperability to various subsystems. As some subsystems can be integrated with other internal or external subsystems through lower-level network stacks, some other subsystems' integration need to be addressed on higher-level network stacks, even at the application level [Bauer et al. 2013]. So, no single rule fits all the possible solutions.

Bringing the topic to a higher level, the intercommunication that is needed to integrate IoT devices with other systems, besides the low-level network protocols, shall deliver simplicity and allows reliable end-to-end communication with the least control overhead, good fault tolerance, and consume the least possible network resources [Bauer et al. 2013] and [Firouzi et al. 2020].

Delivering end-to-end communication is a tricky goal. Achieving this means allowing a high interoperability level, not only M2M (Machine to Machine) communication but also allowing data to be delivered to storage and information systems [Bauer et al. 2013]. Protocols that operate at the application level tend to provide functions that are not present on lower-level layers, and this is why there is a need to use standardized application protocols to achieve the desired interoperability [Bauer et al. 2013].

Communication at the application level allows more complex operations. Several devices use HTTP or WebServices to provide refined resources, but this approach has cons, such as a significant communication overhead as presented by [Firouzi et al. 2020] and [Bauer et al. 2013]. The search for a leaner protocol led to the development of one that could deliver resources with very restricted constraints - the MQTT (MQ Telemetry Transport) [HiveMQ 2020].

2.2.1 MQTT

MQTT was proposed as a protocol in 1999, allowing IoT devices to communicate reliably [MQTT 2022]. The main goal was to present a lightweight protocol with the lowest resource requirements to deliver reliable communication on unreliable networks with low bandwidth and high latency. The main goal was to provide a protocol to be used for pipeline monitoring using devices that consume minimal battery and connect to satellite networks, following requirements such as simplicity of implementation, quality of service data delivery, efficiency on bandwidth and latency, being data agnostic, and allowing continuous session awareness [HiveMQ 2020]. MQTT adopts the Publish/Subscribe approach, in which IoT devices do not directly connect to the client that consumes the messages. This is based on a three-dimensional decoupling approach, where space, time, and synchronization are key points [HiveMQ 2020].

Space decoupling means that the publisher and the subscriber may not know each other, allowing simplicity in the implementation and management; time decoupling means that the publisher and subscriber may not be running (or turned on) at the same time; and third, synchronization decoupling implies that operations on both components do not need to be interrupted during the publish/subscribe task and there are no processing nor network overhead to synchronize elements for communicating [HiveMQ 2020].

MQTT architecture proposes two main elements involved in messaging: clients that act as publishers or subscribers and the broker. The broker is the middleware on which MQTT relies to interact with clients and gather or deliver data, depending on the client's role [HiveMQ 2020]. As told, clients can act as data publishers when they are pushing data gathered into the broker, and it acts as a subscriber when they pull data from a broker [HiveMQ 2020].

The broker is the most important element in architecture as it allows components to interact. The publisher clients connect to the broker and push data to dataspaces called *topics* [MQTT 2022]. Topics are hierarchical structures similar to folders, where data is pushed and can be named freely, regarding the case-sensitive nature of a topic identifier [MQTT 2022]. It is also possible to state that information published to the broker is subject-based, as every data unit pushed to the broker has a subject [HiveMQ 2022]. Data published shall be represented using UTF-8 (Unicode Transformation Format - byte oriented) to avoid problems with data representation [Oasis 2014], as Unicode can generate escape codes that can be misunderstood on different systems; however, MQTT specification indicates that if data is encoded using Unicode, the data packet will be dropped.

2.3 Scalability

Scalability is a sensitive topic on MQTT implementations, as main efforts to scale such environments were made towards clustering [Spohn 2020]. Ruempitak [Ruenpitak et al. 2022] also states that there are three main approaches for scaling MQTT deployments. *Bridging* is the first approach, where brokers are statically and directly connected, which has some drawbacks, such as inefficient message forwarding and possible message looping [Ruenpitak et al. 2022]. The second approach, proposed by [Longo e Redondi 2023], is through implementing spanning tree capabilities on the MQTT protocol, called MQTT-ST [Longo et al. 2020], which allows brokers to be interconnected without message lopping risk. The third approach is through clustering, which is the most adopted solution but relies on a single orchestrator, which can be risky for mission-critical environments, as presented by [Bass 2002] and [DE LACERDA MACHADO Jr., Spohn e Granville 2023].

The cited three approaches to scalability leave a gap in the topic for a reliability-focused approach, which can be filled with the proposal presented by Spohn [Spohn 2020]. Scalability based on a self-organizing federation approach is a different proposal that focus on high availability, but because of its novelty, it is underexplored. Motivated by this panorama, the present work presents a simple but ingenious implementation of the work presented by [Spohn 2020].

3 RELATED WORK

3.1 General Works and First Efforts Towards Scalability

This section presents work efforts toward scalability with a federation approach. It is essential to note the topic's complexity, as years were spent trying to deliver substantial evolution. The topic remains open due to a few factors like the evolution of devices' processing power, developments in connectivity, and more reliable technology, allowing clustering to evolve quickly [DE LACERDA MACHADO Jr., Spohn e Granville 2023] and being an accepted and widely adopted solution. But distributed critical environments need a different approach, focused on the survivability of the infrastructure [Bass 2002]. The following works give an essential overview of the topic's evolution.

3.1.1 InfoBus Repeater: A Secure and Distributed Publish/Subscribe Middleware

The *InfoBus Repeater* is a middleware developed by Uramoto and Maruyama [Uramoto e Maruyama 1999] in 1999. Based on the Infobus API (Application Program Interface), which is an interface that allows Java Beans to intercommunicate [Sagar 2003], the repeater provides distributed communication through a simple federation approach using the publish/subscribe principle.

Infobus is an event-driven communication interface provided by Java[1] that allows data updates to different processes (or software instances) using a publish/subscribe approach [Sagar 2003]. Three types of Java Beans interact with the InfoBus in the architecture: data producers, data consumers, and data controllers. These elements need to join the InfoBus to access notifications generated by the events. Data producers announce the availability of new data, data consumers register to listen to data items that have been published, and data controllers filter and monitor data items [Sagar 2003]. But *InfoBus* has limitations in distributed environments, lacking the ability to communicate to remote instances, leading to the proposal of the *InfoBus Repeater* [Uramoto e Maruyama 1999].

The *InfoBus Repeater* thus provides the regular *InfoBus* resources in distributed networking environments. The repeater connects to *InfoBus* instances and acts as a data publisher and data consumer, monitoring all data items that flow in the instance [Uramoto e Maruyama 1999]. It also monitors data consumers announced inside the *InfoBus* in-

[1]http://java.com

stances and thus monitors which data items are marked as redistributable for later relaying to the remote instances [Uramoto e Maruyama 1999].

By allowing data published on an *InfoBus* instance to be replicated to other instances, the *InfoBus Repeater* acts as a federation agent allowing intercommunication between remote elements, as the same behavior and data availability is expected in all interconnected nodes [Uramoto e Maruyama 1999]. The interconnection between two *InfoBuses* is made using TCP (Transmission Control Protocol), but if more instances need to be interconnected, RMTP (Reliable Multicast Transport Protocol) should be used to avoid data retransmission loops [Uramoto e Maruyama 1999]. The repeater also provides SSL (Secure Socket Layer) authentication to allow secure interconnection between those networked buses [Uramoto e Maruyama 1999].

The idea of an orchestrator that provides message replication through an integrator is not new, but the *InfoBus Repeater* proposal delivered a mature work on the topic, and it stands as a valuable contribution to modern research.

3.1.2 The Federation of Critical Infrastructure Information via Publish-Subscribe Enabled Multisensor Data Fusion

In 2002, Bass [Bass 2002] presented a work that proposed models for federating critical infrastructure monitoring systems in "loosely coupled service-oriented information fusion processes administered by a federation of organizational services," which includes:

- federated sensor, processing, assessment, and storage;
- publication, subscription, and other fusion services;
- attribute-based data and information publishing;
- subscription-based data and information availability.

The models focused on delivering a reliable communication architecture for monitoring critical infrastructure, such as communication networks, electric power grids, intelligence networks, immigration systems, air traffic control, and transportation systems, not limited to those, using a publish/subscribe approach [Bass 2002].

The motivation for using a publish/subscribe approach is its event-oriented nature, as other aspects that are desirable in a monitoring environment, such as time and space decoupling, which means that data publishers and subscribers do not need to be synchronized and also do not need to be aware of their mutual existence [Bass 2002].

One important point is the network topology, which is critical to the federation. Bass [Bass 2002] states that centralized network topologies are unsuitable for critical environments, and a distributed approach is needed. The author analyzes three network topologies - hierarchical, acyclic peer-to-peer, and general peer-to-peer. Hierarchical topologies have cons regarding the servers at the top of the hierarchy, which the upcoming data can overload. This issue can generate a dangerous bottleneck to the monitoring system [Bass 2002]. Acyclic peer-to-peer networks prevent a server overloading by presenting a more distributed topology, but it also has its cons, as they may not deliver a desirable redundancy to server interconnection, which can be seen as single point of failure [Bass 2002]. So, a general peer-to-peer is the adequate network topology for critical monitoring infrastructure, presenting redundancy in links to avoid single points of failure [Bass 2002], but it is an utopic and costly scenario when integrating different networks of diverse entities.

But on the other hand, regarding the uniqueness of each involved party in the network integration, a hybrid model acting in a cooperative way, just like a real-world internet topology, will deliver significant reliability and can fit the singularities of each integrated member network policy [Bass 2002].

To provide data availability over the network, data access points are needed. These access points should rely on an event notification service, which notifies interested parties of new data published over its topics of interest over the distributed and heterogeneous environment [Bass 2002]; the author's point of view on this particular topic enlights one of the foundation points of the present work which is delivering a federated network of MQTT brokers which provides data access to all topics of interest in a distributed environment.

With that, this particular work magnifies the motivations of Spohn's [Spohn 2020] proposal and reinforces the need for a viable solution that fits the singularities of this problem and related environments.

3.1.3 Towards Scalable Publish/Subscribe Systems

Distributed publish/subscribe solutions have evolved in their design and implementation, and distributed architectures have become more commonly adopted, bringing FBR (Filter-Based Routing) as a highly adopted solution on these topologies. As stated by Ji *et al.* [Ji et al. 2015], FBR is not efficient nor flexible, despite being highly adopted as a viable solution for distributed environments. This comes from the coupling of event matching and routing, where each routing decision is based on the resulting event properties. Also, FBR suffers from limitations such as problems supporting general overlay topologies, subscription duplication, redundant and repeated event matching, and a lack of flexible support to overlay reconfiguration [Ji et al. 2015].

Ji *et al.* initially proposed the D-DBR (Dynamic Destination-Based Routing) to overcome these problems by decoupling the P/S (publish/subscribe) system into two independent layers, one responsible for content-based matching and the other responsible for destination-based multicasting [Ji et al. 2015]. This allows topology changes (the overlay) to be held by the multicasting layer, which keeps the routing information updated and addresses message forwarding; supporting overlay reconfiguration introduces a valuable fault-tolerance mechanism and optimizes performance [Ji et al. 2015].

