DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

PAULO CÉSAR LEITE DE MATOS

BSc in Computer Science and Engineering

FAULT-TOLERANT PUBLISH-SUBSCRIBE SYSTEM WITH MULTIPLE DELIVERY GUARANTEES

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

FAULT-TOLERANT PUBLISH-SUBSCRIBE SYSTEM WITH MULTIPLE DELIVERY GUARANTEES

PAULO CÉSAR LEITE DE MATOS

BSc in Computer Science and Engineering

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA School of Science and Technology

Examination Committee

Chair: Miguel Goulão
Associate Professor, NOVA School of Science and Technology

Rapporteur: Miguel Matos
Assistant Professor, IST/UL

Member: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA School of Science and Technology

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

Fault-Tolerant Publish-Subscribe System With Multiple Delivery Guarantees

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.5) [20].

Acknowledgements

Working on this dissertation has been one of the most challenging yet rewarding experiences of my life. Along the way, I have been fortunate to receive the support and encouragement of many incredible people.

First and foremost, I would like to express my gratitude to my adviser, Hervé Paulino, for giving me the opportunity to contribute to such a complex and important project. His guidance and expertise were invaluable throughout the development of the work presented in this dissertation.

I am also grateful to the Department of Computer Science and NOVA LINCS at NOVA School of Science and Technology for providing me the knowledge, tools and resources that were crucial for carrying out this work and preparing me for my future as a professional.

Afterwards, I would like to extend my thanks to all my colleagues and friends with whom I had the pleasure of working with, and that made my university experience more pleasant.

Finally, I wish to express my deepest gratitude to my family for the immense sacrifices and unwavering support throughout these years. To my parents, who were my greatest source of strength, thank you for teaching me the values of hard work, perseverance and self-belief. Your faith in my potential allowed me to overcome the challenges that I found in my way, even in moments when I doubted myself. Without you, this dissertation would not have been possible.

Abstract

Distributed system's ability to scale has become a core requirement to provide services across the globe due to the increase in worldwide connectivity prompting developers to devise solutions able to provide high availability and performance through geo-replicated systems. This strategy brings data closer to clients, however, with data being stored in multiple distant locations update propagation becomes slower, hindering both availability and performance. This is known as the trade-off between availability and data consistency, being more noticeable in situations where not all data or operations have the same consistency requirements. Developers started looking at approaches that support multiple consistency models, more often called mixed consistency model. The reasoning behind this model, is that developers are able to choose the type of consistency each operation requires and can adapt the propagation of operations accordingly.

Ginger is a distributed middleware system that supports the existence of operations with different consistency requirements and capable of disseminating messages according to the total, causal and eventual orders. The current issue with Ginger, is that it does not behave correctly under the occurrence of faults. Therefore, in this thesis, we propose to implement, on top of the already existing dissemination protocol, fault-tolerant mechanisms that ensure its correctness in circumstance where network or process faults occur, taking into consideration current approaches presented in the literature.

In the end of this thesis, we will get an updated version of Ginger with the necessary modifications to ensure the reliable delivery of messages, under the occurrence of faults, while guaranteeing Ginger's dissemination protocol and ordering guarantees are preserved. The experimental results, will demonstrate if the proposed solutions are capable of effectively tolerating faults, if they do not introduce undesired overhead when recovering from them and what is the impact of introducing fault-tolerance in the system, when no failures occur.

Keywords: Distributed systems, Consistency, Publish-Subscribe, Fault-tolerance, Replication, CAP Theorem

Resumo

Com o aumento da conetividade e a necessidade de fornecer serviços à escala global com alta disponibilidade e desempenho, a capacidade de escalar de sistemas distribuídos tornou-se essencial. Para essse efeito, recorrem-se a sistemas geo-replicados, onde dados são armazenados em vários locais distantes. No entanto, a propagação de atualizações torna-se mais lenta, prejudicando a disponibilidade e o desempenho. Isso deve-se ao compromisso entre a disponibilidade e a consistência dos dados, sendo mais notório quando diferentes dados ou operações têm requisitos diferentes de consistência. Os programadores começaram a procurar abordagens que suportem vários modelos de consistência, conhecidos como modelo de consistência mista. Com este tipo de modelos, estes podem escolher o tipo de consistência necessário para cada operação e adaptar a propagação das mesmas em conformidade.

O Ginger é um sistema de middleware distribuído que suporta a execução de operações com multiplas garantias de entrega, capaz de disseminar mensagens de acordo com as ordens total, causal e eventual. O problema atual do Ginger é não se comportar corretamente perante a ocorrência de falhas. Assim, nesta tese, propomos implementar, sobre o protocolo de disseminação já existente, mecanismos tolerantes a falhas que garantam a sua correção em circunstâncias em que ocorram falhas na rede ou nos processos, tendo em consideração as abordagens atuais apresentadas na literatura.

No final desta tese, obteremos uma versão atualizada do Ginger com as modificações necessárias para assegurar a entrega fiável de mensagens, sob a ocorrência de falhas, garantindo que o protocolo de disseminação do Ginger e as garantias de ordenação são preservados. Os resultados experimentais demonstrarão- se as soluções propostas são capazes de tolerar eficazmente as falhas, se não introduzem sobrecargas indesejadas na recuperação das mesmas e qual o impacto da introdução de tolerância a falhas no sistema, quando não ocorrem falhas.

Palavras-chave: Sistemas distribuídos, Consistência, Publicador-Subscritor, Tolerância a falhas, Replicação, Teorema CAP

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

FIFO First-In-First-Out *(pp. 20, 25, 26, 29, 30, 44, 46–48, 50, 54, 55, 68, 95)*

CAP Consistency - Availability - Partition-Tolerance *(p. 2)*

DDS Data Dissemination Service *(pp. 4, 23)*

MDS Metadata Dissemination Service *(pp. 4–6, 23, 25, 26, 29, 30, 34)*

MI middleware instance *(p. 26)*

pub/sub publish/subscribe *(pp. 4–9, 13–17, 19, 23, 24, 26, 30, 54, 56, 58, 68, 69)*

1 Introduction

1.1 Motivation

Worldwide connectivity has led to greatly increase the demand over systems's resources due to the increasing number of clients at different locations. Therefore, scalability has become a crucial goal of developers in the design of distributed systems [27, Chapter 1]. A system is scalable if it is able to handle the addition of resources without suffering a noticeable loss of performance or increase in administrative complexity [21, Chapter 9]. Scalability of a system can be evaluated in different dimensions, however we focus on two that relate more directly to issues we face.

The first dimension relates to the system's size, in the sense that we are able to easily add more users and resources to the system. When a system needs to support more users and resources, having a single server responsible for implementing a service, managing a database and routing messages through many communication channels, highly increases the load on the server, becoming a bottleneck as the number of users and resources increases, and, in case failures occur, it becomes a single point of failure leading the service, data and routing algorithm to become unavailable [27, Chapter 1].

Another dimension used to evaluate scalability is a system's ability to scale geographically when its users and resources lie far apart. Scaling geographically poses a challenge mainly related to the problems of having multiple centralized components in the context of wide-area communications. This network setting inherently leads to performance and reliability issues, due to the inability to instantly transmit messages between locations far apart from each other, leading, for example, to periods of time where the system's state seen by clients in those locations may be different [27, Chapter 1].

Among the different scaling techniques that can be employed, one of the most important is replication. Replication has become crucial in today's distributed systems. It allows to manage and balance the load among components, leading to better performance, but also have replicas of the system closer to its clients, allowing for lower latency [27, Chapter 7]. However, having multiple replicas of a resource raises the problem of consistency among them, since updating one, in the absence of synchronization protocols, leads to replicas

having different copies of the resource.

A well known and famous theorem in the world of distributed systems is known as the Consistency - Availability - Partition-Tolerance (CAP) [4] theorem, which claims that in a network, subject to failures, with shared data it is impossible to simultaneously provide all of the following properties: Strong **C**onsistency, High **A**vailability and **P**artition-Tolerance [28]. The trade-off between performance and consistency has been one of the main challenges developers have faced, in the sense that one can achieve performance by sacrificing consistency or achieve consistency by sacrificing performance [18]. Therefore, developers need to carefully reason about the consistency requirements of their systems and if they are willing to accept the consequences of using solutions that allow lower latency and high availability or using solutions that ensure the safety of data and operations. Currently the most common strategy is to lower the consistency requirements in order to achieve better performance, mainly due to the fact that clients desire to get quick responses to operations they execute.

In scenarios where different operations do not require the same consistency guarantees, running with a single consistency level can severely impact, the latency and availability of operations' results that only require weak consistency, in case the system is using stronger guarantees, introducing unnecessary delay for operations that only require weaker consistency guarantees, or the safety of data, otherwise. It is, therefore, important that systems are able to provide mechanisms that support multiple levels of consistency, so that the execution of operations does not violate the required strong consistency guarantees, ensuring the safety of operations, while allowing operations that do not require the same level of consistency, to execute and make their results available faster to clients.

1.2 Ginger

The work to be developed in this thesis focuses on Ginger [22, 9], a distributed middleware system that coordinates operations between applications and database systems and allows application developers to register services and its operations. Ginger's main purpose is to provide a system that facilitates the development of applications that support mixed consistency guarantees. Inherently, this consistency model requires a dissemination infrastructure able to deliver messages respecting their consistency requirements while ensuring that, in scenarios where components fail, those requirements are not violated.

Ginger follows a layered architecture (Figure 1.1) composed by the following layers:

Front-End API: provides an user-friendly library that allow developers to define transactions and operations. It executes on clients and operations are translated into messages in the middleware.

Middleware: facilitates communication between client applications and databases. This layer is responsible of ensuring that operations execute according to the consistency levels defined by the developer.

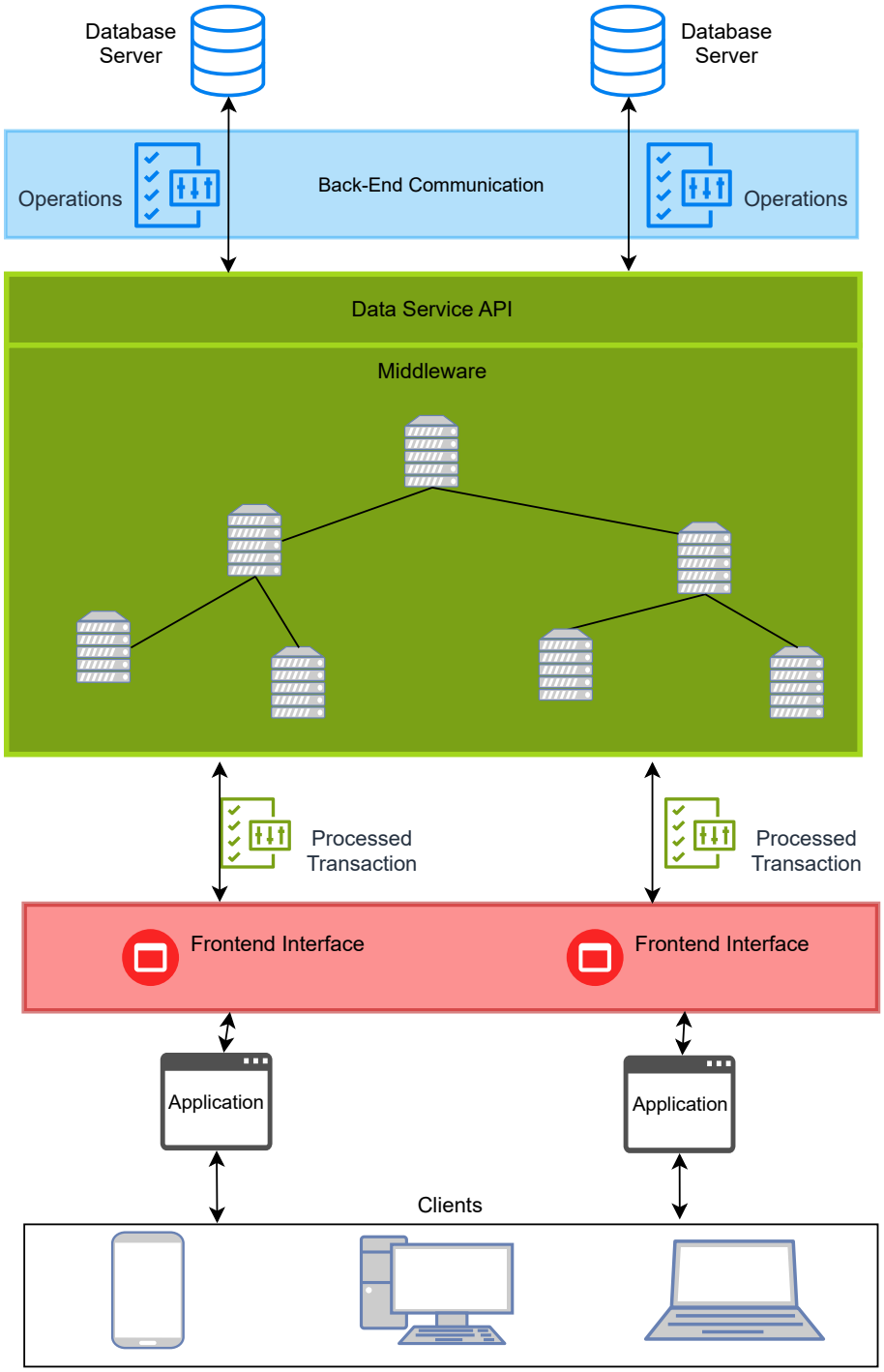


Figure 1.1: Ginger architecture adapted from [22].

Table 1.1: Bank service operations.

Operation	Description	Consistency
getBalance(account)	Returns the *account*'s current balance	Eventual
deposit(account, amount)	Increments the *account*'s balance in *amount*	Causal
withdraw(account, amount)	Withdraws *amount* from *account*'s balance	Linear

Data Service API: provides an API to add service's implementations to the system. A service consists of a well-defined interface that specifies a set of operations accessible to clients. Each implementation must define a *data store* and specify how the operations interact with this *store*.

Backend Communication: handles interactions with data stores, enabling the commit of operations.

Ginger's middleware infrastructure facilitates the dissemination of operations from applications to clients, each connected to a storage system responsible of storing application's data, through a Data Dissemination Service (DDS) and a Metadata Dissemination Service (MDS), in which this thesis will mainly focus.

The MDS follows a publish/subscribe (pub/sub) architecture using a subscription scheme based on topics, built on top of a dissemination tree and composed by several nodes called emphbrokers, responsible by facilitating dissemination of messages from applications to clients, and implements the necessary mechanisms to ensure that operations execute according to the defined consistency level. As the dissemination process is based on topics, when multiple clients subscribe to a topic, those subscriptions create a per-topic sub-tree on top of the already existing dissemination tree and is used to route publications of the associated topic.

Ginger provides applications the ability to support operations with different consistency requirements, supporting the *eventual*, *causal* and *linear* consistencies. The dissemination infrastructure implements delivery mechanisms to ensure messages are delivered appropriately and, according to the consistency level associated with the operations they carry, to every subscriber. In the Ginger's MDS, linear consistency is guaranteed by delivering messages in total order to every client. To ensure causal consistency, messages are delivered in causal order. Finally, since eventual consistency only requires messages to eventually be delivered to all subscribers, there are no ordering requirements on these messages (more details will be provided in section 2.2.1).

1.2.1 Practical Example - Bank Service

Let's take a look at a simple example to highlight how systems benefit from solutions that only route messages to specific receivers and support operations that require different delivery guarantees, particularly from Ginger.

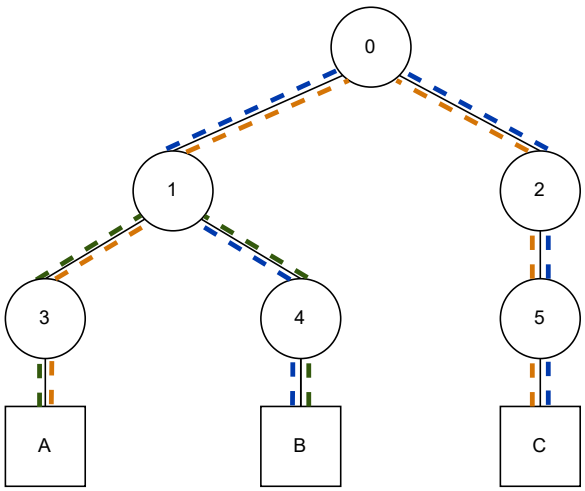


Figure 1.2: Topic sub-trees. The orange line represents the routing tree for topic *A* and the green line for topic *B* and blue for topic *C*

Consider a scenario where a bank is developing a banking service with the operations presented in Table 1.1. The bank's operation is supported by three data stores located at sites *A*, *B* and *C*. To ensure reliability, the system administrators replicate data from each location to one of the other two. For instance, accounts at *A* are also replicated at *B*, those from *B* are replicated at *C*, and finally, accounts from *C* are replicated at *A*.

In this context, each location (country) corresponds to a topic, with this value easily determined from the account's bank identifier, such as it is from an *IBAN* (International Bank Account Number). Once the location is extracted, the account number serves as the key that uniquely identifies the account within the respective data stores. Figure 1.2 illustrates the tree configuration and the per-topic trees created after the subscriptions have been issued.

For the MDS' perspective, when a client connected to *A* wants to perform an operation on their account, that operation is translated into a publication within the pub/sub system, which must be delivered to all subscribers of the *A* topic. For instance, if the client initiates a *deposit* in their account, the corresponding publication must be sent to both *A* and *B*. Subsequently, if the client wishes to verify their account balance and perform a *withdraw* based on the balance returned by the *getBalance* operation, both of these associated messages must also be delivered to both locations, similar to the earlier *deposit* operation.

Furthermore, besides delivering messages to all subscribers, Ginger also ensures the order in which these operations were performed is respected. According to Table 1.1, in the example depicted in Figure 1.3, the *deposit* operation must be delivered in causal order and *withdraw* in total order. Delivery in causal order must respect causal dependencies between messages (Section 2.2.1.2). Thus, since *deposit* does not depend on any other

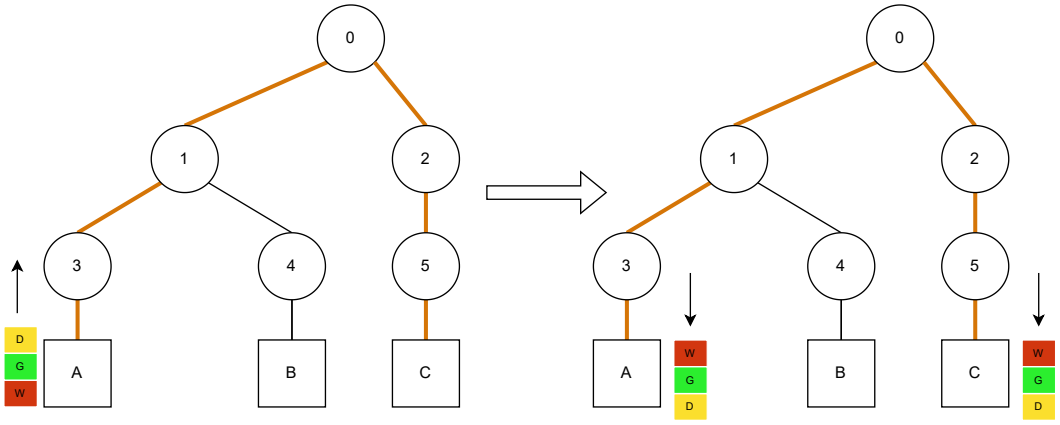


Figure 1.3: Ordered message delivery in Ginger.

operation, it can be delivered without any restriction. This is not the case for operation *withdraw*. Since the client executed *deposit* and *getBalance* before executing *withdraw*, the associated publications must be delivered to all clients in that order to satisfy total order. Thus, messages published before and after the message, in respect to the *withdraw* operation, must be delivered to all clients in the same order. Tree brokers are responsible for guaranteeing that order is respected.

1.3 Problem

In this thesis, we aim to address the challenges associated with fault tolerance in distributed pub/sub systems, specifically focusing on the support for multiple delivery guarantees. Ginger's MDS, a specific implementation of this type of system, currently lacks fault tolerance. As a result, in the event of failures, it is highly probable that it will be unable to maintain its functionality. Additionally, because it supports operations with multiple delivery guarantees, there is a also the risk that these delivery guarantees may be compromised during such failures.

In the case of distributed pub/sub systems built on top of dissemination trees, due to the inherent decoupling between publishers and subscribers (Section 2.3), if a node of the tree fails and becomes unavailable, the branches that derive from it and clients connected to those branches, will become unreachable. Therefore, we need to guarantee that, despite becoming unreachable through the overlay, clients still receive publications associated with their subscribed topics. Additionally to messages not being delivered to all clients in the event of failures, delivery guarantees may also be broken because client operations will only be executed by a subset of clients, leading to inconsistent client states.

Since Ginger's MDS also provides applications the ability to support operations that execute with different delivery guarantees, our problem is not only to guarantee messages are delivered to all clients subscribed to the corresponding topic, in the event of failures, but also to ensure messages are not delivered out of order, i.e., causal and total order must

be upheld as if a failure never occurred, guaranteeing causal and linear consistency.

In Section 2.4.1, we will show that several existing works tackle fault tolerance with different delivery guarantees. However, in contrast with our work, none of them address these issues in systems supporting multiple delivery guarantees.

1.4 Proposal

The goal of this thesis is to develop a fault-tolerant protocol, on top of the already existing dissemination mechanisms implemented by Ginger's pub/sub, able to ensure that when faults occur due to node or communication channel failures, operations can still be disseminated from its source to all destinations, guaranteeing that delivery respects the consistency level assigned to them. Therefore we propose to:

- Develop a tree maintenance protocol to ensure messages can be routed from all publishers to all subscribers.
- Develop a protocol to guarantee messages are not lost and are correctly ordered in the event of failures, with the goal of ensuring the multiple delivery guarantees are respected.
- Assess the developed protocols using different test configurations.

1.5 Contributions

The work developed in this thesis contributes with the implementation of a fault-tolerant pub/sub system with multiple delivery guarantees which, to the best of our knowledge, there is no similar system. Our solution allows systems like Ginger, to reliably deliver messages to clients while guaranteeing the multiple delivery guarantees are respected.

Our work makes the following key contributions:

1. We implemented fault-tolerant mechanisms in a pub/sub system with multiple delivery guarantees.
2. In the context of Ginger, our work contributes with the implementation of a protocol to repair the dissemination tree only using local information. This approach minimizes the number of nodes whose state changes after a topological reconfiguration and limits the load imposed on brokers affected by failures, after the overlay is restored.
3. Our work also contributes with a protocol to effectively restore communication channels between brokers, ensuring messages are not lost, message ordering is preserved and delivery guarantees are maintained.

4. Lastly, our work contributes with an evaluation methodology to assess correctness and performance in different tree configurations and varying communication latency.

1.6 Document Structure

The remainder of the document is structured as follows:

Chapter 2 introduces the concepts related to replication, data consistency, pub/sub systems, particularly reliable pub/sub systems where we highlight some of failures models and adopted strategies and techniques. Afterwards, we present some works related to reliability in distributed systems, particularly in pub/sub systems.

Chapter 3 provides an overview of Ginger's pub/sub system, highlighting how the infrastructure can be configured, how consistency levels translate to message ordering, each broker's state, existing application-level messages and, finally, how messages are processed.

Chapter 4 details the state and algorithms, implemented during this work, utilized by brokers to detect failures, reconnect the overlay and restore the communication channels between brokers, while maintaining the correct message order to ensure that client's databases' state converge.

Chapter 5 presents the methodology used to evaluate our system and an analysis of the tests made, with a focus in proving our solution can guarantee the properties provided by the pub/sub, in the presence of failures.

Chapter 6 highlights some of the conclusions we were able to take from our work and makes some suggestions for future work to further improve the pub/sub's functionalities and performance.

2

Background and Related Work

This chapter aims at covering the necessary concepts and techniques required to address the problem introduced in the previous chapter and present some approaches proposed by researchers to address it. In section 2.1, we describe different replication techniques regarding how and where data is stored and how updates to the data are propagated. Afterwards, in section 2.2, we introduce the concept of consistency models, highlighting the consistency guarantees provided by different models. In section 2.3 we describe the concept of publish/subscribe (pub/sub) systems, the advantage of having clients decoupled in multiple dimensions and the different existing subscription schemes. In Section 2.4, we outline the different kinds of faults that can occur in distributed systems, the requirements of reliable pub/sub and common strategies and techniques utilized to build resilient pub/sub systems. This chapter ends in Section 2.4.1, with the presentation of previous works with different to provide reliability guarantees in similar systems and a table summarizing them.

2.1 Replication

Replication is one of the cornerstones in the design of reliable distributed systems. It is a widely used technique to provide high availability, improved performance, linked to latency, the ability to tolerate individual node losses (fault-tolerance) and improve system's scalability [26]. There are several replication models and techniques one can choose when designing a replicated system. Each approach to the system design can provide different properties to it. Next, we present these techniques.

2.1.1 Total and Partial Replication

When thinking about data location and availability, one can reason about the total and partial replication models.

Total Replication is a model in which every instance of the database has a complete copy of the data, meaning that every storage node contains all the data available in the

system. When a database instance fails, it can be replaced by any other [26]. With this approach, one can achieve high data availability, since data is available in every database instance, faster query execution, especially when clients are scattered around the world by having multiple instances of the data closer to the clients, and improved load balancing, since queries can be executed in any replica. On the other hand, it may introduce issues related to data consistency and slower update process due to requiring that every database instance has a complete copy of the data.

Partial Replication is a model in which, in contrast to total replication, each replica only holds a subset of the database. This can be an interesting approach when storage space is a concern because updates are propagated only towards the affected sites [26]. However, certain sites are not able to execute certain types of transactions, which can affect the ability to load balance.

2.1.2 Active and Passive Replication

The concepts discussed so far, are mainly focused on addressing where data is being stored. However, it is also necessary to consider how data is replicated and how updates are propagated among the different replicas. The active and passive replication models provide two different methods of approaching this issue.

Active Replication is a method in which client operations are forwarded to all replicas in a coordinated order, relying on a protocol to decide the order of execution of such operations. Afterwards, each replica executes the operations independently, reaching a result state identical among all replicas. With this approach, to guarantee that replicas state after executing all operations is identical, client operations are required to be deterministic [19].

Passive Replication is a method in which all client requests are sent to a primary node, which executes the requests and sends update messages, containing the state of the master after executing the request, to the secondary nodes in the same order as the order of the operations executed by the master replica. Unlike active replication, client operations are not required to be deterministic due to the fact that only the master node receives and executes requests. On the other hand, this approach can consume a lot of bandwidth when the result state is large. Also, the primary node might fail and, thus, this method is required to have a replica recovery and master election protocol [19].

