POLITECNICO DI MILANO
Master's degree in Computer Science and Engineering
Department of Electronics, Information and Bioengineering

Distributed MQTT+: development of a pub/sub broker for distributed environments

ANTLab
Advanced Network Technologies LABoratory

Advisor: Prof. Alessando Redondi
Co-Advisor: Eng. Edoardo Longo

Master's Degree Thesis of: Leonardo Stagliánó
Matricola 917310

Anno Accademico 2020-2021

A Nonna Maria, anche se non ci sei più, questo traguardo, che tanto hai sognato di vivere insieme a me, è per te

Sommario

MQTT (Message Queuing Telemetry Transport) è un protocollo publish/ subscribe il cui utilizzo è cresciuto in maniera sempre più importante negli ultimi anni per applicazioni nell'ambito IoT o di reti wireless di sensori. Per questo motivo è stato soggetto a diversi ambiti di ricerca che, negli ultimi anni, hanno portato a una sua estensione sempre più ampia e a possibilità di applicazione sempre più svariati, non pensati quando fu lanciato da IBM. Tra questi, sviluppati proprio nell'ambito del Politecnico di Milano vi è MQTT+, un'estensione del protocollo originario che ha consentito di effettuare operazioni integrate nello standard, aumentando la complessità degli scenari di utilizzo senza però alterare i principi cardine di efficienza e semplicità che gli ideatori avevano come obiettivo principale.
Parallelamente la necessità di creare infrastrutture distribuite, ha imposto di pensare al modo di consentire l'utilizzo di tale protocollo, insieme ad altri suoi omologhi, all'interno di un ambito di cooperazione di macchine dislocate nelle parti più disparate del pianeta.
Da questi due percorsi nasce questo progetto, che cerca di consentire l'utilizzo dell'estensione MQTT+ anche in un contesto distribuito, assumendo al contempo una valenza rilevante anche dal punto di vista dello sviluppo di un sistema di connessione efficiente e resiliente a eventuali malfunzionamenti.

Abstract

MQTT (Message Queuing Telemetry Transport) is a publish/subscribe protocol whose usage is growing a lot in recent years, for IoT applications or Wireless Sensor Networks. For this reason, it was the subject of several research topics that, recently, brought to a wide extension and several application contexts, not even thought when it was launched on the market by IBM. Among these, developed at the Politecnico di Milano, there is MQTT+, an extension of the original protocol that allowed the execution of aggregated operations integrated into the standard, increasing the complexity of the application scenarios without affecting the main principles: efficiency, and ease of use that the developer had as the main goal.
At the same time, the need of creating distributed infrastructures imposed to think how to enable the usage of this protocol, along with others of the same type, in a context in which several machines, located in the most disparate parts of the world, have to communicate.
These paths brought to the birth of this project that tries to allow the usage of the MQTT+ extension also in a distributed context, having also a relevant value, considering the possibility of developing an efficient and resilient interconnection system.

Ringraziamenti

Ringrazio Ida, la persona che mi ha trasferito la dolcezza e la serenità di cui avevo bisogno durante tutto questo periodo, che mi ha portato a completare il tassello mancante di un percorso faticoso ma bellissimo.

Ringrazio i miei genitori, per l'importanza che hanno sempre dato all'istruzione e alla cultura, che ho percepito fin da piccolo e che mi ha portato a realizzare uno dei miei sogni, studiare e conseguire risultati accademici presso questo prestigioso ateneo.

Ringrazio Domenico e Stefano, che in questi anni sono sempre stati vicini a me, mi hanno supportato e mi hanno aiutato a superare più di qualche giorno difficile strappandomi dei sorrisi.

Ringrazio il Politecnico di Milano, per la professionalità e l'umanità che i suoi membri mi hanno sempre trasmesso, in particolare Alessandro Redondi ed Edoardo Longo, che mi hanno aiutato a portare a compimento questo lavoro col loro prezioso aiuto.

Contents

Sommario **1**

Abstract **2**

Ringraziamenti **4**

1 Introduction **1**
1.1 Overview 1
1.2 Project Objective 2
1.3 Thesis Outline 2

2 State of Art **3**
2.0.1 MQTT+ 3
2.0.2 MQTT Bridging 5
2.0.3 PADRES protocol 6
2.0.4 MQTT-ST 6
2.0.5 Other Distributed Publish/Subscribe paradigms . . . 7

3 Project Structure & General Idea **9**
3.1 MQTT+ Distributed Broker 9
3.2 System-level Perspective 10
3.3 Java Server 13

4 Implementation **15**
4.1 Discovery Protocol 15
4.2 Round Trip Time Computation 17
4.3 STP protocol 20
4.3.1 Root Selection 21
4.3.2 Path Computation 23
4.4 Routing Algorithm 25
4.4.1 Algorithm Adaptation 26
4.4.2 Routing Tables 27
4.4.3 Conclusions and clarifications 34

5 Testing Methodology **35**
5.1 Evaluation Targets 35
5.2 Testing Environment 36
5.2.1 Clients locality 38
5.2.2 Bridging topology 40
5.2.3 Convergence experiments 41
5.2.4 Clarifications and other aspects 42

6 Experimental Results **44**
6.1 Network traffic 44
6.2 Workload 47
6.2.1 CPU workload 47
6.2.2 Memory workload 50
6.3 Convergence results 54

7 Final considerations **57**

Bibliografia **59**

List of Figures

3.1 Deployment diagram for the broker 9
3.3 Explanation of various connections of the system 10
3.2 Example of MQTT+ Distributed System 11
3.4 The system from the point of view of the clients 12
3.5 State diagram that describes the Java Server 13

4.1 Example of a discovery message 15
4.2 Example of the logical connections established in a system of 5 brokers 17
4.3 The green node sends the requests for the computation of the RTT estimations 18
4.4 Example of a possible RTT message request sent by the green node 18
4.5 The responses sent as consequence of the figure 4.4 19
4.6 Example of a response message received by the white node . 19
4.7 Representation of what is stored in memory by each node of the network 20
4.8 Message exchange for the first phase of the tree formation protocol 21
4.9 Example of a packet sent in the first phase of the protocol . . 21
4.10 Message sent at the end of the phase described in this subsection 22
4.11 Situation at the end of the first phase of the STP protocol . . 23
4.12 Example of a packet sent by the brokers during the path computation phase 23
4.13 Example of a snapshot of the system at the end of the protocol 25
4.14 Representation of the situation in memory of each broker at the end of the STP protocol 28
4.15 Representation of the system reaction to a publish message on a topic never encountered 29
4.16 How the SRT tables are filled upon receiving the messages of the figure 4.15 30
4.17 Example route followed by a subscription sent to a broker that filled its SRT 31
4.18 Example of the PRTs obtained by the traffic showed in 4.17 . 33

4.19 Example of a routing of a publish message of an already encountered publication topic 33

5.1 Representation of the setting of the network used for the tests 37
5.2 Example of how the clients may be placed in case of 100% locality 38
5.3 Example of how the clients may be placed in case of 50% locality 39
5.4 Example of how the clients may be placed in case of 0% locality 39
5.5 Topology adopted to perform the tests in the bridging case . 40
5.6 How the convergence period is computed after the startup of a broker 41
5.7 How the convergence period is computed after the failure of a broker 42

6.1 Representation of the obtained traffic dimension (in bytes) differentiated by the algorithm and the locality adopted in the experiment 45
6.2 Average usage of the CPU by the 5 brokers during a test case in which clients are placed with 0% locality 47
6.3 Average usage of the CPU by the 5 brokers during a test case in which clients are placed with 100% locality 48
6.4 Average usage of the CPU by the 5 brokers during a test case in which clients are placed with 50% locality 49
6.5 Average usage of the memory by the 5 brokers during a test case in which clients are placed with 0% locality 51
6.6 Average usage of the memory by the 5 brokers during a test case in which clients are placed with 100% locality 52
6.7 Average usage of the memory by the 5 brokers during a test case in which clients are placed with 50% locality 53
6.8 Average convergence performance for the 5 brokers at the startup of the system 55
6.9 Average convergence performance for the remaining 4 brokers at the failure of one of the brokers 55

List of Tables

6.1 Tabular representation of the traffic (in bytes), considering as excluded the part concerning the publishes forwarding, differentiated by the locality and the routing methodology adopted 45
6.2 Tabular representation of the traffic (in bytes) concerning the publishes forwarding, differentiated by the locality and the routing methodology adopted 45
6.3 Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 0% locality 48
6.4 Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 100% locality 49
6.5 Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 50% locality 50
6.6 Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 50% locality 51
6.7 Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 100% locality 52
6.8 Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 50% locality 53

Chapter 1

Introduction

1.1 Overview

The Internet of Things paradigm is becoming more and more important during the last years as new technologies are pushing its potential to a very high level, moving this world increasingly into the users' everyday life. Smart object, sensors, and a lot of other IoT devices are daily in touch with us, and developing new technologies in this branch assumes a lot of relevance as time passes, this is the reason why this project is born, to increase the functionalities of the current available IoT services and to improve the possibility of enjoying them in a distributed fashion, as mobility is one of the crucial properties for new technologies to be successful nowadays.
The main reason for this incredible success can be identified in the large area of interest that this technology paradigm covers: washing machines, smartwatches, doors, windows, smart bulbs, and many other objects that are used every day by everyone can be included in systems and services that are able to ease a lot of daily tasks of the users, allowing also to enlarge these objects' field of action to new perspective never explored before.
MQTT (*Message Queuing Telemetry Transport*) is an IoT protocol of growing importance, its success is based on ease of use and a large field of applicability. It was developed by *IBM* to have a low impact in terms of processing power and it has been thought for all the situations in which the bandwidth is limited. It is a publish/subscribe protocol and at the start, it was developed to work with a single broker that is responsible for the message handling and forwarding operating independently and as self-contained. In the last years, the research focused on the possibility of integrating this protocol in a distributed environment, making the brokers able to cooperate in order to connect clients that are located in different parts of the world. Its standard is available at [1], a new version, the 5.0, was developed recently and contains lots of improvements like topic aliasing, enhanced authentica-

tion, disconnect message to be sent by the broker to specify the reason of disconnection, and so on.

1.2 Project Objective

The project interests a protocol that is assuming a key role in the development of the importance of IoT technologies during the last years, MQTT. It was the subject of research before and expanded in its functionalities with MQTT+ [5], an extension that gave the possibility of performing new operations on the data collected by the brokers adding a simple syntax to the MQTT well-known standard for the subscriptions.
The missing part in the implementation of the cited project was the possibility of using it in a distributed fashion, inserting each broker into an infrastructure that allows collecting and elaborating data from different locations. The prescriptions of the distributed paradigm impose also resilience and recovery to failure, and, as a technological project demands nowadays, efficient exploitation of the computing power of the nodes, these are other objectives to which the project aims.
Making MQTT+ working according to the paradigm just described imposes us to evaluate different alternatives on how the brokers that make up the entire infrastructure have to communicate, forward information, and keep connected. Many choices were faced, so the motivation behind them, the implementation, and the experimental aspects will be discussed later in this paper.

1.3 Thesis Outline

In chapter 2 the reader can find the state of art for distributed publish/subscribe paradigm, more specifically distributed MQTT technologies and their correlation with the described work are detailed.
In chapter 3 there will be the project general idea to fulfill the objectives described before, the design choices, and the structure of the project.
Chapter 4 is focused on the description of the implementation, how the implementation is structured, and the practical actuation of the project.
Chapter 5 contains a discussion of the choice of the evaluation methods for the obtained result, the evaluation targets, and the motivations for these choices.
In chapter 6 the results of the experimental evaluations are presented, there will be also comments on their meaning with respect to the starting goal of the project.
Chapter 7 explains what are the possible future research paths related to the described project and some final considerations.

Chapter 2

State of Art

MQTT+, the technology at the base of this project, can be considered as completely developed as detailed in [5]. Making it work in a distributed environment requires different properties that, as we will see in the following, are not guaranteed simultaneously by any technology available nowadays. It is possible to summarize these properties as follows:

- **Network traffic efficiency**, to not waste bandwidth for the communication among the brokers that make up the network, as the traffic generated by the clients can be a bottleneck itself.
- **Fault resilience**, the system must be able to recreate correctly the connections among the brokers still available after a failure of a single node of the system. This is a quite common property required for a distributed system.
- **Dynamic overlay creation**, in order to exploit those nodes that are more powerful to relay and process packets as much as possible.
- **The orchestration of the system must not be centralized**, no centralized entity must be required to start the system. The brokers have to be able to join without any pre-existing information about the addresses with which they can connect to the other nodes.

These properties were the guideline during the development of the project, since, as can be seen later in this chapter, they are not simultaneously fulfilled by any of the technology already present in the context of the MQTT protocol.

2.0.1 MQTT+

MQTT+ is the technology on which this work is based. It is an enhanced syntax for MQTT that allows to carry out some operations on data appended in the payload of the normal MQTT publish messages, that the

standard protocol wouldn't allow. This is achieved by simply adding to normal subscriptions the operator, preceded by the $ character and followed by the normal topic filter on which the operator has to act. The classes of operations that MQTT+ adds to the standard can be summarized as follows:

- **Rule based operators**: as the name says they are operators that enforce a rule that filters out the messages related to a certain topic to which a client is subscribed. If a publication related to that topic complies with the rule of the operator, it is forwarded to the subscribed client, otherwise, it is discarded.
  For example the syntax **$EQ;value/topic** applied to subscription filters allows specifying a "rule" according to which the broker forwards the publication to a topic that matches the *topic* part of the filter, if and only if its value is exactly equal to the *value* specified in the syntax above.

- **Temporal aggregation operators**: these operators allow to obtain the average, the minimum, the maximum, and the sum of the messages under a certain topic according to a temporal granularity (daily, quarter-hourly, hourly), using all the last messages published under that topic to carry out one of the above-listed operations, limiting the operands to the temporal range specified.
  As an example with the syntax **$DAILYSUM/topic**, the sum of all the payloads of the publications related to the *topic*, in a span of 24 hours from the moment the subscription has been received, are summed up and returned to the subscriber as soon as the timer expires.

- **Spatial aggregation operators**: the allows to specify one of the mathematical operation listed in the previous point of the bullet-list, but the operands of this operations are not selected on a temporal basis, they are chosen with the help of the wildcards already present in the MQTT standard (specified by the # and + characters). It is possible to consider as an element of the operations, publication of different topics that match the subscription filter net of the wildcards. With the syntax **$SUM/topic**, for example, the payload of every publication on a topic that matches the *topic* part of the filter will be summed up and the subscriber will receive the final value, as soon as a publication arrives at the broker, the value is recomputed using the last published value of each topic belonging to the aggregation.

- **Data processing operators**: MQTT+ is able to specify some operators that allow the subscriber to manifest their interest in some particular operation that can be carried out on the payload of a corresponding MQTT message. For example, the operator **$CNTPPL**

allows the client to signal that it is interested in the number of people that are depicted in an image published on the specified topic, without receiving the entire image, but only the result of the image processing operation on the image. There are also other examples of operators but this functionality has for sure a lot of applicability fields.

Moreover, the aggregation operators can be combined together to have more and more possibilities and the last point in the list shows the potential of introducing such a modification to the MQTT standard, allowing also to save bandwidth and enable new application field, also where the constraints are so tight.

2.0.2 MQTT Bridging

Lots of already existing brokers, like **Mosquitto** [7], **HiveMQ** [1], and so on, allows connecting to others through a bridging method, specifying before the startup of the system to which broker connects to and on which topics exchange information. Unfortunately, none of these products guarantee efficiency: since is not possible to know which topics will be injected into the network of brokers and which of them will be connected to a client interested in them, the only way to allow communication among the nodes was to forward every MQTT message generated by the clients. This explains the **lack of efficiency** of this approach, resulting in a lot of wasted traffic between nodes that are not interested in specific messages. Moreover, this approach is **static**, everything has to be done at startup, and disconnection is not properly handled using this type of interconnection.
As an example, Mosquitto allows to specify a configuration file as parameter passed to the launch command of the broker, inside this configuration file it is possible to state with a specific syntax one or more connection with a Mosquitto Broker, like the one below:

```
connection id_2
address 10.0.0.251:1883
topic # both
```

The connection line simply specifies a start of a new bridging connection specification inside the configuration file, the second one specifies to which host connects to through its IP and port, used by the Mosquitto instance. The last line specifies the topic pattern that is the subject of the bridging between the two nodes and the direction of the bridging ("#" means that every topic is shared and "both" that this connection is going to be bidirectional).