Although, D-DBR has scalability issues related to large-scale networks, as each broker has to have knowledge of all brokers in the system; this is why Ji *et al.* proposed the MERC (Match ad Edge and Route intra-Cluster), which aims at allowing scaling P/S environments on large networks deployments by creating clusters of brokers that just need to be aware of other brokers in the same cluster, and interconnection cluster brokers that act on the edges as routers to other clusters.

MERC also uses the same destination-based content routing used in D-DBR, adding a hierarchical approach, organizing the brokers into interconnected clusters through *edge brokers*, i.e., brokers that are part of two or more clusters [Ji et al. 2015]. The destination-based routing is divided into two layers, the intra-cluster, and the extra-cluster event routing, assembled based on the brokers' advertisements of subscriptions [Ji et al. 2015].

Notably, efforts toward delivering an efficient solution to handle data over a distributed environment based on a publish/subscriber approach try to explore decoupling at all levels. This particular work contributed to the present work by providing a more solid foundation for task division in the implementation.

3.2 Foundation Works

The *Foundation Works* to the present work provides the theoretical foundation or the main guidelines for the present work, also bringing a new vision over the implementation. Three main works have contributed to the current work, which will be presented and discussed in this section. The first work is the paper presented by Spohn [Spohn 2020], who proposed the theoretical model for federating MQTT brokers using an *ad-hoc* approach, meaning that there is no need for one broker to connect directly to other federated brokers, just its neighbors, and, on [Spohn. 2021], a first implementation proposal was presented. Ribas [Ribas 2022] presented the first derived implementation of Spohn's work , presenting a Rust-written federator that relies on the attached broker to store information regarding federation and its federated topics.

3.2.1 Publish, Subscribe, Federate!

This is the main foundation work on which this proposal relies. Spohn [Spohn 2020] presented a model for delivering federation resources to MQTT brokers using a simple and innovative approach. IoT deployments rely on brokers to gather and deliver information to clients, and mainly, scaling such systems is typically made using a clustering approach. To deliver high availability, Spohn [Spohn 2020] proposes a federative approach to scale those deployments, focusing on MQTT-based solutions.

The model proposed by Spohn aims at being simple and flexible, bringing communication principles introduced by MANETs (Mobile Ad-Hoc Networks), where network routing and orchestrating are made at the application level, as a network member (or node) may only have direct and limited access to its neighbors [Spohn 2020].

Also, the model proposes resources for dealing with information spreading through the network, based on PUMA (Protocol for Unified Multicast Routing), which relies on meshes for multicast communications [Spohn 2020].

Based on these two pillars, the model proposed by Spohn works on two levels. The first is establishing an overlay network that delivers routing and other high-level network resources to network members, called the *federation* [Spohn 2020]. Over that overlay network, data is forwarded and routed to all groups, or individual members of the federation, allowing end-to-end communication.

As MQTT-based IoT deployments are the environment the model aims to support, dealing with particularities of the publish/subscribe approach is mandatory. The main point of the proposed solution is to allow remote subscribers to a common topic to receive data regarding that particular topic on any interconnected broker. This only can happen if all brokers are aware of the existence of subscribers and their localization. To solve this problem, the model proposes a network-wide announcement for subscribers' presence, assembling a mesh that facilitates data to flow between remote brokers. The mesh format also aims to facilitate routing and managing mesh topics, as described by Spohn [Spohn 2020].

The proposed format for managing the meshes, the overlay network or the topic meshes, is a core element election that plays a referential role in each mesh. All orchestration and data forwarding is made depending on this reference element. The model also proposes a generic message format or data that is used to orchestrate the meshes, which shall carry the following information [Spohn 2020]:

- CoreID
- Distance to the core in hops
- Mesh membership flag
- List of parents

All information regarding core brokers, being the overlay core or a topic core, is known by all federation members and, consequently, information regarding all federated topics. This allows a broker not part of a topic mesh, meaning there are no locally connected subscribers to that given topic, to forward messages to a federated topic, reaching the subscribers to the topic that are connected to remote brokers transparently [Spohn 2020].

The implementation proposed in Section 4.3 tries to be as fiducial as possible to this model.

3.2.2 Federação de Brokers do Protocolo MQTT - Implementação e Análise de Desempenho

This work presents one implementation derived from Spohn's [Spohn 2020] work written in Rust[2] by Ribas [Ribas 2022], which relies on the MQTT broker to store federation-related information.

The implementation uses a topic called `federator` in which information about core announcements, member announcements, routing, and beacons which are messages regarding local subscribers, are stored. The beacon is a registration message published by the client subscribing to a federated topic. So, the subscriber needs to be aware of

[2]https://www.rust-lang.org/

federated topics and register to receive information from the federation if there is any publication on remote brokers, as described by Ribas [Ribas 2022].

This implementation provides a good overview of the challenges in implementing Spohn's model. The implementation proposed in the present master thesis tries to be more transparent to clients, being aware of the existence of a federation, and also tries to detach from the MQTT broker, avoiding its use as a data store.

4 PROPOSAL AND IMPLEMENTATION

4.1 Proposal

The work presented by Spohn [Spohn 2020] was chosen to be the foundation of the present work. His work, as reviewed, proposes some adapting to an existing MQTT broker, allowing intercommunication between brokers and orchestration to assemble an overlay network, allowing topic meshes to be organized and providing data published on a given topic to be propagated to all brokers that have subscribers.

To implement Spohn's work, some points needed to be defined. First, a regular MQTT broker to provide all the core MQTT functionalities. Second, a viable approach for customizing the broker keeping it low on resource usage and being the least intrusive as possible. Third, and last, a programming language that allows implementing the solution without excessive effort.

The chosen MQTT broker for implementing the proposal was Eclipse Mosquitto MQTT broker [Eclipse 2018] which is a very mature, reliable, low profile on resource usage, and highly deployed solution. More details about Mosquitto will be presented ahead, but one important characteristic is that all log data generated can be redirected to a specific topic. This is a game-changer because allows all broker operations to be monitored without deeper customization.

Mosquitto's capability to redirect logs to a topic allowed the possibility to create a module that interacts with the broker in a simple manner, through native MQTT. So, if the broker allows the module to connect using its regular resources, it is also possible to design a module that consumes messages from the log topic to monitor and manage data. Based on this approach, and knowing that Python [Python 2022] has a library that allows interactions with MQTT brokers, it became the chosen programming language to implement the proposed solution.

The library that allows MQTT interactions with Python is called *Paho* [Paho 2022]. With this library, it is possible to subscribe to broker's topics and treat data received, also, it is possible to publish data on the topics. With that, the idea of developing a module that could be attached to the broker providing the needed communication and orchestration functions evolved, and, the development of a wrapper became possible.

Ahead, it will be presented a brief overview of the technologies involved, and, after, the design and implementation of the wrapper

4.2 Resources Overview

4.2.1 Mosquitto

Eclipse Mosquitto MQTT broker [Eclipse 2018] is a well-known and highly deployed MQTT broker that aims to be highly flexible and adequate for IoT deployments, as it is focused on being low profile on computational resources usage. Also, it is open-source software distributed under the Eclipse Public Licence (EPL), which allows audition, free modification, and limited redistribution. Mosquitto broker has also been ported to several platforms and can be installed on almost every operating system.

Current branches of the Mosquitto MQTT broker present two versions of MQTT protocol implementation, the 3.1.1 and the 5.0 [Eclipse 2018]. The main difference between these two versions is that version 5.0 provides additional features such as session/Message Expiry Interval, Reason Code, Topic Alias, User Properties, and Shared Subscriptions. It improves the performance, stability, and scalability of large systems [EMQX 2022]. For this work, the version used was 3.1.1 because of its wide adoption and also for being more mature than 5.0

Mosquitto configuration allows all log data to be redirected to a particular system topic, the `$SYS` topic [Eclipse 2018]. This is one of the most important elements that provided resources for implementing this work. All information regarding topic subscribers is logged and can be used to monitor if there is any subscriber to a given topic, as shown in the following example:

```
log_dest topic
log_type all
```

For a more adjusted option, it is possible to fine-tune which information will be published in the `$SYS$` topic, placing all desired logging levels one per line, as shown:

```
log_dest topic
log_type debug
log_type error
log_type warning
log_type notice
log_type information
log_type subscribe
```

```
    log_type unsubscribe
```

4.2.2 Python

Python [Python 2022] is a powerful and flexible programming language that provides valuable resources for implementing Spohn's [Spohn 2020] proposal. Python is a high-level scripted programming language, not compiled, but with a very good performance and low on computational resources usage. This is why Python was chosen for the implementation.

Also, Python is a very mature programming language, with several libraries to support virtually every computing topic, ranging from low-level networking, data analysis, neural networks, web applications, and other areas [Python 2022].

For dealing with IoT, mainly MQTT-based environments, the Eclipse foundation presented the Paho project, which delivers to various programming languages a resourceful library to deal with MQTT-related applications [Paho 2022].

4.2.3 Paho

The Paho Project results from efforts to deliver resources that provide open-source implementations of open standards protocols aimed at applications for IoT environments [Paho 2022]. Paho, as explained by the Eclipse Foundation [Paho 2022], is a Maori word that means "to broadcast, to transmit, to announce" and fits the main functions that the library delivers.

Eclipse Project tried to facilitate the creation of a flexible ecosystem for IoT environments, delivering value-aggregated tools to integrate M2M (Machine-to-Machine) solutions, and MQTT protocol plays a vital role in this scenario; with the MQTT code made available by its creators, Paho became a framework that was quickly adopted to deal with the technology that was evolving, back in 2010's [Paho 2022].

The first code was made available in C and Java and later ported to other programming languages, such as Python [Paho 2022], and can be installed using the regular Python package manager PIP (Package Installer for Python).

4.3 Model and Implementation

After the initial implmementation presented by Spohn [Spohn. 2021], Ribas [Ribas 2022] introduced a RUST-written federator that uses the broker to store information regarding federated topics and routing information. The federator interacts with the broker using the Paho library provided by the RUST programming language.

To be more detached from the broker, the present work proposes a wrapper that interacts to gather and post information directly related to federated topics, handling routing and monitoring internally without storing functional data in the broker.

The main goal was developing a very light and simple solution that delivers the expected behavior proposed on Spohn's work [Spohn 2020].

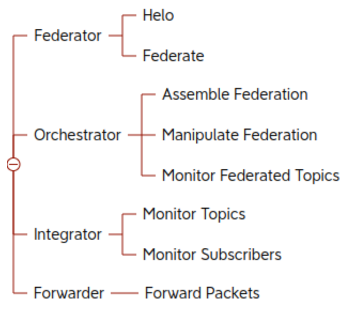
4.3.1 Model

The Main Group is composed of four functional subgroups. These subgroups are named federator, orchestrator, integrator, and forwarder. The first subgroup, the federator, has two primary functions: first, monitor communication links between neighbors using a handshake-like function, and second, generate a regular announcement token used by the wrapper to assemble the federation. The second subgroup deals with orchestrating functions, using information from the brokers to assemble and manage the federation (the overlay network) and monitor and manage information regarding the federated topics. The third subgroup is responsible for interacting with the broker to monitor subscribers and topic data, gathering new data to forward to remote federated brokers, or publishing received data from remote brokers to local topics. And the fourth subgroup is responsible for forwarding packets, avoiding looping forwards and spurious retransmissions.