2.2 Consistency Models

A consistency model refers to the guarantees provided by the system about the order in which operations appear to occur to clients. It determines how data is accessed and updated across multiple nodes, and how these updates are made available to clients.

There are several consistency models, each with its advantages and disadvantages, and can be categorized in two categories: data-centric and client-centric [1, 5].

Data stores provide read and write operations over data items. When a process performs a read operation, it expects the return value to be the result of the last write operation performed on the given data item. However, in the context of distributed systems, it is not possible to have a global clock to synchronize operations and decide which write was performed last. Therefore, we need to provide alternatives that lead to a range of consistency models [27, Chapter 7].

2.2.1 Data-centric consistency models

Data-centric consistency models focus mainly on the synchronization processes among replicas and the ordering of operations, providing guarantees about the ordering of operations performed in all data items [3]. Next, we present consistency models that fit the data-centric consistency model, each providing different properties to the system.

2.2.1.1 Strong Consistency

Strong consistency models, guarantee that all replicas are in a consistent state immediately after an update, before the replica, who receives the request, replies to a client. They ensure that all replicas perceive the same order of data accesses performed by different clients [8].

Linearizability is a consistency model that provides strong consistency guarantees. In this model, operations have associated a time, called *linearization point*, which corresponds to the time a client operation takes effect [1], and is located some time between the operation's time of arrival and the time the process replies to the client. Furthermore, Linearizability requires client operations to be ordered based on their *linearization point* [3].

Sequential Consistency is another model that provides strong consistency guarantees. It requires that all requests are serialized by the same order in every replica and that requests from the same client are executed in the order that they are received by the storage system, i.e., the result of any execution is the same as if the operations by all processes in the data store were executed in some sequential order [27, Chapter 7]. It is weaker than Linearizability since it does not preserve real-time ordering of operations [2].

2.2.1.2 Weak Consistency

A weak consistency model is one that provides very weak or no guarantees on the ordering of operations, i.e., it can be seen as an approach where replicas might by chance become consistent [3].

Causal Consistency is a model where operations have a causal relationship between them, i.e. if an operation A is dependent on the output of an operation B, then operation B must be executed before operation A. For example, if operation B reads a resource and operation A updates the value of the resource based on the value read by B, then A must be executed after B because A depends on B. With this model, it is guaranteed that a client does not read two related write operations in a wrong order and in case the client has read the latest value, then it does not read the stale value [1].

Eventual Consistency is a form of weak consistency, thus it is not required to synchronize replicas as soon as clients send requests, allowing for reduced latency. After a period of time, in the absence of updates or failures in the system, eventually, replicas will converge to a consistent state [3]. However, since updates are performed asynchronously, if other clients read the data item from other replicas, it is not guaranteed that the operation will return the most recent write.

2.2.2 Client-centric Consistency Models

Client-centric consistency ensures that while a client has access to a resource, it will never see a state of the resource older than the current one. However, if multiple clients have access to the same resource simultaneously, then consistency is not guaranteed [1].

Monotonic Reads ensures that when a client reads a version n of a data item, future read operations on the same replica will always return a version of the data item equal or newer than n [3]. From an application perspective data visibility might not be instantaneous but versions become visible in chronological order.

Monotonic Writes is a consistency model in which a write operation performed by a process on a data item k, is completed before successive write operation on k by the same process. Therefore, a write operation on a copy of k is performed only if that copy has been brought up to date by means of any preceding write operation, which may have taken place on other copies of k. In case old writes have not finished, the new write must wait for old ones to finish [27, Chapter 7].

Read Your Writes consistency is provided by data stores, if the effects of a write operation by a process on data item k will always be seen by a successive read operation on k by the same process [27, Chapter 7]. This condition guarantees that a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

Writes Follow Reads consistency guarantees that any successive write operation by a process on a data item k will be performed on a copy of k that is up to date with the value most recently [27, Chapter 7]. A data store provides write follows reads consistency, if a

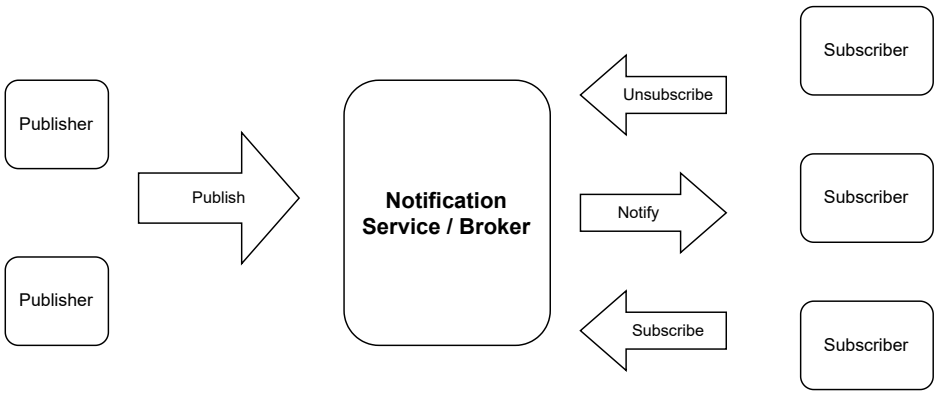


Figure 2.1: Publish/Subscribe architecture

write operation by a process on a data item k following a previous read operation on k by the same process is guaranteed to take place on the same or a more recent value of k that was read.

2.3 Publish/Subscribe Systems

A pub/sub system is one in which clients communicate asynchronously between them by exchanging notifications, relying on a notification service that provides storage and management of subscriptions, and efficient delivery of events [14]. In these systems, there are two types of clients, *producers* and *consumers*, more often called *publishers* and *subscribers*, respectively.

Producers publish events and consumers subscribe to events by issuing subscriptions. Consumers can have multiple active subscriptions and after issuing one, the notification service is responsible for delivering all future notifications when publishers publish an event, subscribed by the subscriber, until the subscription gets canceled. This allows consumers to express their interest in an event or pattern of events. Figure 2.1 shows the architecture of pub/sub systems with a single broker.

This kind of system is a great choice for applications focused on information dissemination, such as newsletters.

2.3.1 Notification Service

The pub/sub system architecture relies on a notification service, composed by a centralized or distributed broker server, responsible for storing publications, managing subscriptions and efficiently delivering notifications to subscribers [14]. It acts as a middleman between publishers and subscribers allowing these to be decoupled from each other. The decoupling provided can be decomposed into three dimensions [14]:

Space decoupling: publishers and subscribers are not required to know each other by relying on the notification service to receive publications from publishers and deliver

notifications to subscribers.

Time decoupling: publishers and subscribers do not need to actively participate in the interaction at the same time. Publishers can publish events while subscribers are disconnected and subscribers can be notified of events while the original publisher of the event is disconnected.

Synchronization decoupling: publishers do not get blocked while publishing events and subscribers get notified asynchronously of an event while performing other concurrent tasks.

2.3.2 Subscription schemes

Subscribers are often more interested in a specific set of events rather than all events. Thus, subscriptions had to be adapted to allow subscribers to specify the set of events they were most interested, which lead to the creation of several subscription schemes. The most widely used subscription schemes are the following [14]:

Topic-based subscription: participants publish events and subscribe to individual topics, identified by keywords.

Content-based subscription: consumers subscribe to events by specifying filters to event's content. Filters are rules that must be met by the content of a publication to be notified by the notification service.

This thesis mainly focuses on a topic-based pub/sub system, therefore, we more detailed insight on this type of system.

In topic-based systems, messages are associated with topics and are selectively routed to destinations with matching interests. Topics are associated with channels, which are queues of messages maintained by brokers, and it's name is used to create the channel. After creation, every channel is uniquely identified by its name, also used as an argument for the functions *publish()* and *subscribe()*. Clients can subscribe to multiple topics and will receive future messages published on those topics [29, Chapter 7].

As stated in [14], the concept of topics is similar to the concept of groups in the context of group communication. In addition, subscriptions can be seen as joining a group, and publishing a message can be seen as broadcasting that message to the members of that group.

2.4 Reliable Publish/Subscribe Systems

Scaling a distributed system requires the addition of components to the system. However, the more components a system is composed of, the more complex it becomes and the probability of components failing also increases. In general, a *failure* occurs when a component of the system fails. According to [27, Chapter 8], failures can be classified as:

Table 2.1: Process fault models based on [28].

Model	Description
Crash failure model	A process permanently stops working
Omission failure model	A process fails to respond to incoming requests
Timing failure model	A process' response is outside a specified time interval
Response failure model	A process' response is incorrect
Byzantine failure model	A process might deviate from its behavior in an arbitrary way

Transient: faults that occur once and then disappear.

Intermittent: faults that occur, then vanishes by itself and reappears repeatedly.

Permanent: faults that continue to exist until the faulty component is replaced.

In order to develop a reliable system, it is important to make assumptions about how a process might fail. Therefore, it is of great importance to define fault models. Table 2.1 classify process failures that can be observed in systems, and faults that occur at the network level.

Detecting failures can seem to be a trivial task however, when dealing with widely scaled systems, it might present itself as a bigger challenge. Therefore, it is important to design robust systems and algorithms that can guarantee the correct behavior of correctly operating processes, in the presence of failures.

In the context of distributed pub/sub systems, it is often necessary to provide strong guarantees about the reliable delivery of information. This is mainly due to the decoupling between publishers and subscribers as mentioned in Section 2.3. Systems can rely, for example, on overlay networks of distributed brokers or group communication and reliable application-layer multicast, such as Scribe [6], to propagate messages between brokers and from publishers to subscribers [14].

Moreover, we should also take into account that brokers might eventually fail and, therefore, system's need to implement mechanisms to detect such failures and recover from them. According to [13], these are some of the techniques and strategies that have been adopted by current implementations:

Planning: estimate the reliability level of several paths from a source to a destination, and pick the one that exposes the highest reliability. This mainly aims at avoiding faults as much as possible.

Reconfiguration: adopt topological reconfiguration to recover connectivity in the forwarding tree after a node or link crashes. Aims at maintaining a consistent connectivity for the system.

Retransmissions: publishers store produced events so that subscribers can request retransmissions when losses are somehow detected.

Epidemic Algorithms: processes exchange at a random time its history of the received notifications with a randomly-chosen set of processes among the ones constituting the system. Subsequently, inconsistencies, i.e., message drops are detected by comparison and corrected through retransmissions.

Forward Error Correction: forward redundant data instead of using retransmissions. Receivers can reconstruct the original event even if some notifications have been dropped during delivery.

Broker Replication: replicates brokers in the notification service. Brokers' state is replicated to its neighbors so in case of a broker failure, it can be easily substituted without losing subscription consistency.

2.4.1 Existing Work

In this section, we present some related works concerning the reliability problem of pub/sub systems. Table 2.2 presents a summary of the solutions reviewed.

Scribe [6] is a scalable application-level multicast infrastructure built on top of Pastry [23], a peer-to-peer location and routing substrate, which forms a robust and self-organizing overlay network in the Internet. Pastry is also used to manage group creation and joining, and to build a per-group multicast tree. Scribe nodes can create, send messages to, and join groups.

In scribe, subscribing to topics leads to establishing a per-topic multicast tree, rooted at the topic's *rendez-vous* point. A topic's *rendez-vous* point, is the node with *nodeId* closer to *topicId*. Nodes subscribe to topics by using Pastry to route JOIN messages.

Regarding reliability, Scribe mostly relies on Pastry to be resilient to node failures and uses TCP to reliably disseminate messages in the presence of unreliable links. In the multicast tree, each parent periodically sends heartbeat messages and children suspect their parent is faulty when they fail receiving heartbeat messages routing a JOIN message to a different parent. Furthermore, all nodes in the system maintain a buffer of received messages, containing the last publications published to the topic. This buffer is used by parents to retransmit messages the new child has missed while detecting and recovering from the old parent's fault. Children also send period messages to parents in order to restate their interest in a topic, otherwise their entry in the forwarding table is discarded. To tolerate *rendez-vous* point's faults, its state is replicated across the k closest nodes to the root in the *nodeId* space. If it fails, the node with the closest *nodeId* to the *topicId* will become the new root. When children detect that their parent has failed, they will be rerouted to the new root.

Epidemic Algorithms for Reliable Content-Based Publish-Subscribe [10, 11] In [10, 11], the authors approach reliability by relying on gossip (or epidemic) algorithms to improve reliability in content-based pub/sub systems and propose three algorithms based on different gossip strategies.

Unlike topic-based pub/sub where topic subscriptions define a group of receivers and messages within a topic can be assigned a sequence number to easily detect losses, patterns in content-based pub/sub do not have the same effect. This is mainly linked to the fact that events can match multiple patterns resulting in routing being performed independently and messages not getting assigned a sequence number. The authors identify the previous problems as the main challenges that have to be addressed when applying reliability to content-based pub/sub.

The first proposed algorithm, uses a strategy based on *push* gossip. At each gossip round, the gossiper randomly picks a pattern p from its subscription table, constructs a digest, a pair containing the source identifier and a sequence number associated top the source, of the identifiers of all the cached events matching p, builds a gossip message containing the digest, labels it with p and propagates the message. Afterwards, when a dispatcher receives a message labeled with p, it checks if it is subscribed to this pattern and if the identifiers contained in the digest correspond to events already received by it. The identifiers of missed events are included in a request message sent to the gossiper, which replies by sending the corresponding events.

The second and third algorithms use a strategy based on *pull* gossip. The former is a *subscriber-based* algorithm in which, when a dispatcher detects a lost event it inserts the corresponding information in a buffer *Lost*. In the next gossip round, it chooses a pattern p among the ones associated with local subscriptions, selects the events in *Lost* related with it, and inserts the corresponding information in a digest attached to a new gossip message. The message is then labelled with p and routed similarly to the *push* solution. Afterwards, when a dispatcher receives the message, it checks its cache against the events requested by the gossiper and, if any are found, sends them back to it. The last solution is a *publisher-based* algorithm. It requires that event sources also cache their published events and that the address of each dispatcher on the path towards a subscriber is appended to the message. It behaves similarly to the previous algorithm, but instead routes gossip messages towards publishers instead of subscribers. The messages are distinguished based on the event source rather than the pattern, and augmented with the information required to route back to the publisher.

Reliable and Highly Available Distributed Publish/Subscribe Service [17] In [17], the authors present a system that uses a topology management and subscription propagation scheme that enables brokers to compute new forwarding paths when failures of nearby neighbors occur.

The system relies on a tree-based overlay, in which brokers maintain a partial view of this tree which includes all brokers within distance $f+1$, where f is the maximum number

of concurrent broker failures that the algorithm tolerates. Such information is stored locally by each broker in a data structure called the topology map.

Brokers also maintain a subscription table containing entries for subscriptions inserted into the system. Subscriptions contain a *from* field that points to another broker located *f+1* hops closer to the subscriber. In the event of failure of one or more neighbors, the broker uses this information and its topology map to reconnect the network topology, and forward publications over new paths towards matching subscribers.

Finally, the algorithm contains a recovery procedure used by failed brokers to re-enter the system. This procedure aims at restoring a broker's routing information according to the current state of the network and is divided in three phases: *synchronization*, *message forwarding* and *termination of recovery*.

Thicket [15] is a decentralized algorithm that builds and maintains multiple trees over a single unstructured overlay network which, in contrast to structured overlays, are more resilient to the dynamics of nodes joining and leaving the system, due to the minimal restrictions imposed by the overlay. Furthermore, the overlay implements a reactive peer sampling service responsible for notifying the Thicket layer whenever there are changes to the node's partial view.

The algorithm employs a gossip strategy to build T divergent spanning trees, where T is the maximum number of trees that can be built in a single overlay, limited by the protocol's *fanout (t)*, and where most nodes are interior in a single tree and leaf in all other trees. The remaining overlay links are used to ensure that all nodes in the system are connected to all trees, detect and recover from tree partitions when nodes fail, ensure that tree heights remain small and ensure that forwarding load of each node is limited by a parameter called *maxLoad*, which must be low enough in order to limit the forwarding load imposed to each node, avoiding overloading.

Nodes also maintain a set of *backupPeers* with the identifiers of the neighbors that are not being used to receive or forward messages in any of the T trees, a set of *activePeers* with the identifiers of neighbors from whom it receives and forwards messages to, a set of *announcements* that store control information, used to detect and recover from tree partitions due to node failures or departures, received by peers in the *backupPeers* set and, finally, a *receivedMessages* set with the identifiers of messages previously received and forwarded by a node.

Thicket implements a tree repair mechanism with the purpose of ensuring that all nodes eventually become connected to all existing spanning trees and recover from tree partitions that might happen due to failure of nodes. It relies on a *SUMMARY* message, containing the identifiers of messages added to the *receivedMessages* set, since the last *SUMMARY* message. When a node receives such message, it verifies if all identifiers are recorded in its *receivedMessages* set. If no messages have been missed, the message is discarded. Otherwise, the node stores in the *announcements* set, a tuple containing the message identifier and the sender of the summary. For each tree t where a message has

been detected has missing, the algorithm initiates a *repair timer*. When the timer expires, if the missing messages have not been received, the node assumes that t has become disconnected from that tree and executes a procedure to repair it. Multiple executions of the repair mechanism may lead to configurations where several nodes are interior in more than one tree. Therefore, Thicket implements a reconfiguration procedure in order to undo such configuration and redistribute the load among all nodes.

P2S [7] is a topic-based and crash-tolerant *Paxos*-based pub/sub middleware based on Goxos, a *Paxos*-based State Machine Replication framework. Paxos is a fault-tolerant consensus protocol in which a set of replicas tries to reach agreement on a value. With this protocol, replicas can reach agreement when at least *f+1* replicas are able to communicate, where f is the number of failures the system can tolerate.

This system leverages Paxos as a way to achieve total message ordering, while providing a mechanism to tolerate broker failures. For that purpose, P2S relies on a set of *2f+1* brokers between publishers and subscribers. With this design, P2S is able to provide fault-tolerance through replication while providing total ordering through Paxos.

In this system, client messages are handled by the Goxos framework. When clients send messages, Goxos treats them as Paxos requests, orders and delivers them to the broker application layer, which forwards the messages to the subscribers according to the message topic.

GEPS [24] is a content-based pub/sub system in which publishers, before publishing, advertise their intent of publishing a message. Advertisements are flooded in the overlay so that all brokers become aware of all advertisements in the system. When a broker receives an advertisement, it stores the advertisement and the broker it came from, in the subscription routing table (SRT). An advertisement contains a type and a set of predicates defined over the attributes of the messages that the advertiser intends to publish.

When a client issues a subscription, it is propagated based on the SRT, in order to propagate the subscription towards the advertisers. Similarly to advertisements, subscriptions contain a type and a set of predicates over the attributes of the publications they are interested in, and each broker stores the subscription and the broker it came from.

In *GEPS*, the overlay tree is extended by establishing additional links between brokers, based on a broker's position in the tree and similarity with other brokers. Similarity between brokers quantifies the commonality of contents routed by two brokers.

Each broker maintains a partial view comprised of the brokers it is aware through the dissemination tree and established extra links. A node's partial view aims at maintaining information about its sibling nodes and it serves as a mechanism to verify the availability of view members and to discover more similar siblings. This can be accomplished by periodically updating it.

CPTCast [12] is a protocol able to build dissemination trees inspired by the tree construction mechanisms of *Plumtree* protocol, that provides causal delivery guarantees. It follows a gossip strategy, which inherently provides redundancy in the dissemination process and, even though processes expect to receive multiple copies of a messages, the protocol has to guarantee that each message is delivered just once by ensuring that each communication channel only carries each message once and messages are delivered in a First-In-First-Out (*FIFO*) order. This can be achieved by certifying that each node only forward messages when they receive it for the first time, which means nodes also require to store each message. Furthermore, when a node receives a repeated message, it acknowledges that the message has been received and transmitted by the sender and it can discard the data related to the message, once it acknowledges that all neighbors received the message.

Contrary to static systems with tree overlays, where guaranteeing *FIFO* delivery of messages is enough to guarantee causal delivery, in dynamic systems, where nodes can join and leave the system, leading to constant reconfiguration of neighborhoods, you also have to guarantee that nodes are able to receive the messages when reconfiguration happen.

The contributions made by this work, aim at developing a protocol able to provide causal delivery of messages under the conditions of dynamic systems, by employing techniques such as control message exchange and message buffering, to guarantee that new unsafe links are progressively integrated in the dissemination tree without compromising the consistency guarantees. Furthermore, to support the dissemination of messages under the process of reconfiguration of the whole tree, the protocol uses an epidemic strategy, dividing peers into *eagerPushPeers* and *lazyPushPeers*, separating each node's neighbors by communication mode. The broadcast tree is constituted by the links where eager push gossip is employed and lazy push links are leveraged as a strategy to achieve reliability by promoting these links to eager push when faults occur.

LoCaMu [25] is an algorithm built on top of an unstructured overlay network, modeled by an undirected, acyclic and connected graph referred to as *base graph*. To tolerate failures, the base graph is augmented with additional edges, creating an augmented graph. Even with redundant paths, messages are routed along the paths defined by the base graph, only using the redundant paths to propagate messages when nodes in the overlay fail. However, in those scenarios messages might be duplicated or re-ordered. Therefore, the algorithm maintains the required metadata to ensure the in order and reliable delivery of messages even when faults occur.

In LoCaMu, it is assumed that nodes can fail and subsequently recover, keeping their state in persistent memory. Faulty nodes may remain unavailable for an arbitrary amount of time. Furthermore, each node has access to a failure detector mechanism, that notifies nodes when neighbors become available or unavailable, allowing nodes to avoid sending messages to unavailable nodes. The recover mechanism is only used to improve the performance of the algorithm and avoids the periodic transmission of messages to

all nodes in its neighborhood, even to the ones that are temporarily unavailable. The algorithm also ensures that unavailable nodes will eventually receive all messages when they recover.

LoCaMu is a localized algorithm where nodes store metadata about the messages sent and received by other nodes into a *safety neighborhood*. When a message is sent and tagged with metadata, it only needs to be tagged with information belonging to nodes part of it's safety neighborhood of the recipient. Safety neighborhoods include all nodes that are at most *2f+1* hops away in the base graph. Nodes are required to keep track of identifiers of messages generated by neighbors in their safety neighborhood. The identifiers of messages originated outside the neighborhood cannot be used and, thus, it is necessary to replace it with a more local identifier. Instead of having a single identifier, messages are assigned with multiple identifiers as they are being forwarded through the network. Each of the identifiers is only valid in a given neighborhood.

Failure-tolerant overlay trees for large-scale dynamic Networks In [16], the authors propose a tree overlay maintenance protocol which utilizes different strategies with the goal of minimizing node degree after a node fails and a reconfiguration takes place In the proposed protocol nodes store information about other nodes in different caches. Additionally, each cache as an update mechanism their own advantages and disadvantages, according to the reconfiguration strategy utilized.

The different strategies proposed can either localize the reconfiguration mechanism to nodes surrounding a failed node or take a more global approach. The main caches, *Regional* and *Global*, store information about other nodes in the overlay and, therefore, require up to date information.

With that information, when a node detects the failure of its parent it can identify candidate parents and send a *ParentRequest* message to the one it picks. When nodes receive a parent request they can either accept or refuse depending on their own node degree, which is limited to a maximum threshold. If accepting a connection would exceed such limit, then nodes can refuse and the requesting node must chose another parent. The authors also propose strategies to soften the max degree restraint to limit scenarios where no candidate parent would accept a parent request.

2.4.2 Discussion

In this Chapter, we introduced the necessary concepts that allow the development of scalable distributed systems that can reliably provide services to their clients with high availability and performance. Finally, we presented an array of existing works that approach fault-tolerance in systems whose structure and consistency guarantees are similar to Ginger's. These papers served both as a support basis to identify failure situations that compromise Ginger's correctness and as an inspiration for the strategies we will employ to ensure it's correctness in such situations. We determined that to support

Table 2.2: Solution properties.

System	Topology	Delivery	Ordering	Fault-tolerance	Strategy	Details	Requirements
Scribe [6]	Tree	No guarantees	FIFO	✓	Reconfiguration	Routing Tables	Each topic has its own multicast tree rooted at a *rendez-vous*, which is the node with *nodeId* closer to *topicId*
P. Costa et al [10]/ [11]	No topology	At-least-once	No order	✓	Gossip	Message Buffering	Pull-based: Information related to lost events is inserted in a buffer and a request is propagated for the missing events. Dispatchers receive the message and verify if the requested event is cached and, in case they have, send a message to the requester. Push-based: Gossiper picks a pattern and propagates a message containing the identifiers of cached messages. Receivers verify if the identifiers correspond to messages already received and reply with a request message.
R. S. Kazemzadeh and H. Jacobsen [17]	Tree	Exactly-once	FIFO	✓	Redundant Paths	Partial view	Each broker maintains in its partial view all replicas within δ+1 distance, where δ is the number of faults to tolerate.
Thicket [15]	Tree	No guarantees	No order	✓	Reconfiguration	Reconfiguration	Nodes send periodic *SUMMARY* messages to it's neighbors to detect either if messages have been missed or neighbors have failed. Use backup peers when peers fail. Reconfigure tree to distribute load among nodes
P2S [7]	Star	No guarantees	Total	✓	Paxos	Replication	2f+1 replicas, where f is the number of faults to tolerate
GEPS [24]	Tree	Causal	FIFO	✓	Redundant Paths	Partial view	Brokers maintain and share a sibling view, based on the level of the tree and content subscribed by other brokers, with the parent and child brokers. Each broker also store the sibling's views of the parent and children.
CPTCast [12]	Tree	Causal	FIFO	✓	Gossip	Partial view	New links are marked as unsafe until every node recognizes it as safe. Partial view of links in eager push mode and lazy push mode. Links in lazy push mode are promoted to eager push when faults occur.
LoCaMu [25]	Tree	Causal	FIFO	✓	Redundant Paths	Partial View	Nodes maintain a "safety neighborhood" with peers that are at most *2f+1* hops away. Nodes store metadata related to messages sent and received by other nodes in that neighborhood. Identifiers of messages originated outside a node's neighborhood are not valid and are required to be replaced.
D. Frey and A. L. Murphy [16]	Tree	No guarantees	No order	✓	Reconfiguration	Partial View	Nodes maintain an *Ancestors* and *Siblings* chains. Additionally, nodes contain variables used to control their degree in order to determine which nodes might be suitable to accommodate others that have become disconnected from the remaining tree.

node failures in dissemination trees we can replicate brokers allowing any replica to replace it in case of failure, extend the tree with extra edges by adding additional edges between brokers not connected through the tree and use gossip strategies to guarantee the delivery of messages. To detect broker failures, brokers can exchange periodic heartbeat messages with its neighbors, which can be perceived as failed when a broker does not receive the periodic message from them. Similarly to detecting broker failures, recovering from them can also be achieved through the exchange of control messages.

