TÉCNICO LISBOA

LoCaPS: Localized Causal Publish-Subscribe

Filipa Salema Roseta Pedrosa

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. Sérgio Marco Duarte

October 2020

Acknowledgments

First, I would like to express my gratitude for all the support that my thesis advisor, Professor Luís Rodrigues of the Instituto Superior Técnico at Universidade de Lisboa, provided me. Thank you for the countless discussions, meetings, and all the feedback throughout the year. All of it made this thesis possible.

I also want to express my gratitude towards my colleagues from room 501 of INESC. You made my days there so much more fun and were always very supportive, making me feel very welcomed.

Finally, I want to thank my dear friends and family, who always supported me during my student journey throughout all these years, and will continue to do so. Thank you, Mariana Farias, Mariana Rosa, Dorin Gujuman, and Tiago Gonçalves for being there for me when I most needed help. Thank you, Pedro Remédios, for doing almost every class project with me and making me laugh every time. Enduring those hard years of IST with you made it much more manageable. Thank you to my wonderful sister, Leonor Pedrosa, and brothers, Petr Terletskiy and Nicolas Pedrosa, for always being so cheerful and making me forget the bad days. Coming this far would not have been possible without any of you.

This work was partially supported by the FCT via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271) and project UIDB/ 50021/ 2020.

Abstract

This thesis addresses the problem of offering low latency to subscribers in a reliable causal publish-subscribe system. The publish-subscribe abstraction has emerged as a fundamental tool to build distributed systems that preserve strong decoupling among information consumers and producers. The most common strategy to implement this abstraction in large-scale systems consists of using a network of message brokers that relay events from publishers to consumers. These brokers require coordination to offer quality of service guarantees to message subscribers. In most systems that enforce reliability guarantees, a subscriber needs to wait until its subscription has been propagated to every broker in the system, and known by all relevant publishers, before starting to receive events. Curiously, this happens even when a subscription is covered by a previously deployed one. To the best of our knowledge, previous reliable causal systems do not focus on reducing the observed latency by subscribers nor on the coverage relationships between subscriptions. In this thesis, we study the properties that need to be satisfied to reduce subscription latency. We also propose a new publish-subscribe system that leverages causal order multicast to offer low subscription latency when subscriptions achieve such properties. Experimental results show that our algorithm can outperform previous solutions in terms of subscription latency.

Keywords: Distributed Systems, Publish-Subscribe, Causality, Reliability, Latency

Resumo

Esta tese aborda o problema de oferecer baixa latência aos subscritores num sistema fiável e causal de edição-subscrição. A abstração de edição-subscrição emergiu como uma ferramenta fundamental para construir sistemas distribuídos que preservam dissociação forte entre consumidores e produtores de informação. A estratégia mais comum para concretizar esta abstração em sistemas de larga escala consiste em usar uma rede de intermediários, que transmitem eventos dos editores para os consumidores. Estes nós intermediários necessitam de coordenação para oferecer garantias de qualidade de serviço aos subscritores de mensagens. Na maior parte dos sistemas que oferecem garantias de fiabilidade, um subscritor precisa de esperar até a sua subscrição ter sido propagada a todos os intermediários no sistema, e ser conhecida por todos os editores relevantes, antes de poder começar a receber eventos. Curiosamente, isto acontece mesmo quando uma subscrição está coberta por uma anteriormente instalada. Tanto quanto sabemos, trabalho anterior em sistemas de edição-subscrição fiáveis e causais não se foca em reduzir a latência observada pelos subscritores nem nas relações de cobertura entre subscrições. Nesta tese, estudamos as propriedades que têm de ser satisfeitas para reduzir a latência de subscrição. Também propomos um novo sistema de edição-subscrição que tira partido de difusão causal em grupo para oferecer baixa latência quando essas propriedades são verificadas. Resultados experimentais mostram que o nosso algoritmo tem melhor desempenho que soluções anteriores em termos de latência de susbcrição.