Figure 4.1 briefly overviews the Main Group and its subgroups. There is also a Support Group, which encompasses six subgroups that aim to provide support functionalities to the broker, such as flushing caches, low-level networking functions to send and receive packets, packet inspection, logging, and debugging, as demonstrated in Figure 4.2.

These two functional groups support all the needed functions allowing a full-functional wrapper. Ahead, a more detailed presentation of each group will be made.

Figure 4.1: Main Functions



4.3.1.1 Main Group Functions

Federator Subgroup:

As presented, there are four main functions that the wrapper relies on. The subgroup that delivers federation capabilities is called *Federator* group and has two functions. As reviewed in Section 3.2.1, the wrapper does not connect to other wrappers besides its neighbors. Because of that, monitoring these links is mandatory to keep up with topology changes, which will consequently change routes.

The *helo* function uses a regular `helo` packet to the neighbors, being responded with a `heloback` packet, assuring that the link is up and the neighbor is ready. It is essential to state here that there is a need for previous knowledge of the neighbors. The `helo` packet sending interval can be adjusted to fit topology singularities, but a notification every ten seconds should work in most cases. When the handshake is made, the wrapper registers internally that there is a link up with that particular neighbor and the remote IP address. This information is stored in a Python list called `activeNeigh`.

The second function inside the *Federator* subgroup is the *federate* function, which also sends announcement packets in regular intervals. The packet is structured as follows:

```
fd(sequenceCount, distanceVector, brokerId)
```

For each new announcement, the wrapper, identified by an integer called `brokerID`, adds one to the `sequenceCount`, which is an integer, indicating the information lifetime. The `distanceVector`, also an integer, is added by one on every forward operation, indicating the number of hops (or in-between wrappers). This information stores the federation topology view on each wrapper, which is assembled by knowing which neighbor delivered the same packet with the smaller `distanceVector`; the resulting data will be stored in memory as a routing table. One can ask about packet looping, but this is avoided by a message-forwarding cache.

Orchestrator Subgroup:

In the *Orchestrator Subgroup*, there are three main functions: the *federation assembly* function, the *federation manipulation* function, and the *federated topic monitoring* function. The first function treats the packets received with other federation members' information. This function filters data inside `fd` packets, organizing the list with information regarding federation members, routes, and distance, which is vital to the wrapper functions. Data is stored inside a Python *list* of *dictionaries*, allowing quick and easy access. The `federation` list is organized as follows, having a *dictionary* item assigned for each federation member:

```
federation = [pAdd1 : {'brokerID': valueBID, 'route': neighIP,
                       'sequenceCount': valueSC,
                       'distanceVector': valueDV},
```

The *federation manipulation* function orchestrates the core election and federation topologies updates. It uses the data inside `federation` list to elect core broker, manage `brokerID` collisions, and manage topology changes.

Federated Topic Monitoring function orchestrates the topic meshes. Information regarding the federated topics, the topic core, the membership flag, and core disputes are handled by this function. The data is stored inside a Python *dictionary* of *lists*, organized as follows:

```
topics={topic1:[coreID,memberFlag],topic2:[coreID,memberFlag]}
```

Integrator Subgroup:

The *Integrator Subgroup* encompasses two functions that are directly related to the wrapper's attachment to the broker: *topic monitoring* and *subscriber monitoring*. These functions rely on the Paho library to subscribe to the broker's topics and monitor the information submitted.

Subscriber monitoring allows the wrapper to identify if there is a new subscriber to a topic and adjust its flag membership on a federated topic. When a client subscribes to a topic, the information is published in the `$SYS` topic, which the wrapper monitors for updates. A message is generated, and if the topic is not federated, the federation announcement is made throughout the federation.

Finally, *Topic Monitoring* is made through the wrapper's ability to monitor all topics in the broker using the `#` topic. It filters data published to federated topics and replicates to the mesh.

Forward Subgroup:

This unfunctional group filters and forwards messages, relying on the message caches. Each message that arrives is cached and forwarded. If a message has already been received, it is ignored. There is a secondary cache for topic messages, as topic messages can also be repeated - a temperature monitor, for example - and this secondary cache avoids looping messages but allows similar messages to be received.

4.3.1.2 Support Group Functions

These subgroups of functions are secondary but not less important. Support functions help orchestrate and allow the wrapper to work as software. The six subgroups of functions are *packet sending*, *packet receiving*, *packet inspection*, *logging*, *debugging*, and *cache flushing*.

Packet Sending:

This subgroup has one primary obvious function: send packets using low-level network functions provided by the *socket* library. But for the wrapper to work, there are three types of packet sending. The first type is made of packets that are sent to the neighbors. The list of active neighbors is the data source used to send these packets. The second type deals with packets that shall be sent to the federation. The difference between the first and the second is that packets sent to the federation include all packets, and packet forwarding excludes resending the packet to the sender.

The third function is routed sending, which sends packets based on their destination. This function needs additional routing information gathered from the `federation` list. So, when a packet is directed to some federation or topic core wrapper, the packet handling is diverse from the packets broadcasted to neighbors (or `helo` packets).

Packet Receiving:

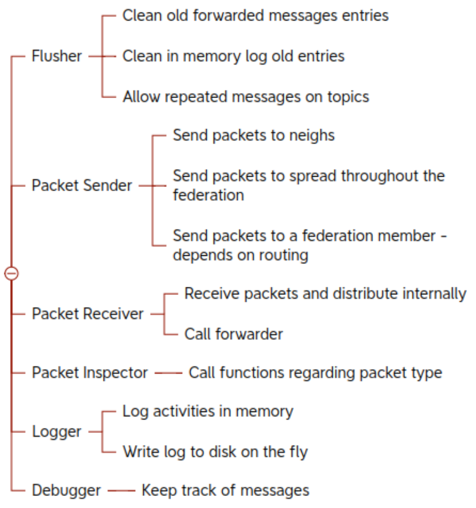
Packet Receiving has two main goals. First, to provide a socket to receive data and handle packets to the packet inspector. Second, handle packets that need to be forwarded to the packet forwarder. These two functions are straightforward but dramatically important as the entry point for messages to the wrapper.

Packet Inspector:

The *packet inspection* function filters packets based on their message type. Message types will be discussed ahead. The inspector calls for each message type a different function or group of functions, handling data in the packet payload.

Logging:

Figure 4.2: Support Functions


Flusher
Clean old forwarded messages entries
Clean in memory log old entries
Allow repeated messages on topics
Packet Sender
Send packets to neighs
Send packets to spread throughout the federation
Send packets to a federation member - depends on routing
Packet Receiver
Receive packets and distribute internally
Call forwarder
Packet Inspector
Call functions regarding packet type
Logger
Log activities in memory
Write log to disk on the fly
Debugger
Keep track of messages

Logging is a very important function as it helps monitor tasks and data being held and allows inferring the software behavior. The log stores information regarding function activities and results, allowing wrapper behavior inspection.

Debugging:

Debugging delivers a more profound experience than logging, as it registers every message that the wrapper sends and receives and gathers information placed inside federation and topic variables.

Flushing:

All messages are placed in caches to avoid infinite message retransmission. Periodical flushing is mandatory to keep a lean memory usage. There are two caches, one for forwarded messages and another for received messages.

4.3.1.3 Message Types

All packets that are exchanged between wrappers carry a message that is identified by two letters, and, additionally, a payload. There are ten different types of messages, presented in table 4.1.

For the neighbor handshake, `hl` and `hb` messages are used. These messages carry no payload and are used just for wrapper mutual announcements. After the initial handshake, the wrapper makes regular announcements using `fd` messages that carry a simple payload, as presented before, with its `brokerID` and two counters, one that is marked every time the message is forwarded counting the distance, and the other marking the sequential number of the announcement; if a message with a lower announcement count is received, it means that information regarding the remote wrapper needs to be updated,

Table 4.1: Message types

`hl`	helo message
`hb`	helo back message
`fd`	federate announce message
`rc`	reconsider `brokerID` message
`ca`	core announcement message
`tc`	topic core announcement message
`rt`	reconsider topic core announcement message
`ts`	topic subscribe message
`tm`	topic data message
`tu`	topology update message

and, consequently, the distance information may change.

When two wrappers announce themselves to the federation with the same `brokerID`, a dispute system is needed; wrappers with colliding identification send `rc` messages to the federation asking for a reconsideration. Wrappers receiving reconsideration notification generate a new random identifier posteriorly announced throughout the federation.

The core election happens during the federation overlay network assembly. The broker with the lowest `brokerID` number is elected the core broker, and it starts sending periodic `ca` messages.

Following the logic proposed by [Spohn 2020], every new subscriber attached to a broker makes the wrapper announce a topic subscription to the federation through a `ts` message. This information is registered by all wrappers, who monitor its brokers for new messages. If a new message is posted on a federated topic, the wrapper assembles a `tm` message and directs it to the topic core broker.

Electing the core broker for the topic is a slightly different task. When a wrapper first announces a topic to be federated, it is accepted as the topic core broker and starts sending periodic `tc` messages. In case of collision, i.e., two brokers announce themselves as core, a `rt` message is exchanged, and the wrapper with the lowest `brokerID` takes the role.

There are two scenarios in that a topology update packet is generated. First is when a core broker stops announcing itself for a long time. The federation "understands" that the core is no longer available, so a new core election is needed. The second is when there are detected topology changes between neighbours, which means routing tables must be updated.

4.3.2 Implementation

Python allowed a very simple implementation of the model. The version used for coding the wrapper was 3.8. Supportive libraries and simplicity in coding helped build a resourceful wrapper with low coding. Low-level network functions are delivered by the `socket` library, allowing the wrapper to communicate through a UDP socket that listen to connections using port 10500. All data packets are "encapsulated" using `pickle` library, which provides data serialization, keeping payload integrity when assembling and disassembling packets.

Paho library, presented in section 4.2.3, allows topic and event monitoring in the Mosquitto broker. Like other functions, such as periodical announcements, debugging, and cache flushing, Paho-related functions rely on the `threading` library, which allows parallelism to be explored in the implementation. Other libraries, such as `copy`, `time`, and `random`, were also used, providing elementary resources for dealing with variables and time-related functionalities.

Depending on data type inference, some variable initialization is needed when dealing with lists or dictionaries. The variable needs to be initialized to store dictionaries inside a list or a list inside a dictionary because the Python interpreter may misinfer the external data type.

Addressing parallelism is a delicate topic; as several tasks were parallelized using the `threading` library, concurrent data access must be managed, and *blocking* the variables is mandatory. Blocking guarantees that data stored inside a variable is not subject to change-in-change, which means that two concurrent threads will not change the data inside a variable simultaneously, guaranteeing data integrity.

The single file wrapper script runs side-by-side with the broker, and it is called from the command line, as it is a simple single file script.

All the implementation and tests were made in a Linux environment. A virtual machine using Ubuntu[1] 22.04 set up with 8GB of RAM, 30GB of disk space, and 2 virtual processors, hosted by a Linux Mint[2] 21-powered laptop. The hypervisor used to power the virtual machine was a regular Oracle VirtualBox[3] 7.0.6 Desktop Edition.