The solutions we propose in Chapter 4 are mainly inspired by the approaches in [16] and LoCaMu [25] despite being a system whose overlay is unstructured, however some of the strategies used in this work can be applied in our work's context.

3

Ginger's Publish-Subscribe

This chapter describes the implementation of Ginger's publish/subscribe (pub/sub) system, detailing how the system's initial configuration is set, each broker's state and the types of application-level messages used to support the multiple delivery guarantees and, finally, how messages are processed to ensure each is correctly ordered.

Ginger uses two dissemination services to deliver messages to clients, the Data Dissemination Service (DDS) and the Metadata Dissemination Service (MDS), the second being the one this thesis focuses on. The MDS follows a topic-based pub/sub architecture built on top of a pre-existing dissemination tree of *brokers*. When a client executes an operation, it is mapped into a publication in the pub/sub and is required to be delivered to all subscribers of the topic the publication is associated with. The MDS is responsible of ensuring publications are delivered to all subscribers according to the consistency level associated with the operation performed by a client. Figure 3.1 illustrates the middleware's architecture.

3.1 Broker configuration and building the overlay

Brokers are the main component of the MDS's dissemination tree and are responsible for receiving, processing and propagating messages while guaranteeing they are delivered according to the consistency level they are labeled with. Brokers utilize configuration files to define a set of parameters required to build and maintain the overlay tree. Those parameters are:

self: indicates the broker's address in in the pub/sub system;

parent: indicates the broker parent's address;

nr-actors: indicates the number of workers a broker has;

During the initialization process, additionally to initializing the variables required to build the tree, *parent* and *self*, brokers initialize a pool of *workers*, whose number is determined by the *nr-actors* parameter, responsible for processing messages received by

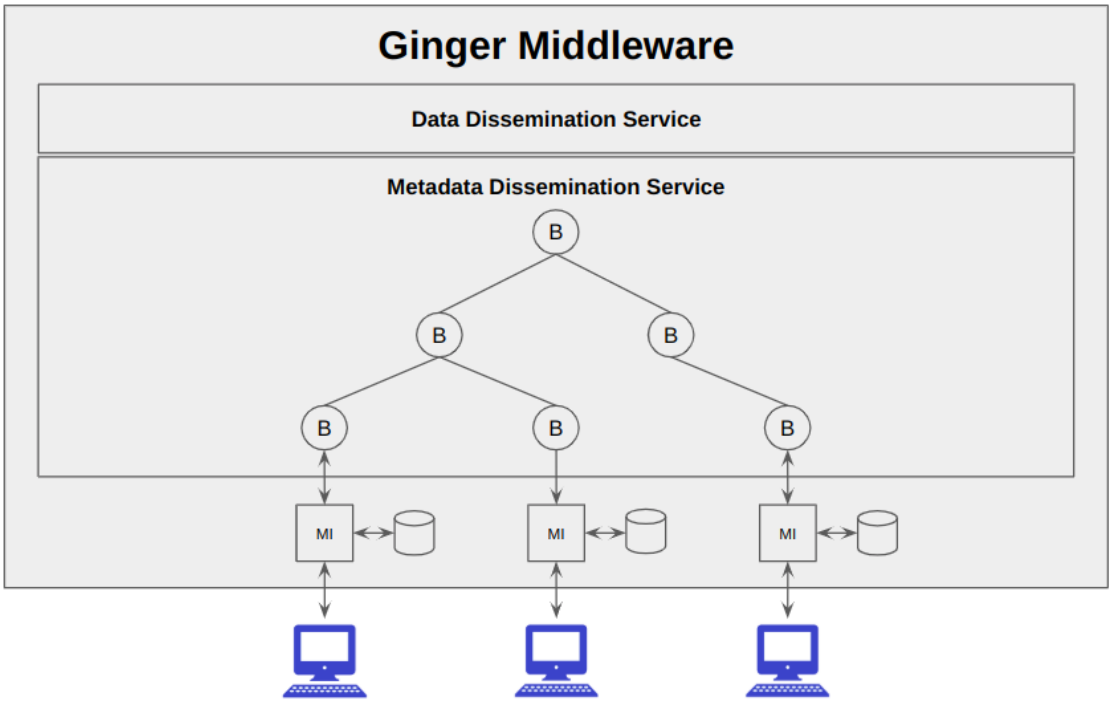


Figure 3.1: Middleware's architecture.

the corresponding broker. Further details on the remaining configuration parameters will be provided in Sections 4.2 and 4.3.

3.2 Consistency levels supported

Messages are processed and delivered according to the consistency level assigned to the publication. Ginger supports *Eventual*, *Causal* and *Linear* consistency levels, which in the context of the pub/sub are mapped into the *Eventual*, *Causal Order* and *Total Order* delivery guarantees, respectively. The following list describes the guarantees associated with each consistency level:

Eventual: brokers only need to ensure that these messages are eventually delivered to all clients subscribed to the topic.

Causal: messages are delivered in causal order, explained in Section 2.2.1.2;

Liner: messages are delivered to all subscriber in total order, i.e., for a given message delivered in total order to all subscribers, messages delivered before and after it are delivered to all subscribers before and after it, respectively;

3.3 Data structures maintained by each broker

Brokers maintain a set of data structures required to process and forward publications across the dissemination tree. The structures associated with the tree overlay and

dissemination mechanism are:

neighbors: stores the addresses of brokers who are directly connected.

parent: a reference to the parent broker's address.

brokerManagers: for each topic, stores an object responsible by processing and ordering incoming messages.

subscribers: addresses of subscribers per topic.

parentRef: an actor object from the Akka framework actor system that represents the parent broker. If the parent subscribes the broker, it references the parent actor in the system otherwise, it is null.

numberOfChildren: number of child brokers that subscribe a topic.

locked: maps key items to message identifiers; informs if a specific key item is blocked at the broker.

linearT: maps publication identifiers to metadata messages labeled with consistency level *linear*; used to store messages labeled as *linear*.

childrenAcks: stores *acknowledgments* received associated with a message identifier labeled as *linear*.

poolDown: queue that stores messages going down in the dissemination tree associated with a topic; allows brokers to process messages in a First-In-First-Out (*FIFO*) order.

poolUp: similar to *poolDown*, but for messages going up in the tree.

keyItemsDown: maps the key items from messages being processed and are going down the tree.

keyItemsUp: similar to *keyItemsUp* but for key items of messages that are going up the tree.

3.4 Message types

In Ginger's MDS, there are different types of messages:

Subscribe

$$Subscribe < topic, path >$$

As the MDS follows a topic-based pub/sub architecture, when a client is interested in receiving messages related to a topic, it sends a *Subscribe* message, to its edge broker. When a broker receives this type of message, it stores the sender's address in the subscribers set associated with the subscribed topic and forwards the subscription to its remaining neighbors. The propagation of this message allows the establishment of sub-trees associated with each topic, used to route messages relative to it. This message also contains a field named *Path* to store a set of the addresses of brokers who received and forwarded the message to its neighbors. The *path*'s relevance will be explained in Section 4.2.1

Generally, messages used to propagate operations have the following structure,

$$Message < topic, identifier >$$

A message contains information regarding the topic to which it is associated and an identifier that uniquely identifies the message. The *identifier* field is a tuple *(clientID, counter)*, where *clientID* identifies the client who performed the operation associated with the message and *counter* is a message counter maintained by the middleware instance (MI) who published it.

Metadata

$$Metadata < keyItems, deliveryGuarantee, processed >$$

Metadata messages, are used to propagate operations between service's clients. The *keyItems* field, contains the keys that uniquely identify data items or structures in a database and is a tool to support the ordering mechanisms. The *deliveryGuarantee* field, indicates the assigned delivery guarantees of the operation executed by the client and the dissemination mechanism utilized by brokers depends on it. Finally, the *processed* flag is used to indicate that a message has been processed, although it is only relevant for messages that are delivered in total order.

3.5 Processing messages

As mentioned previously, brokers maintain message queues that allow processing and forwarding messages in *FIFO* order. To deliver messages with multiple delivery guarantees, brokers must adapt the algorithm to process a message depending on the consistency requirements of the operation executed by the client.

Disseminating messages tagged with eventual consistency is the simpler of cases, since brokers are only required to ensure these messages are eventually delivered to all subscribers of the corresponding topic.

Disseminating messages tagged with causal consistency is also straightforward. Ginger ensures these messages are delivered in causal order, by relying on the message queues to process them in *FIFO* order which is enough to ensure causal order when no faults occur [25].

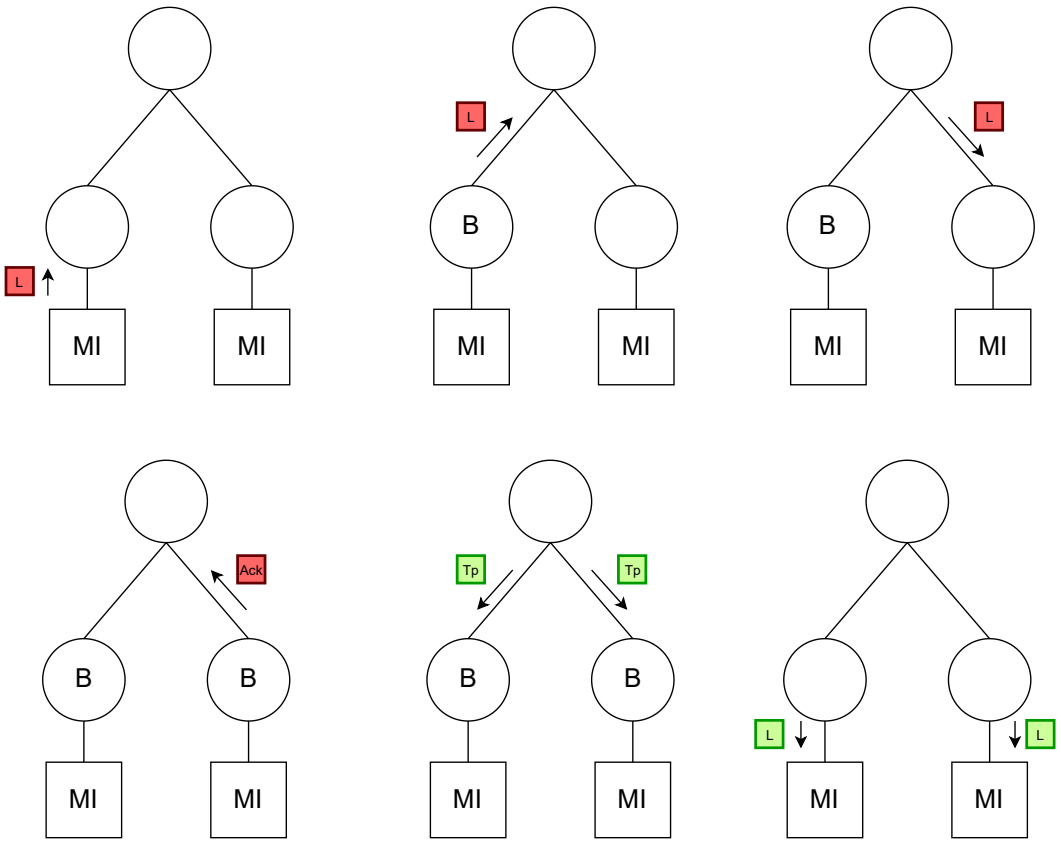


Figure 3.2: Total order delivery protocol.

These queues are also used to process messages tagged with linear consistency however, these messages require a more complex strategy. Total order delivery requires brokers to establish a synchronization point imposing strict ordering constraints on message ordering. For a given message with total ordering guarantees, any prior message received by a subscriber is guaranteed to be received before the message tagged with linear consistency by all subscribers, and any message received after the message by a subscriber is guaranteed to be received after the message by all subscribers. To ensure this, brokers temporarily stop processing messages that contain the same pair *(topic,keyItem)* as the message tagged with linear consistency, achieved by temporarily blocking the given pair at each broker subscribed to the topic when the message is received. Afterwards, if a broker is not the topic's tree root, it propagates the message to its parent, who then propagates the message to its children who are subscribed to the topic and waits for them to acknowledge the message. When all brokers in the topic's sub-tree acknowledge it, if the broker is not the topic's tree root, the process is repeated until it reaches the root. Afterwards, the root propagates a message to its children indicating that the message has been successfully processed and can be delivered to all subscribers. This process is illustrated in Figure 3.2. To support this process, the middleware implements the following messages:

TotalAck

$$TotalAck < keyItems >$$

is used by a broker to notify its parent that every broker below it in the tree as received the corresponding metadata message tagged with linear consistency and temporarily stopped processing messages with the same *keyItems*, which refers to the keys in the metadata message.

TotalProcessed

$$TotalProcessed < keyItems >$$

is a notification sent by the topic's tree root, to notify all brokers the message associated with the key items, has been successfully processed and can be delivered to all subscribers. As in *TotalAck* messages, the *keyItems* field refers to the keys in the corresponding metadata message.

4

Solution Overview

This chapter provides a detailed description of the work developed in this thesis, which considers a crash failure fault model, presented in Table 2.1 and, consequently, we only consider faults resulting from brokers failing and remaining in that state. We start by identifying the main requirements our work needs to fulfill in Section 4.1.

In Section 4.2, we present the first key component of our work, related to Ginger's Metadata Dissemination Service (MDS) dissemination infrastructure. The MDS works on top of a dissemination tree of brokers, and ensuring messages are delivered to all destinations is crucial. In order to deliver messages to all destinations, it is necessary to ensure that the dissemination infrastructure remains connected. Consequently, the first key component of our work is a protocol to maintain and reconnect the overlay in the event of failures. In Section 4.2.1, we show how a failure can compromise the connectivity of the system and the restrictions on message ordering imposed by the multiple delivery guarantees. From there on, until Section 4.2.4, we provide a thorough description of the details of the implemented protocol.

Then, starting in Section 4.3 we delve into the details of how brokers restore the communication channels between them and ensure no data is lost and message order in preserved in that process. These communication channels utilize a First-In-First-Out (*FIFO*) strategy to order messages, property that must be upheld after the overlay is repaired. Moreover, in the event of failures some messages may remain blocked, preventing other published messages from being delivered to all subscribers. The next sections, until Section 9, we detail the necessary steps to detect if messages were missed, to complete any ongoing processing of messages tagged with linear consistency, and how brokers prevent messages from being received out of order. Finally, we highlight how our protocol guarantees causal and total order, in Section 4.4.

During this chapter, whenever we want to refer to the broker's state in algorithms, we use the keyword *this*.

4.1 Identifying solution requirements

As mentioned in previous chapters, the main goal of this thesis is to implement a protocol that allows Ginger's middleware to maintain the overlay's connectivity and the correct message order in the event of failures. However, before discussing the details of this protocol, it is important to outline the MDS's properties that must be preserved:

Eventual delivery: messages published on a topic are required to be eventually delivered to every subscriber. For that purpose, we must ensure the paths between publishers and subscribers remain connected in the event of failures. Therefore, we need to ensure that the dissemination infrastructure remains continuously connected.

Causal order delivery: messages tagged with causal consistency must be delivered in causal order. In the absence of failures, delivering messages in *FIFO* order by relying in *FIFO* communication channels is enough to guarantee causal order [25]. However, we may not be able to guarantee the channels remain intact when failures do occur.

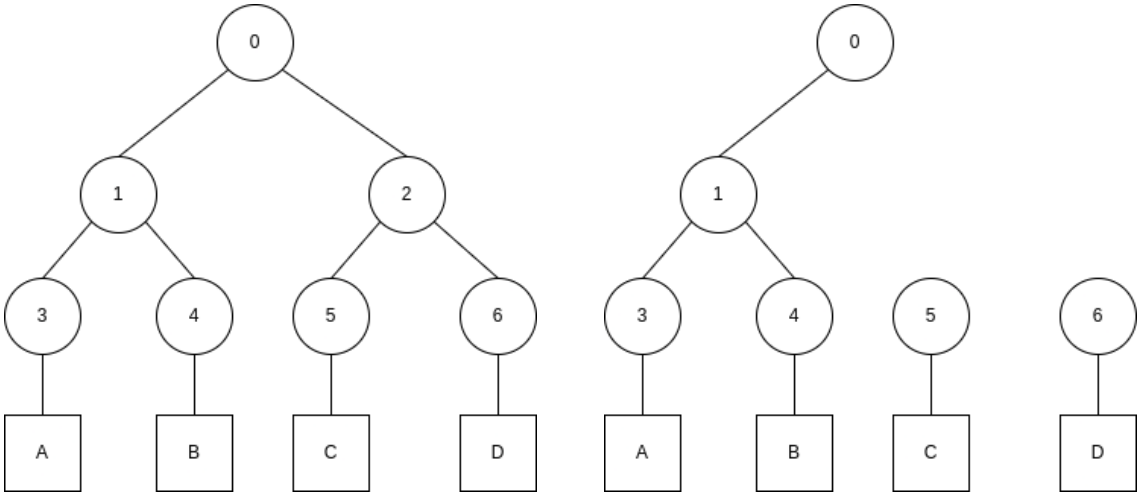
Total order delivery: messages tagged with linear consistency must be delivered in total order. Ginger ensures total order by blocking and unblocking message processing when the necessary conditions are met. If failures occur, brokers may wait indefinitely for such conditions to be met leading them to become unable to process further incoming messages.

4.2 Overlay maintenance protocol

The overlay maintenance protocol implemented in this work is derived from the maintenance strategies outlined in [16], with a particular emphasis on the *Regional* strategy. This approach is selected for its ability to minimize reconfiguration costs by confining reconfiguration to the failed broker's children, aiming at reconfiguring the tree while maintaining it balanced once the reconfiguration process is completed. The protocol is designed to be localized, meaning that brokers only need to communicate with brokers within their local tree view to reconnect the overlay. This approach is beneficial in large-scale systems, as it reduces the number of messages exchanged and the time required to reconfigure the overlay.

4.2.1 Consequences of a failure

In graph theory, a tree is defined as an undirected, connected and acyclic graph in which two vertices are connected by exactly one path. Consequently, if a non-leaf vertex is removed, the tree becomes disconnected. The same concept applies to a tree topology. When a node is removed, the tree becomes disconnected, creating multiple sub-trees where nodes are only able to communicate within their sub-tree. This is a relevant issue, because, in the context of a publish/subscribe (pub/sub) system, published messages



(a) Before failure (b) After node 2 fails

Figure 4.1: Example Ginger middleware overlay. Brokers are represented by circles and clients by squares.

might not be delivered to all of the topic's subscribers, if the topic sub-tree becomes disconnected.

Consider the tree from Figure 4.1 and assume every client subscribes to the same topics. In Figure 4.1a, every broker and client is reachable through the overlay. However, as displayed in Figure 4.1b, the tree becomes disconnected as a result of a failure. In the example, the failure of broker 2 creates three sub-trees each rooted at its direct neighbors. From the moment the failure occurs and onward, in the absence of a mechanism to restore the overlay's connectivity, messages published by clients in different sub-trees will not reach clients in other sub-trees, violating the properties provided by multiple delivery guarantees.

4.2.2 Expanding the neighborhood

To properly repair the tree, brokers cannot rely solely on the knowledge about direct neighbors. To reliably reconnect it, brokers are required to expand their knowledge of the tree, storing information about other brokers beyond their immediate neighbors in a *local tree view*. The local tree view, is composed of a broker's *ancestors*, *siblings* and neighbors. To store brokers in the local tree view, we defined two types of relationships between brokers, *ancestor* and *sibling*. An *ancestor* is defined as a broker with a lower depth in the tree and a *sibling* is a broker with the same parent. Since tree views are designed to be localizable, they only contain the addresses of brokers within a maximum distance of *maxHops*. This parameter is also related to the number of faults that can be tolerated by each broker and, since topic sub-trees need to be preserved, a broker can tolerate a maximum number of faults equal to the number of brokers within its tree view that subscribe to at least the same topics.

To address this, *Subscribe* messages transmitted through the network include information about brokers along the message's path, stored in a field called *path*. A message's path, *Path<pathEntries>*, consists of a set of path entries, *PathEntry<address, hopCount, direction>*. Each entry contains the address, the distance traveled, represented by the number of hops the message has taken since it was added to the path and a flag (*direction*) to indicate if the message was sent upward or downward at the corresponding hop. Information about the message's direction at each hop is relevant to determine relationships between brokers and is explained in the next section. Additionally, each broker stores the information provided by each entry in the path. The message's path also contains a *maxHops* field to indicate the maximum number of hops that each path entry can take before being removed from it.

Determining if a broker is a sibling. Determining if an entry corresponds to a *sibling* can be illustrated by considering the tree from Figure 4.1a. Consider the sub-tree composed by brokers 2, 5, 6. Nodes 5 and 6 are siblings because they have same parent broker. Messages sent between broker 5 and 6 follow the paths, $5 \rightarrow 2$ and $6 \rightarrow 2$, respectively. When messages move from 5 or 6 to broker 2, they travel from child to parent, meaning they move upward in the tree. At broker 2, messages take a final hop to reach each other, either $2 \rightarrow 5$ or $2 \rightarrow 6$, adding up to an hop count of 2 with the initial move being upwards. This movement pattern and hop count allows a broker to conclude an entry correspond to a sibling.

Determining if a broker is an ancestor. Let's make a similar argument on how to determine if an entry corresponds to an *ancestor*. Now, consider the sub-tree composed by brokers 0, 1, 2, 3 and 4. In this tree, brokers 0 and 2 are the *ancestors* of brokers 3 and 4 because their depth in the tree is lower than 3 and 4. To identify if a broker's depth in the tree is lower than other broker, we can once again look at the movement patterns of messages. Messages from 0 reach brokers 3 and 4 by taking two hops downward. Messages from 2 reach brokers 3 and 4 by taking an upward hop, followed by two downward hops. Therefore, based on the messages' movement patterns, a broker can be identified as an ancestor if the message was sent downwards or if the message takes one upward hop and the message follows a downward path onward.

4.2.2.1 Storing path information

Algorithm 1 outlines how information contained in the message's path is handled and stored in the *local tree view*. It starts by calling the procedure *incrementPathEntries* (line 2) that increments each entries' *hopCount* (line 18). Afterwards verifies whether the message's sender is the broker's parent (line 3). This verification ensures that brokers only store information about other brokers reachable through their parent, preventing them from attempting to connect to a broker in the event of failures. If the condition is satisfied, the algorithm iterates over each entry in the path (lines 5-15) and, after each

Algorithm 1: Handling and storing path information

```
1  procedure handlePath(sender , path):
       /* increment each entry's hop count                                  */
2      incrementPathEntries(path)
3      if sender = parent ∨ sender = ⊥ then
4          previousEntry ← ⊥
5          foreach entry ∈ path.getPathEntries() do
6              broker ← entry.address
7              hops ← entry.hopCount
8              direction ← entry.direction
9              if hops ≤ maxHops then
10                 if ¬ containsSibling(entry) ∧ ¬ containsAncestor(entry) ∧ address ∉ neighbors
                     ∧ address ≠ self then
11                     if hops = 2 ∧¬ direction then is sibling
12                         siblings.add(address)
13                     else
14                         if direction ∨ (¬ direction ∧ previousEntry ≠ ⊥∧
                             previousEntry.direction) then is ancestor
15                             ancestors.add(address)
16             previousEntry ← entry
```

iteration, tracks the last entry handled which may be important in the next iteration. It starts by checking if the address is not contained in the *ancestors* set, nor in the *siblings* set, nor in the direct neighbors set and if it is not its own address (line 9), extracting the entry's information if the conditions are met (lines 6-8). Then it verifies if the message was sent upward at the corresponding hop (*direction* is set to *false*) and made two hops in the tree (*hopCount* equals 2) (line 10), recognizing it as a *sibling* and storing it in the appropriate set. Otherwise, if the message was sent downward (*direction* is set to *true*) or if the message was sent upward and the previous entry contained the address of a broker who sent the message downward, then it is recognized as an *ancestor* and stored accordingly (lines 13-14). Finally, it updates the entry previously handled (line 15).

Clients must also be prepared for the possibility of the broker they are connected to failing. To ensure connectivity is preserved, clients also store information about brokers in a tree view. The key difference is that they are not required to organize them according to their position and relationship in the tree however, they must order the addresses in the view based on the number of hops contained in each path entry, prioritizing the selection brokers closer to them. This approach simplifies the view's management process for clients while still ensuring they can maintain connectivity when necessary. Since clients do not receive *Subscribe* messages, their tree view is updated when a

$$TreeViewUpdate < other, nChildren, failed, path >$$

message is received, as a result of a broker establishing a new connection or a broker failing. This message contains the address of the broker who failed or the connection was

Algorithm 2: Handling TreeViewUpdate

```
1  upon onTreeViewUpdate(update):
2      if update.failed then fault triggered the update
3          removeFromTreeView(address)
4      if update.sender ≠ this.parent ∧ update.sender ∈ neighbors then
5          this.neighborNumChildren[update.sender] ← update.nChildren
6      update.path ← handlePath(update.path)
7      if this.parent = ⊥ then
8          update.nChildren ← neighbors.size
9      else
10         update.nChildren ← neighbors.size - 1
11     foreach neighbor ∈ this.neighbors do
12         if neighbor ≠ update.sender ∧ neighbor ≠ this.parent then
13             send(neighbor, update)
14 procedure removeFromTreeView(address):
15     if ∃ ancestor ∈ ancestors: ancestor.address = update.other then
16         ancestors.remove(update.other)
17     else
18         if ∃ sibling ∈ siblings: sibling.address = update.other then
19             siblings.remove(update.other)
```

established to (*other*), a field indicating whether the event that triggered the update was a failure or a new connection (*failed*), the sender's number of children (*nChildren*) and the message's *path*, and is utilized to update local tree views. The path component includes a *PathEntry* for each broker in the sender's *local tree view*.

Updating the tree view is extremely important to prevent brokers and clients from attempting to repair the tree by contacting brokers that have previously failed without them becoming aware of it. An update is triggered whenever a broker establishes a connection with another broker and when a broker fails. In the MDS, brokers communicate with each other by establishing TCP channels between, which are monitored by brokers on each end of a connection. When a fault on one end of the connection occurs, the other broker is notified of that occurrence triggering the update mechanism.

Algorithm 2 outlines the procedure executed by a broker to process a *TreeViewUpdate* message. Brokers start by verifying if the update was triggered due to a failure by checking if the *failed* attribute is set to *true* (lines 2-3). If the update was triggered by a failure and the failed broker was an *ancestor* or a *sibling*, the address is removed accordingly (lines 15-19). Afterwards, if the message was sent by a child, brokers update the *neighborNumChildren* map that stores the number of children of each of a broker's children (lines 4-5). The information about the number of children is used by brokers when one of its children fails in order to determine how many connections a broker must expect. Generally, a broker's number of children is equal to the size of the neighbors set minus one, because the it includes the parent's address. However, if the broker is the root of the tree and does not

have a parent, its number of children is equal to the neighbors set size. Then, brokers handle the information contained in the message's path, update the message's *nChildren* field, to equals their number of children and forward the message to the remaining children (lines 6-13). The message is not forwarded to its parent due to the information contained not being relevant from the receiver's parent perspective.