[1] see www.hivemq.com/hivemq/mqtt-broker

2.0.3 PADRES protocol

To face the problem of efficiency, limiting as much as possible the network traffic, it was necessary to find a protocol that implements a smart way to forward publishes and subscriptions and one suited for this purpose was for sure [3], but in the context of MQTT this protocol has some adjustment to be done, since it is conceptually a protocol thought to work for **content-based** publish/subscribe paradigm, while MQTT, and so MQTT+, are **topic-based protocols**. The content-based paradigms need the introduction of a new type of messages, they are called advertisement messages, they are sent by the publishers, and their role is to perform a notification to the subscribers on how the publication messages will look like. They can specify the range of value of the publication messages or the attributes that will be used inside the packets sent by the publishers.
The idea behind this approach is based on three tables that are used to route the messages of the base publish/subscribe paradigm on which PADRES is founded on the Overlay Routing Table (ORT), the Subscription Routing Table (SRT), and the Publication routing table (PRT). As the names say, the ORT is the table that contains the direction that the advertisement messages must follow, it is a tabular representation of the logical connections that are present in the network at the startup. The SRT, based on what are the route enforced by the advertisement messages, is used to forward the subscription towards the nodes that would be interested in the contents that are going to be published by the clients. The PRT, exploiting the information spread by the circulation of the subscription, assures to route of the publication to the nodes that expressed interest to a matching topic filter.
Based on [3] it was necessary to develop a whole new paradigm to achieve goals as routing correctly publications and subscriptions, preserving bandwidth, maintaining resilience to failures, and making it easy to integrate this new technology with the already existing ones.

2.0.4 MQTT-ST

The main idea behind this approach is to apply the Spanning Tree Protocol in order to connect MQTT brokers to each other in a network in which loops are avoided.
The protocol follows the main phases of the STP and implements the signaling inside the MQTT messages already available according to the standard, in particular at the startup all the nodes sends to their directly connected neighbors a modified CONNECT message, with the most significant bit set, upon the reception of such a message a broker stores all the IPs and ports of the brokers to which it is connected to, along with the Round Trip Time computation and the value of the C parameter, that summarize the compu-

tational resources available on that machine.
Upon the termination of this phase, every node sets itself as the root of the topology and starts sending the messages that are meant to build the tree, including the necessary information to pursue the objective inside the MQTT PINGREQ messages, appending to it the following information: the IP of the currently selected root, the value of C and the cost that the sender is going to pay to reach the root at that moment. The circulation of these messages allow to perform two operations at the same time: the selection of the root, according to the higher value of C, and so choosing the most powerful node as root, and the path computation, according to which each node chooses its next hop toward the root. In the end, upon the reach of convergence, a tree is formed.
As can be seen in [8], this protocol is well suited for the purpose of the creation of an overlay network among MQTT brokers that is failure resilient and dynamically exploit the most powerful nodes of the network, it also guarantees the property of avoiding loops that may create a lot of bandwidth waste on the physical network underlying the logical created by the protocol.
Nevertheless, this approach lacks as long as traffic efficiency is concerned since it is based on bridging (see 2.0.2 of this chapter) and doesn't allow the brokers to create a network without injecting information in the system before the startup.

2.0.5 Other Distributed Publish/Subscribe paradigms

Beyond the PADRES approach previously described, others have been developed recently to face the problem of introducing the Publish/Subscribe paradigm inside a distributed context while trying at the same time to reduce the signaling overhead as much as possible.
In [4] is introduced the concept of *Matchmaker*, a node in the network responsible for the matching between a publish and a subscribe or an advertisement and a subscribe. The paper describes also how the overhead signaling efficiency is affected by the direction to which the subscription and publication messages are routed inside the network with the addition of this component. The two most intuitive approaches are to forward the publication towards the subscriber nodes or conversely the subscription towards the nodes where the publishers are placed, once this forwarding is operated and the matchmaker operates its comparison to detect matches, the messages can correctly circulate inside the distributed system.
Another approach is presented in [6], this has some similarities with the PADRES algorithm since uses a Subscription Routing Table as the base the entire algorithm uses to correctly propagate publications. This approach focuses more on reliability and the possibility of continuously providing a service, being able to tolerate a maximum number of failures without re-

booting the entire system. As PADRES does, it assumes to have an overlay network already set on which it starts to operate, it is represented in memory by a so-called *Topology Map*, as it records the brokers that join and depart from the system. In addition, an STP with different information with respect to PADRES is introduced, it contains the topic filters along with a *from* field, which can be considered as a "network pointer" which points to another broker in a certain range of numbers of "logical" hops, determined by a parameter that enforce the maximum number of failures acceptable by the whole system. This last table is used to route correctly publications as near as possible to the subscriber to a corresponding topic in a similar way it is done in the PADRES approach, nevertheless, this algorithm is centered in failure resilience, trying to implement a high level of reliability, which is not done by the approach chosen by the work presented in this paper, focused on the efficiency of the algorithm and of the data structures.
The last alternative approach that is going to be presented is called Scribe [2], it is an event-notification infrastructure, that is another name for a topic-based publish/subscribe system. As the reader can see the scope of this paradigm is more focused on the field of application of the project presented in this paper, even if it is general and not applied specifically to the MQTT protocol. The APIs that allow operating in the context of Scribe are quite common of any publish/subscribe paradigm, and so similar to the one implemented by MQTT, as it is a protocol that operates in this particular conceptual scheme. The main peculiarity of Scribe is the fact that it is totally decentralized, there is not any figure that is comparable to the broker, it is based on a P2P approach, and so the routing of the messages uses a mechanism thought for this type of systems, the DHTs (Distributed Hash Tables) and more specifically, one of its implementation called Pastry [9]. The DHTs are routing tables based on key-values pairs in which any participating node can retrieve a value associated with a key exploiting the lookup efficiency of the hash tables, usually, the keys are unique identifier associated with the peer belonging to the system and the values are addresses through which the destination can be reached. The main advantage of using such an approach is that the nodes can be easily added and removed with a very low overhead in terms of computational requirements. Pastry is a specific implementation of this general idea, it is quite similar to other approaches like Chord [10], that presents, similarly to Pastry, a circular key-space. The main difference that allows distinguishing this particular instance of DHT with respect to other homologs approaches is the overlay network built on top and used by the DHT itself, this allows Pastry to apply a metric that reduces the cost of routing packets avoiding the need to flood packets.

Chapter 3

Project Structure & General Idea

In this chapter the general structure of the developed software will be presented, in this project, several entities are involved, so it is necessary to explain how they communicate with each other and what are their roles to accomplish the objectives explained before in this paper. The description starts with a system-level design explanation and proceeds to focus on the structure and the role of each entity involved in the project

3.1 MQTT+ Distributed Broker

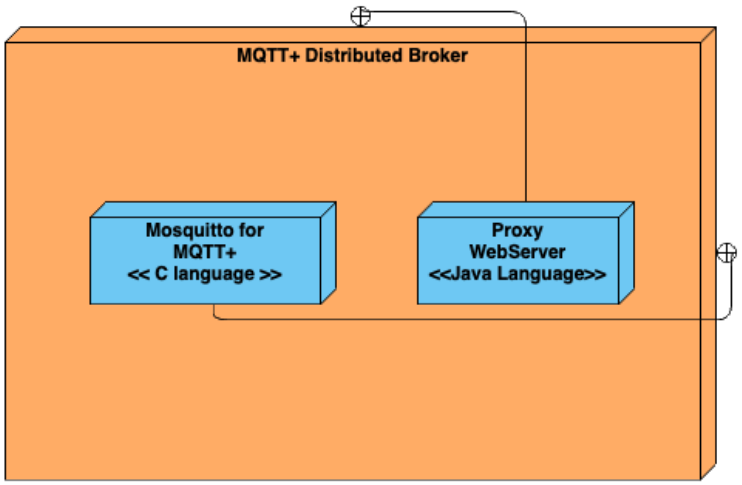

Figure 3.1: Deployment diagram for the broker

The figure 3.1 shows how the broker is structured from a high-level perspective, it can be seen as composed of two communicating entities: a custom

version of the Mosquitto MQTT broker, written in C language, and a web server written in Java.
The Mosquitto broker is responsible for all the well-known operation executed by an MQTT broker, but it also communicates with the server via the HTTP protocol: every publication or subscription is forwarded by the C software to the Java one that adds all the functionalities of MQTT+ explained in [5] and the ones proper of the project described in this paper. The forwarding of the MQTT traffic is the only difference between this version of Mosquitto and the standard version.
The Java Server has three key roles:

- It makes it possible to process the MQTT+ operators.
- It contains all the logic to establish the connection among the brokers.
- It works as a router for the MQTT messages in the network. All the logic to forward smartly the messages to other brokers is in here. This functionality allows avoiding the usage of the bridging offered by the base version of Mosquitto, which is not well suited to obtain a low-traffic message exchange, as explained in 2.0.2.

3.2 System-level Perspective

Observing the whole system from the point of view of the connections among the instances of the brokers, it appears as follows:

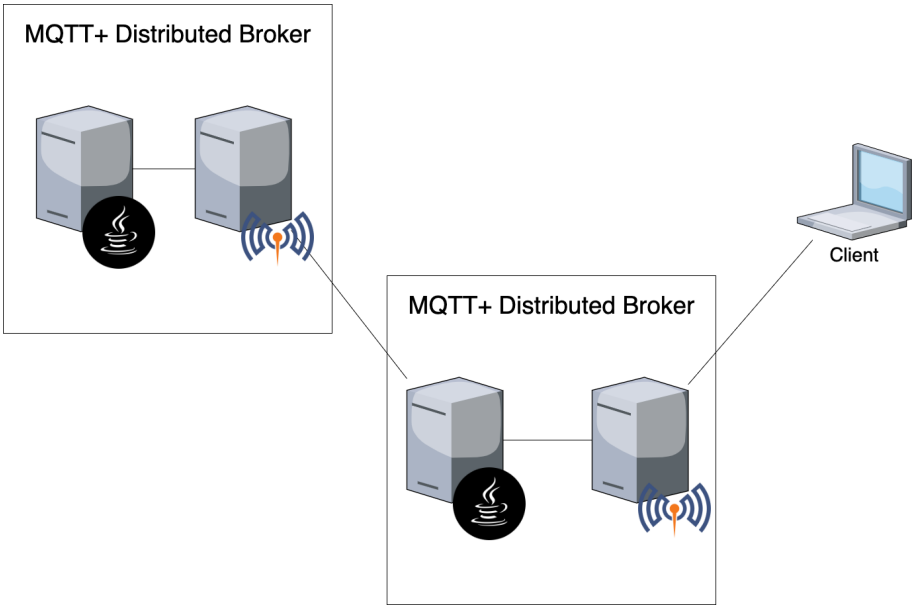

Figure 3.3: Explanation of various connections of the system

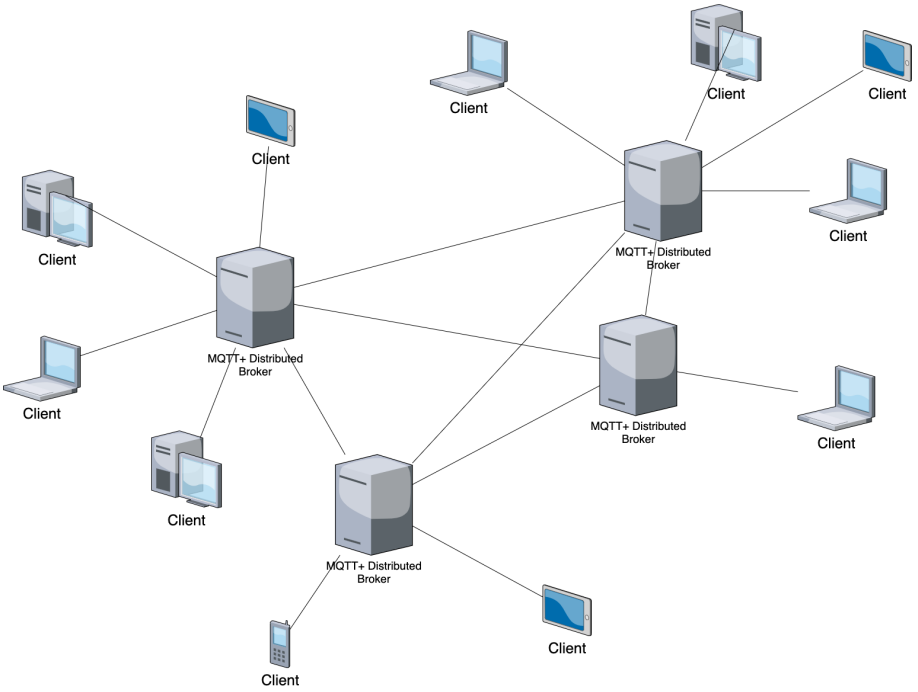

Figure 3.2: Example of MQTT+ Distributed System

In the figure 3.2 every client is connected to a single broker through an MQTT connection established with the Mosquitto "part" of the broker (see section 3.1). Each MQTT+ Distributed Broker knows every other one in the network, from a mathematical point of view it is possible to state that they form a **fully connected graph**. As the figure 3.3 shows, Each MQTT+ Distributed Broker is connected to another instance through an application-layer link between the Java Server of one side and the Mosquitto Broker of the other, indeed the former includes MQTT clients that have the role of forwarding the protocol messages hop by hop, using the mechanism that will be explained in chapter 4. Later in the text, it will be clear how this high number of "logical" connection among the nodes of the network is reduced to form a tree that prevents loop and reduce the total communication delay of the system.
What is important to notice at this step is that the clients see the whole system as follows:

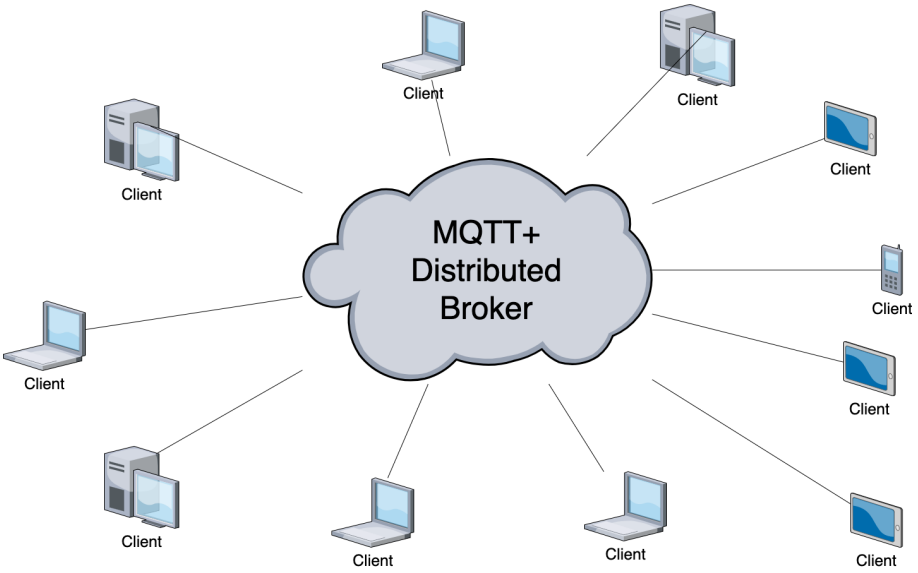

Figure 3.4: The system from the point of view of the clients

The distributed brokers cooperate in order to make the system **appear as a single broker** to which every client is connected. It means that when a client publishes a message, another one, subscribed to a topic that matches the publication of the former, will receive the publication even if they are technically connected to two different instances of the software.
It would be now clear how it is possible to correctly **aggregate** the data produced by the clients, they are simply forwarded through the (application-level) network toward the broker instance connected to the client interested, once it receives the information it can process locally them inside the "Java Server" part of the software. Since MQTT+ has been developed to operate on data published to a broker **on the very same machine**, exploiting its normal functioning implementing a **routing protocol** at an application level, made the entire system work as a **single entity**, splitting the processing overhead over several nodes.
It is important to underline once more that every connection in the figures 3.2 and 3.4 is logical, it means that is established at the application-level of the OSI stack, nevertheless, each line of the figure hides all the lower-level (of the OSI model) links that could be interposed between two nodes, affecting the performances of the information transmission. This aspect is of crucial importance to be ready to understand how the MQTT+ Distributed Brokers arrange themself into a (logical) topology, trying to select as *next-hop* the hosts that guarantee the minimal delay toward the node with the highest computational power, considered as well suited to receive and process the largest number of requests.