To simulate multiple instances of broker/wrapper pairs, an LXD[4] (Linux Contain-

[1]http://ubuntu.com
[2]http://linuxmint.com
[3]http://virtualbox.org
[4]http://linuxcontainers.org/lxd/

ers Daemon) 5.0 environment provided the ability to work with containers that, differently from a regular Docker environment, for example, does not need a new application image to be generated every time the code changes.

LXD delivers full-stack virtualization and containerization experiences. A lean environment where the memory and processing footprint for each instance is shallow but offers a complete operating system experience. Each container used on the implementation and tests provided a fully-functional Ubuntu 20.04 LTS (Long-Term Support) environment with a Mosquitto 3.1.1 broker and Python 3.8.10, followed by Paho-MQTT 1.6.1 library on top. The Mosquitto broker was configured to redirect logs to `$SYS` topic, as explained in 2.2.1, providing the needed resources for the wrapper.

In LXD, as in other virtualization platforms, it is possible to use virtual switches to interconnect groups of instances and provide advanced networking resources. Automation is also possible, but regular resources were used to use the platform to demonstrate the implementation. All instances were initiated and connected to the same virtual switch.

As explained in 4.3.1, the topology is defined by a list of neighbors, so there is no direct relation to the network stack addressing. Then overlay network functions, management, and orchestration are done at the application layer. The unique premise is that all neighbors should be reachable by each other. In the case of a non-reachable neighbor, the link will not be activated, and the initial topology will be diverse from the planned.

The wrapper's code was structured in four blocks as the environment was set up. The first block contains library calls, constants, and variables, as some shall be initiated. The second block encompasses the primary functional group, and the third contains the support functions group. The fourth block contains threading initialization and general program calls.

A rudimentary console was implemented to monitor the wrapper's behavior, which could bring more information about the operation and managed and stored data.

5 VALIDATION AND PERFORMANCE TESTS

After the wrapper's development, validation and testing were necessary to guarantee that the wrapper meets all proposal's principles. To provide a validation environment, LXD was used to instantiate containers just as proposed in the topology presented by Spohn [Spohn 2020], as can be seen in the figure 5.1.

The scope of the implementation encompasses the main MQTT functions. The implementation did not explore two resources: Quality of Service (QoS) and security. They were set out of this study because the main focus is validating an innovative proposal to orchestrate network meshes, and those two topics were considered for future work. Based on that, it is stated that the implementation will only use messages with QoS level 0.

5.1 Validation

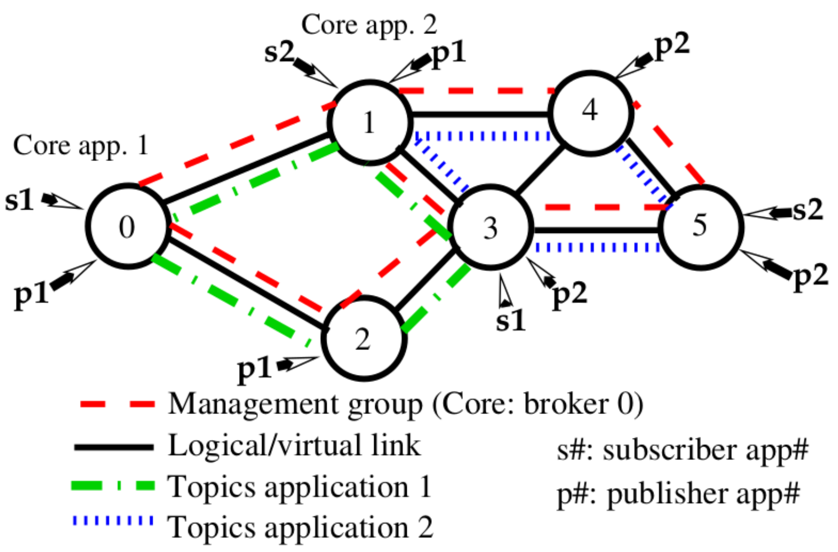
The proposed topology was used to validate the wrapper's adherence to the proposal. To initiate, all wrappers were adjusted to know their neighbors, which was done using the `neigh` constant, where neighbors' IP addresses are stored. After the initial topology adjustment, all the wrapper instances were started.

Through the console, which brings information regarding linked neighbors, federated members, and federated topics, it was possible to observe after the initial handshake that the neighbors' list started to be filled. *Helo* packets were set up to be sent every ten seconds, so it took a little more than ten seconds for every neighbor link to each other.

The overlay network assembly is started parallel to the neighbor handshake. The wrappers send `fd` packets to their neighbors, which are forwarded to propagate through the network. At that time, wrappers started to show remote wrappers information, with their respective `BrokerID`, routes, and distances. With that information, it is valid to state that the federation mechanism was correctly implemented and can be considered accurate. Several simulations were made, and the overlay network converged precisely in all rounds.

At this point, it is essential to disclose that convergence time is relatively high. `fd` messages are sent every twenty seconds. Adding the time for propagation and core dispute treatment, convergence can take more than one minute, depending on the number of wrappers in the topology. Console messages presented in figure 5.2 show `n` as the

Figure 5.1: Validation Topology, extracted from [Spohn 2020]



list of neighbors, `f` as the list of federated brokers, with respective IP address, BrokerID, distance, routes, and information lifetime.

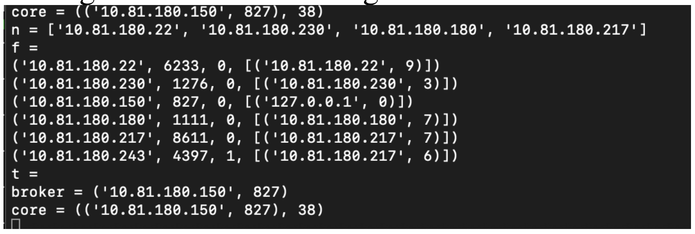
The core election occurs when wrappers and remote wrappers send core announcement messages every twenty seconds. Core announcement packets `ca` propagate and are received by wrappers, which are analyzed. If the core is not elected, the `ca` with the lowest `brokerId` is taken as core. Every twenty seconds, the thread that addresses core election monitors if the federated member with the lowest `brokerID` is the core broker. If not, the wrapper with the lowest `brokerID` is elected, and additionally, the individual wrapper sends additional `ca` announcements to the federation.

The overlay network deals with topology changes by monitoring the lifetime field in federation information. Suppose no `fb` packet is received in one minute, what also does not update lifetime information. In that case, the wrapper understands that there is a topology change, i.e., one of the overlay network wrappers has stopped responding.

This mechanism delivers the desired behavior to deal with topology changes. Orchestration occurs autonomously, allowing wrappers to reorganize routing and information regarding federation members without manual interference. To validate this, random wrapper disconnections were simulated, forcing network partitioning. The overlay network behaved as expected, and in the case of network partitioning, a new core was announced for the coreless part. When two networks join, the dispute mechanism orchestrates the election between duplicated cores.

Convergence time in the case of network partitioning can take up to five minutes, depending on the number of wrappers and federated topics. This happens because of

Figure 5.2: Console image of federated brokers



```
core = (('10.81.180.150', 827), 38)
n = ['10.81.180.22', '10.81.180.230', '10.81.180.180', '10.81.180.217']
f =
('10.81.180.22', 6233, 0, [('10.81.180.22', 9)])
('10.81.180.230', 1276, 0, [('10.81.180.230', 3)])
('10.81.180.150', 827, 0, [('127.0.0.1', 0)])
('10.81.180.180', 1111, 0, [('10.81.180.180', 7)])
('10.81.180.217', 8611, 0, [('10.81.180.217', 7)])
('10.81.180.243', 4397, 1, [('10.81.180.217', 6)])
t =
broker = ('10.81.180.150', 827)
core = (('10.81.180.150', 827), 38)
```

the significant interval times used in the implementation, which can be adjusted. All the timers are set not to flood the network with control packets and keep a low profile on traffic.

The topic meshes assembly is event-driven, triggered by new subscribers' announcements. When an event of this nature happens, the wrapper checks its registers if the topic is already federated. If the topic is not federated, the wrapper sends a `tc` message, announcing itself as the topic core. This message propagates throughout the federation, and wrappers become aware of the topic's existence and where messages shall be routed.

The same principle is adopted to deal with a collision in topic meshes core announcements. If, in an even event, two or more wrappers announce themselves as topic cores, the wrapper with the lowest `brokerID` is elected the core.

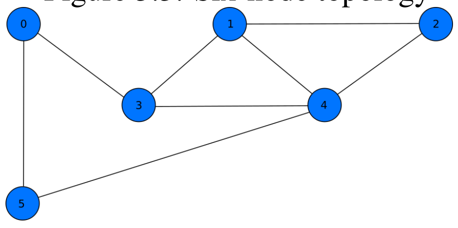
Topic core announcement messages are sent periodically, every thirty seconds, by the topic core wrapper. This message has the same information as a `fd` message, allowing federation members to know their distance in hops to the topic core and the information lifetime. If a topic core stops sending announcements, the nearest topic mesh member assumes the role and starts to send new announcement messages. The timeout for topic core information was set up to two minutes.

Every published message on a broker to a federated topic that has no subscribers to that topic is sent toward the topic core. New topic members announce themselves when a new subscriber is connected to the wrapper's broker, and a `tm` is sent toward the topic core. A mesh membership flag is marked, and the wrapper identifies as part of the mesh. Intermediate wrappers between the new member and the topic core wrapper also adhere to the mesh membership, marking the membership flag on their registers.

In topic meshes, when there is a topology change, as topic membership messages also carry lifetime data, intermediary wrappers that stop receiving these messages become aware that they are no part of the mesh anymore, as no traffic is forwarded, and the topic membership flag is unset.

Topic messages are gathered through general topic monitoring; the wrapper sub-

Figure 5.3: Six-node topology



scribes to the root # topic. Based on the federated topics list, the wrapper filters which topics shall have messages forwarded to the federation. In case of a new message to a federated topic, it is packed in a `tm` packet and sent. This action has no timer adjustments because it is trigger-based; as soon as a message is published, it is sent to the federation.

All validation tests presented satisfactory results, and it is possible to state that the implementation delivered the desired behavior and resources proposed in Spohn's [Spohn 2020] work.

5.2 Performance Tests

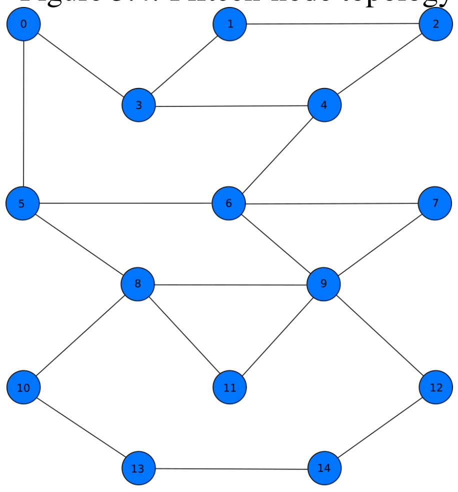
Performance is a very delicate topic in this work. As the implementation aims to be reliable in delivering availability and flexibility, the focus on time-based performance indexes is secondary for the environments which are aimed by the proposed solution.