4.2.3 Handling neighbor failure

As mentioned in Section 4.2.2, faults are detected by monitoring TCP connections established between brokers and their neighbors, when a connection fails, the broker is notified of that event and, if the failed neighbor was its parent it must select a broker within its *local tree view* to reconnect the overlay with otherwise, it must wait for the failed neighbor's children attempting to establish a connection.

To initiate the repair protocol, brokers begin by removing all stored information associated with the failed neighbor (Algorithm 3), namely removing its address from the set of neighbors, the number of children it has, if it is was a child and a broker (Algorithm 3, lines 4-6) and from the set of subscribers of each topic subscribed by it (Algorithm 3, lines 7-8 and 11-22). To remove the failed neighbor from a topic's subscribers set, a broker verifies if the neighbor was subscribed to a given topic (Algorithm 3, line 12-13). Then, if the neighbor was a broker and corresponds to its parent within the topic's sub-tree, the parent's reference is set to an undefined broker otherwise, the number of child brokers subscribed to the topic is decremented (Algorithm 3, lines 14-18). On the other hand, if the neighbor was a client, it is removed from the set of clients subscribed to the topic (Algorithm 3, lines 20-21). As mentioned in Section 4.2.2, detecting a fault triggers an update to brokers' tree view, therefore, during this process brokers send *TreeViewUpdate* messages to the remaining neighbors, allowing them to update their tree views (Algorithm 3, line 9).

The next step depends on the relationship with the failed broker. If a broker was the failed broker's parent it only needs to wait for the failed broker's children to contact it in an attempt to repair the tree by establishing a connection (Algorithm 4, lines 18-22). To do so, the parent is required to know how many connections it must expect and how many it can establish. The number of connections it must expect can be determined by the number of children the failed broker had, information received when a tree view update is triggered. The other requirement is to know how many connections it can establish. In our protocol, brokers have an upper bound on the number of active connections that can be maintained, corresponding to the maximum number of addresses within the *neighbors* set and referred to as *totalDegree*. This parameter aims at avoiding the establishment of connections from all brokers within the sub-trees rooted at a each child in case of successive faults occurring, and maintaining a balanced tree after the tree is reconnected. With this information, brokers are only allowed to establish new connections if the number of active connections is less than *totalDegree*. In other words, new connections can be established if the difference between *totalDegree* and the current number of neighbors,

Algorithm 3: Removing neighbor information after it failing

```
1  function removeNeighbor(neighbor):
2      nChildren ← 0
3      this.neighbors.remove(neighbor)
4      if isBroker(neighbor) then
5          nChildren ← this.neighborNumChildren[neighbor]
6          this.neighborNumChildren.remove(neighbor)
7      foreach topic ∈ this.factory do
8          removeFromTopicData(neighbor, topic)
9      sendTreeViewUpdate(neighbor, true)
10     return nChildren
11 procedure removeFromTopicData(neighbor, topic):
12     failed ← this.subscribers[topic][neighbor]
13     if failed ≠ ⊥ then
14         if isBroker(neighbor) then
15             if failed ≠ parentRef[topic] then
16                 numberOfChildren[topic] ← numberOfChildren[topic] - 1
17             else
18                 parentRef[topic] ← ⊥
19         else
20             if isClient(neighbor) ∧ failed ∈ clients[topic] then
21                 clients[topic].remove(failed)
22         subscribers[topic].remove(failed)
23 procedure sendTreeViewUpdate(other, failed):
24     update ← ⊥
25     if this.parent = ⊥ then
26         update ← TreeViewUpdate(this.neighbors.size, other, failed, Path({}))
27     else
28         update ← TreeViewUpdate(this.neighbors.size -1, other, failed, Path({}))
29     if failed then
30         if other = parent then
31             update.path.addPathEntry(PathEntry(this.self, 0, false))
32             foreach neigbhor ∈ this.neighbors do
33                 send(neighbor, update)
34         else
35             update.path.addPathEntry(PathEntry(this.self, 0, true))
36             send(parent, update)
37     else
38         if other ≠ parent then
39             send(parent, update)
```

Algorithm 4: Handling neighbor failure

```
1  upon onNeighborDown(down):
2      failedNeighbor ← down.neighbor
3      nChildren ← removeNeighbor(failedNeighbor)
4      if failedNeighbor = parent then parent failed
5          if ancestors.size > 0 then
6              address ← sendParentRequest()
7              setConnectionTimer(address)
8          else
9              if this.siblings.size > 0 then
10                 if this.siblings.size ≥ this.residualDegree then
11                     setExpectedConnections(this.siblings.size)
12                 else
13                     setExpectedConnections(this.residualDegree)
14                 foreach sibling ∈ siblings do
15                     send(sibling, RootElection(self, residualDegree, node))
                   /* set a connection timer for an unknown broker          */
16                 setConnectionTimer(⊥)
17             parent ← ⊥
18     else child failed
19         if nChildren ≥ this.residualDegree then
20             setExpectedConnections(this.residualDegree)
21         else
22             setExpectedConnections(nChildren)
23         setConnectionTimer(⊥)
24 function sendParentRequest():
25     this.expectedConnections ← 1
26     (broker, pr) ← getReplacementParent(⊥, ParentRequest(getSubscribedTopics(), false))
27     send(broker, pr)
28     return broker
29 function getReplacementParent((broker, parentRequest)):
30     if (declinedBy = ⊥ ∨ declinedBy.size = 0) ∧ ancestors.size > 0 then
31         broker ← ancestors[0]
32     else
33         if ancestors.size > 0 ∧ ancestors ⊆ declinedBy then
34             broker ← {∃ ancestor ∈ ancestors: ancestor ∉ declinedBy}[0]
35         else
36             if siblings.size > 0 ∧ siblings ⊆ declinedBy then
37                 broker ← {∃ sibling ∈ siblings: sibling ∉ declinedBy}[0]
38     if broker = ⊥ then
39         this.declinedBy ← {}
40         return getReplacementParent((broker, ParentRequest(getSubscribedTopics(),
             true)))
41     return (broker, parentRequest)
42 procedure setExpectedConnections(amount):
43     if this.expectedConnections > 0 then
44         his.expectedConnections ← this.expectedConnections + amount
45     else
46         this.expectedConnections ← amount
```

referred to as *residualDegree*, is greater than 0. In practice, if the failed neighbor's number of children is less than *residualDegree*, the broker can establish connections with all the failed broker's children. Otherwise, it can only establish a number of connections equal to its *residualDegree* (Algorithm 4, lines 19-22). The number of expected connections is stored in the *expectedConnections* counter and decremented each time an expected connection is established, and is used to determine when all expected connections have been established. To set this counter, brokers call the *setExpectedConnections* procedure (Algorithm 4, lines 42-46) that adds a given amount to the *expectedConnections* counter, if the counter is greater than 0, otherwise, the counter is set to the given amount. Finally, the broker initializes a timer, to brokers from waiting indefinitely for connections in scenarios where any of the failed broker's children fail simultaneously. Once the timer expires, the broker executes the procedure detailed in Section 4.2.4. Timers are initialized with a reference to the address of the broker whose connection is expected. However, since the failed broker's parent is waiting for multiple connections to be established, the timer is initialized with a reference to an undefined broker, represented by the symbol $\perp$.

On the other hand, if the failed broker is a broker's parent then two scenarios can occur (Algorithm 4, lines 4-17). In the first one, if the *local tree view* contains addresses in the *ancestors* set, a broker selects an address within its tree view and attempts to establish a connection with the respective broker, by executing the *sendParentRequest* function (Algorithm 4 lines 24-28), where the function *getReplacementParent* selects an address within the tree view (Algorithm 4 lines 29-41), and sends a

$$ParentRequest < subscribedTopics, breakDegree >$$

message, containing a set with the identifiers of the topics subscribed by the broker, *subscribedTopics*, and a flag named *breakDegree* that when set to *true* forces a broker to accept a request, to the selected broker. A topic identifier is only included in the *subscribedTopics* set if the broker is not the topic's sub-tree root and if at least a child broker subscribes that topic. Afterwards the broker waits for a response to the request, indicating if the request was accepted or rejected.

To select a new parent, the *getReplacementParent* function starts by verifying if the set of ancestors is not empty and if a set referred to as *declinedBy* is empty. This set's purpose is to store the addresses of brokers within a broker's tree view to which a broker has sent a *ParentRequest* and rejected it. With this information, brokers only select addresses that are not contained in this set, preventing them from attempting to connect to a broker multiple times in succession.

If both conditions are satisfied, the function selects the first address in the set of ancestors. Otherwise, if the *ancestors* set is not empty and the *declinedBy* set is not empty and the *ancestors* set is a subset of the *declinedBy* set, the function selects the first address in the *ancestors* set that is not contained in the *declinedBy* set. Finally, if the *ancestors* set is empty, the function selects the first address in the *siblings* set that is not contained in the *declinedBy* set. If the *siblings* set is empty or the *declinedBy* set contains all the addresses in

Algorithm 5: Handling parent request

```
1  upon onParentRequest (parentRequest):
2      mySubscribedTopics ← getSubscribedTopics()/* computes the topics subscribed
         by a broker                                                          */
3      allSubscribed ← (∀ topic ∈ parentRequest.subscribedTopics: topic ∈
       mySubscribedTopics)
4      if ((this.residualDegree > 0 ∨ isClient(parentRequest.sender)) ∧ allSubscribed) ∨
       parentRequest.isBreakDegree() then
5          acceptParentRequest(parentRequest)
6      else
7          send(parentRequest.sender, ParentResponse(false))
8          checkExpectedConnections()
9  procedure acceptParentRequest(req):
10     send(req.sender, ParentResponse(true))
       /* verify if the requester's address format fits a broker address
          format in the system                                              */
11     if isBroker(req.sender) then
12         connectToBroker(req.sender)
13         removeFromTreeView(req.sender)
14     else
15         connectToClient(req.sender)
16     addNeighbor(req.sender)
17     foreach topic ∈ req.subscribedTopics do
18         addToTopicData(req.sender,topic)
19     sendLastIds(requester, req.getSubscribedTopics())
20     this.ongoingReconfiguration ← this.ongoingReconfiguration + 1
21 procedure addToTopicData(neighbor,topic):
22     if neighbor ∉ this.subscribers[topic] then
23         this.subscribers[topic].add(broker)
24     if isBroker(neighbor) then
25         if neighbor = this.parent then
26             this.parentRef[topic] ← neighbor
27         else
28             this.numberOfChildren[topic] ← this.numberOfChildren+1
29     else
           /* verify if the neighbor's address format fits a client address
              format in the system                                          */
30         if isClient(neighbor) then
31             this.clients[topic].add(neighbor)
```

the *siblings* set, the function clears the *declinedBy* set and calls itself recursively with the *breakDegree* flag set to *true* to force the next selected broker to accept the next *ParentRequest* (Algorithm 4, lines 29-41). At the end of the procedure, the broker initializes a timer with a reference to the address of the broker selected by the *getReplacementParent* (Algorithm 4, line 7).

When the selected broker receives the request, it must verify whether it can accept it. If *residualDegree* is greater than 0 and it subscribes at least the same topics as the requester,

Algorithm 6: Handling parent response

```
1  upon onParentResponse(response):
2      if response.accepted then parent request accepted
3          this.declinedBy ← ⊥
4          this.parent ← response.sender
5          connectToBroker(response.sender)
6          addNeighbor(response.sender)
7          this.siblings ← {}
8          removeFromTreeView(parent)
9          this.expectedConnections ← this.expectedConnections-1
10         if this.expectedConnections = 0 then cancel the running timer
11             cancelTimer()
12     else parent request rejected
13         cancelTimer()
14         this.declinedBy.add(response.sender)
15         newParent ← sendParentRequest()
16         setConnectionTimer(newParent)
17 procedure addNeighbor(neighbor):
18     this.neighbors.add(neighbor)
19     this.residualDegree ← this.residualDegree + 1
20     if isBroker(neighbor) then
21         this.neighborNumChildren.put(neighbor, 1)
22     sendTreeViewUpdate(neighbor, false)
23 procedure checkExpectedConnections:
24     if this.expectedConnections > 0 then
25         this.expectedConnections ← this.expectedConnections - 1
26     if this.expectedConnections = 0 then
27         cancelTimer()
           /* cancels the running timer                                    */
28     if this.ongoingReconfiguration = 0 then
29         foreach topic ∈ this.getSubscribedTopics() do
30             verifyPendingTotalOperations(true, topic)
```

the request can be accepted (Algorithm 5, lines 4-5) and it answers to the with a

$$ParentResponse < accepted >$$

message containing a single flag named *accepted*, used to indicate if a *ParentRequest* is accepted, if set to *true*, or rejected.(Algorithm 5, line 10) If the request is accepted, a new connection can be established between both brokers (Algorithm 5, lines 11-15), and the requester's address is added to the neighbors set (Algorithm 6, line 16). Then the data associated with each topic is updated to include the new subscriber (Algorithm 6, lines 17-18). Afterwards, the *sendLastIds* procedure is executed, to trigger a synchronization process used to exchange information about the most recent messages that were processed (Algorithm 6, line 19), which will be described in Section 10. Finally, the broker increments the *ongoingReconfiguration* counter, used to track the number of ongoing reconfigurations i.e. the number of synchronizations in progress (Algorithm 6, lines 20).

Upon receiving a response to the request, the requester sets the parent's reference and adds the new parent's address its neighbors set and removes it from the tree view. The *siblings* set is also cleared as the position in the overlay changing, and the broker's *expectedConnections* counter is decremented (Algorithm 6, lines 2-11).

If the request is rejected, a broker must restart the protocol, however it can not select a broker within its tree view that has previously rejected a *ParentRequest*. For that purpose, the *ParentResponse* sender's address is added to the *declinedBy* set and another broker within the tree view not contained in the *declinedBy* set is selected. The selection of an ancestor is prioritized over a sibling if any is available, to avoid the broker's depth in the tree from increasing, leading to longer paths between publishers and subscribers and, consequently, increasing message latency. If no ancestor is available (Algorithm 6, lines 12-23) a sibling is selected. The broker then starts a timer to wait for the new connection to be established.

4.2.3.1 Handling edge broker failure

Edge brokers connect clients to the overlay. In the event of their failure, clients also need to find a broker to reconnect them to it, even if the broker was not previously an edge broker. Since a client has a child/parent relationship with their edge broker, in the event of the edge broker failing, the client must execute the same protocol to reconnect the overlay, just as brokers do.

4.2.3.2 Handling root failure

A particular scenario of faults is root failure, which differs from the general reconfiguration process due to root's children lacking any ancestors and, therefore, must work collectively to decide which among them will assume the root's responsibilities. In order to accomplish that, the root's children run a synchronization algorithm to determine which one takes on that role, triggered when detecting the root broker failed (Algorithm 4, lines 9-16). Each broker forwards to its siblings a *RootElection* message including information about the current *residualDegree* and the broker's identifier. Additionally, if the number of siblings is greater than its *residualDegree*, the *expectedConnections* counter is set to the broker's *residualDegree* otherwise, it is set to the number of siblings (Algorithm 4, lines 10-13). Then each broker starts a timer for an unknown broker, since lacking the knowledge of which broker will replace the root, and collect a set of these messages.

After receiving a *RootElection* message from all siblings, each broker can independently determine which of them will become the next root by utilizing a deterministic protocol, which does not required any further message exchange to select the new root. Algorithm 7 selects the next root based on the following selection criteria:

1. The broker with the highest *residualDegree* is chosen as the new root.

Algorithm 7: Root election algorithm

```
1  procedure onRootElection (re)
2      if this.siblingSync = ⊥ then
3          this.siblingSync ← {}
4      if this.siblingSync.size = this.siblings.size then
5          maxDegree ← this.residualDegree
6          maxNodeID ← this.node
7          foreach rootElection ∈ this.siblingSync do
8              siblingDegree ← re.degree
9              siblingNodeID ← re.nodeID
10             if siblingDegree > max then
11                 maxDegree ← siblingDegree
12                 maxNodeID ← siblingID
13                 this.parent ← re.sender
14             else
15                 if siblingDegree = max ∧ siblingNodeID < nodeID then
16                     maxNodeID ← siblingNodeID
17                     this.parent ← re.sender
18         if parent = ⊥ then
19             this.siblings ← {}
20             if this.siblings.size > 0 then
21                 if this.residualDegree ≥ this.siblings.size then
22                     this.expectedConnections ← this.siblings.size
23                 else
24                     this.expectedConnections ← this.residualDegree
25             else
                   /* cancels the running timer                          */
26                 cancelTimer()
27         else
28             send(this.parent, ParentRequest(self, getSubscribedTopics(), false))
29             removeSibling(this.parent)
30             this.expectedConnections ← 1
31             this.declinedBy ← {}
32         this.siblingSync ← ⊥
33         setConnectionTimer(this.parent)
```

2. If multiple siblings have *residualDegree* equal to the highest, the one with lower *nodeID* is selected.

Each broker starts by setting *maxDegree* equal to their *residualDegree* and *nodeID* to their identifier (lines 5-6). Afterwards, brokers iterate through each *RootElection* message to find the broker with *maxDegree* and the corresponding *maxNodeID* (lines 7-17). For each message, brokers start by extracting the necessary data into *siblingDegree* and *siblingNodeID*, and check if *siblingDegree* is greater than the current *maxDegree*. If it is, *siblingDegree* becomes the new *maxDegree*, *siblingNodeID* becomes the new *maxNodeID* and the sender's address becomes the new *parent*. Otherwise, if *siblingDegree* is equal to *maxDegree* and the current *maxNodeID* is greater than *siblingNodeID*, the latter becomes the new *siblingNodeID*

and the sender's address becomes the *parent*.

When the algorithm terminates, if a broker detects the parent reference has not been set or is *null* (lines 18-26), it will become the root and wait for connections from all siblings or from a subset of them, depending on its *residualDegree*. Other brokers upon realizing they are not the new root proceed as in the general reconfiguration process, sending a *ParentRequest* to the newly elected root, incrementing *expectedConnections* and initializing the *declinedBy* set (lines 28-31). Finally, each starts a connection timer (line 33).

4.2.4 Dealing with simultaneous failures

Algorithm 8: Handling expired timer

```
1  procedure onTimerExpired (timer):
2      if timer.broker ≠ ⊥ then
3          removeFromTreeView(timer.broker)
4          this.expectedConnection ← this.expectedConnections-1
           /* sendParentRequest returns the address of the broker selected to
              replace the parent                                              */
5          newParent ← sendParentRequest()
6          setConnectionTimer(newParent)
7      else
8          if this.siblingSync ≠ ⊥ ∧ this.siblingSync = this.siblings.size then
               /* set with the addresses of siblings who sent their message to
                  execute the RootElection algorithm                          */
9              siblingsUp ← {}
10             foreach rootElection ∈ this.siblingSync do
11                 siblingsUp.add(rootElection.sender)
12             foreach sibling ∈ this.siblings do
13                 if sibling ∉ siblingsUp then
14                     sibling.remove(sibling)
               /* execute RootElection algorithm                              */
15         else
16             this.expectedConnection ← this.expectedConnections-1
               /* verifies if the reconfiguration process is still ongoing or
                  has finished                                                */
17             checkOngoingReconfiguration()
```

In Section 4.2.3, we mentioned brokers utilize a counter named *expectedConnections* to determine whether whether all expected connections have been established. This approach has an issue because brokers might become unreachable due to brokers within the tree view failing simultaneously and, consequently, the reconfiguration process might never terminate, leading brokers to wait indefinitely for the remaining expected connections. We also mentioned that when a fault is detected brokers only wait for connection during a limited time interval, by initializing a timer whose value is defined in broker's configuration file. From the failed broker's children perspective, the timer is initialized with a reference to the broker selected to replace the parent by the function *getReplacementParent* from

Algorithm 4. From the parent's perspective, the timer is initialized to an undefined broker, due to the lack of knowledge about which brokers may attempt to connect to it.

When the timer expires, a broker executes the procedure outlined in Algorithm 8. It starts by verifying if the timer contains a reference to a specific broker. In that case (lines 3-6), it was attempting to establish a connection to another broker. Since the selected one did not respond to the *ParentRequest* or it accepted the request but failed to establish a connection, it's address is removed from the broker's tree view, preventing it from attempting another connection to the same broker that may lead to another timer expiration. Then it decrements the *expectedConnections* counter and attempts to connect to another broker, by executing the procedure as if the *ParentRequest* was rejected by the previous broker.

If the timer references an undefined broker, either means the broker was waiting for a *ParentRequest* (lines 16-17) or was waiting for *RootElection* messages in order to determine which broker is taking on the root's responsibilities (line 9). If the broker was waiting for a *ParentRequest*, it decrements the *expectedConnections* counter and verifies if the reconfiguration is still in progress. On the other hand, if the broker was waiting for the necessary messages to determine who becomes the next root, it removes from the *siblings* set, all siblings from whom it did not receive a *RootElection* message and terminates the election algorithm (lines 8-12). This approach is utilized due to the fact that we only consider a crash failure fault model.

4.3 Restoring communication channels and maintaining message order

After reconnecting the overlay, the next step is restoring the *FIFO* communication channels. Additionally, it is necessary to ensure brokers do not miss messages lost due to failures and any pending message processing can be completed. This phase also requires brokers involved in the reconfiguration process to temporarily stop processing and sending incoming messages to neighbors,preventing them from being processed prior to missed messages. Otherwise, if messages were missed, there is a high likelihood that clients will receive messages out of order, breaking message causality property and total order delivery. To effectively stop processing messages, when the *expectedConnections* and *ongoingReconfiguration* counters, used to know how many connections to expect and with how many other brokers a broker is synchronizing with, respectively, or when the *missedMessages* map, used to store the last identifier of missed messages a broker is expected to receive, is not empty, new incoming messages are stored in a per-topic queue referred to as *onHold*.

Algorithm 9: Holding incoming *Metadata* messages during an ongoing reconfiguration.

```
1  upon receiveMetadata (metadata, sender):
2      id ← metadata.id
3      topic ← metadata.topic
4      if this.missedMessages ≠ ⊥ ∧ sender = this.parent then
5          missed ← this.missingMessages[topic]
6          expected ← false
7          if id.clientId ∈ missed ∧ missed.messageIs ≥ id.messageId then
8              expected ← true
               /* queues message for processing                    */
9              send(brokerManagers[topic], metadata)
10             if this.missingMessages[topic][id.clientId].messageId = id.messageId then
11                 this.missingMessages[topic].remove(id.clientId)
12         if this.missingMessage[topic].size = 0 then
13             this.missingMessage.remove(topic)
14             if this.missingMessages.size = 0 then
15                 this.ongoingReconfiguration ← 0
16                 this.missingMessages ← ⊥
17         if ongoingReconfiguration = 0 ∧ expectedConnections = 0 then
18             foreach topic ∈ getSubscribedTopics() do
19                 verifyPendingTotalOperations(topic)
20             releaseOnHoldMessages()
21         else
22             if ¬expected then
23                 onHold[topic].add((metadata))
24     else
25         if this.ongoingReconfiguration > 0 ∨ this.expectedConnections > 0 ∨ this.siblingsSync
             ≠ ⊥ then
26             onHold[topic].add((metadata))
27 procedure releaseOnHoldMessages():
28     foreach topic ∈ getSubscribedTopics() do
29         if onHold[topic] ≠ ⊥ ∧ onHold[topic].size > 0 then
30             foreach message ∈ onHold[topic] do
31                 send(this.brokerManagers[topic], metadata)
32             onHold[topic] ← {}
```

4.3.1 Holding incoming messages during reconfiguration

To prevent messages from being processed and delivered out of order, whenever there is an incoming message and there is an ongoing reconfiguration (*ongoingReconfiguration* or *expectedConnections* counters are greater than zero) or the *missedMessages* map is not *null*, brokers stores it in the *onHold* queue if it is not waiting for that message.

Upon receiving a message, a broker checks if it is waiting for any missed message and if the received one, is one that it waiting for, by verifying if the *missedMessages* map is not empty and the message's sender was the new parent (Algorithm 9, line 4).

Otherwise, the message is put on hold (Algorithm 9, lines 24-26). In the first scenario, it needs to verify if the received message is one that was requested to its parent or not. For that purpose, it initializes a flag, referred to as *expected*, indicating if it was waiting for the incoming message's identifier (Algorithm 9, line 6). If *missedMessages* contains an entry for the message's *clientId* field and the entry's *messageID* is greater or equal to the message's *messageID*, it implies the broker was waiting for the received message. In that case, the *expected* flag is set to true and the message sent for processing (Algorithm 9, line 7-9). Additionally, if the message's *messageId* equals the identifier stored in *missedMessages*, all messages expected related to a *clientId* were received and, consequently, the entry is removed from the map and new messages related to that client are put on hold (Algorithm 9, line 10-11). Therefore, the *expected* flag is set to *true* and the message is queued for processing (Algorithm 9, lines 7-8). Afterwards, the entry is removed from the *missingMessages* map, if the incoming message's *messageId* is equal to the one stored in the entry associated with the client's identifier (Algorithm 9, lines 10-11). If the entry contained in the *missingMessages* map related to the message's topic becomes empty, the entry is removed from the map, and if it becomes empty, the *ongoingReconfiguration* counter is set to zero, indicating the broker is no longer waiting for missed messages, and the map set to *null*, represented by the symbol $\perp$ in the algorithm (Algorithm 9, lines 12-16).

At this point, if both the *ongoingReconfiguration* and *expectedConnections* counters are set to zero, the reconfiguration process is completed and brokers must verify if any messages tagged with total order delivery were being processed and the fault lead them to remain blocked, before releasing the messages being held. The algorithm to execute that verification and complete the total order delivery algorithm is detailed in Section 4.3.2.4. Otherwise, if *expected* flag is set to *false*, the message was not missed by the broker and must be put on hold (Algorithm 9, line 21-23). Furthermore, to release messages on hold, the broker iterates through each topic's *onHold* queue, and, if it is not empty, queues each message for processing, clearing it at the end (Algorithm 9, lines 27-32)

4.3.2 Detecting missed messages

To detect if messages were missed, brokers need to track the most recent processed *Metadata* messages in buffers. For that purpose, each broker keeps a per-topic message buffer (*metadataBuffer*) and stores all incoming *Metadata* messages after being processed. Since messages are disseminated in *FIFO* order through the communication channels, messages must be stored in the same order to accurately replicate the dissemination behavior, which is crucial to ensure missed messages are retransmitted in the correct order. As new messages arrive older ones become irrelevant and keeping them is not necessary therefore, buffers are queues with an upper bound on its size, allowing old messages to be removed as new ones arrive, thereby reducing memory usage as only most recent messages are kept. Buffer size is limited by the configurable parameter *bufferSize*,

which translates to the environment variable *bufferSize*. Configurable buffer sizes allow to adapt buffer's maximum size to the systems' and machines' requirements and available resources, respectively. The trade-off between small buffers and larger buffer is that, in the event of failures, the likelihood of missed messages being lost is greater if buffers are not big enough however, larger buffers might lead to increased memory consumption.