Palavras-chave: Sistemas Distribuídos, Edição-Subscrição, Causalidade, Fiabilidade, Latência

Contents

Acknowledgments iii

Abstract v

Resumo vi

List of Tables x

List of Figures x

Glossary 1

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Results 3

1.4 Research History 3

1.5 Thesis Outline 4

2 Background **5**

2.1 Definitions 5

2.1.1 Event Graph and Subscription History 5

2.2 Properties of Publish-Subscribe Systems 7

2.2.1 Delivery Order 7

2.2.2 Reliability 8

2.2.3 Other Relevant Features 9

2.2.4 Network Overlays and Fault Tolerance 9

2.2.5 Event Routing 10

2.3 Subscription Latency 11

2.3.1 Reducing Subscription Latency 12

2.4 Summary 12

3 Related Work **15**

3.1 Relevant Properties 15

3.2 Systems 16

3.2.1 LoCaMu 16

3.2.2 SIENA 17

3.2.3 Gryphon 18

3.2.4 δ-fault-tolerant 19

3.2.5 Dynamic Message Ordering for Publish-Subscribe 21

3.2.6 XNET 22

3.2.7 Semi-Probabilistic Publish-Subscribe 23

3.2.8 GEPS 24

3.2.9 JEDI 25

3.2.10 Sequencing Graph 26

3.2.11 Epidemic Algorithms for Publish-Subscribe 26

3.2.12 VCube-PS 28

3.2.13 P2PPS 28

3.2.14 Publish-Subscribe Middleware with DSM 29

3.2.15 Comparison 29

3.3 Summary 32

4 Subscription Semantics **33**

4.1 Semantics 33

4.2 Sufficient Conditions for Semantics Enforcement 35

4.2.1 System Model 35

4.2.2 Subscription Stability 37

4.2.3 Stability-Based Conditions 37

4.2.4 Evaluating Full Stability 38

4.2.5 Impact on Latency 39

4.3 A Necessary (and Sufficient) Condition for Semantics Enforcement 39

4.3.1 Causality-Based Condition 39

4.3.2 Leveraging Causality 40

4.3.3 Correctness 41

4.4 Leveraging Coverage 43

4.4.1 Single-Prefix Coverage 44

4.4.2 Multi-Prefix Coverage 45

4.5 Subscription Coverage Optimized Algorithm . . . 47
4.5.1 Correctness . . . 48
4.6 Summary . . . 50

5 LoCaPS **51**
5.1 Goals . . . 51
5.2 LoCaPS . . . 51
5.2.1 LoCaMu . . . 52
5.2.2 LoCaPS Algorithm . . . 54
5.3 Implementation . . . 55
5.3.1 Development Environment . . . 56
5.3.2 Framework . . . 56
5.3.3 LoCaPS . . . 56
5.4 Summary . . . 57

6 Evaluation **59**
6.1 Goals . . . 59
6.2 Experimental Settings . . . 59
6.3 Analyzing the Sufficient and Necessary and Sufficient Conditions . . . 60
6.4 Analyzing Subscription Coverage . . . 62
6.5 Analyzing Localized Algorithms . . . 64
6.6 Discussion . . . 64
6.7 Summary . . . 65

7 Conclusions **67**
7.1 Future Work . . . 67

Bibliography **68**

List of Tables

3.1 Systems' guarantees and strategies for each topic. 32

List of Figures

2.1 Message and execution flow in a Pub/Sub system. 6
2.2 Examples of subscription event histories. 7

3.1 Addressed concerns by respective system. 30

4.1 Subscription semantic examples. 34
4.2 Single path subscription coverage. 44
4.3 Multi-path subscription coverage. 45

5.1 LoCaMu's underlying acyclic graph and extended graph. 53

6.1 LoCaPS vs Delta under different settings 61
6.2 LoCaPS vs the Delta and Gryphon algorithms 62
6.3 LoCaPS vs Delta under different neighborhood scenarios 63