Compared to cluster-based solutions, convergence in the proposed solution is exceptionally high. Clustering mainly aims for throughput maximization, which means convergence time is essential. But, because clustering-based solutions rely on a unique entity to orchestrate topology, and topology is mainly disposed of parallel elements, delivering high availability needs additional resources, which can consume elevated computational resources [Spohn 2022].

Bass' [Bass 2002] work states that IoT-based solutions for critical-distributed environments are highly indicated and remarks that traffic throughput is less important than availability. Monitoring such infrastructure usually tolerates higher convergence time, with few exceptions. Based on this understanding, it is possible to state that delivering high availability to IoT-based deployments in distributed networks accepts convergence time of minute greatness.

Suppose that in a public illumination environment, one small sector fails. The time needed to repair or recover from failure has more-than-an-hour greatness, sometimes one day. In a distributed network environment, it can be supposed that one or two

Figure 5.4: Fifteen-node topology



network elements will fail, and the topology will change. If part of the network traffic is compromised for 5 minutes, it will not significantly affect the monitoring liability.

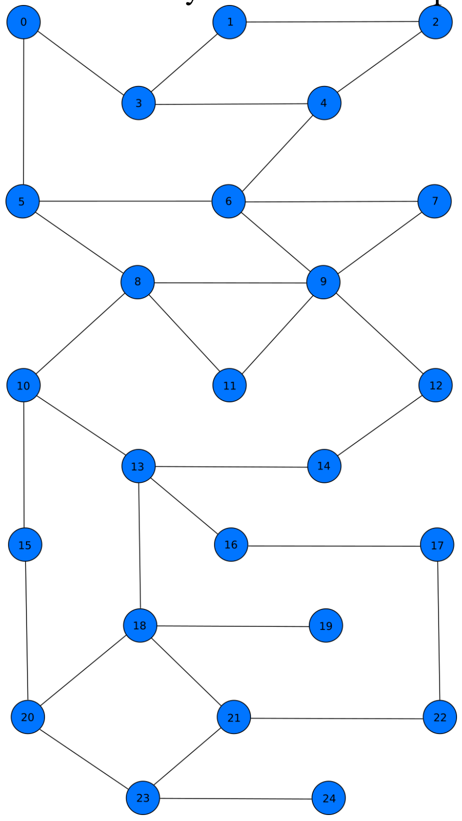
This means that the convergence time observed to rearrange the overlay network, which took one to five minutes, depending on the number of wrappers. Three scenarios were assembled to watch the solution's behavior, one with six wrappers (fig. 5.3), one with fifteen wrappers (fig. 5.4), and one with twenty-five wrappers (fig. 5.5).

The convergence time for federation topology in the six-wrapper topology stands around one minute. Tests were made removing wrappers number three and number one and three simultaneously. The convergence time was unaffected if any elements played the core wrapper role. When analyzing topic mesh behavior, the convergence time was higher, standing around two minutes to converge, including the cases where the failed wrappers were playing topic core roles. There were six different federated topics, with five subscribers and five publishers connected randomly.

When the number of connected wrappers is higher, convergence time is also higher, but there is no direct time-population ratio. Timers are essential in avoiding network flooding and delivering acceptable behavior to the self-managed meshes. Tests were made with one to four failed wrappers, randomly chosen. It is important to note that failing nodes on the edge of the mesh do not significantly affect performance. Convergence in the fifteen-wrapper topology took up to two minutes for the overlay network and between two and three minutes for the topic mesh.

In a more dense topology, tests were made with up to eight randomly chosen failed wrappers, and the convergence time for the overlay network was between two to three minutes and between two to five minutes for the topic mesh. There was no significant

Figure 5.5: Twenty-five-node node topology



time variation when node edges failed.

Network partitioning was also tested, with a higher time-to-converge in the coreless partitions, but the difference remained below half a minute.

One can argue that missed messages can not be recovered, which is an undeniable truth. But when remembering the scope of the proposed solution, missed messages are tolerable, as the environments for which this proposal fits have a high tolerance for this scenario.

6 CONCLUSION

This work presented one Python-written and wrapper-based implementation proposal to the innovative work published by Spohn [Spohn 2020], which proposes a self-managed network of interconnected brokers for high-availability environments for IoT. This means the environment shall address availability besides time-based performance, tolerating high-time convergence behavior.

This work resulted in a wrapper that can be attached to a Mosquitto MQTT broker, delivering the ability to interconnect with other brokers and make topics and their data available network-wide. The wrapper development was aimed at simplicity and low use of computational resources.

The wrapper coding was made over a single file, using multithreading resources and relying upon MQTT-aimed support libraries, allowing easy integration to the MQTT Broker. Because of its simplicity in integrating with the Mosquitto MQTT broker, it is proper to state that any MQTT broker implementation can be used if it allows subscriber information to be published on an internal topic. In Mosquitto, the subscriber information log can be directed to the `$SYS` topic, which the broker accesses. Data is filtered to gather information related to topics that have been subscribed. All other data regarding topics are gathered from the # topic.

The implementation was tested in three different topologies, each with six, fifteen, and twenty-five interconnected wrappers. Tests were made aiming for two objectives: first, to validate the implementation, as it is based on an innovative model, and second, to test convergence times in cases of node failures.

The first objective was achieved, as the wrapper behaved as proposed in all tested environments. The activities of listing connected neighbors, assembling the federation, electing the federation core wrapper, addressing disputes with the federation core wrapper election, identifying new topic subscribers and federating new topics, announcing federated topic core, addressing federated topic core disputes, and, finally, addressing topology changes, where handled correctly on all proposed environments, allowing confidence in the implementation.

Performance tests were made focusing on high availability and resilience. All tests showed the brokers' ability two deliver these two properties, not focusing on convergence time. The proposed applications for the solution are high-time tolerant, allowing some data to be missed and minute-high convergence.

Tests showed that convergence could take up to five minutes, depending on the scenario, but these numbers are acceptable. There was no direct ratio between the number of wrappers in the network and the time to converge, but, as expected, denser topologies needed more time.

The number of publishers and subscribers was out of scope, and the simulations used a fixed number of topics, subscribers, and publishers connected randomly to the brokers. This was because the main goal was validating the implementation behavior and convergence time on the overlay network, mainly and secondarily convergence time on topic meshes, not including traffic volume interference.

The results showed that the proposed solution confidence is high in test environments but still has open topics that must be addressed to deliver a fully-functional product for real-world use.

6.1 Open Topics and Future Work

This work aimed at implementing and validating the theoretical proposal but focusing mainly on functionality. This led to a few open topics that can be further investigated and implemented. It is known that further development is needed to improve resources related to security and the quality of service that MQTT provides.

Regarding security, MQTT provides authentication and encryption. Authentication can be implemented in several ways, such as ACLs (Access Control Lists), RADIUS integration, and certificates, as regular clients can authenticate to the broker without issues [MQTT 2022]. But the wrapper, as it connects to the broker, can also be authenticated using Paho library resources. Also, authentication between wrappers was out of the scope of the present work, but it also can be in the open topics list.

Encryption is also a desired resource to enforce security between brokers. Python offers several libraries that can be used to implement such functionality [Python 2022], and strategies to implement it can be explored and turned into research work.

Further work regarding the present implementation can encompass deep research into performance issues, fine-tuning for faster convergence time, and exploring different and denser traffic scenarios.

REFERENCES

BASS, T. The federation of critical infrastructure information via publish-subscribe enabled multisensor data fusion. In: **Proceedings of the Fifth International Conference on Information Fusion. FUSION 2002. (IEEE Cat.No.02EX5997)**. [S.l.: s.n.], 2002. v. 2, p. 1076–1083 vol.2.

BAUER, M. et al. Iot reference model. In: ____. **Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 113–162. ISBN 978-3-642-40403-0. Disponível em: <https://doi.org/10.1007/978-3-642-40403-0_7>.

CASTELLANI, A. P. et al. Architecture and protocols for the internet of things: A case study. In: **2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)**. [S.l.: s.n.], 2010. p. 678–683.

DE LACERDA MACHADO Jr., J. F.; SPOHN, M. A.; GRANVILLE, L. Z. Client-Transparent and Self-Managed MQTT broker federation at the application layer. In: **2023 International Conference on Computing, Networking and Communications (ICNC): Network Algorithms and Performance Evaluation (ICNC'23 NAPE)**. Honolulu, USA: [s.n.], 2023.

ECLIPSE. **Eclipse Mosquitto**. 2018. Accessed: 2022-12-14. Disponível em: <https://mosquitto.org/>.

EMQX. **EMQX Website**. 2022. <https://emqx.com>. Accessed: 2022-08-19.

FIROUZI, F. et al. Iot fundamentals: Definitions, architectures, challenges, and promises. In: ____. **Intelligent Internet of Things: From Device to Fog and Cloud**. Cham: Springer International Publishing, 2020. p. 3–50. ISBN 978-3-030-30367-9. Disponível em: <https://doi.org/10.1007/978-3-030-30367-9_1>.

HIVEMQ. **MQTT MQTT 5 Essentials**. [S.l.]: HiveMQ GmbH, 2020. ISBN 9783000679131.

HIVEMQ. **HiveMQ Website**. 2022. <https://hivemq.com>. Accessed: 2022-04-19.

Ji, S. et al. Towards scalable publish/subscribe systems. In: **2015 IEEE 35th International Conference on Distributed Computing Systems**. [S.l.: s.n.], 2015. p. 784–785.

LONGO, E.; REDONDI, A. E. Design and implementation of an advanced mqtt broker for distributed pub/sub scenarios. **Computer Networks**, v. 224, p. 109601, 2023. ISSN 1389-1286. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1389128623000464>.

LONGO, E. et al. Mqtt-st: a spanning tree protocol for distributed mqtt brokers. In: **ICC 2020 - 2020 IEEE International Conference on Communications (ICC)**. [S.l.: s.n.], 2020. p. 1–6.

MQTT. **The standard for IOT messaging**. 2022. <https://mqtt.org/>. Accessed: 2023-03-10.

NAIK, N. Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP. In: **IEEE International Systems Engineering Symposium (ISSE)**. Vienna, Austria: [s.n.], 2017. p. 1–7.

OASIS. **MQTT 3.1.1 Specification**. 2014. Accessed: 2022-12-14. Disponível em: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718009>.

PAHO, E. **Eclipse Paho Website**. 2022. <https://www.eclipse.org/paho/>. Accessed: 2022-10-20. Disponível em: <https://pypi.org/project/paho-mqtt/>.

PYTHON. **Python Website**. 2022. <https://python.org>. Accessed: 2022-10-20.

RIBAS, N. K. **Federação de Brokers do Protocolo MQTT: Implementação e Análise de Desempenho**. Trabalho de conclusão de curso - Bacharelado em Ciência da Computação — Universidade Federal da Fronteira Sul, 2022.

RUENPITAK, C. et al. Scalable distributed broker system for very large mqtt networks. In: **2022 19th International Joint Conference on Computer Science and Software Engineering (JCSSE)**. [S.l.: s.n.], 2022. p. 1–6.

SAGAR, A. **Take a ride on the Infobus**. 2003. Accessed: 2023-04-10. Disponível em: <https://www.comscigate.com/JDJ/archives/0302/sagar/index.html>.