To detect if and which messages were missed, parents share information contained in their *metadataBuffer* with new children in order to compare it with their own message buffer and find which messages are contained in the parent's buffer but not in their own buffers. When comparing buffers, children simultaneously detect which messages have been missed by both. The use of buffers aim at avoiding storing version vectors due to requiring a version entry for each client, which is a solution that does not scale as the number of clients increases.

4.3.2.1 Synchronizing message buffers

Algorithm 10: Sending information contained in buffers to the child broker.

```
1 procedure sendLastIds(destination,subscribedTopics)
2     buffer ← {}
3     foreach topic ∈ subscribedTopics do
4         senderIsClient ← ∃ client ∈ this.clients[topic]: client = destination
          /* determines if the buffer is being sent to a client                */
5         foreach metadata ∈ this.metadataBuffer[topic] do
6             if ¬ senderIsClient ∨ (senderIsClient ∧ metadata.deliveryGuarantee ≠ TOTAL ∨
               (metadata.deliveryGuarantee = TOTAL ∧ metadata.processed)) then
                  /* put function replaces the value for the corresponding key
                     if the key exists in the map                               */
7                 lastIdsPerClient.put(metadata.id.clientId, metadata.id)
8         buffer.add(topic, lastIdsPerClient)
9     send(destination, ParentBuffer(buffer))
```

The protocol to restore a communication channel begins after accepting a *ParentRequest*. To trigger the protocol, after accepting a *ParentRequest*, the new parent sends to the new child a

$$ParentBuffer < map >$$

message containing information about the messages contained in its buffers. The *map* field contains for each topic subscribed by the new child, a mapping of the last message identifiers received from each client. The procedure to send the *ParentBuffer* message is outlined in Algorithm 10.

To map the last received identifier per topic and client, a broker iterates through its *metadataBuffer*, in *FIFO* order, and puts an entry *(clientId, messageIdentifier)* in the map, corresponding to the buffer's topic. If it finds multiple messages with the same *clientId*, it replaces the current mapped entry with a new one, containing a more recent

messageIdentifier. This ensures the most recent received identifier per client is mapped (Algorithm-10, line 7).

The content sent in the *buffer* field is also dependent on the new child's type. For instance, if the new child is a client, a broker cannot send the identifiers of messages, tagged with total order delivery, that have not been processed. Otherwise, in the synchronization protocol detailed in Section 4.3.2.2, messages may be detected as missing and might be delivered before being completely processed, ensured by the conditional statement in line 6 of Algorithm 10. Additionally, parents only send to a child its buffers associated with the child's subscribed topics, guaranteed by the argument *subscribedTopics*, preventing them from detecting messages from irrelevant topics as missing.

4.3.2.2 Handling parent buffer

Upon receiving the a *ParentBuffer* message, a broker initializes maps *missed*, *parentMissed* and *missedMessages* (Algorithm 11, lines 2-4). These store, for each topic, the identifier of the last message received per client, a set of the messages missed by the parent, and the identifier of last message processed by the parent per client, respectively.

First a broker adds the parent to the topic's subscribers set (Algorithm 11, line 6). Then, initializes an empty set in the *missed* and *parentMissed* maps for the topic being handled, stores a copy of the parent buffer's mappings for the current topic in *missedMapping*, utilized to remove mappings in the buffer determined by the broker as received, (Algorithm 11, lines 8-11). Afterwards, it initializes the *myLastIdPerClient* map, to map its last received message identifiers per client (Algorithm 11, line 12). Lastly, it initializes a set called *found* that stores the client identifier of the identifiers stored in the parent's mapping, when found within the broker's buffer.

The first step is to identify the last messages received by the broker within its buffer and messages missed by the parent, simultaneously. For that purpose, the broker iterates through its buffer and stores the current message's identifier in the *myLastIdPerClient* map (Algorithm-11, line 16). The *put* function adds to the map a *(key, value)* entry, where the key is the client identifier and the value is the message identifier. If there is already a mapping for the client identifier, the *put* function replaces the current value with a new one. Additionally, to identify messages missed by the parent while iterating through the buffer, if the broker finds within the buffer a message whose identifier is contained in the parent's mapping, by checking if the current message's identifier is equal to the identifier stored in the parent's mapping for the client identifier, it stores the client's identifier in the *found* set and removes it from *missedMapping* as it the broker did not miss that message (Algorithm 11, lines 17-20). That information is then used to find messages missed by the parent. Since messages are stored in *FIFO* order, messages stored at higher indexes imply they were processed more recently therefore, if at a given iteration, if the *found* set contains the current message's client identifier, the message is deemed as missed since it was more recently processed, and added to *parentMissed* (Algorithm 11, lines 21-23).

Algorithm 11: Handling parent buffers.

```
1  upon onParentBuffer(parentBuffer):
       /* map from topics to a stack of message identifiers                */
2      missed ← {}
3      parentMissed ← {}
4      this.missedMessages ← {}
5      foreach (topic, mapping) ∈ parentBuffers do
6          addToTopicData(parent, topic)
7          missed[topic] ← {}
8          parentMissed[topic] ← {}
9          missedMapping ← mapping
10         myBuffer ← this.metadataBuffer[topic]
           /* computes the ids of the messages in the buffer               */
11         myIdBuffer ← getBufferedIDs(this.metadataBuffer)
12         myLastIdPerClient ← {}
           /* stores the clientId of a message contained in the parent's
              mapping and found within the buffer                          */
13         found ← {}
14         foreach metadata ∈ myBuffer do
15             id ← metadata.id
16             myLastIdPerClient.put(id.clientId,id)
17             if id.clientId ∈ mapping then
18                 if id = mapping[id.clientId] then
19                     found.add(id.clientId)
20                     missedMapping.remove(clientId)
21                 else
22                     if id.clientId ∈ found then
23                         parentMissed[topic].add(Metadata(id, topic, this.self,
                             metadata.keyItems, metadata.processed,
                             metadata.deliveryGuarantee))
24         foreach (client,id) ∈ mapping do
25             if (clientId,id) ∉ myIdBuffer then
26                 missed[topic].put(clientId, id)
27         if missedMapping ≠ {} then
28             this.missedMessages[topic] ← missedMapping
29     send(parent, LostMessageRequest(parentMissed, missed))
30     if this.missedMessages ≠ {} then
31         this.missedMessage ← ⊥
32         this.ongoingReconfiguration ← this.ongoingReconfiguration - 1
33         if ongoingReconfiguration = 0 ∧ expectedConnections = 0 then
34             foreach topic ∈ getSubscribedTopics() do
35                 verifyPendingTotalOperations(topic)
36             releaseOnHoldMessages()
```

Afterwards, for each entry in the parent's mapping, if the broker's buffer does not contain the mapped identifier, the identifier is added to the *missed* (Algorithm 11, lines 24-26). Lastly, if *missedMapping* is not empty, the broker stores it within the *missedMessages* map (Algorithm 11, lines 27-28).

After completing the algorithm for each topic in the parent's buffer, both brokers and parents need to receive the missed messages. For that purpose, brokers utilize a *pull*-based strategy to request the missed messages and *push*-based strategy to retransmit the messages missed by the parent. To accomplish this, brokers send a

$$LostMessageRequest < missed, parentMissed >$$

which contains the *missed* map containing the identifiers of the last messages received by the broker per client and the *parentMissed* map containing the messages missed by the parent per topic (Algorithm 11, line 30).

At the end of the algorithm either the broker missed messages, which means the *missingMessages* map is not empty and must wait for them, or it is empty which implies it did not miss any messages (Algorithm 11, lines 30-36). In the second scenario, the broker decrements the *ongoingReconfiguration* counter and if it becomes zero, the reconfiguration has successfully been completed if brokers are not expecting further connections and there is no ongoing reconfiguration (*expectedConnections* and *ongoingReconfiguration* are equal to zero). If that condition is satisfied, before proceeding, brokers need to verify if any *Metadata* message tagged with total order delivery is blocked and the processing was interrupted due to the occurred fault. This is extremely important because that process must be finished otherwise, conflicting messages will not be allowed to be processed. The verification algorithm will be shown and detailed in Section 4.3.2.4. Finally, brokers release all messages that were put on hold during this period (Algorithm 11, line 36).

4.3.2.3 Handling a message request

Upon receiving a *LostMessageRequest*, the receiver must find the messages requested by the sender and retransmit them in *FIFO* order. For that purpose, it executes a similar procedure as the new child when identifying missed messages. For each entry in the *missed* map received in the message, the broker initializes a *found* set (Algorithm 12, line 3), whose purpose is the same of the *found* set in the procedure to identify missed messages, which is to store client identifiers if the buffer contains a message whose identifier is mapped. Therefore, if the *missedMapping* in line 2 contains an entry of the current message's client identifier, that identifier is stored in *found* (Algorithm 12, line 8). Similarly, to the previous procedure, if in the following iterations the current message's client identifier is within the *found* set, the message is retransmitted to the requester (Algorithm 12, lines 9-11). After retransmitting all missed messages, messages in the *parentMissed* map are queued for processing (Algorithm 12, lines 12-13). Finally, the broker verifies if the any reconfiguration is ongoing (Algorithm 12, lines 14).

Algorithm 12: Handling a *LostMessageRequest*

```
1  upon onLostMessageRequest(request):
2      foreach (topic,missedMapping) ∈ request.missed do
3          found ← {}
4          foreach metadata ∈ this.metadataBuffer[topic] do
5              identifier ← metadata.id
6              if id.clientId ∈ missedMapping then
7                  if missedMapping[id.clientId].messageId = id.messageId then
8                      found.add(id.clientId)
9                  else
10                     if id.clientId ∈ found then
11                         send(request.sender, metadata)
12     foreach (topic, messageList) ∈ request.parentMissed do
13         send(this.brokerManagers[topic], messageList)
14     checkOngoingReconfiguration()
```

What happens if any requested message is no longer contained in the parent's buffer? Although it is not yet implemented, if the parent's buffer does not contain a requested message, the parent raises an exception indicating to the client the operation was not executed, allowing the client to repeat the operation and every operation dependent on it. If the lost message required eventual delivery then, further messages from that client can be processed. If the message is required to be delivered in causal order or total order then, no further messages can be processed until the expected one arrives.

4.3.2.4 Finishing processing total order operations

Each broker involved in the reconfiguration process executes this algorithm once per subscribed topic. For clarity reasons and since the algorithm depends on the current stage of total order processing, we divided the algorithm into segments each corresponding to a different stage. Furthermore, this algorithm may send redundant messages to ensure processing can be completed and we will also show how brokers can detect and correctly process redundant messages. After finding the stage in which message processing is, the algorithms replicates the behavior of total order processing.

Processing Stage - Publication for a given pair *(topic,keyItems)* that is blocked and has received *TotalAck* from all its children When a broker receives a *Metadata* message to be delivered in total order from a neighbor, it blocks processing messages for the pair *(topic,keyItems)* contained in it. To do so brokers initialize a list destined to store *TotalAck* acknowledgments from its children. This list is stored in the *childrenAcks* map, which stores a *TotalAck* list for each *(topic, keyItems)*. Brokers use this map to track which children have acknowledged the corresponding message to be delivered in total order. After storing the acknowledgment in the list, the broker forwards the *Metadata* message

Algorithm 13: Algorithm to progress when a message is blocked and a broker has received a *TotalAck* from all its children

```
1  procedure verifyPendingTotalOperations(topic):
2      if this.childrenAcks[topic][id] ≠ ⊥ then
3          if this.childrenAcks[topic][id].size = this.numbreOfChildren[topic] ∨
             this.numberOfChildren[topic] = 0 then
4              if this.parentRef[topic] ∉ this.subscribers[topic] then
5                  this.childrenAcks[topic].remove(id)
6                  msg ← this.linearT[topic][id]
7                  msg.processed ← true
8                  this.processedTotalOrderMetadataIds[topic].add(msg)
9                  foreach subscriber ∈ this.subscribers[topic] do
10                     if subscriber ≠ this.parentRef[topic] then
11                         if subscriber ∈ this.clients[topic] then
12                             send(subscriber, msg)
13                         else
14                             send(subscriber, TotalProcessed(id, topic, msg.sender,
                                 msg.keyItems))
15             else
16                 if metadata.sender ≠ parentRef[topic] then
17                     send(parentRef[topic], metadata)
18                 else
19                     send(parentRef[topic], TotalAck(id, topic, metadata.keyItems))
20         else
21             this.childrenAcks[topic][id] ← {}
22             foreach subscriber ∈ this.subscribers[topic] do
23                 if subscriber ≠ this.parentRef[topic] ∧ subscriber ∉ this.clients[topic] then
24                     send(subscriber, metadata)
```

to the remaining neighbors and waits for them to acknowledge the message's reception by sending a *TotalAck*. When all children acknowledge the message then, which implies the corresponding list in the *childrenAcks* map contains an acknowledgment for each child, it either sends a *TotalAck* message to the parent or the *Metadata* message, depending if the message's sender was the parent or a child, respectively. Afterwards, brokers wait to receive a *TotalProcessed* notification from the parent in order to unblock the *(topic,keyItems)* pair and process further messages.

If the failure occurs when a broker finds itself in a scenario where it has received a *TotalAck* from all its children and it has not sent the corresponding *Metadata* or *TotalAck* to its parent, it needs to either send the *Metadata* message to the parent, if its sender was a child, or a *TotalAck* if its sender was the previous parent. To do so brokers execute Algorithm 13. Brokers start by verifying if the *childrenAck* map contains the list for the corresponding *(topic,identifier)* pair (line 2). If it contains that list, the broker verifies if the number of *TotalAck* contained in it is equal to the number of children subscribed to the topic, which is either zero or greater than zero (line 3). In that case, the broker needs to

either send an acknowledgment to the parent or the *Metadata* message. To do so, broker verify if the parent subscribes to the topic (line 4). If the parent subscribes to the topic (lines 17-20), brokers check if the message's sender was not the parent, sending the *Metadata* message to it in that situation (lines 17-18). Otherwise, brokers send a *TotalAck* to the parent. If the parent does not subscribe to the topic (lines 5-15), it implies the broker is the topic's tree root and it must notify its children the message has been processed. For that purpose, they send *TotalProcessed* notifications to all children brokers (lines 14-15) and the *Metadata* message to connected clients (lines 11-12), if any are connected.

Algorithm 14: Algorithm to progress from waiting for *TotalAck* from children but the *childrenAcks* map does not contain the list.

```
1  procedure verifyPendingTotalOperations(topic):
2      if metadata.sender ∈ this.neighbors ∧ metadata.sender ≠ parentRef[topic] ∧
         this.numberOfChildren[topic] then
3          this.childrenAcks[topic][id] ← {}
4          this.childrenAcks[topic][id].add(TotalAck(id, topic, metadata.sender,
             metadata.keyItems))
5          foreach subscriber ∈ this.subscribers[topic] do
6              if subscriber ≠ this.parentRef[topic] ∧ subscriber ∉ this.clients[topic] ∧ subscriber ≠
                 metadata.sender then
7                  send(subscriber, msg)
8      else
9          if metadata.sender ∉ this.neighbors ∧ this.parentRef[topic] = ⊥ then
10             this.childrenAcks[topic][id] ← {}
11             foreach subscriber ∈ this.subscribers[topic] do
12                 if subscriber ≠ this.parentRef[topic] ∧ subscriber ∉ this.clients[topic] ∧
                     subscriber ≠ metadata.sender then
13                     send(subscriber, msg)
```

Processing stage - Publication for a given pair *(topic,keyItems)* that is blocked and has not yet received *TotalAck* from all its children. In this stage, there are two possible scenarios. In the first, the broker received the *Metadata* message from one of its children but the list for the pair *(topic,keyItems)* does not exist (Algorithm 14, lines 2-9). The other possible scenario occurs when the broker performing this procedure becomes the topic's tree root due to the former failing, the message's sender was the former root and it had already acknowledged the message to its previous parent (lines 10-15).

A broker finds itself in the first situation by verifying if the message's original sender was one of its neighbors, other than its parent, and if any of its children subscribe to the topic (Algorithm 14, line 2). If the condition is met, it initializes a list to store acknowledgments, in the *childrenAcks* map, and adds a *TotalAck* to it (Algorithm 14, lines 3-4). In the standard processing algorithm the original *Metadata* message is handled as a *TotalAck* sent by the message's sender, therefore, this aims at replicating that behavior. Finally, the *Metadata* message is sent to every neighbor subscribed to the topic, with exception to the parent and

the message's sender. Afterwards, the broker waits for the acknowledgments allowing the standard processing mechanism to execute (Algorithm 14, lines 7-9).

Since this message may have already been received and acknowledged by some of the children and the corresponding *(topic,keyItems)* pair is already locked, when brokers receive a *Metadata* message and verify that the corresponding *(topic, keyItems)* pair is already blocked, they resend the acknowledgment to the parent.

If the condition for the first scenario is not met, the broker checks if the message's sender was not a direct neighbor, which implies the sender failed, and it either has become the topic's tree root or it already was (Algorithm 14, lines 11-12). To handle this scenario, the broker sends the *Metadata* message to all topic's subscribers and wait for the acknowledgments (Algorithm 14, lines 13-15). Once again, the broker waits to receive the necessary acknowledgments from its children and, afterwards, the standard protocol is executed.

4.4 Ensuring causal and total order delivery

We have presented the protocols to ensure messages are delivered to every destination and messages are not lost in the event of failures. We also detailed how our work guarantees that messages tagged with total order delivery can be unblocked if the occurred fault does not allow the processing to be completed. However, we haven't clarified how causal and total order delivery are guaranteed.

According to [25], to ensure causal order in acyclic overlays it is only required to ensure messages are propagated in *FIFO* order. In the absence of faults the pub/sub already propagates messages in that order therefore, we are only required to ensure lost messages are delivered in the same order and the overlay reconfiguration protocol does not create any cycles. During this Chapter, we highlighted that metadata buffers store *Metadata* messages in *FIFO* order and, in case any are identified as missing during the buffer synchronization phase, retransmitting them in *FIFO* order guarantees it is not violated. Furthermore, we guarantee the overlay remains acyclic by preventing brokers from connecting to any of its descendants.

And what about total order delivery? To guarantee total order delivery we do not need any more complex mechanisms. Consider that brokers were processing a message tagged with total order delivery for a given *(topic,keyItems)* pair. During that time, if a broker receives a message containing the same pair, the middleware guarantees it is not processed before the total order message's processing is finished, by temporarily blocking all messages with the same pair and storing them in queues. Only when processing is completed can the pair be unblocked and queued messages be processed. Our work replicates that behavior by holding messages received in a queue while a reconfiguration is in progress and by making sure any total order messages being processed is unblocked before releasing the messages received during that time, ensuring messages are not delivered out of order.

In conclusion, our protocol works side-by-side with the underlying dissemination mechanisms previously implemented. Therefore, besides maintaining *FIFO* ordering, we only need to guarantee that processing messages who require total order delivery and whose processing has not been finished before the fault occurring, is completed before resuming standard message dissemination.

4.5 Summary

In this Chapter, we presented the mechanisms implemented to provide Ginger with the support to tolerate faults.

We started the Chapter by identifying the requirements of our solution. Afterwards, we presented the first component of our work, which is a localized overlay maintenance protocol that reliably reconnects the overlay after detecting a broker has failed. Furthermore, it guarantees the tree remains balanced and the reconfiguration costs fall on brokers in logical proximity to the broker that failed.

Finally, we presented the second component of our work, which is a protocol that guarantees messages are not lost and its ordering is maintained in order to deliver messages according to the necessary delivery guarantees.

The following chapter presents the conducted experimental work.

5

Experimental Evaluation

5.1 Goals

Our work contributes with the development of a protocol to provide fault-tolerance capabilities in pub/sub systems with multiple delivery guarantees. With the goal of evaluating our protocol's validity, we established a set of test experiments under a baseline network configuration and varying parameters, with the purpose of simulating real-world scenarios. Our experiments focus on two key aspects: *correctness* and *performance*.

When evaluating our protocol's *correctness*, the goal is to ensure the pub/sub system is able to deliver all messages to the corresponding topic's subscribers according to the respective delivery guarantees, even in the presence of failures. This involves verifying if messages are consistently delivered according to the defined delivery guarantees and that they are neither lost nor duplicated during a failure. Additionally, we also must ensure the overlay reconfiguration protocol is performed correctly and broker's acquired knowledge, beyond the initial configuration, is correctly stored and updated.

In terms of *performance*, the evaluation aims at measuring the system's efficiency at restoring its normal functionality under different failure scenarios. These includes measuring the cost of performing topological adaptations, measured as latency of reconfiguration, in the event of failures. Topological adaptations occur from different perspectives, which we take into account in the different scenarios. Topological changes may also have a direct impact in the latency of messages that were already being processed before failures occurring, mainly due to brokers involved in the reconfiguration temporarily suspending message processing and holding incoming messages until the reconfiguration as terminated.

To carry out our experiments we defined a set of questions and metrics we want to answer and measure, to effectively evaluate our work.

5.1.1 Correctness

To evaluate correctness, we defined the following questions :

- Is the broker's view of the tree correctly updated, on brokers joining and leaving the overlay?
- In the presence of one or more faults, are the children's *ParentRequest* correctly handled with regard to the handler's current *residualDegree*?
- If a broker's *ParentRequest* is refused, is it able to find another broker to connect to?
- In the presence of one or more faults, are there any messages lost?
- Do any messages, to be delivered in total order, remain in the state they were before the fault and their processing finished afterwards?
- Can our protocol ensure the delivery guarantees are upheld?
- When the overlay tree's root fails, can one of its children replace it?

The metric used to assess these questions is *yes* or *no*, since our protocol either behaves correctly or not.

5.1.2 Performance

The following questions were used to evaluate the performance of our protocol, in the event of failures one or more failures:

- What is the cost of reconfiguration, measured in reconfiguration time, taking into account the number of rejected *ParentRequest*?
- How impactful is the reconfiguration process in messages, regarding their latency?
- How impactful is the number of keys items in message latency, in the presence of failures?

5.2 Methodology

To evaluate our work, we implemented a simple *BankService*, that simulates a banking service and provides the following operations:

getBalance(account): returns the current balance of *account*. Is eventually delivered to every client.

deposit(account, amount): adds *amount* to the *account*'s balance. Delivered in causal order.

withdraw(account, amount): removes *amount* from the *account*'s balance. Delivered in total order.

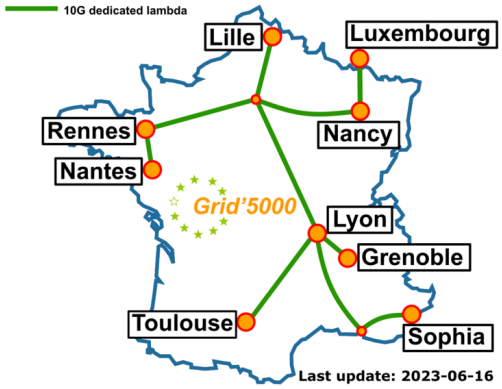


Figure 5.1: Grid'5000 backbone. Taken from Grid'5000's homepage.

Table 5.1: Specification of machine utilized during evaluation.

CPU	Memory	Storage	Network
1x 18 Core Intel Xeon Gold 5220	96 GB	1x480GB SSD + 1x960GB SSD	2x25Gbps

transfer(account1, account2, amount): removes *amount* from *account1*'s current balance and adds *amount* to *account2*'s current balance. Delivered in total order.

applyInterest(account): computes the *account*'s new balance, based on a fixed interest rate. Delivered in total order.

In our benchmark, a topic corresponds to a *BankService* whose operation is supported by multiple data stores replicated at different sites. To replicate service's data, each site issues a subscription in the pub/sub to the respective topic. In each service, each bank account's number serves as the key that uniquely identifies the account within the data stores. This service was used to evaluate both the correctness and performance of our work.

To carry out our experiments, we relied on the services provided by the Grid'5000 cluster[1]. Grid'5000 is a large scale cluster used for experiment-driven research in all areas of computer science, with more focus on parallel and distributed computing. Among other key features, we chose to use it due to its highly configurable environment and large amount of resources provided. The cluster's resources are distributed across France's territory linked through a high speed network. In our experiments we utilized a machine with the specifications found in Table 5.1.

The overlay utilized to run our experiments can be found in Figure 5.2. This overlay is composed by eight brokers, represented by circles, and five clients, represented by squares. Links in the overlay were assigned a communication latency, in milliseconds, each calculated with reference to a latencies table[2]. Moreover, to assign them, we categorized each link in one of two categories: *inter-region* and *intra-region*. An *inter-region* link is one

[1] https://www.grid5000.fr/w/Grid5000:Home
[2] https://www.cloudping.co/grid/p_10/timeframe/1D

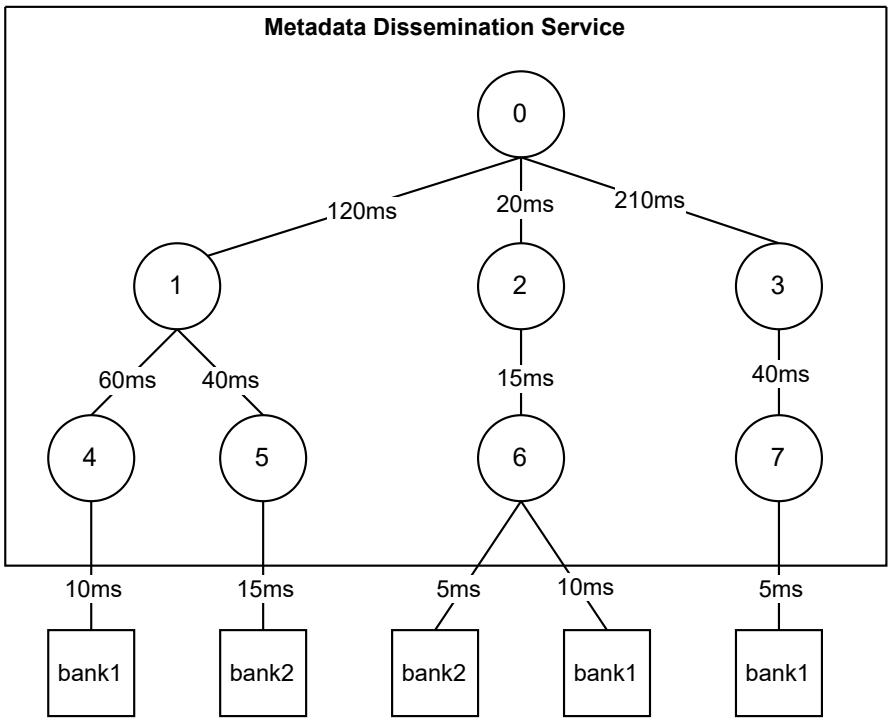


Figure 5.2: Simulated network used during experiments. The numbers over the links correspond to the message's latency in each of them.