3.3 Java Server

This entity can be seen as a state machine, it is detailed as follows:

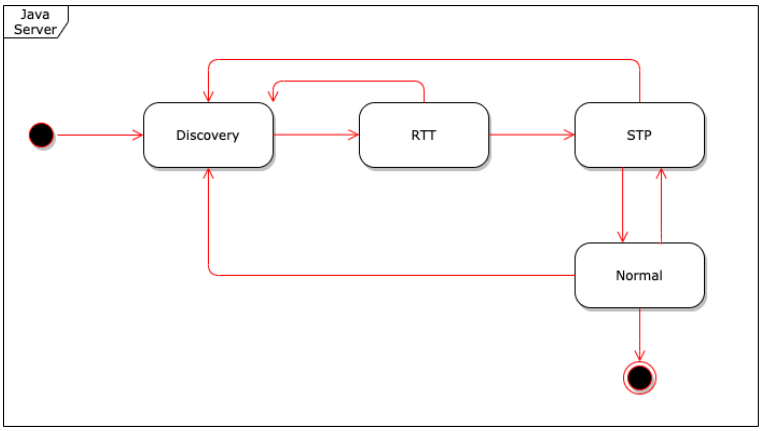

Figure 3.5: State diagram that describes the Java Server

In the following we will describe the meaning of each status and each connection between states, it has to be noticed that the implementation of the protocols will be discussed in the next chapter, as will be done with every implementation detail:

- **Discovery**: in this state, the server communicates with the other nodes to discover their IP addresses and the ports through which it can reach their processes running the protocols that they implement. Since no prior information is available, it sends packets to a multicast group joined by the nodes that are interested to join a network.

- **RTT**: it is improper to define it as a state, since the computation of the *Round Trip Time* between a server and another one is continuously executed, however, it is mandatory to have at least one computation of RTT for each discovered node to proceed to the next state. After the first "round" of computation of the RTTs, the server has all the information needed to compute the overlay network of the distributed system.

- **STP**: here the server will execute all the processing to have a list of *neighbors* to be able to apply the routing protocol that will be described in the implementation details of the project. As can be seen, even if the name of the state refers to the *Spanning Tree Protocol*, the protocol implemented in this state it is a modified and simplified version of it, the applicability of these simplifications is due to particular conditions that are guaranteed by the discovery protocol.

- **Normal**: it can be considered as the nominal state of the server, the one in which it will be for most of the time and the only one in which the requests forwarded by the connected Mosquitto Broker can be processed since all the steps for the correct communication of the server with the other nodes in the network have been executed. While the server is in any other state than this, the requests are put in a queue of execution and they are executed as soon as this step is reached.

The reader is now ready to analyze all the possible connections among the states described above. As can be seen, by the graph, the **Discovery** state can be reached by any state, there are multiple motivations to explain this concept: every time a disconnection to a previously discovered node is detected, the server returns to this state to know which servers are still on. On the other hand, if a node wants to join a network when the already present serves are doing other operations, the server has to start all the procedure to determine its neighbors from scratch.
The **STP** state can be reached by the "nominal" state whenever the network conditions change since the server continuously monitors its RTT with every node in the network, it periodically checks whether the structure of the tree formed is outdated concerning the time to reach every other server. All the other connections have been detailed in the explanation of the statechart above.

Chapter 4

Implementation

In this chapter the reader can acquire knowledge about what has been done from an implementation point of view to pursue the objectives previously explained in the paper, following the conceptual model presented in the last chapter.

4.1 Discovery Protocol

One of the most important features of the project is the fact that the broker that is obtained at the end of its development would have the ability to insert itself into a network of brokers without any information given by the user, autonomously discovering the other nodes. This task is accomplished by a simple protocol that requires the brokers that want to become part of a system, to join a specific multicast group, in order to be able to send and receive discovery messages like the following:

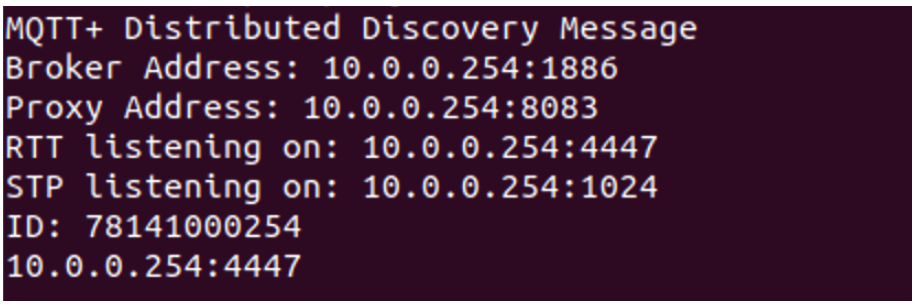

Figure 4.1: Example of a discovery message

The message below is sent via a UDP packet to the multicast address previously specified. The fields of the packets are:

- **The header**: as it will be possible to see later, each message sent by the MQTT+ Distributed Broker has a header to specify to which

protocol it belongs to.

- **Broker Address**: this field contains the address of the Mosquitto broker process of the sender that will be used to forward the MQTT messages.
- **Proxy Address**: it is the address of the Java Server, this value was mainly included in order to test the system in a local environment to distinguish the different instances of the same Java process.
- **RTT listening on**: the sender says to the receiver where it is going to listen to Round Trip Time request messages.
- **STP listening on**: the address from which the sender expects to receive packets of the STP protocol, used to arrange the topology of the system.
- **ID**: this unique identifier is used by the receivers in order to detect whether the sender is going to start a new session of discovery or not and to detect retransmission of the same packer, indeed since the underlying transport protocol in unreliable and the servers may be started at different moments, each instance transmits the same packet (with the same information and the same ID) multiple times until the end of this phase in its own instance.

This procedure starts at the very same time the Java Server is launched, during the period of time the protocol is running inside the process, it continuously sends and receive packets until a timer expires, which will terminate the sending part, in order to pass to the next step (see figure 3.5), but never stops the receiver since the server has to be able to listen to discovery requests in case of disconnections or insertion of new nodes in the system. The timer is postponed every time a discovery message is received from another node, this lets the procedure be sufficiently confident to terminate when convergence is reached. Obviously, this implementation hasn't been thought to protect the server against security threats, since it is not the main topic of the research carried out by who wrote the code. The duration of the timer clearly was the subject of an empirical choice.
The situation, from a connection point of view, at the end of this state of the Server state machine can be synthesized with this graph:

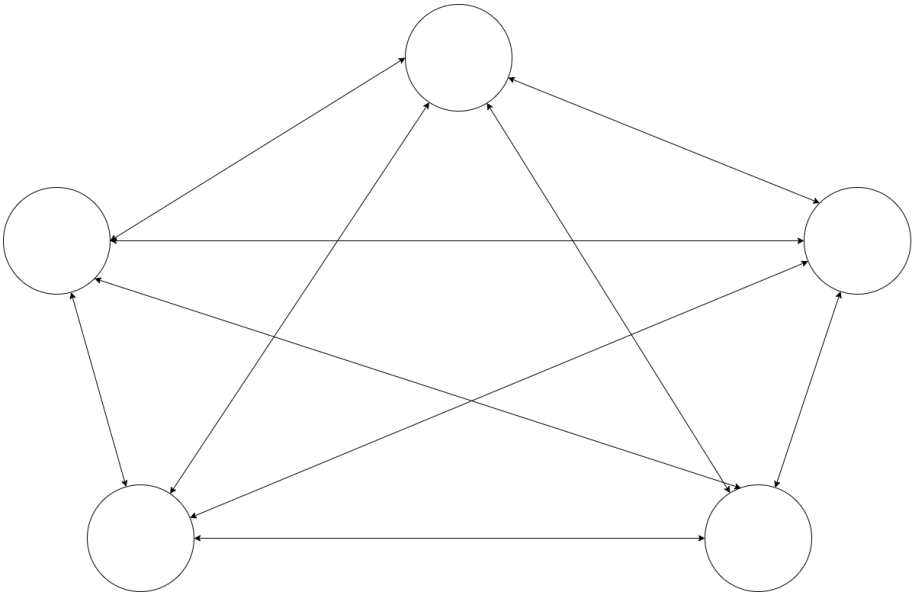

Figure 4.2: Example of the logical connections established in a system of 5 brokers

Each node of the graph represents an MQTT+ Distributed Broker, while the arcs (it is important to notice that they are bidirectional) symbolize the fact that the two connected nodes know each other addresses to proceed in the following operations of the protocol.

4.2 Round Trip Time Computation

The MQTT+ Distributed Broker is now ready to start the computation of an estimation of the *Round Trip Time* required to communicate with the previously discovered nodes of the network.

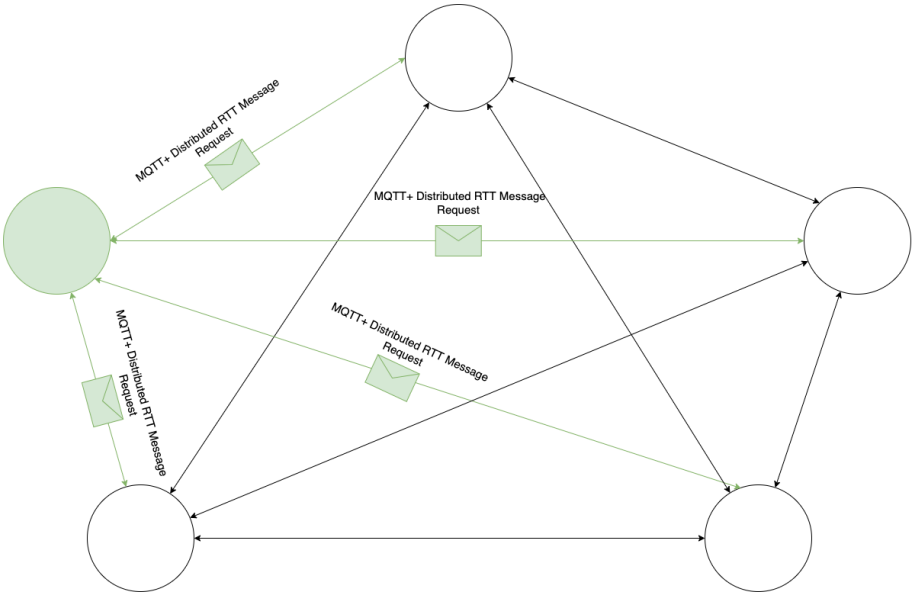

Figure 4.3: The green node sends the requests for the computation of the RTT estimations

The reader can imagine assuming the role of the green node in the figure 4.4 but has to keep in mind that what is happening in one direction of each link, is happening into the other one as well.
The messages sent by the highlighted broker are of the following type:

```
Sent: MQTT+ RTT Request:da191000252
10.0.0.252:8081
```

Figure 4.4: Example of a possible RTT message request sent by the green node

The packet contains simply two crucial contents:

- A **request identifier**: it is used to be able to transmit the same message several times if no response is received by the broker that originally sent the request. In particular, internally, the server keeps a timer for each request code, whenever this timer expires, it sends again the same packet to the original destination, this procedure is repeated at most three times after which the connection is considered no more active and the Discovery procedure is restarted.
  The code is also useful to understand upon the reception of a response to which message the received packet corresponds.

- The **sender of the packet**, expressed as the IP address of the host plus the port used by the server to receive the forwarded traffic by the Mosquitto broker.

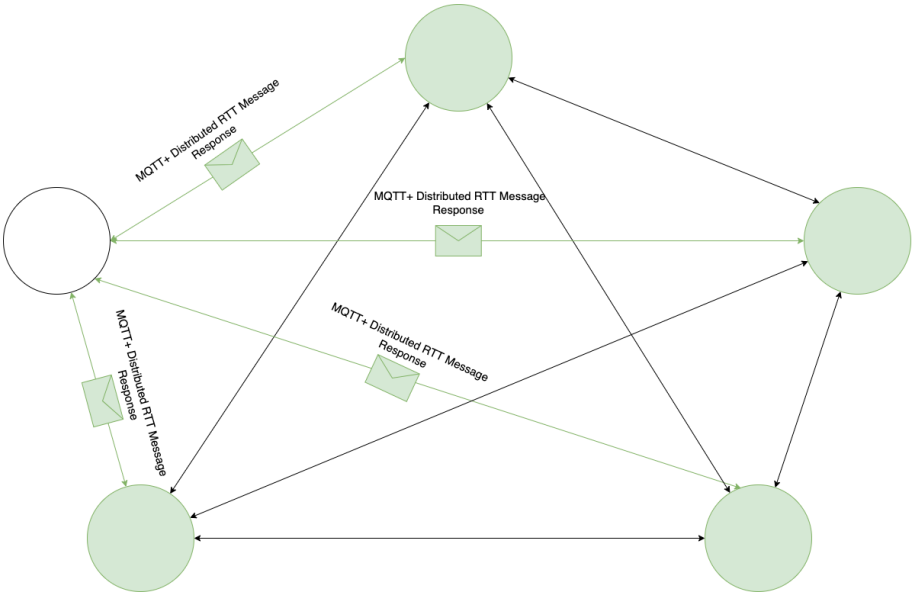

Figure 4.5: The responses sent as consequence of the figure 4.4

In the figure 4.5 the brokers previously contacted by the white node reply to the request message:

```
Packet received: MQTT+ RTT Response:1c641000252
10.0.0.253:8082
```

Figure 4.6: Example of a response message received by the white node

The structure of the response is symmetric with respect to the request:

- The **request identifier** is the same that was received in the previous request message.
- The **sender** is one of the node highlighted in green in the figure 4.5.

The total Round Trip Time for each couple of request-response messages is computed as follows: as soon as the sender (the node with IP 10.0.0.252) submits the request (fig. 4.4) it saves the instant of time in which it happens, upon reception of the response (fig 4.6), it memorizes at which time this happens and carries out the difference between the instant the request started and the time the response arrived. Every node does the same a number of times that is equal to the number of the discovered node continuously until the server is up.
At the end the situation will be like this:

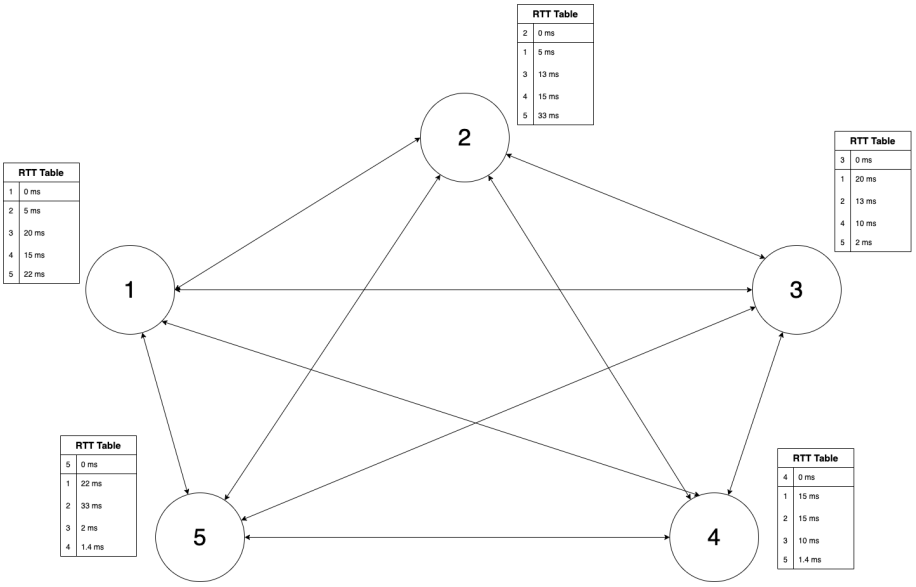

Figure 4.7: Representation of what is stored in memory by each node of the network

The nodes save in memory, into a table (it is a HashTable in the Java implementation), a record whose key is the IP of the node for which it is stored the RTT and whose value is the computed numerical value.
This snapshot represents the system after *one round* of computations, but after a short sleep time, the nodes will execute again the same operations, in the meanwhile, they are ready to execute the next step of the state machine.

4.3 STP protocol

The standard *Spanning Tree Protocol* (IEEE 802.1D) is used to prevent possible loops into switching loops and consequent broadcast storm, nevertheless, in our case the main hypothesis underlying the theoretical background of that protocol **doesn't hold**. The starting point is different, because *every node knows how the network is composed and what is its distance from every other broker*, instead the original STP protocol is based on the premise that the single node hasn't the complete knowledge of the network. This condition allows the application of some simplification to the standard protocol that will be explained in the following. Based on what's said above it is possible to distinguish two phases of the protocol:

1. Root Selection
2. Path Computation

The list presented above is ordered because the two phases are executed one after the other and the reader will discover their implementation in the following sections.