SCHIFFER, A. **How a fish tank helped hack a casino**. 2017. <https://www.washingtonpost.com/news/innovations/wp/2017/07/21/how-a-fish-tank-helped-hack-a-casino/>. Accessed: 2023-02-19.

SPOHN., M. An endogenous and self-organizing approach for the federation of autonomous mqtt brokers. In: INSTICC. **Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS,**. [S.l.]: SciTePress, 2021. p. 834–841. ISBN 978-989-758-509-8. ISSN 2184-4992.

SPOHN, M. A. Publish, subscribe and federate! **Journal of Computer Science**, Science Publications, v. 16, n. 7, p. 863–870, Jul 2020. Disponível em: <https://thescipub.com/abstract/jcssp.2020.863.870>.

SPOHN, M. A. On mqtt scalability in the internet of things: Issues, solutions, and future directions. **J. Electron. Electric. Eng**, v. 1, n. 1, 2022. Disponível em: <https://ojs.wiserpub.com/index.php/JEEE/issue/view/jeee.v1i12022>.

STAMPS, M. **Are Internet of Things Devices Invading your Privacy?** 2021. <https://blog.techguard.com/internet-of-things-invading-your-privacy>. Accessed: 2023-03-22.

URAMOTO, N.; MARUYAMA, H. Infobus repeater: a secure and distributed publish/subscribe middleware. In: **Proceedings of the 1999 ICPP Workshops on Collaboration and Mobile Computing (CMC'99). Group Communications (IWGC). Internet '99 (IWI'99). Industrial Applications on Network Computing (INDAP). Multime**. [S.l.: s.n.], 1999. p. 260–265.

VMWARE. **RabbitMQ**. 2022. Accessed:2023-01-04. Disponível em: <https://www.rabbitmq.com/>.

ZOUGANELI, E.; SVINNSET, I. E. Connected objects and the internet of things — a paradigm shift. In: **2009 International Conference on Photonics in Switching**. [S.l.: s.n.], 2009. p. 1–4.

APPENDIX A — RESUMO EXPANDIDO

Federação de Brokers MQTT Transparentes ao Cliente e Auto Gerenciada em Nível de Aplicação

Escalabilidade em sistemas para IoT é um tópico que ainda apresenta espaço para inovação. Estes ambientes possuem características muito particulares, como, por exemplo, a necessidade de gerenciamento cuidadoso de tráfego e escalabilidade. Visando entregar recursos simples e focados na conservação de energia, resistenstes a ambientes de rede com características não ideais de latência e perdas, o protocolo MQTT suge como um divisor de águas e passa a ser amplamente adotado. Porém, soluções de escalibilidade para ambientes de IoT têm focado intensamente em soluções baseadas em cluster pela facilidade de gerenciamento e pela evolução tecnológica, que entregou equipamentos e soluções de conectividade e de processamento mais confiávies [DE LACERDA MACHADO Jr., Spohn e Granville 2023].

Porém, ambientes distribuídos possuem características um pouco diferentes, que, através de um ambiente com escalabilidade centralizada como clusters, não possuem soluções maduras, ou estas são comerciais, sem um detalhamento da arquitetura e comportamento. Outro ponto a ser levantado é a necessidade de um elemento central para orquestração em ambientes centralizados, papel feito pelo balanceador de carga, ou *load balancer*, que oferece risco de disponibilidade por ser um ponto único de falha na arquitetura [Spohn 2022].

Visando contornar esse risco em ambientes distribuídos que requeiram alta disponibilidade, Spohn [Spohn 2020] propôs um modelo de solução focada em escalabilidade horizontal para brokers MQTT que se utiliza de princípios de comunicação para redes *ad hoc*, onde tarefas de alto nível, como roteamento e orquestração, são tratadas em nível de aplicação, uma vez que os elementos interconectados em rede não necessariamento podem ter capacidade de comuncação direta.

O modelo proposto por Spohn divide a orquestração da solução em dois elementos distintos. A orquestação da federação que trata da organização de todos os elementos membros, criando uma rede *overlay* que entrega comnuicação fim-a-fim. E, sobreposta a essa rede, se organizam malhas vinculadas a tópicos, criando um elemento sobreposto com orquestração autônoma, destacada da rede *overlay*. As malhas vinculadas a tópicos permitem que assinantes de tópicos recebam atualização de dados publicados independentemente do broker no qual estejam conectados, e de igual forma, independentemente do broker onde a informação seja publicada.

O modelo proposto por Spohn é o fundamento do presente trabalho, que tem como objetivo apresentar uma proposta de implementação do modelo. Para isto, a linguagem de programação Python foi escolhida, pois entrega recursos para integração com soluções baseadas em MQTT, além de possuir uma diversidade de bibliotecas signigicativa que suportam o desenvolvimento rápido e simples de aplicações. A implementação foi realizada através do desenvolvimento de um *wrapper* que se integra a uma instância regular do broker Eclipse Mosquitto de maneira transparente, e oferece os recursos de federação da proposta fundamental.

A implmementação possui dois grandes grupos funcionais, sendo um grupo principal, com quatro subgrupos - federador, orquestrador, integrador e encaminhador - e um de suporte, com seis funções secundárias, mas não menos importantes - envio e recebimento de pacotes de rede, inspetor de pacotes, *logging* e [**?**], e um gerenciador de *cache*.

A orquestrção é feita através de dez diferentes tipos de mensagens, suportando tanto a orquestração da rede *overlay* como as malhas de tópicos, além de permitirem notificações de assinantes de tópicos federados bem como o transporte dos dados publicados nos respectivos tópicos.

Para validar a implementação do modelo, foram, inicialmente feitos testes utilizando uma topologia com seis elementos, interligados aleatoriamente. Os testes permitiram identificar que a implementação apresentou o comportamento esperado.

Para a realização de testes de escala, foram utilizados três cenários diferentes, com topologias de seis, quinze e vinte e cinco elementos. Os testes tiveram como objetivo verificar o tempo de convergência da orquestração e a correta reorganização das malhas de forma autônoma em caso de falha de elementos aleatórios.

Observou-se que o tempo de convergência é relativamente alto em comparação a soluções *clusterizadas*, porém, como o ambiente de alta disponibilidade para o qual a solução foi pensada suporta este perfil de tempo, entendeu-se que o comportamento estava adqueado.

Finalmente, concluiu-se que a implmentação entregou o comporatmento desejado, porém deixando alguns pontos a serem explorados futuramente, como refinamentos em relação ao tempo de convergência e a implmementação de recursos de autenticação e segurança, que estiveram fora do escopo do presente trabalho.

APPENDIX B — PUBLISHED PAPER - ICNC 2023 - QUALIS A2

J. F. d. L. Machado, M. A. Spohn, and L. Z. Granville, "**Client-Transparent and Self-Managed MQTT Broker Federation at the Application Layer**," 2023 International Conference on Computing, Networking and Communications (ICNC), Honolulu, HI, USA, 2023, pp. 603-607, doi: 10.1109/ICNC57223.2023.10074556.

Abstract: The use of IoT devices for monitoring and automation became very disseminated. Also, as a consequence, IoT scalability issues evolved into one of the main challenges on large deployments. One of the most adopted architectures for the communication between IoT devices and other information systems is based on the Publish/Subscribe paradigm, mainly embraced by MQTT-capable devices. Some implementations aimed to solve scalability challenges in those environments, mainly using clustering solutions, while a few considered federation approaches. Existing solutions are predominantly proprietary, lacking public documentation, and may be considered incipient. In the present work, we propose a client-transparent and self-managed solution for scaling MQTT brokers using a federation approach through a Python-written wrapper, providing federation functionalities and message routing without customizing regular brokers. While clustering solutions usually target throughput improvement, the federation approach explores higher availability through distributed architecture. We present a validation to expose our solution's flexible availability and capability to deal with topology change issues.

URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=10074556isnumber=10073976>

Client-Transparent and Self-Managed MQTT Broker Federation at the Application Layer

José Fernando de Lacerda Machado Jr.[*], Marco Aurélio Spohn[†], Lisandro Zambenedetti Granville[*]
[*]Institute of Informatics - Federal University of Rio Grande do Sul - Porto Alegre, Brazil
[†]Federal University of Fronteira Sul - Chapecó, Brazil
lacerda.machado@ufrgs.br, marco.spohn@uffs.edu.br, granville@inf.ufrgs.br

Abstract—**The use of IoT devices for monitoring and automation became very disseminated. Also, and as a consequence, IoT scalability issues evolved into one of the main challenges on large deployments. One of the most adopted architectures for the communication between IoT devices and other information systems is based on the Publish/Subscribe paradigm, mainly embraced by MQTT-capable devices. Some implementations aimed to solve scalability challenges on those environments, mainly using clustering solutions, while a few considered federation approaches. Existing solutions are predominantly proprietary, lacking public documentation, and may be considered incipient. In the present work, we propose a client-transparent and self-managed solution for scaling MQTT brokers using federation approach through a python-written wrapper, providing federation functionalities and message routing without customization of regular brokers. While clustering solutions usually target throughput improvement, the federation approach explores higher availability through distributed architecture. We present a validation to expose our solution's flexible availability and its capability to deal with topology change issues.**

I. Introduction

The MQ Telemetry Transport (MQTT) protocol has been widely adopted on Internet of Things (IoT) devices communication [1] [2], not only because of its low data overhead but also because of its reliability and strong standardization. That is due not only to MQTT's maturity but also to its ease of implementation. As a result, the adoption and deployment of MQTT-enabled devices and the solutions based on the platform became popular [3].

MQTT employs the Publish/Subscribe (P/S) paradigm where a broker intermediates the communication between publishers (*e.g.*, IoT devices) and subscribers (*e.g.*, other devices or applications that consume the publishers' offered information). Because IoT devices (playing the role of either publishers or subscribers) usually suffer from limited resources (battery capacity, processing power, and communication support), MQTT-based solutions aim at dealing with the limitations of IoT deployed setups [4]. Scaling up such systems by supporting an increasing number of publishers and subscribers also has to consider scaling up MQTT brokers to avoid them becoming bottlenecks in environments with varying amounts of flowing information. There exist several solutions for MQTT-based setups that employ clustering [5] [6] [7] as a technique to deal with the brokers' scalability issue. Unfortunately, most of those solutions are only available in commercial products. As an alternative to the clustering techniques, the use of federation techniques [8] are being considered, but still less mature than clustering.

Although attempts to solve the scalability problem in MQTT are in place, as mentioned above, the central fact is that MQTT implementations with improved scalability are scarce and often limited to proprietary products. As such, scaling up MQTT environments by employing a public, freely available solution is still a need. Implementing such functionality is thus an opportunity and potential game-changer, providing elastic capabilities for a broad set of application scenarios.

In a previous work [9], we proposed a self-managed MQTT federation that offers the first movement toward building and maintaining an overlay mesh network of autonomous brokers. Clients (*i.e.*, publishers, and subscribers) communicate over this self-organizing mesh network with low control overhead. The materialization of the above solution can range from a more intrusive one (*i.e.*, requiring changes to the broker) to the one in which the federation mechanism, which internally also relies on the P/S paradigm, stays exclusively at the application layer (*i.e.*, brokers remain unchanged) [10].