Table 5.2: Latency ranges assigned to each link category.

Category	Latency Range
Intra-Region	5ms-70ms
Inter-Region	70ms-300ms

whose ends are in different earth's regions. So, for example, if one end of its ends is located in Europe and another in North America then the link between them would belong to this category. On the other hand, an *intra-region* link is one that connects any brokers in the same region. This categorization was made with the purpose of assigning latencies between brokers to effectively simulate a system with clients in multiple regions. Table 5.2 displays the latency ranges used to assign a latency to each link category.

In practice, we first decided to place broker 0, the tree's root, in the European region. Afterwards, we decided to place its children in three different regions. Broker 1 was placed in North America, broker 2 in the same region as broker 0 and broker 3 in the Asian continent. Then, to more accurately assign latencies between brokers 0 and 3 and brokers 0 and 1, we decided to get more specific with the latency ranges between these regions, according to our reference table. The region-specific latency ranges can be found in Table 5.3. Finally, we took average values and assigned them to the links. As for the

Table 5.3: Latency ranges between Europe and other regions.

North America	Asia
70ms-160ms	180ms-300ms

remaining ones, since they are all in the *intra-region* category, we assigned values within the category's latency range. These latencies were applied using the *traffic control* (*tc*) tool, bundled in the *iproute* package available on Debian Linux. With this tool, we set different network packet queuing disciplines, to apply a delay on outgoing packets to a given destination, in order to emulate the latency between brokers.

After assigning latencies between brokers, we moved into setting up multiple test scenarios, introducing different number of faults in the system, by killing the processes associated to each broker, to verify if brokers behave according to their relationship with the failed brokers and with the goal of assessing the protocol's efficiency, taking into consideration the failed broker's position, its number of children and subscribed topics. Each test configuration can be found in a configuration file, which contains the necessary information to build the initial overlay, to define broker's and client's parameters, to define the time at which a specific broker fails and test specific parameters, which include:

- *n_key_items*: number of key items per topic.
- *n_operations*: total number of operations performed by each client, which translate to publications in the middleware.
- *eventual*: percentage of messages, tagged as *eventual*, published.
- *causal*: percentage of messages, tagged as *causal*, published.
- *linear*: percentage of messages, tagged as *eventual*, published.
- *msg_interval*: time, in milliseconds, between the publication of two consecutive messages.

The broker and client parameters are the following:

- *maxHops*: the maximum distance a *PathEntry* travels in the tree. Can also be seen as the maximum distance, measured in message hops, between a broker and others it knows of.
- *timerDuration*: the maximum time a broker or client waits for a connection to be established, after their parent failing.
- *bufferSize*: maximum length of per-topic buffers.
- *nr-actors*: number of workers per broker.

- *total-degree*: maximum number of active connections a broker can maintain.

Among these parameters, *nr-actors* and *total-degree* are broker specific. Finally, failures are configured by indicating the broker's identifier and the time, after the beginning of the run, to fail. An example configuration file can be found in Annex I.

The test scenarios, include one with no failures that served as a comparable baseline. In each scenario we varied, the failed broker's position in the tree, the number of topics subscribed by the failed broker, which is dependent on the topics subscribed by its children, and the number of failures. In terms of scenario parameters, a test can be defined by the tuple:

$$test < n\_key\_items, n\_operations, eventual, causal, linear, msg\_interval, n\_faults >$$

In terms of broker and client parameters, these were fixed across all experiments. The respective parameters are highlighted in Listings 5.1 and 5.2

Listing 5.1: Parameters utilized by brokers in each scenario

```
broker_parameters{
    nr-actors=10
    total-degree=3
    bufferSize=150
    timerDuration=2
    maxHops=3
}
```

Listing 5.2: Parameters utilized by clients in each scenario

```
mi_params{
    maxHops=3
    bufferSize=150
    timerDuration=2
}
```

Concerning correctness, the methodology is to answer the questions, defined in Section 5.1.1, for all scenarios. Concerning performance the methodology is to compute statistics for each scenario and for groups of scenarios. Due to the lack of time, each scenario was ran 5 times and the respective statistics were computed. The scenarios are the following:

Broker 1 fails some time after starting: brokers 4 and 5 must connect to a new parent and lost messages must be received by both.

Broker 2 fails some time after starting: brokers 1, 2 and 3 must execute the algorithm to determine which of them becomes the new root.

Brokers 1 and 4 fail simultaneously: the client connected to broker 1 must find a new broker to connect it to the middleware.

Brokers 0 and 1 fail simultaneously: brokers 2 and 3 must start the algorithm to determine the new root broker however, they are not aware of broker 1's failure.

Broker 0's children fail some time after starting: brokers 4, 5, 6 and 7 must find a new parent.

5.3 Correctness

Concerning correctness of the overlay maintenance protocol, we must analyze broker's log files. It is correct if the following properties are observed in them:

1. Brokers send *NeighborStatus* messages when a neighbor's status changes (neighbor added or failed).

2. Brokers store addresses according to the number of hops and movement pattern of path entries.

3. Brokers correctly remove failed brokers from their tree view, when receiving an update.

4. Brokers increment and decrement their *residualDegree* when removing or adding neighbors.

5. Brokers execute the root election protocol if the root fails.

6. Brokers reject *ParentRequest* if *residualDegree* equals zero or are not subscribed to any of the requester's topics.

7. Brokers prevent messages from being processed out of order, by holding incoming messages during a reconfiguration.

8. Brokers can detect messages missed and guarantee they are delivered in the order they are processed.

5.3.1 Updating tree view

In Section 4.2.2.1, we highlighted how important updating tree view is. In Annex II, we provide a walkthrough of a broker's log file showcasing tree view updates triggered by adding neighbors to the neighbors set.

5.3.2 Incrementing and decrementing *residualDegree*

Correctly updating *residualDegree* ensures brokers do not accept connections from more brokers than they can handle. In Annex IV, we provide an example log file of a broker changing its *residualDegree*, triggered by changes in its neighborhood (adding or removing brokers).

5.3.3 Sending and1 handling *ParentRequest*

After a broker's parent failing, brokers must take appropriate action at attempting to repair the overlay. In this case, a broker must contact another in its tree view. In Annex IV, we also provide an overview of a broker taking those appropriate steps.

5.3.4 Ensuring total order message's processing is completed

In Section 4.3.2.4, we presented the steps brokers take to verify if a failure interrupted the processing of total order *Metadata* messages and how it can be completed. Annex V contains an example of a broker's log file, correctly performing the steps to ensure total order message's processing is completed after a failure occurring.

5.3.5 Executing root election algorithm

The logs in Annex VI, correspond to excerpts of a scenario where the root fails. In this situation root's children must work collectively to determine which among them becomes the new root of the overlay. The logs in the referenced annex show an example of the correct behavior.

5.3.6 Preventing messages from being processed out of order during an ongoing reconfiguration

In Section 4.3.1, we explained that to prevent messages from being processed and delivered out of order, during an ongoing reconfiguration, brokers hold incoming messages and once they determine the reconfiguration has been completed, they release them. Annex VII, contains an example log file of that behavior.

5.3.7 Executing multiple rounds of *ParentRequest*, correctly detecting missed messages and correctly rejecting requests

Performing topological reconfigurations, may require multiple rounds of connection attempts, depending on broker degree and topics subscribed by brokers selected to replace another's parent. Despite the multiple rounds, every time a *ParentRequest* is received, it must be handled correctly taking into account the previous properties. Moreover, even if it takes multiple rounds to find a replacement for the parent broker, a broker must correctly detect missing messages and the new parent must retransmit them in the correct order.

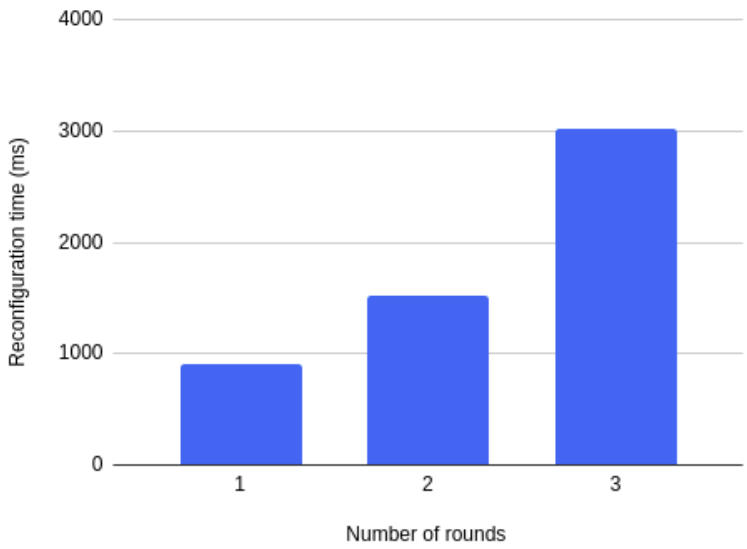


Figure 5.3: Reconfiguration time measured in milliseconds, taking into account the number of rounds required to establish a connection with the new parent, and receive all missed messages.

Annex VIII, contains an example of log showing the process is correctly performed and is supported by a text explaining its contents. Furthermore, Annex IX, contains tables showing the correct ordering of messages for a given key.

5.4 Performance

In this section, we provide an analysis of the performance results obtained. We ran multiple test scenarios, in which we varied the number of faults and number of keys in the system, to measure the cost imposed by a reconfiguration, measured in *reconfiguration time*, in milliseconds, and the average message latency, in milliseconds. Since we are working with multiple delivery guarantees, we grouped messages by delivery guarantees and calculated the average latency for each. As mentioned in Section 5.2, these tests were ran in a machine with a simulated environment, therefore, they may not correctly translate to a real world scenario. Since we also ran them in a single machine, multiple other factors may have interfered on our performance evaluation, such as scheduling and parallel tasks performed by other users due to our access level not allowing us to utilize better machines.

5.4.1 Reconfiguration cost

The experiment displayed in this section, aims at measuring the overhead introduced by a reconfiguration. The overhead is measured in *reconfiguration time*, in milliseconds, and the results obtained are displayed in the chart from Figure 5.3.

We define reconfiguration time as the time elapsed between the moment a fault is detected and the moment when the last missed message is received by a broker. Since the

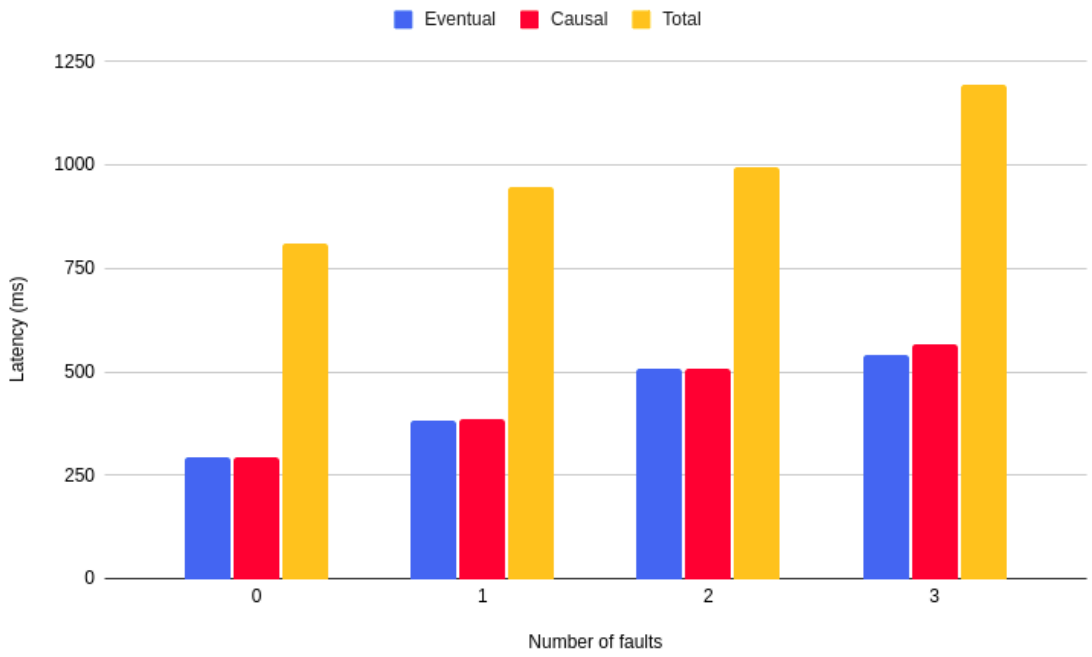


Figure 5.4: Message latency in the absence of failures and in the presence of one or more failures.

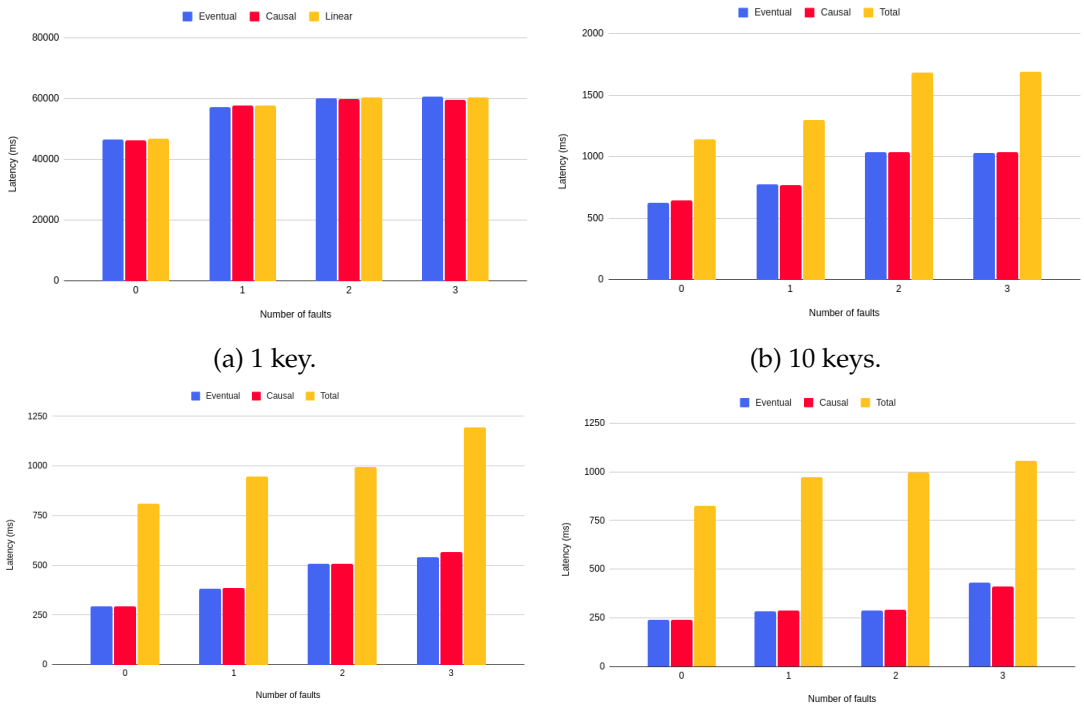
reconfiguration is executed in rounds composed of sending a *ParentRequest* and receiving the corresponding *ParentResponse*, it is expected that the more rounds it takes for a broker to complete the reconfiguration, the longer it will take for the it to be completed increasing the reconfiguration time. To conduct this experiment, we executed tests in which brokers would have their *ParentRequest* rejected once or more, either by receiving a *ParentResponse* informing the request was rejected or a timer expiring, which is handled as a rejection. In this case, we induced faults in the root's children to simulate these occurrences. The tuple that represents these tests is

$$test < 100, 1000, 0.4, 0.3, 0.3, 50, f >\ ,\ f \in \{0, 1, 2, 3\}$$

By analyzing the results, we were able to determine that the cost of reconfiguration is directly linked to the number of rounds needed for a broker to successfully establish a connection with the new parent and receive all missed messages, as the average time of reconfiguration increases as the number of rounds increases. The chart in Figure 5.3 proves this claim since each column's height, which represents the reconfiguration time, increases with the increase in rounds performed until the reconfiguration process is concluded.

5.4.2 Impact of failures in message latency, within multiple brokers' *tree views*

In Section 5.4.1, we determined that the reconfiguration process has a cost attached to it and, therefore, we expected faults to have an additional impact on message latency.



(a) 1 key.

(b) 10 keys.

(c) 100 keys.

(d) 1000 keys.

Figure 5.5: Message latency with varying number of faults and number of key items.

To confirm our expectations, we executed experiments where we varied the number of simultaneous faults occurred, within multiple brokers' *tree views*, and compared the results with a scenario where no faults occur. The tuple used for this experiment is equivalent to the one used in the experiments from Section 5.4.1:

$$test < 100, 500, 0.4, 0.3, 0.3, 50, f >\ ,\ f \in \{0, 1, 2, 3\}$$

The results are displayed in Figure 5.4, that shows the average message latency, in milliseconds, for each delivery guarantee provided. The results show that, in general, messages tagged with eventual and causal delivery guarantees are delivered faster to clients than messages tagged with total delivery. This is due to the delivery mechanism utilized by brokers to deliver messages in total order.

We expected message latency to be directly linked to the latencies between brokers in the absence of faults. Moreover, since the occurrence of faults adds a reconfiguration that imposes an overhead, it is expected that the more faults that occur the greater the impact that each fault imposes and, consequently, the observable latencies will increase. This fact is corroborated by the chart in Figure 5.4, which shows a correlation between the number of faults and message latency since column's height, which represents average latency, increases as the number of faults also increases.

5.4.3 Impact of key items in message latency when failures occurs

Now that we measured the cost of reconfiguration and how multiple failures can affect message latency, we also wanted to determine how the number of keys in the system. The *keyItems* field in *Metadata* messages is essential for the correct dissemination of messages especially when messages tagged with total order delivery are in play, given that to ensure total order, brokers temporarily block messages related by the keys contained within *keyItems*. The greater the number of existing keys, the lower the likelihood of messages being related by the keys within *keyItems*, consequently, decreasing the likelihood of messages being blocked. The experiment displayed in this section aims at measuring how impactful is the number of keys in message latency when coupled with the occurrence of faults. This experiment corresponds to the tuple

$$test < k, 500, 40\%, 30\%, 30\%, 50, f >,\ k \in \{1, 10, 100, 1000\} \land f \in \{0, 1, 2, 3\}$$

and the results are displayed in Figure 5.5. The results display the average latency, in milliseconds, of messages tagged with different delivery guarantees for different number of keys and faults occurred, and show that the higher the number of keys, the lower the average message latency, which can be explained by the number of conflicting messages that are blocked during the time a message tagged with total order delivery is being processed, and the associated cost of reconfiguration. A small number of keys increases the chance of conflicting messages that are blocked, leading to an increase in message latency since conflicting messages are only processed after delivering the total order message. Moreover, similarly to the experiment from Section 5.4.2, the more faults that occur, the higher the impact on message latency.

6

Conclusions

In this dissertation, we presented our approach at developing a protocol to provide fault-tolerance capabilities to a pub/sub system who delivers messages from publishers to subscribers according to multiple delivery guarantees. Our work resulted in the implementation of localized overlay maintenance protocol, inspired in the overlay maintenance strategies presented in [16] and by LoCaMu [25], to effectively reconnect tree overlay networks. This protocol aims at minimizing reconfiguration costs by confining the topological adaptions to failed node's children and at minimizing the load imposed on nodes after the necessary topological changes occur, achieved by limiting node's degree. The developed protocol also ensures messages are not lost as a consequence of nodes failing, by restoring the communication channels and message flow between nodes. When restoring communication channels, nodes need to guarantee that any messages missed by new neighbors are delivered in the same order received by a node, which in this case *FIFO* order, since our pub/sub disseminates messages in that order. Since our systems delivers messages according to multiple delivery guarantees, the protocol also needs to ensure those delivery guarantees are not violated. For instance, during reconfiguration if a node is processing a message that needs to be delivered in total order to all subscribers, it needs to ensure no other message is processed before its processing is completed. Our protocol ensures that by holding *Metadata* messages while a reconfiguration is ongoing, preventing them from being processed out of order and, consequently, from violating delivery guarantees.

The results analyzed in Chapter 5, prove the protocol's correctness in terms of reconnecting the overlay, ensuring messages are not lost and delivery does not violate the multiple delivery guarantees as a result of node failures. The results show nodes are able to find a node to replace their parent, after it fails. Additionally, they show nodes can detect messages missed by neighbors and deliver them in the order they were received and message processing can be completed before processing incoming messages after the reconfiguration is completed. Furthermore, message latency is directly linked to communication latency between nodes and our solution only has a slight impact in message latency in the presence of failures.

6.1 Future Work

During the development of our work we noticed that, although the pub/sub system provides a great baseline to develop services that provide developers the ability to register operations with different delivery guarantees, some aspects can still be improved.

Improved parent selection criteria. When a fault is detected, brokers select a replacement for their parent based on the distance, regarding the number of message hops, to brokers within their local tree view. This might lead brokers to establish a connection with a broker in a distant geographical location increasing the latency of communication. Another improvement that we propose is improve the criteria used to store brokers in the local tree view, since we are dealing with a topic-based pub/sub system and brokers are stored in tree views based on their position in the overlay tree. These criteria can primarily take subscribed topics into consideration since topic trees are required to be preserved once a fault occurs.

Network partition-tolerance. As mentioned in Chapter 4, our work considers a crash failure fault model and, therefore, only failures resulting from brokers crashing are considered. In real world applications, the ability to tolerate network partitions is desirable and necessary to ensure the correct behavior of the system. Therefore, we propose the implementation of a protocol to detect and handle the occurrence of network partitions.

Concurrency control. As of now the middleware utilizes a very basic strategy to approach the problem of shared data among different processes. In many situations, the lack of good concurrency control strategies can greatly impact the system's performance. Therefore, as future work we propose the implementation of a more robust concurrency control strategy in order to improve system's performance when multiple processes are accessing the same shared resources.

Failure recovery. During our work we developed a protocol to effectively repair the middleware overlay, when brokers fail. However, if, for instance, a broker recovers from a failure, the middleware does not have a mechanism to support its recovery, by reintegrating the broker in it. The recovery protocol allows the overlay to return to the state before a failure occurring.

Dynamic joining mechanism. As of now, the pub/sub overlay is pre-configured, in configuration files. This puts a lot of responsibility in system's developers when configuring the initial connections between brokers. If a broker is not correctly configured, it may not connect to the expected broker which may lead to a decrease in performance and message latency. Therefore, we propose the development of a protocol that allows brokers to dynamically join the overlay, by contacting other brokers already in it.

Garbage collection mechanism. In our approach, brokers store processed messages in buffer and use them to identify missed messages if a fault is detected. In our implementation, buffers have an upper bound on their length. If buffers are not large enough there might be a chance of a broker requesting a message and that message no longer being contained in the buffer. On the other hand, if buffers are too large, a broker's memory usage may be increased unnecessarily when no faults occur. For that purpose, we propose the implementation of a garbage collection mechanism to find a middle ground between both the highlighted downsides of our solution.

Bibliography

[1] H. N. S. Aldin et al. "Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications". In: *CoRR* abs/1902.03305 (2019). arXiv: 1902.03305. URL: http://arxiv.org/abs/1902.03305 (cit. on pp. 11, 12).

[2] H. Attiya and J. L. Welch. "Sequential Consistency versus Linearizability". In: *ACM Trans. Comput. Syst.* 12.2 (1994), pp. 91–122. DOI: 10.1145/176575.176576. URL: https://doi.org/10.1145/176575.176576 (cit. on p. 11).

[3] D. Bermbach and J. Kuhlenkamp. "Consistency in Distributed Storage Systems - An Overview of Models, Metrics and Measurement Approaches". In: *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*. Ed. by V. Gramoli and R. Guerraoui. Vol. 7853. Lecture Notes in Computer Science. Springer, 2013, pp. 175–189. DOI: 10.1007/978-3-642-40148-0_13. URL: https://doi.org/10.1007/978-3-642-40148-0%5C_13 (cit. on pp. 11, 12).

[4] E. A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. Ed. by G. Neiger. ACM, 2000, p. 7. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502 (cit. on p. 2).

[5] S. Burckhardt. "Principles of Eventual Consistency". In: *Found. Trends Program. Lang.* 1.1-2 (2014), pp. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011 (cit. on p. 11).

[6] M. Castro et al. "Scribe: a large-scale and decentralized application-level multicast infrastructure". In: *IEEE J. Sel. Areas Commun.* 20.8 (2002), pp. 1489–1499. DOI: 10.1109/JSAC.2002.803069. URL: https://doi.org/10.1109/JSAC.2002.803069 (cit. on pp. 15, 16, 22).

[7] T. Chang et al. "P2S: a fault-tolerant publish/subscribe infrastructure". In: *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*. Ed. by U. Bellur and R. Kothari. ACM, 2014, pp. 189–197. DOI: 10.1145/2611286.2611305. URL: https://doi.org/10.1145/2611286.2611305 (cit. on pp. 19, 22).

[8] H.-E. Chihoub et al. *Consistency Management in Cloud Storage Systems*. 2014 (cit. on p. 11).

[9] L. Chula. "Disseminação de metadados com diferentes garantias de ordenação". Master's thesis. Monte de Caparica, Almada: NOVA School of Science and Technology, 2023-03 (cit. on p. 2).

[10] P. Costa et al. "Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation". In: *24th International Conference on Distributed Computing Systems (ICDCS) 2004, 24-26 March 2004, Hachioji, Tokyo, Japan*. IEEE Computer Society, 2004, pp. 552–561. DOI: 10.1109/ICDCS.2004.1281622. URL: https://doi.org/10.1109/ICDCS.2004.1281622 (cit. on pp. 17, 22).

[11] P. Costa et al. "Introducing reliability in content-based publish-subscribe through epidemic algorithms". In: *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems, DEBS 2003, Sunday, June 8th, 2003, San Diego, California, USA (in conjunction with (SIGMOD/PODS))*. Ed. by H. Jacobsen. ACM, 2003. DOI: 10.1145/966618.966629. URL: https://doi.org/10.1145/966618.966629 (cit. on pp. 17, 22).

[12] P. H. B. Dias. "Tree-based Decentralized and Robust Causal Dissemination". MA thesis. NOVA School of Science and Technology, 2020-04 (cit. on pp. 20, 22).