4.3.1 Root Selection

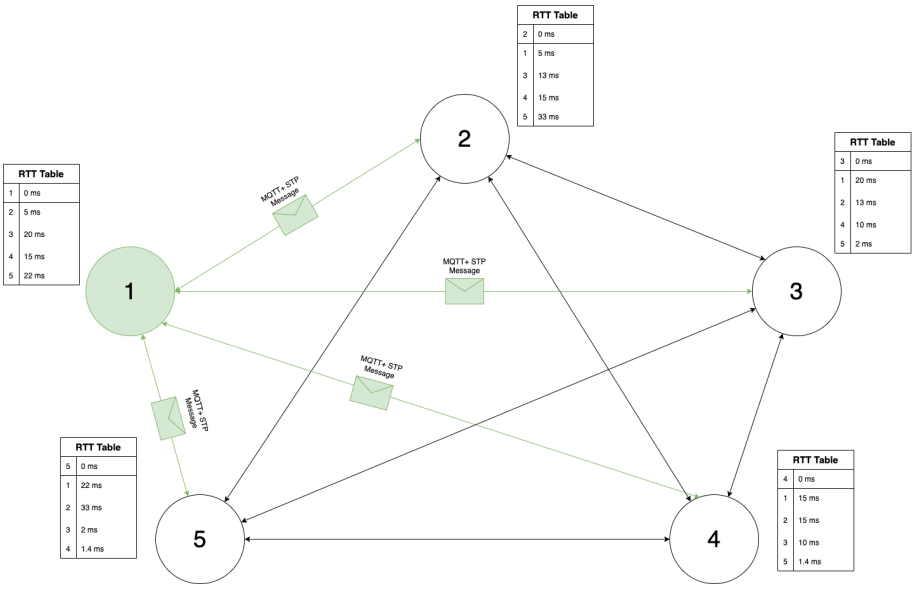

Figure 4.8: Message exchange for the first phase of the tree formation protocol

The figure 4.8 shows the first phase of the protocol from the point of view of the node "1". In the very same way the node did in the RTT computation, it sends to all the other brokers of the network the same message, obviously, it uses different ports as indicated by the discovery messages we saw in section 4.1.
The packet injected into the network, in this case, is like the following:

```
STP PACKET SENT: MQTT+ STP Message
Root: 10.0.0.252:8081
M: 1.2291304E7
L: 2304.0
P: 0
Source: 10.0.0.252:8081
```

Figure 4.9: Example of a packet sent in the first phase of the protocol

It is now possible to analyze each field of the packet:

- **Root**: here the sender signals who is the node selected as its root, in this phase this field always matches the "Source" field since at start-up time every node chooses itself as the main node of the tree because it hasn't any information about the others.
- **M**: this is one of the fields used by the nodes to apply the criterion for

the selection of the root node, it is the memory capacity of the sender, expressed in kB.

- **L**: the second parameter used for the selection of the root node, expresses the CPU clock frequency of the sender, expressed in MHz.
- **P**: this field is always 0 in the Root Selection phase because it is a value useful for the path selection as it gives the value of the distance (in terms of Round Trip Time previously computed) between the sender and the root, considering that when this message is sent the sender selected itself as root, it is perfectly logical to have 0 as the value of this message part.
- **Source**: this field indicates the sender of the packet, as can be seen in the figure 4.9 it is, in fact, equal to the "Root" field.

The root selection is performed every time a packet of the type above is received by a node, it simply adds together the parameters L and P and compare this sum with the one computed with the parameters of the currently selected node (at start-up this corresponds to itself), if the intermediate value is greater than the current one, then the root changes. In the case of ties, the root with the lowest IP address is chosen as the root.
Once every node collected all the STP root messages it finishes this phase, having selected a node as the root of the tree that is going to be formed and sends a message as the one that is presented here:

```
STP PACKET SENT: MQTT+ STP root selection completed
Source: 10.0.0.252:8081
```

Figure 4.10: Message sent at the end of the phase described in this subsection

Once a node receives this packet from every other node of the network it can safely proceed to the next phase the protocol, at the end of the first phase the situation appears like this:

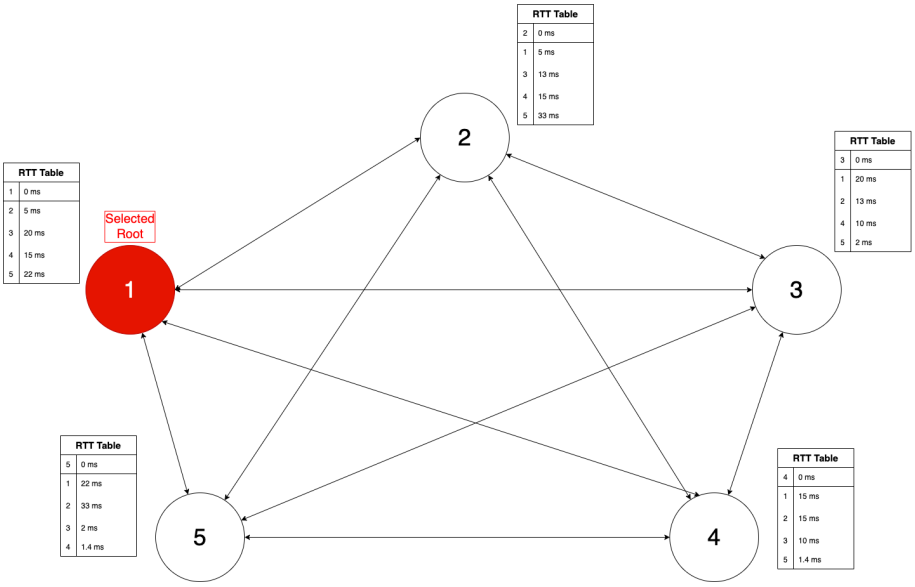

Figure 4.11: Situation at the end of the first phase of the STP protocol

The figure 4.11 reports an example of a possible convergence of the first phase: the node "1" has been selected as the root node of the tree, the reader can be sure that the choice is agreed by every node of the network since the situation in the picture is a snapshot of the system after the reception of the messages showed in the figure 4.10.

4.3.2 Path Computation

Once the previous phase has been completed the nodes collaborate in order to form the Spanning Tree that has the minimum path cost. The situation at this point is identical to the one presented in the figure 4.8, the messages sent are analogous as well but they differ in the content of the P field, as discussed in the subsection dedicated to the explanation of the Root Selection phase. It is possible to show an example of a packet sent during this step of the protocol:

```
STP packet content: MQTT+ STP Message
Root: 10.0.0.251:8080
M: 1.2291304E7
L: 2304.0
P: 374318205
Source: 10.0.0.253:8082
```

Figure 4.12: Example of a packet sent by the brokers during the path computation phase

As can be seen by the figure and anticipated previously during the expla-

nation of the protocol, the messages, during this phase, present two main differences compared to the ones of the type shown in the figure 4.9:

- The *Root* and the *Source* fields are now different, in particular, this message is not sent by the root node, so it presents as root the address of that broker and as sender the indication its own IP.
- The *P* field is now different from 0, it is a direct consequence of what is said in the previous point of this list, since the selected root is not equal to the node that is the source of the packet, its cost is different from 0 and corresponds to the cost that the sender pays to reach the root in terms of RTTs.

The receiver of such a packet is now going to compare its own cost to reach the root with respect to the value sent by the other node, if this value plus the cost to get to the sender is lower than the one it has in its *RTT Table*, then it sets the sender of the packet as new root (since it is the new next hop to the root) and updates the cost value. An attentive reader may notice that since the root has 0 as path cost to itself and the RTTs can be assumed greater than 0 for all the other nodes in the network (at least in a real environment), it never updates its root and cost value. If those changes occur inside the data structure of the receiver, it sends new packets to all the nodes of the network, to notify the new values and let the other nodes take advantage of this new information.
Sooner or later no more new messages will be sent into the network and the receiver thread of each node will be closed as their timer expires (those timers represent the period in which the brokers wait for a new message to arrive and are postponed upon reception of each STP message, an analogous behavior seen in the RTT and Discovery protocols). An example of the snapshot of the system at the end of this step may be the following:

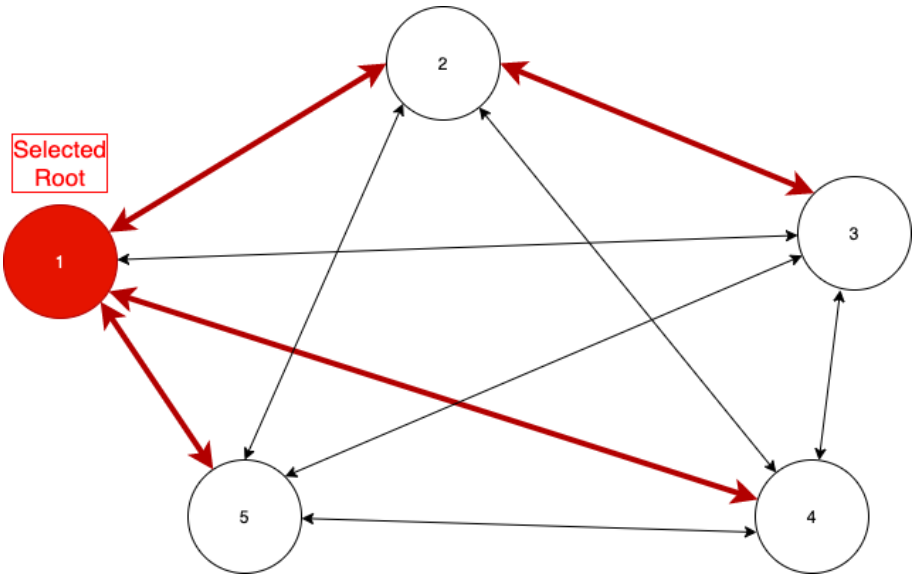

Figure 4.13: Example of a snapshot of the system at the end of the protocol

The red links are the logical connections that are active from now on until a new run of the protocol will be executed. It is important to notice that the other potential connections are not thrown off, the nodes will keep them in memory in case of another execution, at least if no disconnection among the network will happen, in this case, everything will be restarted from scratch. As soon as the tree is formed the nodes will use the continuous computation of RTT to monitor their distance to the nodes that have been elected as root during the *Root Selection* phase.
The information about the neighbors that are the result of this protocol will be used to actuate the routing mechanism that will be discussed in the next section.

4.4 Routing Algorithm

The algorithm that will be presented in this section takes a lot of inspiration from [3] but, as it has been said during the introductory part of the paper, this protocol is designed to be applied to content-based publish/subscribe paradigms while MQTT is a topic-based protocol, for this reason, it may be worthy to spend some time to understand how the original protocol is adapted to a different scope, to be ready, in the successive sections, to understand how it is implemented in the specific case of this project.

4.4.1 Algorithm Adaptation

The main clarification that has to be done at this point is what are content-based publish/subscribe paradigms and how they differ from topic-based ones. In a content-based protocol, messages are only delivered to a subscriber if the attributes or the content of those messages match the constraints defined by the subscriber, while in a topic-based one the subscribers will receive all messages published to the topics to which they subscribe, the topics are in fact called *logical channel* as they represent a logical source from which they receive contents by the clients. Based on these definitions it appears clear that the original paradigm in which PADRES operates is based on the ***values*** published by each client rather than on how they are labeled, this justifies the presence of particular messages called *advertisement* that are injected into the network before clients publish messages. In the content-based paradigm, publications are based on messages that present attributes and values assigned to them, these attributes are sort of keys to which the corresponding values are assigned. Advertisements are messages in which the publisher notifies potentially interested clients on the type and range of values it will associate to each attribute of its forthcoming publications.
Two examples of publication and advertisement are reported below:

```
P: [class, 'STOCK'], [symbol, 'YHOO'], [open, 25.2],
[high, 43.0], [low, 24.5], [close, 33.0],
[volume, 170300], [date, '12-Apr-96']
```

```
A: [class, eq4, 'STOCK'], [symbol, isPresent, @STRING],
[open, >, 0.0], [high, >, 0.0], [low, > ,0.0], [close, >, 0.0],
[volume, >, 0], [date, isPresent, @DATE]
```

As it is possible to notice the complexity of these messages is very high due to the presence of a typing mechanism and comparison operators for each of these types.
Even if the *class* attribute may represent a point of conjunction between content-based and topic-based approaches, as its role can be in some sense compared to the topic of the latter, the presence of advertisements in a protocol like MQTT has absolutely no sense, moreover, the addition of a new type of message into the consolidated standard of this paradigm was not the intent of this project. According to the standard (see [1], section 1.5.3) an MQTT topic must comply with the UTF-8 encoding for strings with a length range that goes from 0 to 65535 bytes, the project exploits the freedom in the design process of topics by creating a new type of messages that are simple publication with an additional part appended to the topic.
It is now possible to see an example of this new type of message, explain the reason behind this addition, and see how it is linked to the advertisement concept of the content-based paradigm:

```
MQTT publish message
topic: room1/sens0/temp@10.0.0.251:1883
payload: 81.0
```

The message presented above is an example of the MQTT publish packet, lots of its fields have been neglected because they are not important at this stage of the discussion. The addition on which the reader may focus its attention is on the *topic* field, it is totally compliant to the standard but at the same time, it presents a "strange" final part. The idea behind this addition is that the '@' character is not used too much inside the topic design of the MQTT protocol, but at the same time, it expresses well the meaning of this type of message. At the end of the topic, it is possible to find the origin of the publication, more precisely *the broker that originally received the publication of that message from a client.*
The message is the result of post-processing, since upon receiving a message from its connected broker, that in turn received it from a connected client, the Java Server appends this additional information to the topic string, this operation is done only for the first message received by the server that belongs to a certain topic.
Similarly to what is done by publishers in the content-based paradigm, this new type of message signals what are the contents published by a certain publisher, plus its broker location. The reader will understand the importance of this addition later in the explanation of the routing protocol adopted, that from now on it's analogous to the one presented in [3] with some significant modification that will be explained in the next sections.

4.4.2 Routing Tables

Talking about routing algorithms imposes to include into the discussion the routing tables, that represent its core. This routing algorithm uses three tables, the reader may notice that the meanings and the names of these tables are quite similar to the ones described in [3].

- **ORT** (*Overlay Routing Table*): this table is the result of what is described in the section 4.3, in fact, it contains the list of the neighbors of each broker, the representation of the figure 4.13 may now be extended in this way:

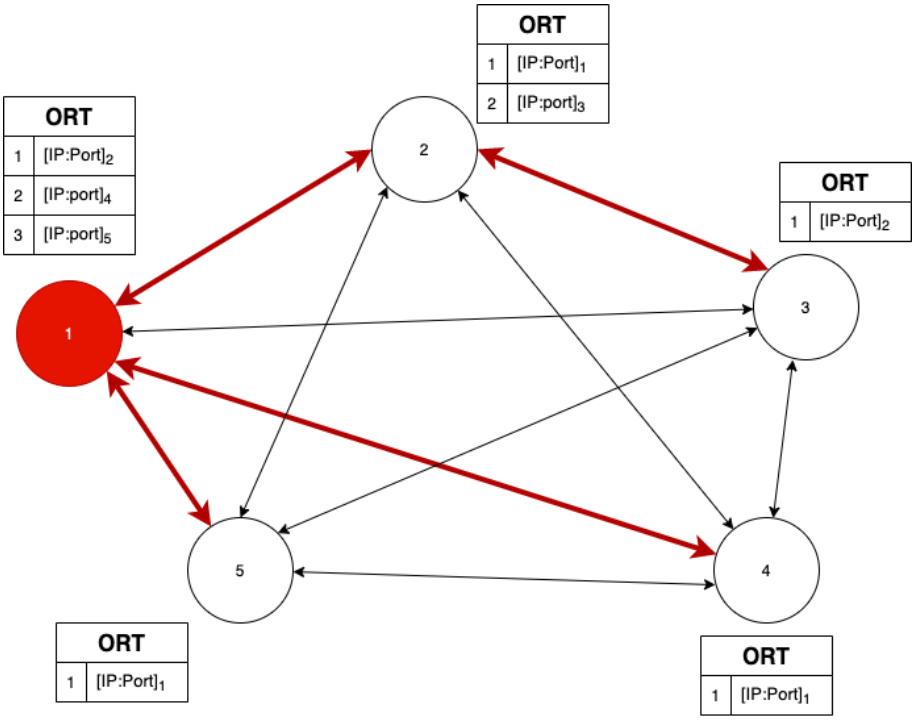

Figure 4.14: Representation of the situation in memory of each broker at the end of the STP protocol

it is important to understand the meaning of each entry of the table, they are written in the form $[\text{IP:Port}]_x$ and its meaning is: *IP and port (separated by a colon) of the instance of Mosquitto running on the node x*. Notice that, since the link are bidirectional, every node stores its neighbors addresses but at the same time is saved into the ORT table of each of them.
They are used in order to route the advertisement messages to the neighbors of each node, more specifically, every time a server receives a published message on a topic never encountered before, it forwards a message of the form represented above, appending the address of its connected Mosquitto instance at the end of the topic string, in order to make the forwarding process to be perfectly inserted into the MQTT standard, it is performed by using a simple MQTT publish. This message is in turn forwarded by the nodes which receive it to their own neighbors, substituting the sender address with their own.