In this paper, we advance our previous research by presenting the design, implementation, and case study of a federation module for MQTT as a wrapper. Our solution, written in Python 3.8 and attached to a Mosquitto 3.1.1 MQTT broker, is flexible, easy-to-implement, and better scales up because of our self-managed federation approach. Our implementation seeks to be the least intrusive possible to the ordinary MQTT environment.

The remainder of this paper has the following organization. In Section II, we review related work. In Section III, we introduce our MQTT wrapper and detail how a set of wrappers in the mesh network federate. We show our implementation and case study in Section IV. In Section V, we conclude this paper with final remarks and future work.

II. Related Work

Several research efforts have addressed scaling up P/S systems, some based on clustering and others based on federation strategies. Bakker and Pattenier [11] present federation strategies on networked systems. They focus on two leading solutions: federation for connection-oriented networks and federation for connection-less networks, as those based on TINA-C NRA (Telecommunications Information Networking

Architecture Consortium - Network Resource Architecture), mainly employed in telecommunications environments.

Uramoto and Maruyama [12] present the InfoBus Repeater application, conveying a unique approach for scaling up MQTT environments throughout a bus. The application, written in Java, acts as a middleware that allows intercommunication between group members. When a member joins the bus, it informs its status as a publisher, subscriber, or both. The bus is a single point of failure and a bottleneck, providing limited scalability.

Bass [13] carried out a structured analysis of a P/S federated network approach for critical infrastructure environments. The work evaluates assembling models, the compliance of existing solutions, and the security aspects and characteristics of possible topologies. Although it is primarily theoretical, the work delivers a broad view of possible scenarios and solutions for assembling federated networks of sensors.

Thean *et al.* [14] presents a work based on clustering MQTT brokers to deal with edge computing demands. They propose an architecture for scaling a cluster of container-based MQTT brokers on the edge of the environment, each acting as a bridge to the brokers placed on a cloud infrastructure. Even though their solution provides enhanced scalability results, the container orchestrator remains a single point of failure.

In previous work, we [8] presented a generic approach for federating MQTT brokers following a mesh-assembling mechanism over an overlay network. The solution includes an overlay mesh network that provides the primary communication system, allowing the arrangement of topic meshes, reaching all clients for any federated topic, regardless of where the client connects. A new mesh forms whenever the first subscriber connects to any broker. All brokers with subscribers for the same topic, and any broker that interconnects them, integrate the topic mesh. A mesh has a core broker that coordinates the mesh construction and maintenance. The routing of topic messages begins by forwarding them toward the corresponding topic core, but as soon as the message reaches a mesh member, the message spreads throughout the mesh.

Afterward, we [10] proposed an implementation based on an endogenous approach [9]. The solution employs the native P/S mechanism for managing the meshes and routing of messages. Subscribers must send a beacon message to advertise themselves, but the federation of brokers is primarily transparent to the clients.

III. Federation proposal

We present a self-managed wrapper-based federation solution coded in Python and integrated into the Eclipse Mosquitto 3.1.1 MQTT broker. A wrapper is software capable of interacting with the MQTT broker transparently. For that, the wrapper monitors the broker's log, which has its data redirected to a topic called `$SYS`. The wrapper gathers data from all topics, being possible to monitor specific topics and subtopics for later forwarding its data to all federated members. This strategy makes it possible to adapt this solution to other known MQTT broker implementations capable of redirecting log data to a specific topic. Each broker to be federated shall have a wrapper attached and be responsible for communicating with the neighboring set of brokers and wrappers.

The wrapper follows the principles of the federation mechanism proposed by [8], on which the brokers federate through their neighbors, building a mesh. Each federated broker forwards messages through the mesh, allowing communication between brokers that are not neighbors. Information needed to manage the federation includes a broker identification number (*BrokerID*), the distance in hops to the *core broker* (*i.e.*, the broker that coordinates the mesh), the *mesh membership flag* (signaling whether the broker is in a given mesh), a list of neighboring brokers, and the desired mesh redundancy (achievable when the overlay topology allows).

A. Architecture

The main element of our architecture is the wrapper that interacts with the broker and allows communication with neighboring wrappers and their related brokers. The wrapper interacts with the broker monitoring its logs and all the topics and subtopics, so we assume that the wrapper can access all data generated on the broker.

The wrappers interact with other wrappers through network sockets, allowing data exchange. We chose UDP for transporting data between brokers, as a missing packet may arrive on a wrapper through different paths, so delivery confirmation is not critical. The main goal is to deliver data published on a given topic on a broker with a federated wrapper by forwarding it to another broker through its wrapper where there are connected subscribers to that given topic. Figure 1 presents the schematic architecture.

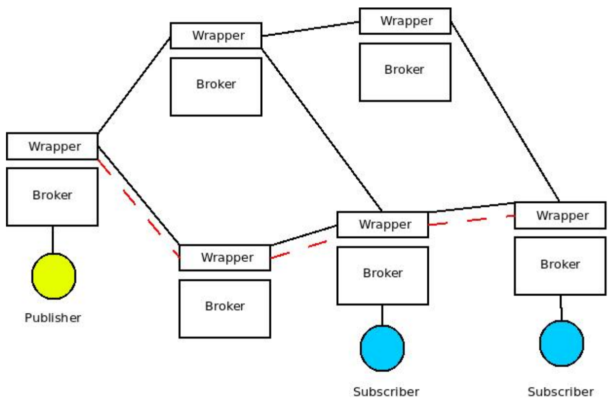


Fig. 1. System's architecture overview

B. Behavior

The wrappers' federation process encompasses two main phases. First, the federation assemblage, regarding the overlay infrastructure that provides communication among federated wrappers. Second, the mesh building and maintenance provide the means for transporting data between wrappers connected to brokers with publishers and subscribers. Regular topics with subscribers will be called federated topics.

1) Federation Assemblage: Federation assemblage assumes that every wrapper knows its neighbors' IP addresses. Periodical *hello* messages are exchanged between neighbor wrappers to keep track of their online status. Announcement messages with an identification number (BrokerID) for the wrapper are also sent periodically, which are forwarded through their neighbors to the other interconnected wrappers, allowing the assemblage of the overlay network (i.e., nodes learn about all federated brokers and their distances). As the announcements propagate, the wrapper nodes settle collisions by randomly selecting a new ID.

As the federation starts, there is a need to elect a core wrapper to coordinate the updates on the federation topology and other management matters. The election is based on the BrokerID number, winning the node with the lowest ID. This core wrapper will be called the management core.

2) Topic meshes: Topic meshes are the meshes of wrappers where there are subscribers to a given topic. They start along the federated network. When a wrapper detects a first subscriber for any topic for which there is no mesh, the wrapper advertises itself as the core for the new mesh. Through a *core announcement* message, wrappers learn how to reach any previously established core. Management and topic cores are similar, with the latter being the reference point for starting topic meshes, while the former orchestrates the overlay network. The node with the lowest ID wins the dispute in a core announcement contention.

If there is a topic core on a topic with a new subscriber, the wrapper sends a mesh membership announcement toward the topic core. If there are two or more paths towards the core with the same distance, it is possible to handle redundancy. The new membership may trigger intermediate wrappers to join the mesh to keep it connected. As for publications, the topic core is the reference target: the wrapper sends the message toward the core, and once reaching it or a mesh member first, the message spreads throughout the mesh. Wrappers keep a local cache to avoid sending the same message indefinitely, considering that the same publication may arrive through different paths.

IV. Implementation and Validation Tests

In our implementation, we have used Python version 3.8, supported by libraries providing MQTT functions such as threading, serialization, and socket connections. The Paho-MQTT library is also used and plays the most crucial role in the solution, allowing the interaction between the wrapper and the broker and entitling monitoring of subscriptions and messages on topics.

The wrapper has five main functional groups and the initial data setup, with this latter including information regarding IP addresses and ports, constants, initial broker identification, and the list of neighbors. By default, wrapper instances communicate through UDP port 10500.

The first functional group handles low-level network-related activities, such as packet sending and receiving. For performance purposes, we used UDP on the transport layer. There are two main functions - packet sending and packet receiving. Packets that need to be redirected are also handled over these functions.

The second functional group handles the operational functions. Packets have a function identifier and are handled according to their *message types*. A packet's structure comprises a *message type* field, followed by the data regarding that particular message.

```
('fd', (14, 1, ('10.81.180.217', 7535)))
```

There are ten different types of messages: hello (hl), hello back (hb), federate (fd), reconsider (rc), core announcement (ca), topic core (tc), reconsider topic core (rt), topic subscribe (ts) topic message (tm) and topology update (tu). Federate, core announcement, and topic core announcement have detailed information on message sequence numbering for control purposes. For instance, the structure of a federation message is as follows:

```
('fd', (14, 1, ('10.81.180.217', 7535)))
           (seq, dist, (ip, id)
```

The *fd* field indicates the message type, followed by a tuple that carries the announcement sequence number, the hop distance from the announcer, and its BrokerID - which also consists of a tuple holding the IP address and the node's virtual ID. The source broker defines the sequence number, while the distance field changes as the message moves farther from the source.

A group of processes runs as *daemons*. These processes are responsible for the hello, federation, core announcements, MQTT broker monitoring, and maintenance. By default, nodes transmit federation and core announcements every 20 s and hello messages every 5 s. The latter group comprises additional functions, such as log generating, cache flushing, and debugging.

The environment for validating our solution consisted of a regular desktop computer (4th generation quad-core i7 with 8GB of RAM) running Linux Mint 20.3 and Oracle VirtualBox 6.1.38 hypervisor with a Ubuntu 22.04 virtual machine guest, using 4GB of RAM and 35GB of disk space. LXD 5.0 was used to host LXC containers inside the virtual machine, using the mainstream Ubuntu 20.04 LTS image. Each container had Mosquitto 3.1.1 installed alongside Python 3.8.10, with Paho-MQTT 1.6.1 library installed through Pip.

For redirecting log messages generated by Mosquitto to the `$SYS` topic, `/etc/mosquitto/mosquitto.conf` needed to be configured with the lines below:

```
log_dest topic
log_type all
```

Each container had only one network interface installed, named 'eth0', connecting to a virtual bridge 'lxdbr0' on the Ubuntu 22.04 host (this is necessary because the solution uses the network interface name to gather its IP address). Each broker/wrapper set on the network corresponds to one container instance. The configuration regarding the neighbors is in a list in the wrapper, as mentioned before. Example:

```
neigh = [('10.81.180.180'),
        ('10.81.180.217')]
```

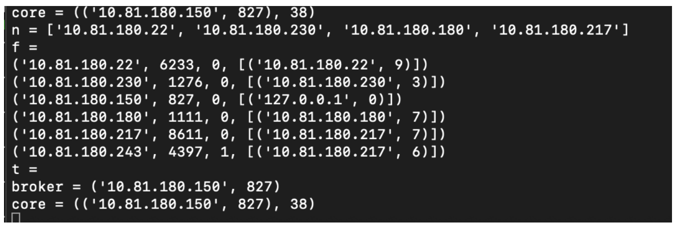
A. Environment remarks and considerations

LXC delivers a handful of scenarios better than a Docker-based environment because it provides a complete operating system experience with minimal memory and processing footprint. Also, Docker needs a new image of the application to be generated each time the source code is changed. With LXC, creating new instances and cloning running instances is straightforward, making scaling the system a trivial task.