Chapter 1

Introduction

This thesis addresses the problem of offering low latency to subscribers in a reliable causal publish-subscribe system. Typically, systems that enforce these properties have a higher latency than others with weaker guarantees in terms of ordering and reliability. For properties to be enforced they require the subscription to be propagated to every broker in the overlay. Not surprisingly, this results in higher latency. On the other hand, best-effort systems have used covering relations between subscriptions to reduce latency. However, to the best of our knowledge, previous reliable publish-subscribe systems have not been able to exploit coverage to speed up subscriptions. In this thesis, we define the necessary and sufficient conditions needed to achieve reliable causal delivery. Interestingly, these conditions do not require every publisher to receive the subscription to ensure reliable delivery to the subscribers. We then analyze how systems can use subscription coverage to decrease latency, and we propose an algorithm that offers both strong delivery guarantees and low subscription latency.

1.1 Motivation

The publish-subscribe abstraction [EFGK03] has emerged as a fundamental tool to build distributed systems that preserve strong decoupling among participants. These can be either information producers or consumers. Producers are named *publishers* and generate *events*. An event is a data unit that can be modeled as a tuple containing multiple fields. Consumers are named *subscribers*, which receive events they *subscribe* to. Participants can express interest in a given content by subscribing to a topic. Systems that support this type of subscription are referred to as topic-based publish-subscribe systems. Alternatively, participants can express constraints on the event's fields. This type of system is called a content-based publish-subscribe system.

The most common strategy to implement a large-scale publish-subscribe system consists of using a network of message brokers that can relay events from publishers to consumers. Publishers connect to a broker of this network and forward events to it, while subscribers connect to other brokers and express interest in events with their subscriptions. Brokers coordinate with each other to make sure events are forwarded in the broker overlay, from publishers to interested subscribers. Messages are matched with subscriptions by brokers to verify if they are relevant to a subscriber. Brokers need to coordinate to offer quality of service guarantees to message subscribers. Typically offered properties are reliable (gapless) FIFO delivery, reliable causal delivery, or even reliable ordered delivery of events.

In this thesis, we are particularly interested in studying the subscription latency in reliable publish-subscribe systems. This latency corresponds to the delay between the time a subscriber performs a subscription and the time at which it receives the first event from any publisher. Said delay varies between different systems: it is a function of the number of steps the algorithm needs to execute, in the routing broker overlay, to deploy the state required to enforce the desired reliability guarantees.

We consider only two variants of reliable delivery: gapless FIFO delivery (GFD) and gapless causal delivery (GCD). Informally, GFD ensures that, once a subscriber starts receiving events from a given publisher, it receives all subsequent events produced by that publisher that match the subscription. Gryphon [ZSB04, BSB$^+$02] is a well-known system that offers this type of reliability. GFD is defined for each publisher, regardless of the interactions between publishers through their events. These interactions are rather common and occur when a client is both a publisher and a subscriber. GCD is a stronger reliability criterion for publish-subscribe systems that avoids anomalies resulting from missing cause-effect relations among events. The cause-effect relations are established from several interactions among publishers. Examples of systems that offer GCD are [CDNF01, PRS96, PLTP08, NDA$^+$14, dAADJ$^+$19].

This work answers the following interesting question: *given that GCD is stronger than GFD, is the subscription latency for the former necessarily greater than for the latter?* Surprisingly, we show that the answer is *no*. In fact, by ensuring causality in the propagation of messages among event brokers, we demonstrate that GCD can be guaranteed quickly. It becomes possible to enforce GCD as soon as a condition is met: All brokers *in a single path*, from *one* of the publishers to the subscriber, must know the subscription. This property is substantially weaker than the one considered by existing implementations, such as [KJ09, dAADJ$^+$19]. These require the subscription to be known by all brokers in *all paths* from *all* publishers.