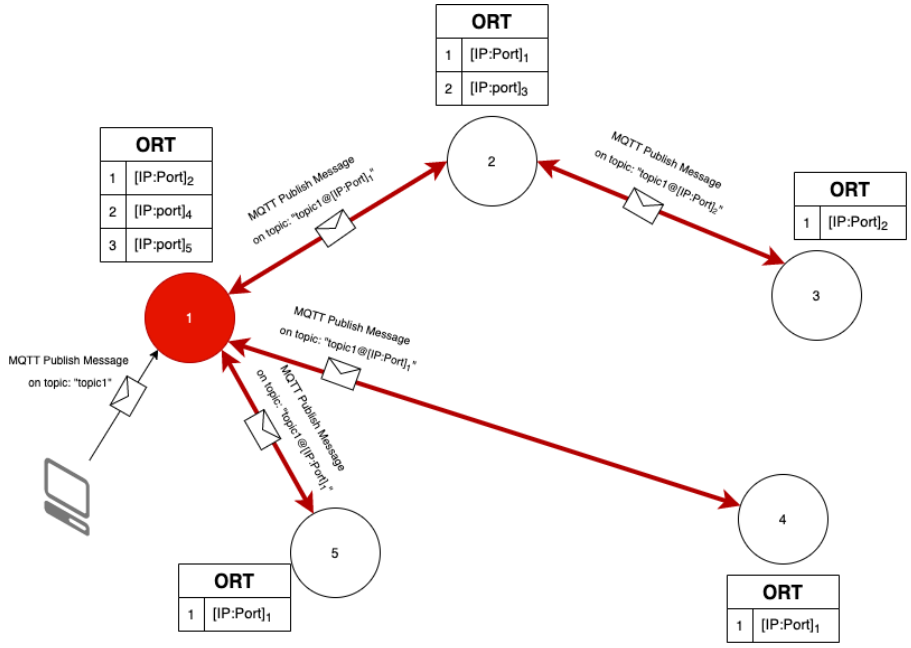

Figure 4.15: Representation of the system reaction to a publish message on a topic never encountered

Two clarifications are important after the observation of the figure above: for clarity of representation, the connections that are not active into the ORT of the nodes are omitted but logically they are always present in the brokers memory, the links are bidirectional but the messages that are forwarded always follow only one direction because upon receiving such a message the ORT table is examined by each node in order to avoid looping.
The messages are obviously not sent by the nodes in parallel as it may seem observing the figure, they are sent after the reception of each node from its "predecessor".

- **SRT** (*Subscription Routing Table*): as the name suggests this table is used by the algorithm to route the subscriptions to the correct nodes. Their filling is a direct consequence of the reception of the messages as showed in the figure 4.15. The situation after the processing of the above mentioned packets can be summarized in this way:

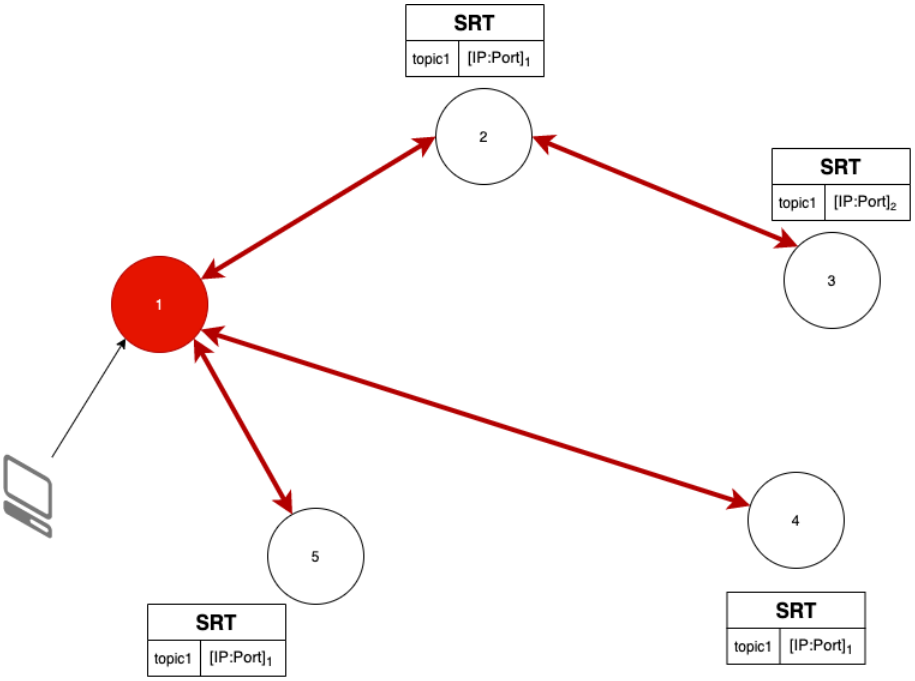

Figure 4.16: How the SRT tables are filled upon receiving the messages of the figure 4.15

The SRT table is implemented through an *HashMap* in which the keys are the topics and the values associated are represented by the addresses of the brokers to which a subscription that arrives must be delivered.
It is now possible to see what happens at the arrival of a subscription message by a client, sent to a broker which filled its SRT.

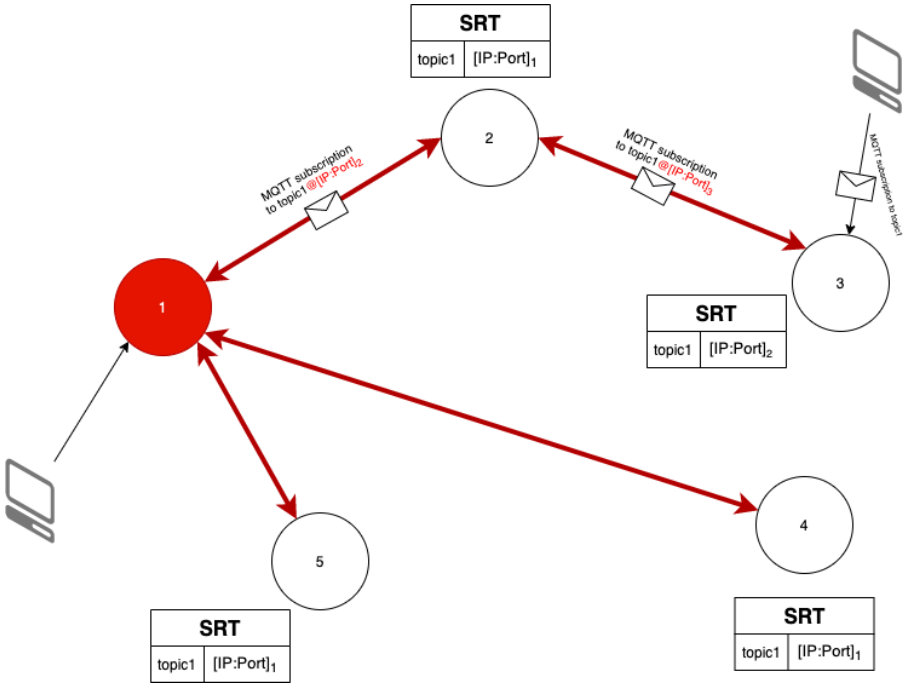

Figure 4.17: Example route followed by a subscription sent to a broker that filled its SRT

The figure above shows how the subscription received by the client connected to the broker "3" is routed to the root according to the content of the *Subscription Routing Tables* encountered during the path. Upon the arrival of a subscription, the Java Server verifies whether its topic filter matches any of the topics that form the set of the keys of the SRT table saved in memory if so it forwards the message to the corresponding Mosquitto broker saved as a value of that corresponding key. The reader may ask what happens in case of circulation of subscription messages before publication messages, the answer to this legitimate doubt is that the servers are able to buffer the received subscription and start their circulation into the distributed system as soon as a corresponding publication arrives, in the end, it allows to fill correctly the PRTs and the system can be considered as working exactly as explained above. The brokers store into a data structure the subscription filters they encounter and the corresponding broker to which each subscription has been forwarded, for example, in case of the arrival of a subscription before any other publication message, the cited data structure stores the arrived topic filter, leaving empty the part of the data structure that is dedicated to the broker that has been advised for that specific topic. As soon as a publication, with a topic that matches the subscription filter previously received, arrives, the

broker verifies that the list of the advised broker about that subscription is empty and so proceed to forward the subscription, establishing a situation in which the future publication can be correctly routed.
Notice that the examples are using simple topics, without any wildcard, to simplify the explanation of the protocol, however, the mechanism is the same in the case of more complex topic filters or publication topics with more of one level since the program inside the Server is able to test the matching of all these more sophisticated cases.
It may appear clear that the publication reaches the root since it was the broker that received a corresponding publication and it doesn't go further because the nodes "4" and "5" are not interested in a subscription with that topic filter at that moment. This is a first example of how this mechanism reduces the traffic on the network, in fact, the usage of the communication implemented by bridging would have replicated the message on the other two links that are excluded in this case.

- **PRT** (*Publication Routing Table*): The reader may have noticed that the explanation of the part highlighted in red into the subscription messages of the figure 4.17 has been previously neglected. The motivation is that those red parts regard the table that is going to be explained now and they were difficult to be contextualized before.
  The messages that are forwarded by the Java Server have a small part appended to the subscription filters that resemble the "extended" concept of advertisement introduced talking about publications, in fact, the concept is quite similar, it is used by the receiving node to be able to update the PRT table, so to store what is the destination of a potential publish message with a topic that matches the received subscription filter (all but the part highlighted in red). Moreover, the forwarding of a subscription is executed in a similar way with respect to the publish messages, they are simple subscription messages, compliant to the MQTT standard, sent to the interested node by a client that disconnects itself immediately after the packet is sent, as it may create additional traffic remaining connected and subscribed to a topic that is not the one subject to the ongoing network traffic.
  Therefore the PRTs appear like this (notice that the keys of the tables are subscription filters, not publication topics):

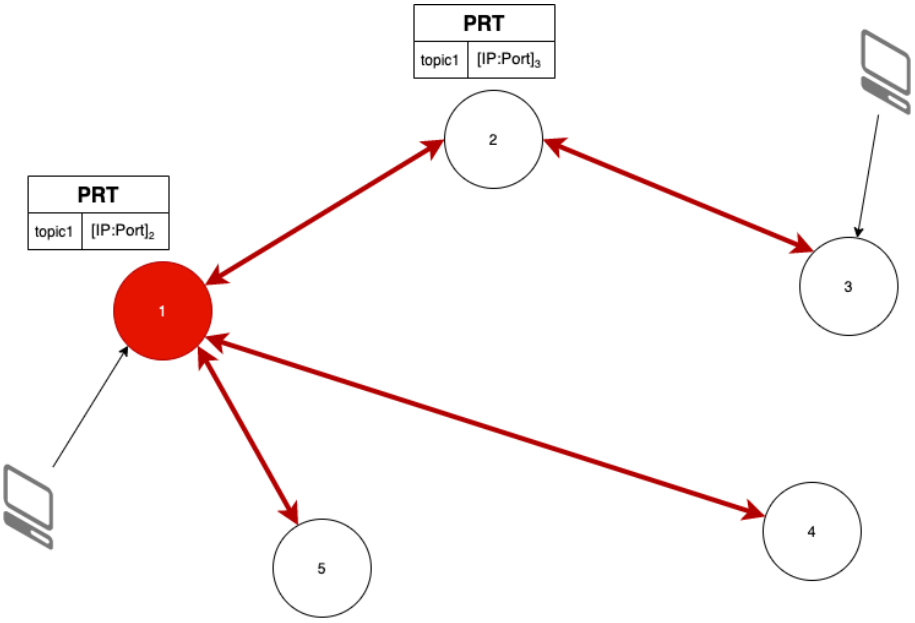

Figure 4.18: Example of the PRTs obtained by the traffic showed in 4.17

When the client connected to the broker "1" issues a publish message, it follows the path designated by the PRTs and reaches the brokers connected to the subscriber interested in it, avoiding the proliferation of messages in parts of the network that can be neglected.

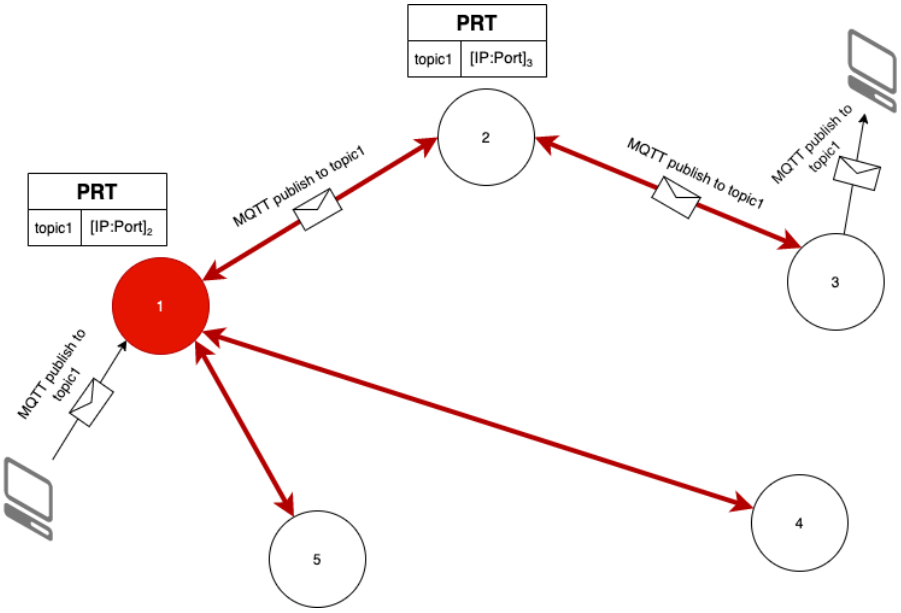

Figure 4.19: Example of a routing of a publish message of an already encountered publication topic

No additional information is necessary to route the message correctly as the data previously memorized in the PRTs are sufficient to accomplish this task. Notice that the publication arrives directly to the interested client (see the black arrow out of the client "3" in the figure 4.19), as the node "2" connects directly to the Mosquitto broker running on the node "3", that is the broker to which the client sent the original subscription.

4.4.3 Conclusions and clarifications

The algorithm presented was explained by means of a very simple example, avoiding the complexity of topics with wildcards, multiple levels, and also neglecting the usage of MQTT+ operators into subscription messages, nevertheless, it is clear that what has been said in the previous section can be easily applied to these cases. In particular, the operators of MQTT+ can be treated as simple operators as soon as the reader considers that the Java Servers contain all the logic to treat the messages received by the Mosquitto brokers in order to process and correctly aggregate the data that is stored locally, assuring the correct arrival of data, as this protocol does, allows to take for granted the fact that the operators of the new paradigms will work correctly.

Chapter 5

Testing Methodology

In this chapter, the reader can find a description of the choices made to evaluate the results of the implementation conceptually described in the previous sections.
This overview starts with the description of what are the objectives of the evaluation and the reason behind their choice, then the description moves to the testing environment, its components, and a justification of why this path has been preferred to obtain significant results to evaluate the implementation of the project.

5.1 Evaluation Targets

In order to properly understand what are the evaluation targets may be useful to recall what are the objectives of the entire project. As explained in chapter 2, the main objectives of the system are basically the following:

- **Network traffic efficiency**: it is important that the impact of the developed system on the network traffic among the brokers is lower than the one enforced by the technologies already present on the market. According to this concept, it should be clear that a measure of the dimension of the traffic generated by the solution described in this paper, with respect to the one derived from the usage of the bridging mode, is a good metric to understand whether this target has been achieved or not.

- **Dynamic overlay creation, fault resilience, independent and self-contained system orchestration**: these three targets of the project are tightly coupled to each other and have a strong link with the evaluation presented in the previous point. It is clear that more system functionalities mean more traffic: to correctly form the tree, to verify the connection with the other brokers, and to discover those processes, all things that the normal bridging simply can't do, the project

adds a part of network traffic that cannot be neglected, however, it is desirable that the resulting overhead allows having a resulting amount of data injected in the network at least comparable to the total traffic obtained by means of bridging.