B. Validation and testing

For validation purposes, we used the scenario proposed in [8] (Figure 4). The validation aims to check the assemblage of the overlay network and identify whether the wrapper behaves as desired, handling federated topics and message routing.

After configuring and starting all the wrappers on network nodes, the nodes populate their federation list. Figure 2 shows the node's three federation lists (`n =` is the list of neighbors, `f =` is the list of federated nodes, followed by the broker IP and node's ID, then the management core).



```
core = (('10.81.180.150', 827), 38)
n = ['10.81.180.22', '10.81.180.230', '10.81.180.180', '10.81.180.217']
f =
('10.81.180.22', 6233, 0, [('10.81.180.22', 9)])
('10.81.180.230', 1276, 0, [('10.81.180.230', 3)])
('10.81.180.150', 827, 0, [('127.0.0.1', 0)])
('10.81.180.180', 1111, 0, [('10.81.180.180', 7)])
('10.81.180.217', 8611, 0, [('10.81.180.217', 7)])
('10.81.180.243', 4397, 1, [('10.81.180.217', 6)])
t =
broker = ('10.81.180.150', 827)
core = (('10.81.180.150', 827), 38)
```

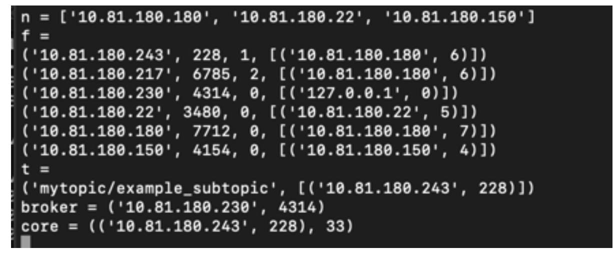
Fig. 2. Node's 3 federation list

The next step consists of simulating a topic subscription and observing the federated topic evolution. We start with a subscriber at node zero. Figure 3 depicts the node's five debugging outputs, showing the federated topic `mytopic/example_subtopic`, and the core for that given topic is node zero - the node that we have the subscriber attached to. Publications for the related topic are forwarded toward node zero and flood the mesh once reaching it.

To test publication routing, we did a publication on node four, monitoring the wrapper, gathering the publication, and sending it toward the core through node one. Next, a publication starts on node five, which has route redundancy towards the core. In this case, we randomly choose one of the neighbors in the message-forwarding process. In all situations under consideration, the publications successfully reach the subscriber.

C. Performance analysis

When analyzing our solution's performance, it is imperative to differentiate the key performance indicators between



```
n = ['10.81.180.180', '10.81.180.22', '10.81.180.150']
f =
('10.81.180.243', 228, 1, [('10.81.180.180', 6)])
('10.81.180.217', 6785, 2, [('10.81.180.180', 6)])
('10.81.180.230', 4314, 0, [('127.0.0.1', 0)])
('10.81.180.22', 3480, 0, [('10.81.180.22', 5)])
('10.81.180.180', 7712, 0, [('10.81.180.180', 7)])
('10.81.180.150', 4154, 0, [('10.81.180.150', 4)])
t =
('mytopic/example_subtopic', [('10.81.180.243', 228)])
broker = ('10.81.180.230', 4314)
core = (('10.81.180.243', 228), 33)
```

Fig. 3. Example of a federated topic with a subscriber attached to node 0

clustering and federation. Clustering relies on aggregating computational resources as a single block, having the load balancer as the main bottleneck and single point of failure. On the other hand, the federation relies on orchestrating distributed resources with multiple access points. Therefore, one could argue that the federation first targets the service's high availability, while clustering aims at high throughput.

The performance analysis evaluates the message delivery reliability while dealing with changes to the overlay topology. For the case studies, we consider the analysis starting after an initial configuration for the federation of brokers is running.

The first test consists of running a subscriber for a non-existent topic. The corresponding node advertises itself as the core for the new topic. The topic mesh construction begins as new subscribers for the same topic connect to different nodes. To test the message routing, a client starts publishing from a node outside the mesh. Initially, messages are directed toward the core, spreading throughout the mesh once reaching the core or a mesh member.

We evaluate the core election process by simultaneously instantiating two subscribers to the same topic in separate nodes. For a while, two core announcements wander around the federation, but eventually, the core with the greater ID gives way to the one with the minor ID. Once nodes learn about the remaining core, the mesh construction process converges. In the case of network partitioning, there will be two different scenarios. In the first, the slice that kept the existing core will rely on it to identify disconnected nodes, and the remaining nodes will be informed to update their databases. In the second, the remaining brokers on the coreless slice will identify that they are not receiving core information updates and will orchestrate a new core election.

V. Conclusions

This work presents a new variant for the federation approach introduced in previous works [8]–[10]. Our solution differs by adopting a wrapper cooperating with the broker while communicating to other wrappers running on neighboring brokers. This proposal is ongoing work and lacks functionalities available on a regular Mosquitto MQTT broker, such as QoS controls and authentication, which are part of future work. However, we achieve our main objective: to provide and evaluate a self-managed federation of MQTT brokers.

We noticed that the initial federation process (*i.e.*, related to the overlay network) requires more time than anticipated. It

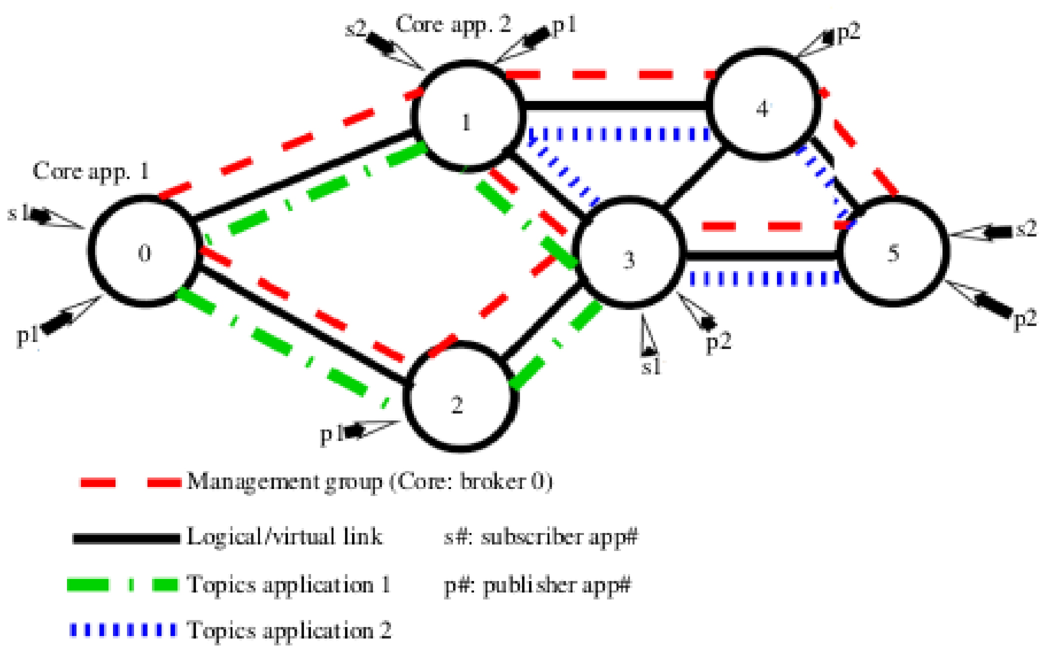


Fig. 4. Validation topology

shows it is worth improving all the topology-related services, from the initial overlay establishment to all the resulting maintenance. We are working on new mechanisms for better handling joining and leaving the federation network.

Monitoring active subscribers is still an open issue. For now, we assume the connection between a subscriber and the broker is stable and remains connected indefinitely. On the other hand, intermittent connecting clients might get new IDs when reconnecting, which becomes a broader monitoring burden.

The present work opens an extensive list of possibilities for improving the federation of MQTT brokers. Compared to other solutions, mainly proprietary market-driven solutions, our solution covers a particular domain, and the initial prototype shows our proposal's potential. The increased availability comes from the self-managing mesh approach, which is central to our work. In case of network partitioning or general connectivity problems, the service is still available for the clients in the same partition. As partitions merge again, the system converges quickly with reduced control overhead.

References

[1] M. Houimli, L. Kahloul, and S. Benaoun, "Formal Specification, Verification and Evaluation of the MQTT Protocol in the Internet of Things," in *International Conference on Mathematics and Information Technology (ICMIT)*, Adrar, Algeria, Dec. 2017, pp. 214–221.

[2] N. Naik, "Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP," in *IEEE International Systems Engineering Symposium (ISSE)*, Vienna, Austria, Oct. 2017, pp. 1–7.

[3] M. Kashyap, V. Sharma, and N. Gupta, "Taking MQTT and NodeMcu to IOT: Communication in internet of things," *Procedia Computer Science*, vol. 132, pp. 1611 – 1618, 2018, international Conference on Computational Intelligence and Data Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050918308585

[4] I. Made Wirawan, I. Dwi Wahyono, G. Idfi, and G. Radityo Kusumo, "Iot communication system using publish-subscribe," in *2018 International Seminar on Application for Technology of Information and Communication*, 2018, pp. 61–65.

[5] HiveMQ, "HiveMQ website," https://hivemq.com, 2022, accessed: 2022-04-19.

[6] EMQX, "EMQX website," https://emqx.com, 2022, accessed: 2022-08-19.

[7] VerneMQ, "VerneMQ website," https://vernemq.com, 2022, accessed: 2022-08-19.

[8] M. A. Spohn, "Publish, subscribe and federate!" *Journal of Computer Science*, vol. 16, no. 7, pp. 863–870, Jul 2020. [Online]. Available: https://thescipub.com/abstract/jcssp.2020.863.870

[9] M. Spohn., "An endogenous and self-organizing approach for the federation of autonomous mqtt brokers," in *Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS,*, INSTICC. SciTePress, 2021, pp. 834–841.

[10] N. K. Ribas and M. A. Spohn, "A new approach to a self-organizing federation of mqtt brokers," *Journal of Computer Science*, vol. 18, no. 7, pp. 687–694, Jul 2022. [Online]. Available: https://thescipub.com/abstract/jcssp.2022.687.694

[11] J. H. L. Bakker and F. J. Pattenier, "The layer network federation reference point-definition and implementation," in *TINA '99. 1999 Telecommunications Information Networking Architecture Conference Proceedings (Cat. No.99EX368)*, 1999, pp. 125–127.

[12] N. Uramoto and H. Maruyama, "Infobus repeater: a secure and distributed publish/subscribe middleware," in *Proceedings of the 1999 ICPP Workshops on Collaboration and Mobile Computing (CMC'99). Group Communications (IWGC). Internet '99 (IWI'99). Industrial Applications on Network Computing (INDAP). Multime*, 1999, pp. 260–265.

[13] T. Bass, "The federation of critical infrastructure information via publish-subscribe enabled multisensor data fusion," in *Proceedings of the Fifth International Conference on Information Fusion. FUSION 2002. (IEEE Cat.No.02EX5997)*, vol. 2, 2002, pp. 1076–1083 vol.2.

[14] Z. Y. Thean, V. Voon Yap, and P. C. Teh, "Container-based mqtt broker cluster for edge computing," in *2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, 2019, pp. 1–6.