[13] C. Esposito. "A tutorial on reliability in publish/subscribe services". In: *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*. Ed. by F. Bry et al. ACM, 2012, pp. 399–406. DOI: 10.1145/2335484.2335537. URL: https://doi.org/10.1145/2335484.2335537 (cit. on p. 15).

[14] P. T. Eugster et al. "The many faces of publish/subscribe". In: *ACM Comput. Surv.* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078. URL: https://doi.org/10.1145/857076.857078 (cit. on pp. 13–15).

[15] M. F. S. Ferreira, J. Leitão, and L. E. T. Rodrigues. "Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay". In: *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*. IEEE Computer Society, 2010, pp. 293–302. DOI: 10.1109/SRDS.2010.19. URL: https://doi.org/10.1109/SRDS.2010.19 (cit. on pp. 18, 22).

[16] D. Frey and A. L. Murphy. "Failure-Tolerant Overlay Trees for Large-Scale Dynamic Networks". In: *Proceedings P2P'08, Eighth International Conference on Peer-to-Peer Computing, 8-11 September 2008, Aachen, Germany*. Ed. by K. Wehrle et al. IEEE Computer Society, 2008, pp. 351–361. DOI: 10.1109/P2P.2008.30. URL: https://doi.org/10.1109/P2P.2008.30 (cit. on pp. 21, 22, 30, 68).

[17] R. S. Kazemzadeh and H. Jacobsen. "Reliable and Highly Available Distributed Publish/Subscribe Service". In: *28th IEEE Symposium on Reliable Distributed Systems (SRDS 2009), Niagara Falls, New York, USA, September 27-30, 2009*. IEEE Computer Society, 2009, pp. 41–50. DOI: 10.1109/SRDS.2009.32. URL: https://doi.org/10.1109/SRDS.2009.32 (cit. on pp. 17, 22).

[18] B. Kemme and G. Alonso. "Database Replication: a Tale of Research across Communities". In: *Proc. VLDB Endow.* 3.1 (2010), pp. 5–12. DOI: 10.14778/1920841.1920847. URL: http://www.vldb.org/pvldb/vldb2010/pvldb%5C_vol3/TY02.pdf (cit. on p. 2).

[19] Y. Liu and V. Vlassov. "Replication in Distributed Storage Systems: State of the Art, Possible Directions, and Open Issues". In: *2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2013, Beijing, China, October 10-12, 2013*. IEEE Computer Society, 2013, pp. 225–232. DOI: 10.1109/CyberC.2013.44. URL: https://doi.org/10.1109/CyberC.2013.44 (cit. on p. 10).

[20] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[21] B. C. Neuman. *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1994. Chap. Scale in distributed systems, p. 68 (cit. on p. 1).

[22] L. M. D. Rocha. "Ginger: A Transactional Middleware with Data and Operation Centric Mixed Consistency". Master's thesis. Monte de Caparica, Almada: NOVA School of Science and Technology, 2020-11 (cit. on pp. 2, 3).

[23] A. I. T. Rowstron and P. Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*. Ed. by R. Guerraoui. Vol. 2218. Lecture Notes in Computer Science. Springer, 2001, pp. 329–350. DOI: 10.1007/3-540-45518-3_18. URL: https://doi.org/10.1007/3-540-45518-3%5C_18 (cit. on p. 16).

[24] P. Salehi, C. Doblander, and H. Jacobsen. "Highly-available content-based publish/-subscribe via gossiping". In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*. Ed. by A. Gal et al. ACM, 2016, pp. 93–104. DOI: 10.1145/2933267.2933303 (cit. on pp. 19, 22).

[25] V. Santos and L. Rodrigues. "Localized Reliable Causal Multicast". In: *18th IEEE International Symposium on Network Computing and Applications, NCA 2019, Cambridge, MA, USA, September 26-28, 2019*. Ed. by A. Gkoulalas-Divanis, M. Marchetti, and D. R. Avresky. IEEE, 2019, pp. 1–10. DOI: 10.1109/NCA.2019.8935065. URL: https://doi.org/10.1109/NCA.2019.8935065 (cit. on pp. 20, 22, 26, 30, 54, 68).

[26] E. Spaho, L. Barolli, and F. Xhafa. "Data replication strategies in P2P systems: A survey". In: *2014 17th international conference on network-based information systems*. IEEE. 2014, pp. 302–309 (cit. on pp. 9, 10).

[27] A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN: 978-90-815406-2-9 (cit. on pp. 1, 11, 12, 14).

[28] A. S. Tanenbaum and M. van Steen. *Distributed systems, 3rd Edition*. Pearson Education, 2018. ISBN: 978-0-13-239227-3 (cit. on pp. 2, 15).

[29] S. Tarkoma. *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012 (cit. on p. 14).

I

Example test configuration file

Listing I.1: Example configuration type.

```
broker0{
    nodeID=0
    self="akka://PubSubSystem@localhost:3000/user/treeBroker"
    parent=""
}
broker1{
    nodeID=1
    self="akka://PubSubSystem@localhost:3001/user/treeBroker"
    parent="akka://PubSubSystem@localhost:3000/user/treeBroker"
}
broker2{
    nodeID=2
    self="akka://PubSubSystem@localhost:3002/user/treeBroker"
    parent="akka://PubSubSystem@localhost:3000/user/treeBroker"
}
broker3{
    nodeID=3
    self="akka://PubSubSystem@localhost:3003/user/treeBroker"
    parent="akka://PubSubSystem@localhost:3001/user/treeBroker"
}
broker4{
    nodeID=4
    self="akka://PubSubSystem@localhost:3004/user/treeBroker"
    parent="akka://PubSubSystem@localhost:3002/user/treeBroker"
}
mi0{
    replicaID=0
    self="akka://middleware@localhost:2500/user/middleware"
```

```
29      my-broker-data="akka://PubSubSystem@localhost:3504/user/treeBroker"
30      my-broker-metadata="akka://PubSubSystem@localhost:3004/user/treeBroker"
31      topic="topic1"
32  }
33  mi1{
34      replicaID=1
35      self="akka://middleware@localhost:2501/user/middleware"
36      my-broker-data="akka://PubSubSystem@localhost:3503/user/treeBroker"
37      my-broker-metadata="akka://PubSubSystem@localhost:3003/user/treeBroker"
38      topic="topic2"
39  }
40  broker_params{
41      nr-actors=5
42      total-degree=3
43      msg-buffer-size=10
44      connection-timer-seconds=3
45      path-entry-max-hops=3
46  }
47  mi_params{
48      path-entry-max-hops=3
49      msg-buffer-size=5
50      connection-timer-seconds=3
51  }
52  scenario_params{
53      n_key_items=100
54      n_operations=200
55      eventual=0.4
56      causal=0.3
57      linear=0.3
58      msg_interval=100
59  }
60  failure{
61      broker=2
62      time=57
63      broker=4
64      time=60
65  }
```

II

Sending and handling *NeighborStatus* - Adding neighbor

Listing II.1: Broker 1 sending and handling *NeighborStatus* messages when neighbors are added to neighbors set.

```
1 class: TreeBroker | message: adding neighbor
    akka://PubSubSystem@10.16.1.107:3000/user/treeBroker. Can receive 3
    connections
2 class: TreeBroker | message: sending NeighborStatus [nChidlren=0,
    other=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,
    failed=false, path=([])] to:
    akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
3 class: TreeBroker | message: connection to
    akka://PubSubSystem@10.16.1.107:3000/user/treeBroker established
4 class: TreeBroker | message: received NeighborStatus [nChidlren=0,
    other=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker,
    failed=false,
    path=([(address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,
    hops=0, down=true),
    (address=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker,
    hops=1, down=false)])] from
    Actor[akka://PubSubSystem@10.16.1.107:3000/user/treeBroker#-434148671]
```

```
5 class: TreeBroker | message: received NeighborStatus [nChidlren=1,
    other=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,
    failed=false,
    path=([(address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,
    hops=0, down=true),
    (address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,
    hops=1, down=false),
    (address=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker,
    hops=1, down=false)])] from
    Actor[akka://PubSubSystem@10.16.1.107:3000/user/treeBroker#-434148671]
6 class: TreeBroker | message: adding sibling
    akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
7 class: TreeBroker | message: received NeighborStatus [nChidlren=2,
    other=akka://PubSubSystem@10.16.1.114:3003/user/treeBroker,
    failed=false,
    path=([(address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,
    hops=0, down=true),
    (address=akka://PubSubSystem@10.16.1.114:3003/user/treeBroker,
    hops=1, down=false),
    (address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,
    hops=1, down=false),
    (address=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker,
    hops=1, down=false)])] from
    Actor[akka://PubSubSystem@10.16.1.107:3000/user/treeBroker#-434148671]
8 class: TreeBroker | message: adding sibling
    akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
```

In this Annex II, we show broker log files to show that, when adding a neighbor, a broker always sends *NeighborStatus* messages to its neighbors and correctly adds information about its other neighbors to the message's path. When

akka://PubSubSystem@10.16.1.108:3000/user/treeBroker

which in the simulated network corresponds to its parent, was added to the neighbor's set, since no other neighbor had been added and the message was sent to its parent, the path field is empty. Before adding brokers 4 and 5, which are its neighbors in the network, it received from the parent two *NeighborStatus* messages, corresponding to when brokers 2 and 3 were added to the parent's neighbors set, respectively. In these messages, the message's path contained information about all parent's neighbors. When handling the message's path, it correctly handled the entries for brokers

akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
akka://PubSubSystem@10.16.1.114:3003/user/treeBroker

who were added to the siblings set. In Section 4.2.2, we defined *sibling* as a broker with the same parent and can be determined if its path entry's hop count equals two and the movement's direction is upward. In the log files, *down=true* is equivalent to *direction=true*. On reception, the message's path contained the following entries:

(address=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker, hops=1, down=false)
(address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker, hops=1, down=false)

After both entries' hop count were incremented, from one to two, and since the *down* field, which is the message's direction, is set to *false*, then each entry corresponds to a sibling.

III

Storing ancestors in message's path.

Listing III.1: Broker 4 handling path entries and storing ancestors.

```
1 class: TreeBroker | message: received NeighborStatus [nChidlren=1,
    other=akka://PubSubSystem@10.16.1.121:3004/user/treeBroker,
    failed=false,
    path=([(address=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker,
    hops=0, down=true),
    (address=akka://PubSubSystem@10.16.1.121:3004/user/treeBroker,
    hops=1, down=false),
    (address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,
    hops=1, down=true),
    (address=akka://PubSubSystem@10.16.1.114:3003/user/treeBroker,
    hops=2, down=false),
    (address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,
    hops=2, down=false)])] from
    Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-262011069]
2 class: TreeBroker | message: adding ancestor
    akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
3 class: TreeBroker | message: adding ancestor
    akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
```

```
4 class: TreeBroker | message: received NeighborStatus [nChidlren=2,
    other=akka://PubSubSystem@10.16.1.26:3005/user/treeBroker,
    failed=false,
    path=([(address=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker,
    hops=0, down=true),
    (address=akka://PubSubSystem@10.16.1.26:3005/user/treeBroker, hops=1,
    down=false),
    (address=akka://PubSubSystem@10.16.1.121:3004/user/treeBroker,
    hops=1, down=false),
    (address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,
    hops=1, down=true),
    (address=akka://PubSubSystem@10.16.1.114:3003/user/treeBroker,
    hops=2, down=false),
    (address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,
    hops=2, down=false)])] from
    Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-262011069]
5 class: TreeBroker | message: adding sibling
    akka://PubSubSystem@10.16.1.26:3005/user/treeBroker
6 class: TreeBroker | message: received Subscribe [topic=topic2,
    path=([(address=akka://PubSubSystem@10.16.1.108:3001/user/treeBroker,
    hops=0, down=true),
    (address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,
    hops=1, down=true),
    (address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,
    hops=2, down=false),
    (address=akka://PubSubSystem@10.16.1.38:3006/user/treeBroker, hops=3,
    down=false)])] from
    Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-262011069]
7 class: TreeBroker | message: adding ancestor
    akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
```

We now analyze if *ancestors* are correctly stored. In Section 4.2.2, an *ancestor* as any broker whose tree depth was lower than the path's handler depth. Annex III displays an excerpt of broker 4's logs (Listing III.1). In the simulated network broker 4's ancestors are 0, 2 and 3. Therefore, when a *Subscribe* or *NeighborStatus* message contains an entry with their addresses, their addresses must be added to the broker 4's *ancestors* set. Moreover, we stated that, when handling a message's path, to identify an ancestor either its path entry's move was downward (*direction* is set to *true*) and its hop count is greater than one, or the move is upward (*direction* is set to *false*) and the previous entry's address belong to an ancestor. In the log file, we can identify three ancestors were added with addresses, in lines 2, 3 and 7.

Let's analyze the corresponding entries:

(address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,hops=1,down=true)
(address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,hops=2,down=false)
(address=akka://PubSubSystem@10.16.1.114:3003/user/treeBroker,hops=2,down=false)

After incrementing the hop count they become:

(address=akka://PubSubSystem@10.16.1.107:3000/user/treeBroker,hops=2,down=true)
(address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,hops=3,down=false)
(address=akka://PubSubSystem@10.16.1.114:3003/user/treeBroker,hops=3,down=false)

Broker 0 (akka://PubSubSystem@10.16.1.107:3000/user/treeBroker) is correctly identified as an *ancestor* due to *hops=2* and *down=true*. As for brokers 2 ((address=akka://PubSubSystem@10.16.1.111:3002/user/treeBroker,hops=3,down=false)) and 3 ((address=akka://PubSubSystem@10.16.1.114:3003/user/treeBroker,hops=3,down=false)), their entries indicate an upward movement (*down=false*) and, at the moment they were handled, the previous handled entry is broker 0's one. Since broker 0 was already stored in the *ancestors* set, they are correctly identified as ancestors and added to the corresponding set.

IV

Incrementing and decrementing *residualDegree* when adding and removing neighbors.

Listing IV.1: Broker 7 increments and decrements its *residualDegree* and send a *ParentRequest* when it detects its parent failed.

```
1 class: TreeBroker | message: adding neighbor
    akka://PubSubSystem@10.16.1.114:3003/user/treeBroker. Can receive 2
    connections
2 class: TreeBroker | message: adding neighbor
    akka://middleware@10.16.1.57:2504/user/middleware. Can receive 1
    connections
3 class: ConnectionMonitor | message: lost connection to
    akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
4 class: TreeBroker | message: removing neighbor
    akka://PubSubSystem@10.16.1.114:3003/user/treeBroker. Can receive 2
    connections
5 class: TreeBroker | message: expecting connections from 1 new neighbors
6 class: TreeBroker | ERROR: parent failed, sending ParentRequest
    [subscribedTopics=[topic1], breakDegree=false] to
    akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
7 class: TreeBroker | message: sending ParentRequest
    [subscribedTopics=[topic1], breakDegree=false] to:
    akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
8 class: TreeBroker | message: setting a timer for expected connections
9 class: TreeBroker | message:
    akka://PubSubSystem@10.16.1.107:3000/user/treeBroker accepted parent
    request, connecting...
```

```
10 class: TreeBroker | message: adding neighbor
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker. Can receive 1
       connections
11 class: TreeBroker | message: removing ancestor
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
```

Listing IV.2: Broker 0 receives *ParentRequest* and send a response.

```
1 class: ConnectionMonitor | message: lost connection to
      akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
2 class: TreeBroker | message: removing neighbor
      akka://PubSubSystem@10.16.1.114:3003/user/treeBroker. Can receive 1
      connections
3 class: TreeBroker | message: expecting connections from 1 new neighbors
4 class: TreeBroker | message: setting a timer for expected connections
5 class: TreeBroker | message: received ParentRequest
      [subscribedTopics=[topic1], breakDegree=false] from
      Actor[akka://PubSubSystem@10.16.1.4:3007/user/treeBroker#170697787]
      my residual degree is 1
6 class: TreeBroker | message: sending ParentResponse [accepted=true] to:
      akka://PubSubSystem@10.16.1.4:3007/user/treeBroker
7 class: TreeBroker | message:
      Actor[akka://PubSubSystem@10.16.1.4:3007/user/treeBroker#170697787]
      subscribed topic 'topic1'
8 class: TreeBroker | message: adding neighbor
      akka://PubSubSystem@10.16.1.4:3007/user/treeBroker. Can receive 1
      connections
```

We now analyze if broker's *residualDegree* is incremented and decrement under network dynamics. Listing IV.1 contains an excerpt of broker 7's log file, whose parent is broker 3. Each broker's initial *totalDegree* is three as displayed in Listing 5.1. Whenever broker 7 adds a neighbor (lines 1 and 2) to its neighbors set, the log indicates *residualDegree* is decremented ("Can receive **2** connections" → "Can receive **1** connections"). Then, when it detects a connection to a neighbor was lost (line 4), we can observe it is incremented ("Can receive **1** connections" → "Can receive **2** connections").

Now let's take a look to brokers sending and handling *ParentRequest* messages. In line 3 of Listing IV.1, broker 7 detected the connection to its parent was lost. Therefore, it must select a broker from its tree view to replace its parent. Broker 7's tree view contains brokers 0, 1 and 2 in the ancestors set. Line 10 of its log shows it sent a *ParentRequest* to broker 0, which corresponds to the first ancestor due to it being closer, in terms of hops, than the remaining ancestors.

When broker 0 detected broker 3's failure (Listing IV.2, line 1), broker 3's address is removed from its neighbors set and its *residualDegree* is incremented (line 2). Then, in line 3, the log states that broker 0 is expecting a *ParentRequest* from one broker, which equals to the number of children broker 3 had. Upon receiving the request (line 5), it states its *residualDegree* is equal to one ("my residual degree is **1**"), implying it can accept a connection. Therefore, it sends a *ParentResponse* to the requester with the *accepted* flag set to *true* (line 6). It is also observable, that broker 7 is added to the *topic1*'s subscribers set (line 7).

V

Ensuring total order message's processing is completed.

Listing V.1: Determining pending total order messages being processed and taking approprate action. Broker 0 log.

```
1 class: TreeBroker | message: Verifying pending total order messages for
    topic: topic1
2 class: TreeBroker | message: locked transactions [Metadata
    [topic='topic1', id=TransactionIdentifier [replicaID=3,
    messageID=224], consistency= Linear]]
3 class: TreeBroker | message: children acks: [LinearAck [topic='topic1',
    id=TransactionIdentifier [replicaID=3, messageID=224]]]
4 class: TreeBroker | message: waiting for LinearAcks for transaction:
    TransactionIdentifier [replicaID=3, messageID=224], resending
    Metadata [topic='topic1', id=TransactionIdentifier [replicaID=3,
    messageID=224], consistency= Linear] to children
5 class: TreeBroker | message: Received LinearAck id=TransactionIdentifier
    [replicaID=3, messageID=224] from:
    akka://PubSubSystem@10.16.1.4:3007/user/treeBroker
6 class: Broker | message: receiveLinearAck LinearAck [topic='topic1',
    id=TransactionIdentifier [replicaID=3, messageID=224]] from
    akka://PubSubSystem@10.16.1.4:3007/user/treeBroker
7 class: TreeBroker | message: Received LinearAck id=TransactionIdentifier
    [replicaID=3, messageID=224] from:
    akka://PubSubSystem@10.16.1.4:3007/user/treeBroker
8 class: Broker | message: receiveLinearAck LinearAck [topic='topic1',
    id=TransactionIdentifier [replicaID=3, messageID=224]] from
    akka://PubSubSystem@10.16.1.4:3007/user/treeBroker
```

```
9  class: TreeBroker | message: Received LinearAck id=TransactionIdentifier
       [replicaID=3, messageID=224] from:
       akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
10 class: Broker | message: receiveLinearAck LinearAck [topic='topic1',
       id=TransactionIdentifier [replicaID=3, messageID=224]] from
       akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
11 class: Broker | message: ==========linearT was ordered=========
12 class: Broker | message: sending LinearProcessed [topic='topic1',
       id=TransactionIdentifier [replicaID=3, messageID=224]] to
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#614734481]
13 class: Broker | message: sending LinearProcessed [topic='topic1',
       id=TransactionIdentifier [replicaID=3, messageID=224]] to
       Actor[akka://PubSubSystem@10.16.1.111:3002/user/treeBroker#515967547]
14 class: Broker | message: sending LinearProcessed [topic='topic1',
       id=TransactionIdentifier [replicaID=3, messageID=224]] to
       Actor[akka://PubSubSystem@10.16.1.4:3007/user/treeBroker#171461907]
```

Listing V.2: Receiving and acknowledging *Metadata* message. Broker 7 log

```
1  class: TreeBroker | message: Verifying pending linear transactions for
       topic: topic1
2  class: TreeBroker | message: locked transactions [Metadata
       [topic='topic1', id=TransactionIdentifier [replicaID=3,
       messageID=243], consistency= Linear], Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=228], consistency=
       Linear], Metadata [topic='topic1', id=TransactionIdentifier
       [replicaID=4, messageID=220], consistency= Linear]]
3  class: TreeBroker | message: waiting for processed confirmation for
       transaction with id: TransactionIdentifier [replicaID=3,
       messageID=243]
4  class: TreeBroker | message: waiting for processed confirmation for
       transaction with id: TransactionIdentifier [replicaID=0,
       messageID=228]
5  class: TreeBroker | message: waiting for processed confirmation for
       transaction with id: TransactionIdentifier [replicaID=4,
       messageID=220]
6
7  class: TreeBroker | message: received LinearProcessed [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=228]] from:
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
8
```

```
9  class: TreeBroker | message: received LinearProcessed [topic='topic1',
       id=TransactionIdentifier [replicaID=4, messageID=220]] from:
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
10
11 class: TreeBroker | message: Received Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=3, messageID=243], consistency=
       Linear] from:
       Actor[akka://PubSubSystem@10.16.1.107:3000/user/treeBroker#1126404846]
12 class: Broker | message: unlocking id= TransactionIdentifier
       [replicaID=0, messageID=228]
13 class: Broker | message: sending Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=228], consistency=
       Linear] to
       Actor[akka://middleware@10.16.1.57:2504/user/middleware#141630053]
14 class: Broker | message: unlocking id= TransactionIdentifier
       [replicaID=4, messageID=220]
15 class: Broker | message: sending Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=4, messageID=220], consistency=
       Linear] to
       Actor[akka://middleware@10.16.1.57:2504/user/middleware#141630053]
16 class: Broker | message: TransactionIdentifier [replicaID=3,
       messageID=243] was already acknowledged to parent, resending ack....
17 class: Broker | message: sending LinearAck [topic='topic1',
       id=TransactionIdentifier [replicaID=3, messageID=243]] to
       Actor[akka://PubSubSystem@10.16.1.107:3000/user/treeBroker#1126404846]
```

In the log file, we observe broker 3 failed and broker 7 established a connection to broker 0. Listing V.1 displays broker 0's log, where it found that message with identifier *(3,224)* was being processed when broker 3 failed. Furthermore, we can also observe the message processing is in a stage where broker 0 is waiting for *LinearAck* acknowledgements from its children. In Section 4.3.2.4, we mentioned that when a broker finds the message processing in this stage, it needs to resend the corresponding *Metadata* message to its children and wait for acknowledgements, which it does in line 4. Then, lines 5, 7 and 9 indicate broker 0 received the acknowledgements needed to complete the message's processing and notify its children.

VI

Executing root election algorithm

Listing VI.1: Broker 1 starts root election algorithm.

```
1  class: ConnectionMonitor | message: lost connection to
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
2  class: TreeBroker | message: removing neighbor
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker. Can receive 1
       connections
3  class: TreeBroker | message: expecting connections from 2 new neighbors
4  class: TreeBroker | message: sending RootElection [residualDegree=1,
       nodeID=1] to: akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
5  class: TreeBroker | message: sending RootElection [residualDegree=1,
       nodeID=1] to: akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
6  class: TreeBroker | message: executing RootElection protocol; need to
       receive RootElection from 2 siblings
7  class: TreeBroker | message: setting a timer for expected connections
8  class: TreeBroker | message: received RootElection [residualDegree=2,
       nodeID=2] from
       Actor[akka://PubSubSystem@10.16.1.111:3002/user/treeBroker#1819788342]
9  class: TreeBroker | message: received RootElection [residualDegree=2,
       nodeID=3] from
       Actor[akka://PubSubSystem@10.16.1.114:3003/user/treeBroker#-135339189]
10 class: TreeBroker | message: sending ParentRequest
       [subscribedTopics=[topic1, topic2], breakDegree=false] to:
       akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
11 class: TreeBroker | message: expecting connections from 1 new neighbors
12 class: TreeBroker | message: setting a timer for expected connections
```

Listing VI.2: Broker 2 starts root election algorithm.

```
1  class: ConnectionMonitor | message: lost connection to
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
```

```
2  class: TreeBroker | message: removing neighbor
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker. Can receive 2
       connections
3  class: TreeBroker | message: expecting connections from 2 new neighbors
4  class: TreeBroker | message: sending RootElection [residualDegree=2,
       nodeID=2] to: akka://PubSubSystem@10.16.1.108:3001/user/treeBroker
5  class: TreeBroker | message: sending RootElection [residualDegree=2,
       nodeID=2] to: akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
6  class: TreeBroker | message: executing RootElection protocol; need to
       receive RootElection from 2 siblings
7  class: TreeBroker | message: setting a timer for expected connections
8  class: TreeBroker | message: received RootElection [residualDegree=2,
       nodeID=3] from
       Actor[akka://PubSubSystem@10.16.1.114:3003/user/treeBroker#-135339189]
9  class: TreeBroker | message: received RootElection [residualDegree=1,
       nodeID=1] from
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#355261381]
10 class: TreeBroker | message: I will be the next root, can receive 2
       connections
11 class: TreeBroker | message: expecting connections from 2 new neighbors
12 class: TreeBroker | message: setting a timer for expected connections
13 class: TreeBroker | message: received ParentRequest
       [subscribedTopics=[topic1, topic2], breakDegree=false] from
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#355261381]
       my residual degree is 2
14 class: TreeBroker | message: sending ParentResponse [accepted=true] to:
       akka://PubSubSystem@10.16.1.108:3001/user/treeBroker
15 class: TreeBroker | message:
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#355261381]
       subscribed topic 'topic1'
16 class: TreeBroker | message:
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#355261381]
       subscribed topic 'topic2'
17 class: TreeBroker | message: adding neighbor
       akka://PubSubSystem@10.16.1.108:3001/user/treeBroker. Can receive 2
       connections
18 class: TreeBroker | message: received ParentRequest
       [subscribedTopics=[topic1], breakDegree=false] from
       Actor[akka://PubSubSystem@10.16.1.114:3003/user/treeBroker#-135339189]
       my residual degree is 1
```

```
19 class: TreeBroker | message: sending ParentResponse [accepted=true] to:
       akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
20 class: TreeBroker | message:
       Actor[akka://PubSubSystem@10.16.1.114:3003/user/treeBroker#-135339189]
       subscribed topic 'topic1'
21 class: TreeBroker | message: adding neighbor
       akka://PubSubSystem@10.16.1.114:3003/user/treeBroker. Can receive 1
       connections
```

Listing VI.3: Broker 3 starts root election algorithm.

```
 1 class: ConnectionMonitor | message: lost connection to
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker
 2 class: TreeBroker | message: removing neighbor
       akka://PubSubSystem@10.16.1.107:3000/user/treeBroker. Can receive 2
       connections
 3 class: TreeBroker | message: expecting connections from 2 new neighbors
 4 class: TreeBroker | message: sending RootElection [residualDegree=2,
       nodeID=3] to: akka://PubSubSystem@10.16.1.108:3001/user/treeBroker
 5 class: TreeBroker | message: sending RootElection [residualDegree=2,
       nodeID=3] to: akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
 6 class: TreeBroker | message: executing RootElection protocol; need to
       receive RootElection from 2 siblings
 7 class: TreeBroker | message: setting a timer for expected connections
 8 class: TreeBroker | message: received RootElection [residualDegree=2,
       nodeID=2] from
       Actor[akka://PubSubSystem@10.16.1.111:3002/user/treeBroker#1819788342]
 9 class: TreeBroker | message: received RootElection [residualDegree=1,
       nodeID=1] from
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#355261381]
10 class: TreeBroker | message: sending ParentRequest
       [subscribedTopics=[topic1], breakDegree=false] to:
       akka://PubSubSystem@10.16.1.111:3002/user/treeBroker
11 class: TreeBroker | message: expecting connections from 1 new neighbors
12 class: TreeBroker | message: setting a timer for expected connections
13 class: TreeBroker | message:
       akka://PubSubSystem@10.16.1.111:3002/user/treeBroker accepted parent
       request, connecting...
14 class: TreeBroker | message: adding neighbor
       akka://PubSubSystem@10.16.1.111:3002/user/treeBroker. Can receive 2
       connections
```

In this situation, brokers 1 (Listing VI.1), 2 (Listing VI.2) and 3 (Listing VI.3) must execute the root election algorithm. In the log files, line 1 shows brokers detecting the root's failure. Additionally, in lines 4 and 5 of these Listings, we observe each of them send a *RootElection* message to each of the siblings. In this case, broker 2 becomes the new root as stated in line 10 of its log (Listing VI.2).