- **Workload**: the list is not ordered by importance but this evaluation objective is for sure the most important. Obviously, it is necessary to verify whether the implementation can be correctly executed by a server machine that usually has an MQTT broker installed on it.
  In order to be able to verify this requirement is crucial to monitor the CPU and memory loads of the processes that make up the new MQTT+ broker. It is possible to predict in advance that the implementation described in these pages has a higher workload with respect to the one enforced by the normal MQTT+ broker, this should be the result of the additional processing activities required by the routing algorithm and the additional functionalities not present in the bridging mode that imply more parallel activities and allocated data structures. An acceptable result should be reasonable CPU and memory requirements, easily satisfiable by a server

- **Convergence performances**: Another important goal to achieve is to have a convergence of the algorithm that makes up the tree of brokers that should add a reasonable delay to the computation of the normal messages. As we said previously in the general description, in particular in the section 3.3, while the Server is busy with the necessary operations to correctly establish the connections with the other brokers, avoiding loops, it queues the MQTT/MQTT+ messages and postpones the processing of this packets until the whole process is completed, this explains why it is essential to have a quick transition from the intermediate states to the nominal one.
  In particular the analysis will be focused basically on two aspects, how much time it takes at the startup to form the tree, and what is the duration of the period between a failure in the system and its complete recovery by the remaining brokers.

The testing environment presented below pursue the goal of measuring the above-described objectives putting the system in a scenario that tries to be as much realistic as possible.

5.2 Testing Environment

After the explanation of which are the goals of the testing process of the project, the reader is ready to know how these objectives have been pursued. The testing environment is mainly based on a platform called *Containernet*

[1], it is a tool that allows simulating a network, setting its topology, and tuning its parameters, such as the links delay and bandwidth. The advantage of using such a tool is that the results of the simulations of the system obtained by means of this emulator can be considered for sure more realistic than the ones retrieved by simply executing the processes locally on a single machine. On the other hand, this type of approach is convenient for its flexibility and integration with a very simple programming language like Python. Moreover using this type of experimental environment allows another level of flexibility considering future works and research in this area, allowing to execute new tests and experiments by simply modifying the images inside the containers, leaving the infrastructure totally untouched.
In addition to being able to properly place and connect network devices such as switches and routers, it allows placing nodes executed inside independent Docker containers.

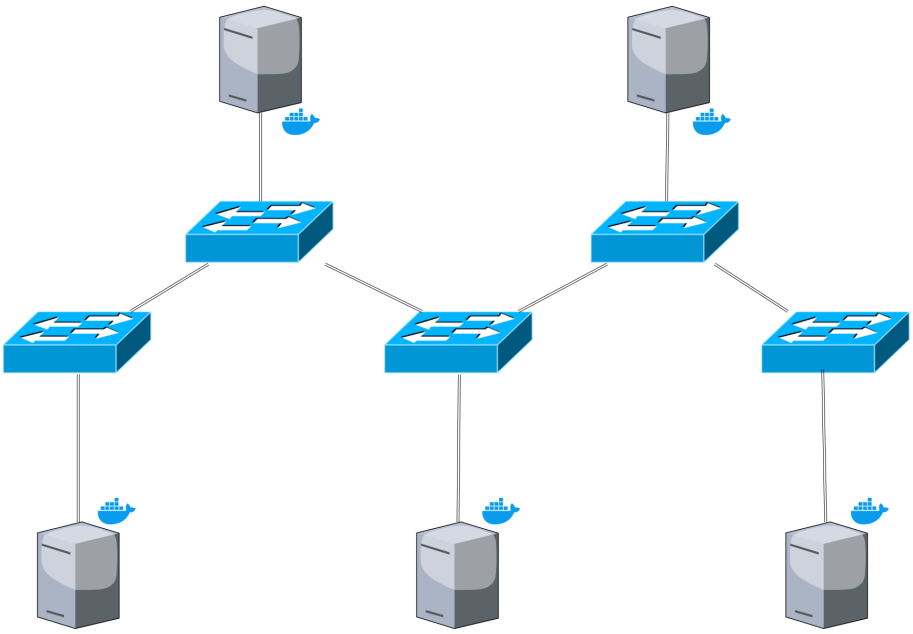

Figure 5.1: Representation of the setting of the network used for the tests

The figure 5.1 shows the setting of the network made up through Containernet to perform the test. The number of selected brokers is five since it represents a meaningful example of a distributed system that is, at the same time, easy to control and trace. As can be seen, the broker is executed inside Docker containers based on OpenJDK 11. The original Mosquitto broker and the Java Server are the compounds of each of the servers visible in the figure above, executed on top of the above-cited base image.
The network is very simple and the Brokers are placed all in the same

[1]see https://containernet.github.io/

(switched) network, the reason for this choice is that the functionalities that require to join and send messages to a multicast group have to be correctly routed by gateways that link networks to each other, it can be easily done by the maintainer of the network by simply adding rules to the IP table of the routers, but from the testing perspective, modifying these tables would require an effort that doesn't make any difference in terms of reliability of the collected results. The links between a broker and its corresponding switch have 1 ms of delay and its bandwidth is 1 Mb/s, these settings are the same for every broker-switch link in the network.

5.2.1 Clients locality

The clients are placed in the network differently according to the test case, the possibilities explored are basically three:

- **100% Locality**: Publishers and subscribers are connected to the same broker.

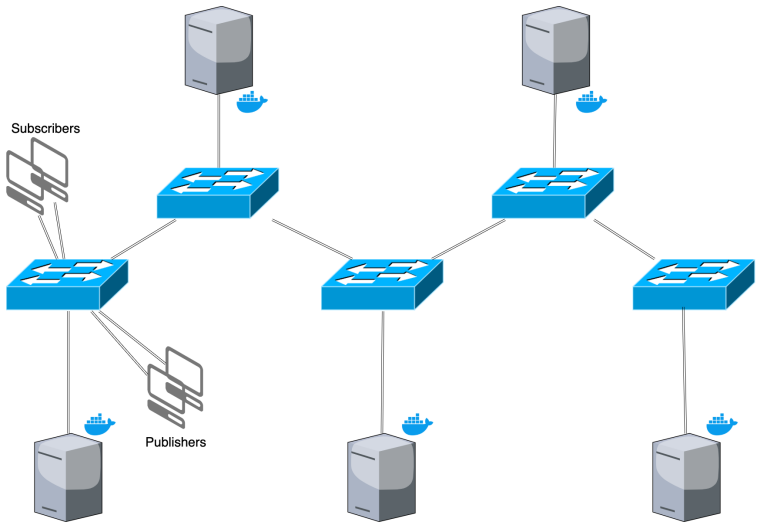

Figure 5.2: Example of how the clients may be placed in case of 100% locality

  The figure above shows the physical location of the clients, they are directly connected to the switch to which the brokers they are connected to are connected.

- **50% Locality**: Publishers and subscribers are connected to the brokers randomly according to a uniform distribution.

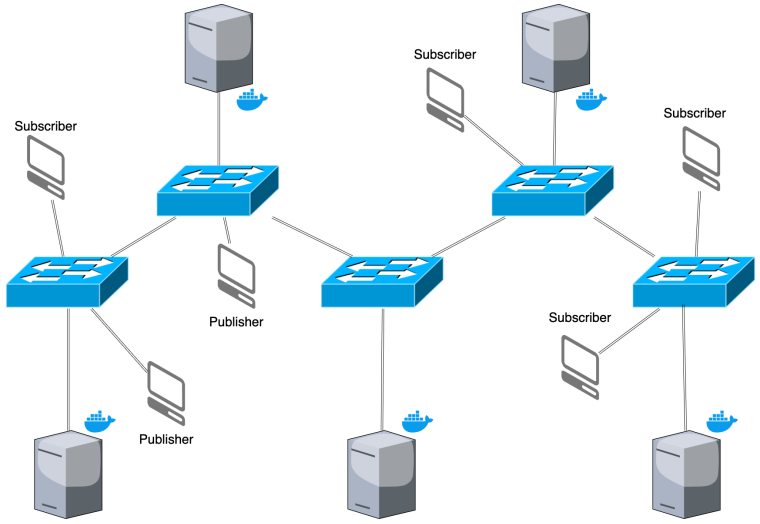

Figure 5.3: Example of how the clients may be placed in case of 50% locality

- **0% Locality**: All the publishers are connected to the same broker, similarly, the subscribers, but the two brokers to which the two classes of clients are connected are different.

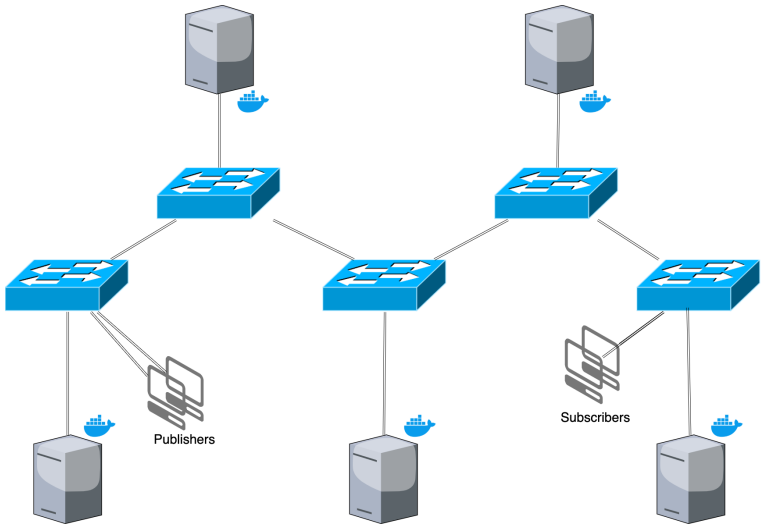

Figure 5.4: Example of how the clients may be placed in case of 0% locality

The localities presented in the bullet list above are the parameters on which each experiment on the traffic is based, as can be seen in the next chapter, as these conditions change the results obtained are quite different. The reader can imagine that the most favorable conditions are guaranteed by the 100% locality of the clients, as the inter-broker traffic of the system is very limited, the reason behind this consequence is that it is not needed to

deliver the publish messages to the other brokers of the system, since the subscribers interested in them are connected to the very same machine of the publishers, the routing algorithm presented in the section 4.4 is able to detect this peculiarity exploiting this aspect, while the normal bridging mode among the brokers continues to blindly forward messages all over the network.

5.2.2 Bridging topology

One of the most important aspects that will be analyzed concerns a deep comparison between the performances obtained with the solution presented in the previous chapter and the ones achieved through the usage of the bridging mode of Mosquitto, as it represents the base of the implemented broker to communicate with the Server. According to this premise, it is useful to show here what is the chosen topology in order to perform the experiments in the cases in which the bridging mode is adopted:

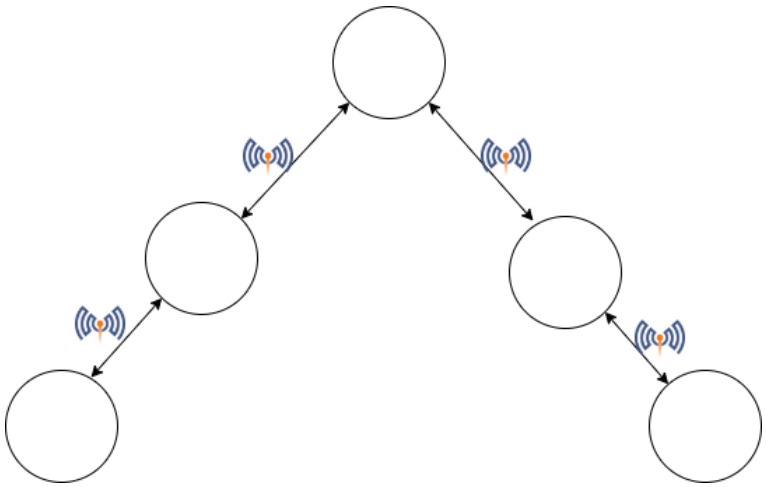

Figure 5.5: Topology adopted to perform the tests in the bridging case

The figure 5.5 shows a "logical" topology enforced on top of the physical one, that remains the same for both cases (bridging and distributed) and is depicted in the figure 5.1, the Mosquitto icons that are placed next to each link express that the overlay connections are established through the bridge configuration this broker offers to its customer. Once more, it is important to highlight that the choice of enforcing a topology is obliged by the fact that this particular mode is static, it imposes to make decisions on which brokers connect together before the startup of the entire system, the logical option was to opt for a balanced tree of 5 nodes, while the dynamic approach described in this paper doesn't give the possibility to know in advance what is the topology that is going to be enforced at a logical level, because con-

sidering as known the physical topology parameters, the computation of the RTT takes into account also the congestion of the node, creating a sort of variability.

5.2.3 Convergence experiments

Calculating the convergence performances of the algorithm would not be simple to be done if different physical machines were used, luckily the choice of Docker results to be useful also in this case, as it guarantees that the different containers are based on the same clock, the one of the machine on which the tests have been executed. This peculiarity brings to the possibility of having a rough estimate of the intervals of time to complete the made up of the tree in different situations.

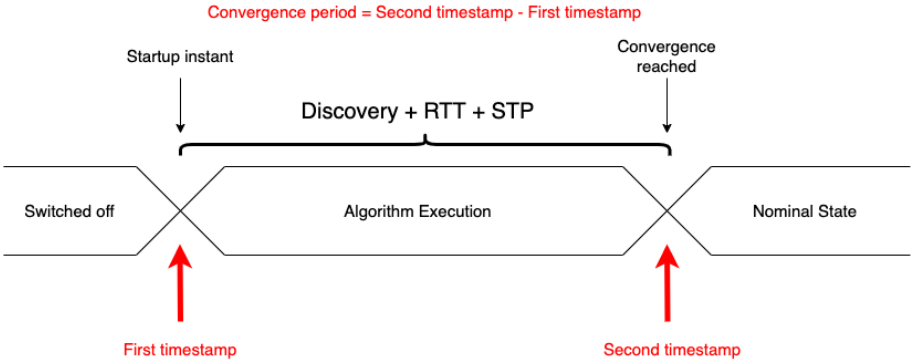

Figure 5.6: How the convergence period is computed after the startup of a broker

The figure 5.6, shows the temporal diagram of a broker, it also shows the time instants that are of interest in the test case represented. As can be seen, by the diagram the test is not interested in the handle of the traffic by the broker, it simply computes the difference between two time instants, it is clear that this type of computation does not guarantee a high precision but still gives an idea on what is the time interval in which the requests are queued at the startup.

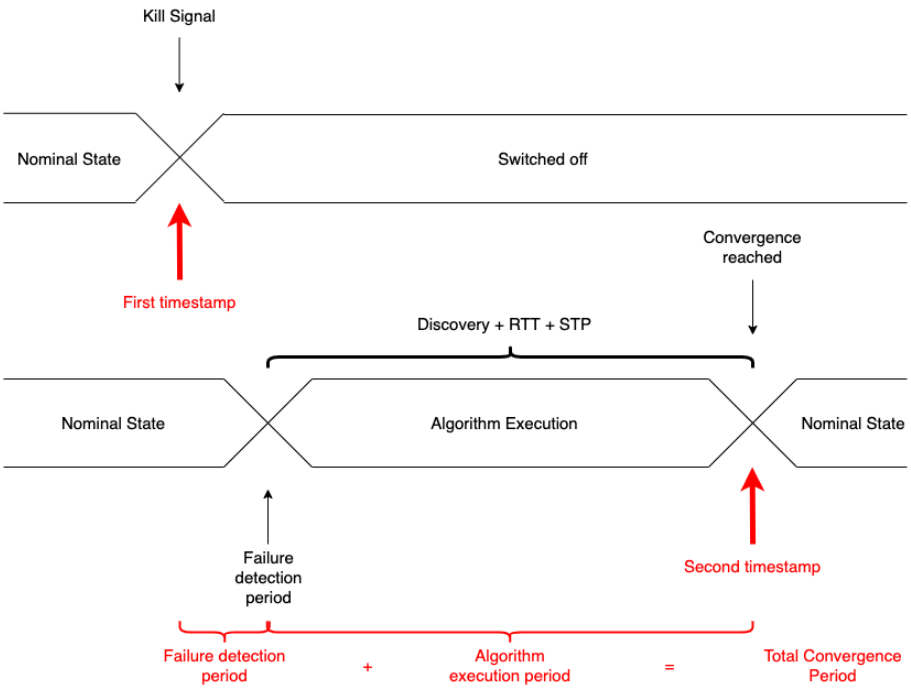

Figure 5.7: How the convergence period is computed after the failure of a broker

In the figure 5.7, the reader finds two temporal diagrams: the former is of the broker that is going to fail during the test, the latter is of one of the remaining brokers. This is the case in which the presence of a synchronized clock is useful, as can be seen in the figure 5.7, the timestamps stored to obtain the final period of time needed to reach again the nominal state are taken from different machines, the presence of clock drifts would have decreased the reliability of the obtained results. This second time period includes two terms: the interval necessary to detect the failure and the actual execution time of the three steps that in the figure are called *Discovery, RTT* and *STP*. Notice that the failure detection penalty is paid only once, as soon as a broker detects this event it restarts the Discovery protocol, by sending a multicast packet to all the servers that joined the group, this message also notifies the need of recomputing the tree and so the other brokers stop to ping the machines they are connected to, restarting the procedure from scratch.

5.2.4 Clarifications and other aspects

Some clarifications about the messages injected by the clients in the network are of extreme importance:

- from the traffic point of view, the heavier operators that can be possible to use from the set offered by MQTT+ are the spatial aggregations

(such as $AVG, $MIN, $MAX ..), a change on any value of the aggregation, results in a change of the aggregated value and this modification should be visible immediately by the subscribers. To execute test cases that are complete but not too long to be executed, this was the preferred solution, as temporal aggregations would require a lot of time to generate a sufficient load, and data processing operators would be too heavy to allow to execute multiple brokers on the same machine.

- from the functional point of view, given from granted that the server that handles the new functionalities of MQTT+ works correctly, testing the functioning of the system using the spatial aggregation would be exactly the same, as the data processing and temporal aggregation operators at the end of the day are represented by numbers inside the publish messages to their subscribers. Testing that the spatial operators work allowed to be sure that also the other operators do the same, as the communication between brokers is focused on publication and the results of the aggregations or other operators are processed by each single broker and sent to the nearest subscriber.

Beyond the isolation of the execution of the processes, using the brokers inside the Docker containers has an additional advantage, it allows to monitor some of the usage statistics of each container, treating them as separate machines with respect to the allocated resources. In particular, the command *docker stats* returns, every time it is called, the percentage of resources (CPU and Memory) the container is using, against the ones allocated to it. Allocating the same resources to the containers during the tests in which the traffic is measured, made it possible to obtain the differences in terms of resource usage by the two alternatives subject to the previously described trials.

Chapter 6

Experimental Results

In this chapter, the reader will be able to see what are the results obtained by the experimental methodology introduced in previous pages. In the first section, it is possible to find the discussion about the results concerning the traffic, while the following ones are dedicated to the workload and the convergence performance of the system.

6.1 Network traffic

The main goal of the project is to apply a new concept in the way the MQTT messages are routed among brokers that belong to a communicating system. In the following results, we expect to see a significant difference in terms of traffic generated to correctly route the publish messages among the nodes that make up the distributed system.
The test case executed has the following features:

- There are 5 clients that subscribe to the same filter topic but each of them uses a random MQTT+ operator selected from the spatial aggregation operators previously introduced in this paper. They are all located on different docker containers, with different IPs.
- There are 20 publishers, each of them publishes 50 messages, there are 5 docker containers, and so 5 different IPs, each of them executes 4 processes in which it is simulated the process of publishing 1 message every second. Every different publisher publishes to a different topic, every topic selected by them matches the filter previously described concerning the subscribers, this means that the messages generated by this class of clients are entirely useful to generate corresponding traffic for the subscribers.
- Each subscription message has a dimension of 89 bytes on average.

- Each publish message has a dimension of 98 bytes on average, the payload of the publications is the timestamp of the time instant in which the single message has been sent.

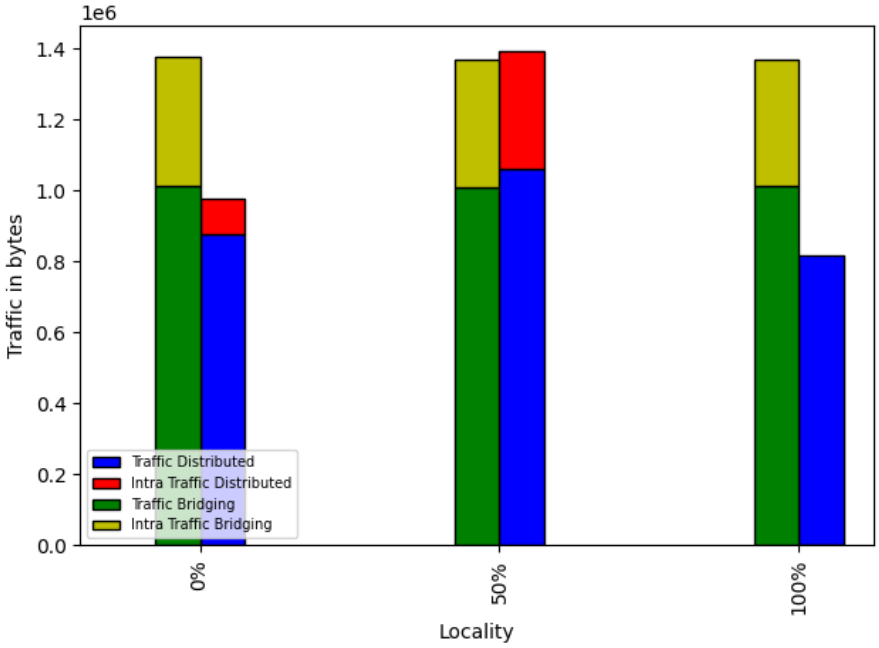

Figure 6.1: Representation of the obtained traffic dimension (in bytes) differentiated by the algorithm and the locality adopted in the experiment

	0%	**50%**	**100%**
Distributed	877280	1060326	817676
Bridging	1011320	1009339	1014473

Table 6.1: Tabular representation of the traffic (in bytes), considering as excluded the part concerning the publishes forwarding, differentiated by the locality and the routing methodology adopted

	0%	**50%**	**100%**
Distributed	98505	334224	0
Bridging	363872	359648	356414

Table 6.2: Tabular representation of the traffic (in bytes) concerning the publishes forwarding, differentiated by the locality and the routing methodology adopted

Once informed about the dimension of the traffic that represents the input of the system, the reader is ready to analyze the results obtained by running

the test cases previously characterized.
Each vertical bar represented in the figure 6.1 depicts the total traffic generated by each test case, differentiated by the technology used and the locality applied to place the clients. Every bar has a further subdivision, on top of each of them, it is possible to find the cumulative dimension of a particular class of packets generated during the run of each test, this category of traffic represents the messages used by the brokers to route the publications to the other node of the network.
Observing the figure it is clear that the main goal of the project has been achieved, as the total traffic dimension is lower for the distributed cases than the bridging ones, except for the 50% locality, moreover the reduction in terms of traffic generated to route the publications is significant, as the 0% locality case with the developed protocol has a total dimension of this class of packets that is one order of magnitude lower than the one obtained with the same locality and the bridging used to connects the broker. The 100% locality shows a cumulative traffic dimension of this class of packets equal to 0, which means that no useless information is routed to the brokers to which no clients are connected, while the bridging method does not exploit this additional advantage to reduce the number of bytes exchanged by the nodes.
Concerning the 50% locality case, it is likely that disseminating clients in a totally random way all over the network, brings the distributed case to have very similar traffic with respect to the particular class of packets highlighted in the figure, paying a lot more in terms of the signaling overhead, as can be guessed following the explanation in the previous chapter of this paper, concerning the routing strategy adopted. The signaling overhead to keep up the system, computing the RTTs, filling the routing table, detecting failures, during the entire test case is included in the "residual" part of the bars in the figure, placed at the bottom, as can be noticed, has a low impact in the 0% and 100% locality total traffic, while in the 50% one it assumes a certain relevance as the blue bar is higher than the green one of the corresponding case, revealing that, excluding the common traffic, this class of packets has a higher weight with respect to the messages needed to correctly set the and keep running the bridging mode among the nodes.
Figure 6.1 also reveals the smartness of the developed protocol with respect to the usage of blind flooding of messages all over the logical links that connect the brokers. The behavior of the brokers during the three tests executed using the bridging mode appears to be quite "flat" across the different cases, while the modified PADRES algorithm shows that it correctly exploits the differences in terms of client placing, in particular, these features are visibly observing the highlighted traffic (the yellow and red bars of the figure), since the bridging shows similar results, while the distributed cases can be recognized by only observing this particular class of traffic.

6.2 Workload

When the CPU and Memory usage is concerned, the reader must be informed about what is the execution environment in which those experiments have been performed. The machine on which the test cases have been executed is a Linux (with Ubuntu 18.04 release) one, with 12 GB of memory and a CPU that is an *Intel i9-9880H* clocked to *2,30 GHz* with Turbo-Boosting that brings the **16 Cores** up to *4,80 GHz*.

6.2.1 CPU workload

In the following some graphics and data that refer to a comparison on the CPU usage by the implemented algorithm with respect to the execution of brokers in bridging mode are shown:

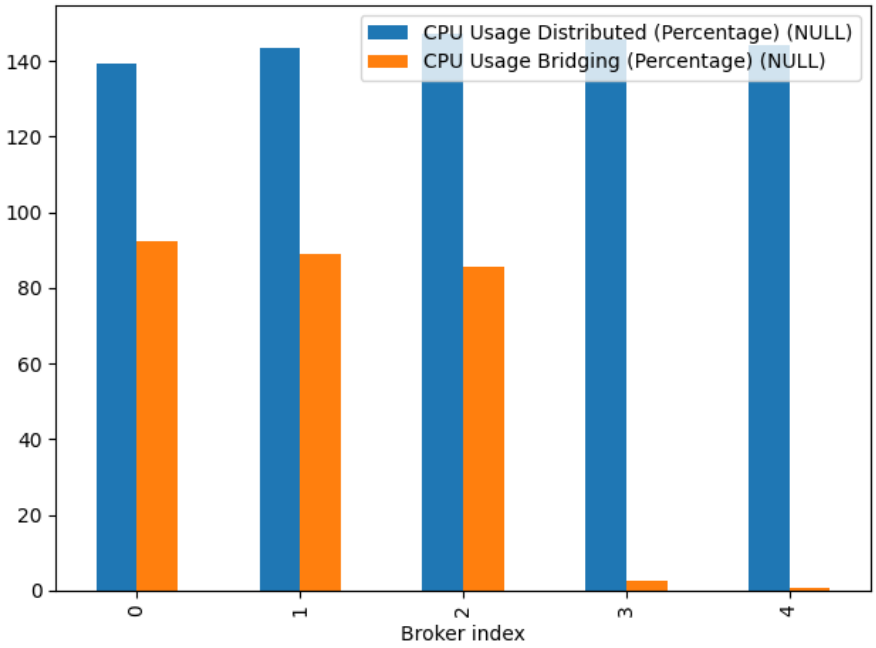

Figure 6.2: Average usage of the CPU by the 5 brokers during a test case in which clients are placed with 0% locality

	Distributed % CPU usage	Bridging % CPU usage
Container 1	139.163	92.198
Container 2	143.590	88.869
Container 3	147.346	85.453
Container 4	145.880	2.500
Container 5	144.320	0.844

Table 6.3: Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 0% locality

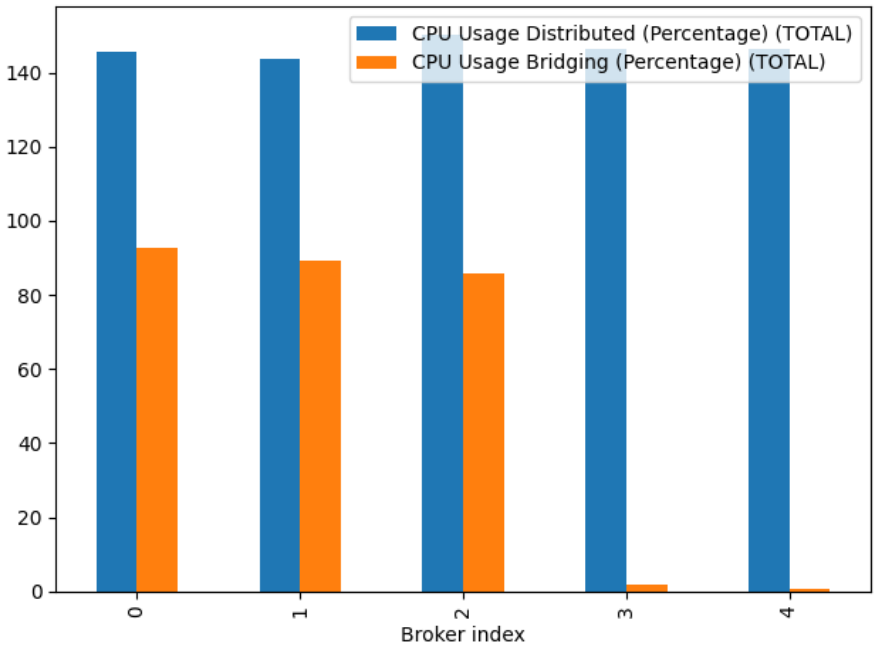

Figure 6.3: Average usage of the CPU by the 5 brokers during a test case in which clients are placed with 100% locality

	Distributed % CPU usage	Bridging % CPU usage
Container 1	145.520	92.823
Container 2	143.798	89.442
Container3	150.352	85.822
Container 4	146.213	1.684
Container 5	146.434	0.813

Table 6.4: Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 100% locality

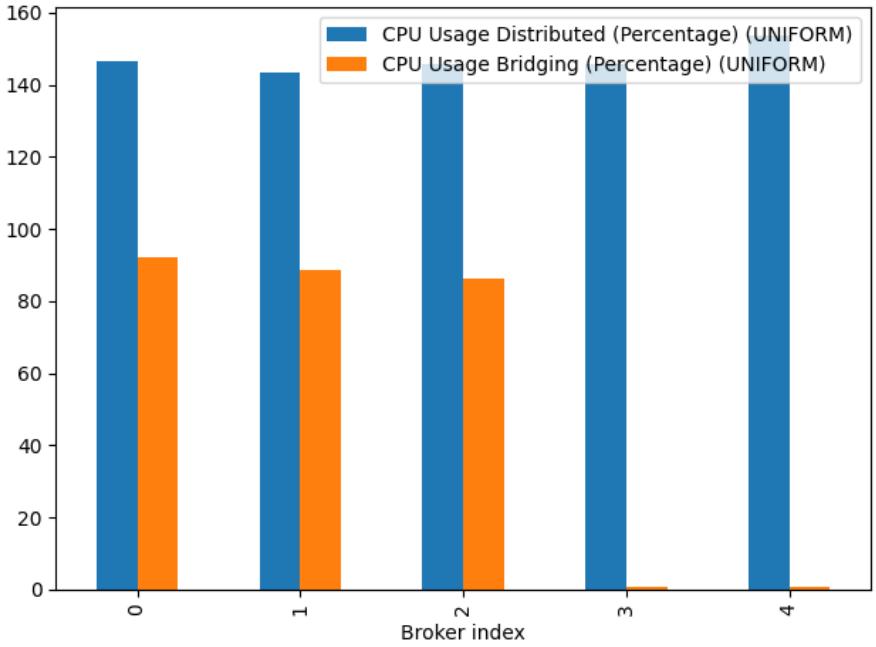

Figure 6.4: Average usage of the CPU by the 5 brokers during a test case in which clients are placed with 50% locality

	Distributed % CPU usage	Bridging % CPU usage
Container 1	146.590	92.125
Container 2	143.205	88.794
Container 3	145.844	86.121
Container 4	145.858	0.806
Container 5	153.646	0.771

Table 6.5: Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 50% locality

The results that are shown in the three figures and tables above reveal that the broker implemented is quite CPU demanding, as they show an average usage of the CPU that is beyond 100%, this is not surprising in a multi-core environment, it simply means that the processing operations required by each container use more than one processing unit on average in a single test-case.
The usage pattern of the CPU for the distributed case appears to be quite flat, so the reader can imagine that most of the excess percentage with respect to the bridging case is due to the operations that are not implemented in the latter that is heavily influenced by the traffic that has to handle during the test, so being affected by where the clients are placed.
The tables also show that the leaf of the bridging topology has low usage of the CPU, as they received all the messages without having to forward them to the other nodes.