VII

Holding incoming messages during an ongoing reconfiguration.

Listing VII.1: Broker 0 holding incoming messages between the time it detected the connection to broker 3 was lost, until the reconfiguration is determined to be completed

```
1  class: ConnectionMonitor | message: lost connection to
       akka://PubSubSystem@10.16.1.114:3003/user/treeBroker
2                                          .
3                                          .
4                                          .
5  class: TreeBroker | message: Received Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=229], consistency=
       Eventual] from:
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-450113151]
6  class: TreeBroker | message: holding Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=229], consistency=
       Eventual]
7
8  class: TreeBroker | message: Received Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=230], consistency=
       Causal] from:
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-450113151]
9  class: TreeBroker | message: holding Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=230], consistency=
       Causal]
10
11 class: TreeBroker | message: Received Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=231], consistency=
       Causal] from:
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-450113151]
```

```
12 class: TreeBroker | message: holding Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=231], consistency=
       Causal]
13
14 class: TreeBroker | message: Received Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=232], consistency=
       Eventual] from:
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-450113151]
15 class: TreeBroker | message: holding Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=232], consistency=
       Eventual]
16
17 class: TreeBroker | message: Received Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=233], consistency=
       Linear] from:
       Actor[akka://PubSubSystem@10.16.1.108:3001/user/treeBroker#-450113151]
18 class: TreeBroker | message: holding Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=233], consistency=
       Linear]
19                                          .
20                                          .
21                                          .
22 class: TreeBroker | message: reconfiguration completed in 91 ms
23
24 class: TreeBroker | message: releasing messages on hold for topic
       'topic1'...
25 class: TreeBroker | message: releasing Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=229], consistency=
       Eventual]
26 class: TreeBroker | message: releasing Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=230], consistency=
       Causal]
27 class: TreeBroker | message: releasing Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=231], consistency=
       Causal]
28 class: TreeBroker | message: releasing Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=232], consistency=
       Eventual]
```

```
29 class: TreeBroker | message: releasing Metadata [topic='topic1',
       id=TransactionIdentifier [replicaID=0, messageID=233], consistency=
       Linear]
```

In this log, broker 7 detects broker 3 has failed (line 1). As we can observe, during the time between detecting broker 3's failure until it determines the reconfiguration is completed (line 22), it received five *Metadata* messages which were put on hold due to an ongoing reconfiguration. Once it determines the reconfiguration was completed (line 22), it releases the messages in the same order *FIFO* order of reception.

VIII

Handling rejected *ParentRequest* and detecting missed messages.

Listing VIII.1: Broker 5 loses connection to broker 1 and has to execute multiple rounds of connection attempts. Then synchronizes its buffer with the new parent's (broker 0) and detects some messages have been missed.

```
1  class: TreeBroker | message: received LinearProcessed [topic='topic2',
       id=TransactionIdentifier [replicaID=2, messageID=445]] from:
       akka://PubSubSystem@10.16.1.2:3001/user/treeBroker
2  class: Broker | message: unlocking id= TransactionIdentifier
       [replicaID=2, messageID=445]
3  class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=445], delivery=Linear, keyItems=[-162550485],
       topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
4
5  class: ConnectionMonitor | message: 08:44:17: lost connection to
       akka://PubSubSystem@10.16.1.2:3001/user/treeBroker
6
7  class: TreeBroker | message: removing neighbor
       akka://PubSubSystem@10.16.1.2:3001/user/treeBroker. Can receive 2
       connections
8  class: TreeBroker | message: expecting connections from 1 new neighbors
9  class: TreeBroker | ERROR: parent failed, sending ParentRequest
       [subscribedTopics=[topic2], breakDegree=false] to
       akka://PubSubSystem@10.16.1.14:3000/user/treeBroker
10 class: TreeBroker | message: sending ParentRequest
       [subscribedTopics=[topic2], breakDegree=false] to:
       akka://PubSubSystem@10.16.1.14:3000/user/treeBroker
11
```

```
12 class: TreeBroker | message: setting a timer for expected connections
13
14 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=453], delivery=Linear, keyItems=[-162550419],
       topic='topic2'] from:
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
15 class: TreeBroker | message: holding Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=453], delivery=Linear, keyItems=[-162550419],
       topic='topic2']
16
17 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=454], delivery=Eventual,
       keyItems=[-162550472], topic='topic2'] from:
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
18 class: TreeBroker | message: holding Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=454], delivery=Eventual,
       keyItems=[-162550472], topic='topic2']
19
20 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=455], delivery=Eventual,
       keyItems=[-162550402], topic='topic2'] from:
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
21 class: TreeBroker | message: holding Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=455], delivery=Eventual,
       keyItems=[-162550402], topic='topic2']
22
23 class: TreeBroker | message:
       akka://PubSubSystem@10.16.1.14:3000/user/treeBroker declined parent
       request
24 class: TreeBroker | message: expecting connections from 1 new neighbors
25 class: TreeBroker | ERROR: parent failed, sending ParentRequest
       [subscribedTopics=[topic2], breakDegree=false] to
       akka://PubSubSystem@10.16.1.54:3004/user/treeBroker
26
27 class: TreeBroker | message: setting a timer for expected connections
28
29 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=456], delivery=Eventual,
       keyItems=[-162550414], topic='topic2'] from:
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
```

```
30 class: TreeBroker | message: holding Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=456], delivery=Eventual,
       keyItems=[-162550414], topic='topic2']
31
32 class: TreeBroker | message:
       akka://PubSubSystem@10.16.1.54:3004/user/treeBroker declined parent
       request
33 class: TreeBroker | message: expecting connections from 1 new neighbors
34 class: TreeBroker | ERROR: parent failed, sending ParentRequest
       [subscribedTopics=[topic2], breakDegree=false] to
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
35 class: TreeBroker | message: sending ParentRequest
       [subscribedTopics=[topic2], breakDegree=false] to:
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
36
37 class: TreeBroker | message: setting a timer for expected connections
38
39 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=457], delivery=Eventual,
       keyItems=[-162550395], topic='topic2'] from:
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
40 class: TreeBroker | message: holding Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=457], delivery=Eventual,
       keyItems=[-162550395], topic='topic2']
41
42 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=458], delivery=Linear, keyItems=[-162550461],
       topic='topic2'] from:
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
43 class: TreeBroker | message: holding Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=458], delivery=Linear, keyItems=[-162550461],
       topic='topic2']
44
45 class: TreeBroker | message: 08:44:17:
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker accepted parent
       request, connecting...
46 class: TreeBroker | message: adding neighbor
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker. Can receive 2
       connections
```

```
47 class: TreeBroker | message: removing ancestor
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
48
49 class: TreeBroker | message: received parent buffer
50
51 class: TreeBroker | message:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
       subscribed topic 'topic2'
52 class: TreeBroker | message: parent subscribes to the topic => topic2
53 class: TreeBroker | message: handling buffer for topic: topic2
54
55 class: TreeBroker | message: last received by parent:
       {1=TransactionIdentifier [replicaID=1, messageID=451],
       2=TransactionIdentifier [replicaID=2, messageID=455]}
56 class: TreeBroker | message: my buffer : [TransactionIdentifier
       [replicaID=1, messageID=445], TransactionIdentifier [replicaID=2,
       messageID=442], TransactionIdentifier [replicaID=1, messageID=446],
       TransactionIdentifier [replicaID=2, messageID=443],
       TransactionIdentifier [replicaID=1, messageID=447],
       TransactionIdentifier [replicaID=2, messageID=444],
       TransactionIdentifier [replicaID=1, messageID=448],
       TransactionIdentifier [replicaID=2, messageID=445],
       TransactionIdentifier [replicaID=1, messageID=449],
       TransactionIdentifier [replicaID=2, messageID=446],
       TransactionIdentifier [replicaID=1, messageID=450],
       TransactionIdentifier [replicaID=2, messageID=447],
       TransactionIdentifier [replicaID=1, messageID=451],
       TransactionIdentifier [replicaID=2, messageID=448],
       TransactionIdentifier [replicaID=1, messageID=452]]
57 class: TreeBroker | message: last received by me:
       {1=TransactionIdentifier [replicaID=1, messageID=452],
       2=TransactionIdentifier [replicaID=2, messageID=448]}
58 class: TreeBroker | message: parent missed: TransactionIdentifier
       [replicaID=1, messageID=452]
59
60 class: TreeBroker | message: missed: {topic2={2=TransactionIdentifier
       [replicaID=2, messageID=455]}}
```

```
61 class: TreeBroker | message: sending LostMessageRequest
       [missed={topic2={2=TransactionIdentifier [replicaID=2,
       messageID=448]}}, lastProcessedIds={topic2={1=TransactionIdentifier
       [replicaID=1, messageID=447], 2=TransactionIdentifier [replicaID=2,
       messageID=445]}}] to:
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
62
63 class: TreeBroker | message: parent missed [Metadata
       [id=TransactionIdentifier [replicaID=1, messageID=452],
       delivery=Linear, keyItems=[-162550467], topic='topic2']] from topic
       'topic2'
64 class: TreeBroker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=452], delivery=Linear, keyItems=[-162550467],
       topic='topic2'] to:
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
65
66 class: TreeBroker | message: connection to
       akka://PubSubSystem@10.16.1.43:3002/user/treeBroker established
67
68 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=449], delivery=Causal, keyItems=[-162550483],
       topic='topic2'] from:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
69
70 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=450], delivery=Eventual,
       keyItems=[-162550436], topic='topic2'] from:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
71 class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=449], delivery=Causal, keyItems=[-162550483],
       topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
72
73 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=451], delivery=Eventual,
       keyItems=[-162550448], topic='topic2'] from:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
```

```
74 class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=450], delivery=Eventual,
       keyItems=[-162550436], topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
75 class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=451], delivery=Eventual,
       keyItems=[-162550448], topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
76
77 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=452], delivery=Eventual,
       keyItems=[-162550398], topic='topic2'] from:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
78
79 class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=452], delivery=Eventual,
       keyItems=[-162550398], topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
80 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=453], delivery=Eventual,
       keyItems=[-162550445], topic='topic2'] from:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
81 class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=453], delivery=Eventual,
       keyItems=[-162550445], topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
82
83 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=454], delivery=Eventual,
       keyItems=[-162550446], topic='topic2'] from:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
84 class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=454], delivery=Eventual,
       keyItems=[-162550446], topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
85
86 class: TreeBroker | message: Received Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=455], delivery=Linear, keyItems=[-162550421],
       topic='topic2'] from:
       Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
```

```
87 class: Broker | message: sending Metadata [id=TransactionIdentifier
       [replicaID=2, messageID=455], delivery=Linear, keyItems=[-162550421],
       topic='topic2'] to
       Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
88
89 class: TreeBroker | message: reconfiguration completed in 579 ms
90
91 class: TreeBroker | message: Verifying pending linear transactions for
       topic: topic2
92 class: TreeBroker | message: locked transactions [Metadata
       [id=TransactionIdentifier [replicaID=1, messageID=449],
       delivery=Linear, keyItems=[-162550437], topic='topic2'], Metadata
       [id=TransactionIdentifier [replicaID=1, messageID=450],
       delivery=Linear, keyItems=[-162550462], topic='topic2'], Metadata
       [id=TransactionIdentifier [replicaID=1, messageID=452],
       delivery=Linear, keyItems=[-162550467], topic='topic2'], Metadata
       [id=TransactionIdentifier [replicaID=2, messageID=448],
       delivery=Linear, keyItems=[-162550480], topic='topic2']]
93 class: TreeBroker | message: waiting for processed confirmation for
       transaction with id: TransactionIdentifier [replicaID=1,
       messageID=449]
94 class: TreeBroker | message: waiting for processed confirmation for
       transaction with id: TransactionIdentifier [replicaID=1,
       messageID=450]
95 class: TreeBroker | message: waiting for processed confirmation for
       transaction with id: TransactionIdentifier [replicaID=1,
       messageID=452]
96 class: TreeBroker | message: waiting for processed confirmation for
       transaction with id: TransactionIdentifier [replicaID=2,
       messageID=448]
97
98 class: TreeBroker | message: releasing messages on hold for topic
       'topic2'...
99 class: TreeBroker | message: releasing Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=453], delivery=Linear, keyItems=[-162550419],
       topic='topic2']
100 class: TreeBroker | message: releasing Metadata [id=TransactionIdentifier
       [replicaID=1, messageID=454], delivery=Eventual,
       keyItems=[-162550472], topic='topic2']
```

```
101 class: TreeBroker | message: releasing Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=455], delivery=Eventual,
        keyItems=[-162550402], topic='topic2']
102 class: TreeBroker | message: releasing Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=456], delivery=Eventual,
        keyItems=[-162550414], topic='topic2']
103 class: TreeBroker | message: releasing Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=457], delivery=Eventual,
        keyItems=[-162550395], topic='topic2']
104 class: TreeBroker | message: releasing Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=458], delivery=Linear, keyItems=[-162550461],
        topic='topic2']
105
106 class: TreeBroker | message: received LinearProcessed [topic='topic2',
        id=TransactionIdentifier [replicaID=1, messageID=449]] from:
        akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
107 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=453], delivery=Linear, keyItems=[-162550419],
        topic='topic2'] to
        Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
108
109 class: TreeBroker | message: received LinearProcessed [topic='topic2',
        id=TransactionIdentifier [replicaID=1, messageID=450]] from:
        akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
110 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=454], delivery=Eventual,
        keyItems=[-162550472], topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
111 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=455], delivery=Eventual,
        keyItems=[-162550402], topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
112
113 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=455], delivery=Eventual,
        keyItems=[-162550402], topic='topic2'] to
        Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
```

```
114 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=454], delivery=Eventual,
        keyItems=[-162550472], topic='topic2'] to
        Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
115 class: TreeBroker | message: received LinearProcessed [topic='topic2',
        id=TransactionIdentifier [replicaID=2, messageID=448]] from:
        akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
116 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=456], delivery=Eventual,
        keyItems=[-162550414], topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
117 class: Broker | message: unlocking id= TransactionIdentifier
        [replicaID=1, messageID=449]
118 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=457], delivery=Eventual,
        keyItems=[-162550395], topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
119 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=449], delivery=Linear, keyItems=[-162550437],
        topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
120 class: Broker | message: unlocking id= TransactionIdentifier
        [replicaID=2, messageID=448]
121 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=456], delivery=Eventual,
        keyItems=[-162550414], topic='topic2'] to
        Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
122 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=2, messageID=448], delivery=Linear, keyItems=[-162550480],
        topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
123 class: Broker | message: unlocking id= TransactionIdentifier
        [replicaID=1, messageID=450]
124 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=457], delivery=Eventual,
        keyItems=[-162550395], topic='topic2'] to
        Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
```

```
125 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=458], delivery=Linear, keyItems=[-162550461],
        topic='topic2'] to
        Actor[akka://PubSubSystem@10.16.1.43:3002/user/treeBroker#1644622344]
126 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=450], delivery=Linear, keyItems=[-162550462],
        topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
127
128 class: TreeBroker | message: received LinearProcessed [topic='topic2',
        id=TransactionIdentifier [replicaID=1, messageID=452]] from:
        akka://PubSubSystem@10.16.1.43:3002/user/treeBroker
129 class: Broker | message: unlocking id= TransactionIdentifier
        [replicaID=1, messageID=452]
130 class: Broker | message: sending Metadata [id=TransactionIdentifier
        [replicaID=1, messageID=452], delivery=Linear, keyItems=[-162550467],
        topic='topic2'] to
        Actor[akka://middleware@10.16.1.6:2501/user/middleware#-1438451775]
```

Listing VIII.2: Broker 0 accepts broker 4's request and rejects broker 5's request because its *residualDegree* is zero.

```
1 class: ConnectionMonitor | message: 08:44:17: lost connection to
      akka://PubSubSystem@10.16.1.2:3001/user/treeBroker
2 class: TreeBroker | message: removing neighbor
      akka://PubSubSystem@10.16.1.2:3001/user/treeBroker. Can receive 1
      connections
3 class: TreeBroker | message: expecting connections from 1 new neighbors
4 class: TreeBroker | message: setting a timer for expected connections
5 class: TreeBroker | message: 08:44:17: received ParentRequest
      [subscribedTopics=[topic1], breakDegree=false] from
      Actor[akka://PubSubSystem@10.16.1.54:3004/user/treeBroker#1051990613]
      my residual degree is 1
6 class: TreeBroker | message: sending ParentResponse [accepted=true] to:
      akka://PubSubSystem@10.16.1.54:3004/user/treeBroker
7 class: TreeBroker | message:
      Actor[akka://PubSubSystem@10.16.1.54:3004/user/treeBroker#1051990613]
      subscribed topic 'topic1'
```

```
8  class: TreeBroker | message: sending ParentBuffer
       [buffer={topic1={0=TransactionIdentifier [replicaID=0,
       messageID=451], 3=TransactionIdentifier [replicaID=3, messageID=454],
       4=TransactionIdentifier [replicaID=4, messageID=451]}}] to:
       akka://PubSubSystem@10.16.1.54:3004/user/treeBroker
9  class: TreeBroker | message: adding neighbor
       akka://PubSubSystem@10.16.1.54:3004/user/treeBroker. Can receive 1
       connections
10 class: TreeBroker | message: 08:44:17: received ParentRequest
       [subscribedTopics=[topic2], breakDegree=false] from
       Actor[akka://PubSubSystem@10.16.1.56:3005/user/treeBroker#833134505]
       my residual degree is 0
11 class: TreeBroker | message: declined parent request, cannot receive
       further connections
12 class: TreeBroker | message: sending ParentResponse [accepted=false] to:
       akka://PubSubSystem@10.16.1.56:3005/user/treeBroker
13 class: TreeBroker | message: expected 1 connections
14 class: TreeBroker | message: still waiting for 0 connections
15 class: TreeBroker | message: canceling timer
16 class: TreeBroker | message: ongoingReconfiguration: 1
```

Let's analyze the log file from Listing VIII.1 in Annex VIII, which corresponds to an excerpt of broker 5's log file. In this scenario, broker 5 detected its parent, broker 1, failed (line 1) implying it must attempt to connect to a broker in its tree view, which contains brokers 0, 4, 2 and 3. It first attempts to connect to broker 0 (line 5), which is its first ancestor. Meanwhile it receives *Metadata* messages from a client, which it correctly puts on hold. When the *ParentResponse* arrives (line 19), it states broker 0 declined its request. Therefore, it must attempt to connect to another broker in its tree view. Broker 0 rejected the request because, as can be observe in Listing VIII.2, it had already accepted broker 4's request (line 5), which lead its *residualDegree* to become 0, as it stated when receiving broker 5's request, "my residual degree is **0**" (line 10). As highlighted in Section 4.2.3, when a *ParentRequest* is rejected, the address of the broker who rejected it, is added to the *declinedBy* set in order to prevent brokers from attempting successive connections to the same broker. Since broker 0's address is now contained int it then, it will attempt to connect to its sibling, which is this case is broker 4 (line 21). Since broker 4 is not subscribed to any of broker 5's subscribed topics, the request must be rejected, which we can observe it was in line 28. Once again its address is added to the *declinedBy* set. Now, broker 0 will attempt to connect to a broker in the *ancestors* set, whose address is not contained in *declinedBy*. It selected broker 2 and sent a *ParentRequest* (line 31). Once again, it received messages from a client which it correctly put on hold. When the *ParentResponse* arrives (line 41), it stated that broker 2 accepted its request, adding broker 2 to the neighbors set, and must expect

to receive information about the last received messages in the new parent's buffer.

When that information is received (line 45), the new parent is added to *topic2*'s subscribers set (line 47-48). In the parent's buffer, the last received messages where *(1,451)* and *(2,455)* (line 51), corresponding to *(replicaID,messageID)* in the log file. Afterwards broker 5 iterated over its buffer to map its last received messages, for each tuple *(replicaID, messageID)*. In this case, *(1,452)* and *(2,448)* were the last received messages (line 53). By comparing its last received messages' identifiers with the parent's, it detects parent only missed message with identifier *(1,452)* since parent's last received message with *replicaID = 1* was *(1,451)*, retransmitting it to the parent. Afterwards, it states it missed message with identifier *(2,455)* (line 56), putting it in the *missedMessages* map implying that reconfiguration is only completed when the message with that identifier is received. Then, it sends a *LostMessageRequest* to its parent containing a mapping of the last received identifier per *replicaID* and the identifiers of the last processed total order messages contained in its buffer (line 57). In this case, the map contains an entry for *replicaID=2* and the identifier *(2,448)*, which is the last received. When the parent receives this message, it retransmits all messages with *replicaID=2* and *messageID* greater than 248 contained in the parent's buffer. Therefore, broker 5 must receive messages *(2,449)*, *(2,450)*, *(2,451)*, *(2,452)*, *(2,453)*, *(2,454)* and *(2,455)*, in that order. As we can observe, from line 64 to 82, these messages were received and sent in the correct order. As identifier *(2,455)* was received in line 82 then, broker 5 is able to determine the reconfiguration is completed and is allowed to verify if any total order messages are currently being processed (lines 87-89).

In lines 89 to 92, the log states broker 5 is waiting for processed confirmation for message identifier *(1,449)*, *(1,450)*, *(1,452)* and *(2,448)*, which, in other words, means its parent is now responsible for ensuring the corresponding *LinearProcessed* notifications are sent. These notifications will be received because, when broker 5 sent the *LostMessageRequest*, it indicated to its parent the identifiers of the last processed total order messages, allowing the parent to send the ones it is waiting for. In lines 102, 105, 111 and 124, the notifications where received and the respective messages are delivered to broker 5's children. At this point, we are able to conclude no messages were lost due to broker 1's failure and brokers correctly performed all steps to ensure they were neither lost nor processed out of order, even when multiple *ParentRequest* were rejected.

Tables IX.1 and IX.2 show an excerpt of the file generated by replicas 1 and 2, with the delivery order for operations performed in account **162,550,483**.

IX

Example message ordering for a specific key.

Table IX.1: Excerpt of file generated by replica 1. This table shows the delivery order of messages for account number 162,550,483

replicaID	messageID	Delivery Guarantee	keyItem
1	42	CAUSAL	162,550,483
2	47	CAUSAL	162,550,483
2	64	TOTAL	162,550,483
2	84	EVENTUAL	162,550,483
1	98	EVENTUAL	162,550,483
2	95	CAUSAL	162,550,483
2	102	TOTAL	162,550,483
1	131	EVENTUAL	162,550,483
2	131	EVENTUAL	162,550,483
2	133	CAUSAL	162,550,483
1	189	TOTAL	162,550,483
2	265	CAUSAL	162,550,483
2	295	TOTAL	162,550,483
2	399	EVENTUAL	162,550,483
2	449	CAUSAL	162,550,483
1	464	EVENTUAL	162,550,483
2	465	TOTAL	162,550,483
1	501	CAUSAL	162,550,483
2	519	TOTAL	162,550,483
2	545	CAUSAL	162,550,483
1	675	TOTAL	162,550,483
1	840	EVENTUAL	162,550,483
2	907	EVENTUAL	162,550,483
1	909	TOTAL	162,550,483
1	938	TOTAL	162,550,483

Table IX.2: Excerpt of file generated by replica 2. This table shows the delivery order of messages for account number 162,550,483

replicaID	messageID	Delivery Guarantee	keyItem
1	42	CAUSAL	162,550,483
2	47	CAUSAL	162,550,483
2	64	TOTAL	162,550,483
2	84	EVENTUAL	162,550,483
2	95	CAUSAL	162,550,483
1	98	EVENTUAL	162,550,483
2	102	TOTAL	162,550,483
2	131	EVENTUAL	162,550,483
1	131	EVENTUAL	162,550,483
2	133	CAUSAL	162,550,483
1	189	TOTAL	162,550,483
2	265	CAUSAL	162,550,483
2	295	TOTAL	162,550,483
2	399	EVENTUAL	162,550,483
2	449	CAUSAL	162,550,483
1	464	EVENTUAL	162,550,483
2	465	TOTAL	162,550,483
1	501	CAUSAL	162,550,483
2	519	TOTAL	162,550,483
2	545	CAUSAL	162,550,483
1	675	TOTAL	162,550,483
1	840	EVENTUAL	162,550,483
2	907	EVENTUAL	162,550,483
1	909	TOTAL	162,550,483
1	938	TOTAL	162,550,483