6.2.2 Memory workload

The situation is quite different as long as memory usage is concerned, the reader should remember that the algorithm presented uses more data structures and the use of the Java part by this new broker is higher than the one that exploits bridging, this results in a potential higher usage of the memory available, however, the load on this resource is not so heavy, as it can be considered grossly twice the usage of a normal broker
The percentages of memory occupation are so quite easy to be managed by a server that has to run an MQTT broker:

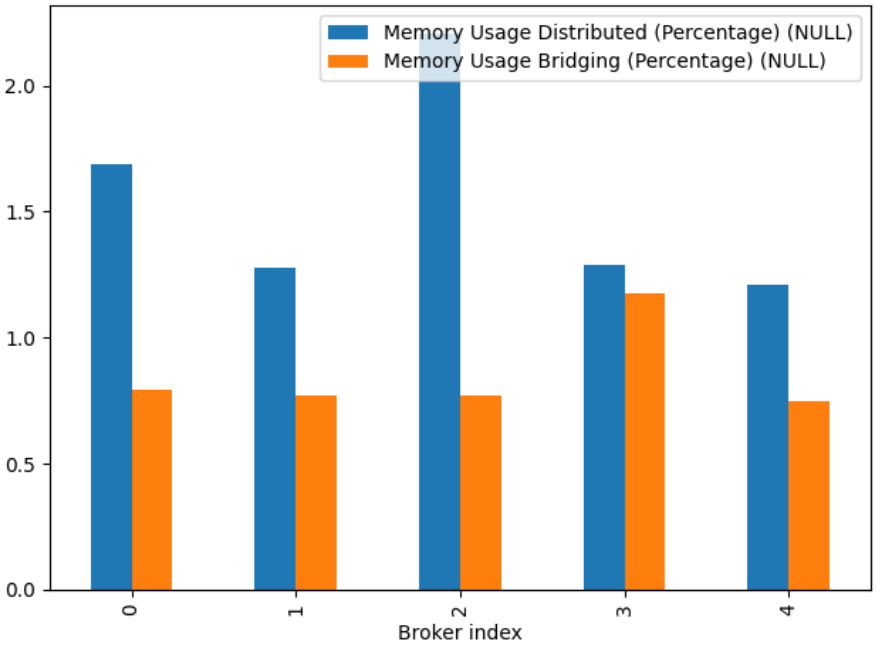

Figure 6.5: Average usage of the memory by the 5 brokers during a test case in which clients are placed with 0% locality

	Distributed % Memory usage	**Bridging % Memory usage**
Container 1	1.689	0.790
Container 2	1.280	0.769
Container 3	2.208	0.770
Container 4	1.286	1.177
Container 5	1.211	0.750

Table 6.6: Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 50% locality

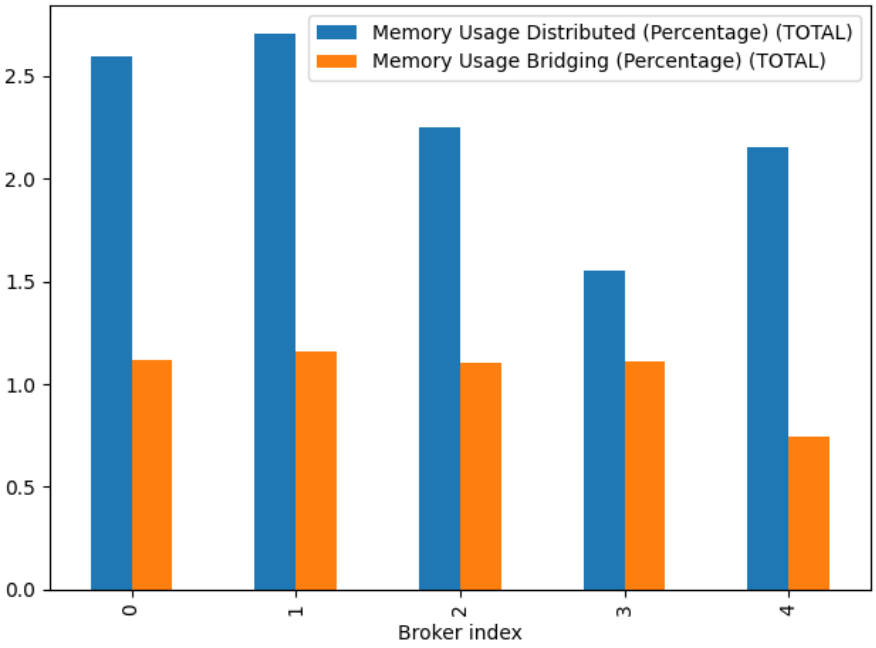

Figure 6.6: Average usage of the memory by the 5 brokers during a test case in which clients are placed with 100% locality

	Distributed % Memory usage	**Bridging % Memory usage**
Container 1	2.596	1.116
Container 2	2.709	1.157
Container 3	2.249	1.107
Container 4	1.550	1.112
Container 5	2.154	0.747

Table 6.7: Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 100% locality

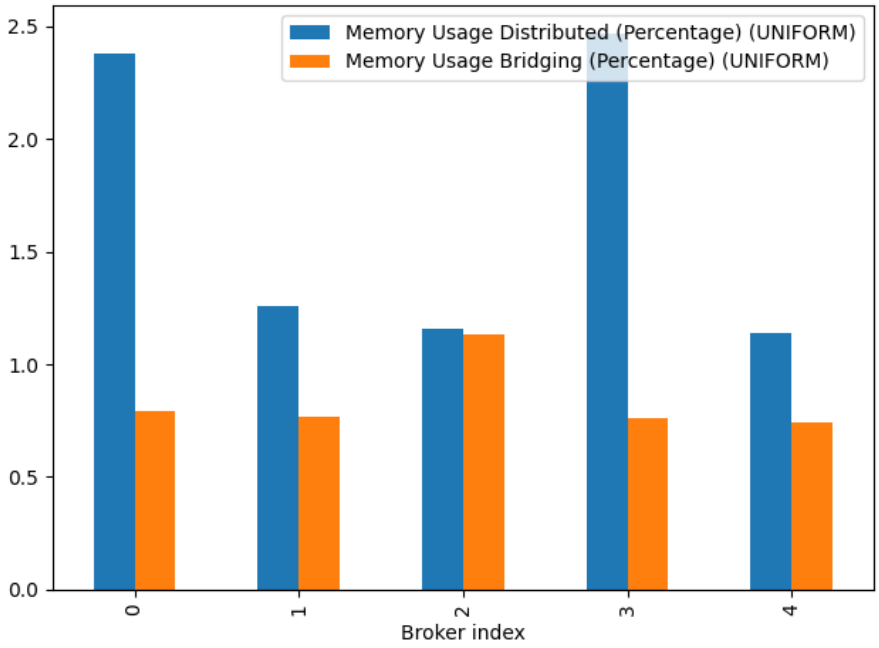

Figure 6.7: Average usage of the memory by the 5 brokers during a test case in which clients are placed with 50% locality

	Distributed % Memory usage	**Bridging % Memory usage**
Container 1	2.383	0.794
Container 2	1.258	0.768
Container 3	1.157	1.135
Container 4	2.471	0.764
Container 5	1.140	0.744

Table 6.8: Average percentages of CPU resources of the host machine used by each container during a test case in which clients are placed according a 50% locality

What was said about the CPU can be said also about the memory usage, it is a quite reasonable demand of resources considering that in some cases it is similar in the two approaches and the Java Virtual Machine is not optimized especially concerning the object and dynamic allocation, while the C language, used to write the Mosquitto part of the broker, is tailored for this purpose. The reader should also remember that the PADRES approach, and in turn its adaptation developed in this project, uses additional data structures like the routing tables (ORT, PRT and SRT), moreover the other protocols (Discovery, RTT and STP) need more data structure and more

data to be kept in memory to work. Despite what has been said, the two approaches appear to be at least comparable in terms of demand of resources, even if the comparison in this particular category results in an advantage for the bridging approach.

6.3 Convergence results

One of the other aspects that is important to test about the developed project is the convergence performance when the system is subject to events like the startup of the system and the failure of one of the brokers of the network. The reader can find in the chapter 5 the description of how these convergence performances have been computed but it would be useful also to detail how these experiments have been carried out:

- Concerning the startup convergence, the 5 brokers, previously used to test the workload and the traffic performances, have been started one after the other and the moment in which the tree was formed was measured for each of those nodes. These measurements have been performed 10 times and the results that will be presented to the reader are the average values of these iterations of the executed tests.

- The failure convergence has been tested as follows: the 5 brokers have been started, then one of the containers executing a broker received a kill signal simulating a connection tear down, or, generally speaking, a malfunction that would not allow the node to correctly reply to the RTT messages to the other brokers, the time instant in which the remained brokers agreed about the new topology has been recorded. Similar to the previous case described the results that will be shown are the average of 10 measurements.

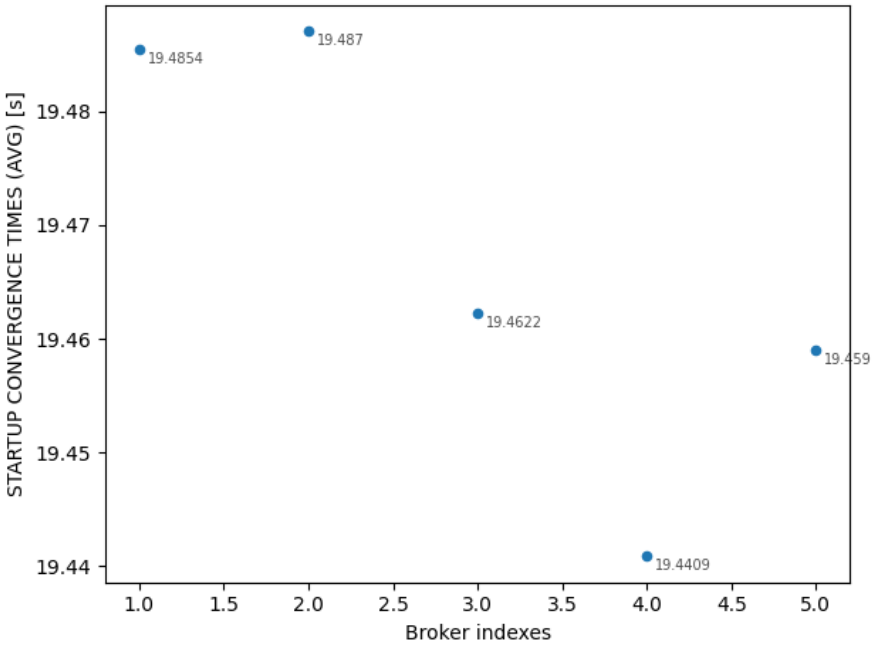

Figure 6.8: Average convergence performance for the 5 brokers at the startup of the system

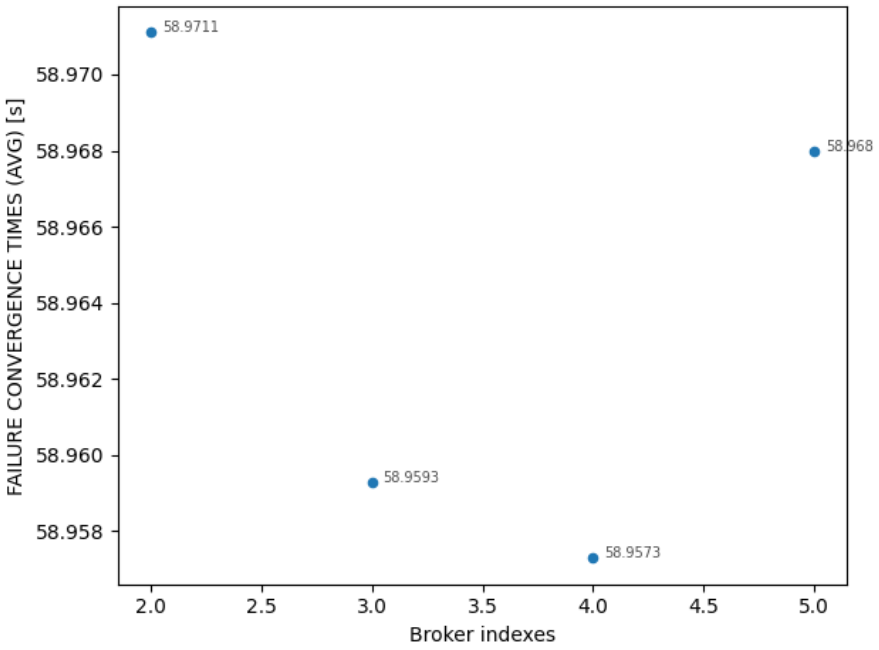

Figure 6.9: Average convergence performance for the remaining 4 brokers at the failure of one of the brokers

The figures 6.8 and 6.9 shows two different classes of results, as the numerical difference suggests, the second measurements take into account also the time the system takes to recognize the failure and to complete the procedure as explained in the chapter 5. Considering that these intervals are period of time in which the system queues the requests forwarded by the Mosquitto broker, sent either by clients or by the Java part of other MQTT+ brokers, it would be satisfactory to have reasonable values that would not force the broker to arrive to a possible loss of requests or to a malfunctioning, more in general. The reader should agree to the fact that these values are absolutely reasonable, moreover, it should be considered that they are heavily influenced by some timers that at an implementation level can be reduced or increased in future versions of the protocols, involved in the convergence process of the system.
The last important consideration that has to be done is that the measurements depicted in the two figures above have a very low standard deviation (this is the main reason behind the choice of not reporting it onto the figures), this suggests that the protocols reach the convergence on different brokers with a very low difference in time, so the orchestration among the brokers allows to have a good synchronization that results in a good efficiency in case the system encounters high traffic in input, preserving a good consistency among the data circulating among the network nodes.

Chapter 7

Final considerations

According to the results presented in the previous section of this chapter, the main goals presented at the beginning of this paper have been achieved.

- The traffic generated by the solution developed showed a significant reduction with respect to the alternative that would have allowed to let the MQTT+ paradigm working also in a distributed environment, the only exception is represented by the 50% locality case.
  In order to be able to obtain measurements that are always below the ones that are given by the bridging case, one of the main paths of research that can be followed is to reduce the signaling overhead, that the protocols developed in this paper needs to operate. In particular, concerning the UDP packets, it would be possible to further reduce the information conveyed by these messages, standardizing some fields and improving the dimension of the packets with respect to the considered KPI.
  Furthermore it would be possible to carry out some analysis about the ideal period of time across RTT messages sent by the same source to the same recipient, this would give a good trade-off between the number of RTT messages injected in the network during a certain period of time and good performances in terms of failure detection and topology re-computation, following changes of the network conditions.

- The developed protocols require a reasonable amount of resources to be executed, nevertheless, it is possible to further reduce this aspect. One of the ways to pursue this goal is to integrate the operations that are now carried out inside the Java Server to a code that has a better usage of the resources, moving downwards the level of abstraction of the development but keeping the conceptual scheme of the implementation intact. This would allow the developer to control the number of threads spawned by the broker and to allocate and deallocate carefully the data structure necessary to the various functionalities that

are now implemented inside the proxy.
The most intuitive way to do so would be to implement entirely the Java server functionalities inside the Mosquitto broker, this would improve also the easiness as far as deploying is concerned, as only one binary file would be necessary, while now the Mosquitto binary has to be executed at the same time of the server JAR which requires the installation of Java on the Server machine.

- The convergence performances showed in the section 6.3 can be considered good enough but they could be improved by taking care of the traffic generated by increasing the frequency of the exchange of signaling messages. These improvements are strongly linked to the first point of this list, as said before, in fact, in order to obtain an optimal trade-off between convergence performances and low traffic overhead a specific study in this sense should be carried out.

One final consideration regards the testing environment, as its re-usability would allow to easily test the new research path results obtained by changing the implementation of the broker, in particular, the complete isolation of the containers with respect to the network simulation given by the mininet framework, on which Containernet is based, allows to simply change the deployment principles and the conceptual structure of this complex and heterogeneous system, without any further effort by the developers.
Although the original goal of the project was to allow the possibility to extend MQTT+ to be used in a distributed environment, it is clear that what has been developed has a certain relevance also considering the base protocol, MQTT, since the routing principles, along with the algorithms that are useful to establish and keep connected multiple brokers, represent also an important contribution to have an efficient and fault resilient connection among those components of the original version of the protocol.

Bibliography

[1] Andrew Banks and Rahul Gupta. Mqtt version 3.1.1 oasis standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 2014.

[2] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized publish-subscribe infrastructure. In *Proceedings of the 3rd International Workshop on Networked Group Communication (NGC'01)*, volume 2233, pages 30–43. Citeseer, 2001.

[3] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The padres distributed publish/subscribe system. In *FIW*, pages 12–30. Citeseer, 2005.

[4] Zihui Ge, Ping Ji, Jim Kurose, and Don Towsley. Matchmaker: Signaling for dynamic publish/subscribe applications. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, pages 222–233. IEEE, 2003.

[5] Riccardo Giambona, Alessandro EC Redondi, and Matteo Cesana. Mqtt+ enhanced syntax and broker functionalities for data filtering, processing and aggregation. In *Proceedings of the 14th ACM International Symposium on QoS and Security for Wireless and Mobile Networks*, pages 77–84, 2018.

[6] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Reliable and highly available distributed publish/subscribe service. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 41–50. IEEE, 2009.

[7] Roger A Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265, 2017.

[8] Edoardo Longo, Alessandro EC Redondi, Matteo Cesana, Andrés Arcia-Moret, and Pietro Manzoni. Mqtt-st: A spanning tree protocol for distributed mqtt brokers. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2020.

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[10] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.