One of the core characteristics of publish-subscribe systems is the covering relation that

exists between subscriptions. Informally, subscription S_a is said to cover subscription S_b if all events that match S_b also match S_a. Another enthralling question we address is the following: *can coverage relations be used to decrease the subscription latency? If yes, in which conditions?* Covering relations have been used to decrease subscription latency in best-effort systems [CRW01, SDJ16]. However, to the best of our knowledge, previous reliable publish-subscribe systems have not been able to exploit coverage to speed up subscriptions. Therefore, we also study the conditions that allow a system to decrease subscription latency for covered subscriptions.

Finally, we introduce this dissertation's resulting algorithm, which supports GCD and makes use of our findings. LoCaPS expands previous work on localized causal multicast [SR19] to implement efficient subscription protocols. These offer low subscription latency when brokers observe suitable coverage relations between a new subscription and previously deployed ones. Experimental results show that by taking into consideration coverage relationships, the system can considerably reduce the latency observed by subscribers.

1.2 Contributions

This thesis makes two contributions:

- It states the necessary and sufficient conditions for implementing Gapless FIFO Delivery and Gapless Causal Delivery;
- It proposes a novel algorithm, named LoCaPS, which leverages these conditions and the notion of subscription coverage to reduce the subscription latency;

1.3 Results

The work described in this dissertation has achieved the following results:

- An implementation of LoCaPS on the PeerSim [MJ09] simulator;
- An experimental evaluation of the system and a comparison of its performance with other algorithms that provide similar properties.

1.4 Research History

This work was performed in the context of a research project, named Cosmos, that aims at offering causal consistency on the network edge. My work builds on the work of Valter Santos,

which has designed LoCaMu[SR19], acting as a Localized Reliable Causal Multicast layer to propagate both subscriptions and events in a broker network. In LoCaMu, multicast groups are static and, therefore, on-line subscriptions are not supported. We were interested in studying if LoCaMu could help in the implementation of a publish-subscribe system that could offer strong guarantees and fast subscriptions. Our system LoCaPS has been implemented as an extension to LoCaMu.

This work was partially supported by the FCT via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271) and project UIDB/ 50021/ 2020.

1.5 Thesis Outline

The rest of the document is organized as follows: Chapter 2 presents an overview of the different properties and characteristics of publish-subscribe systems; Chapter 3 describes all the background related to our work, categorizing several publish-subscribe systems under distinct properties; Chapter 4 defines conditions to enforce GCD semantic and presents a novel algorithm that supports it, as well as optimizations leveraging already deployed subscriptions; Chapter 5 describes LoCaPS, an implementation that materializes our findings; Chapter 6 presents the results of the evaluation performed on each algorithm and compares our system's performance with previous work and Chapter 7 concludes this dissertation, with a summary of the presented contributions.

Chapter 2

Background

This chapter begins with Section 2.1 by introducing concepts relevant to describe characteristics of publish-subscribe systems. Section 2.2 presents an overview of the essential properties of these systems. Then Section 2.3 describes in detail what subscription latency is and which methods are usually employed by systems aiming to reduce it.

2.1 Definitions

The role of a publish-subscribe system is to deliver messages produced by publishers to all interested subscribers. Different systems ensure distinct properties on the history of events relayed to subscribers, in terms of reliability and order. The set of properties for message delivery defines the level of service a system provides. In turn, these properties have an impact on the time taken for a subscriber to start receiving events after a subscription is issued. In this section, we make an overview of the essential properties provided by several publish-subscribe systems, of how these properties affect subscription latency, and of the different techniques that can be used to reduce this latency.

2.1.1 Event Graph and Subscription History

Before we discuss the properties of publish-subscribe systems in more detail, we introduce the concepts of *event graph*, *subscription history*, subscription *starting cut*, and subscription *ending cut*. These concepts are introduced by relying on the example depicted in Figure 2.1. We assume a publish-subscribe system, with multiple publishers, where each one produces a sequence of events.

Events can be causally related. We use the notion of causal order from Lamport[Lam19], where if two events $e_{1.1}$ and $e_{1.2}$ produced by the same publisher p_1, where $e_{1.2}$ is produced

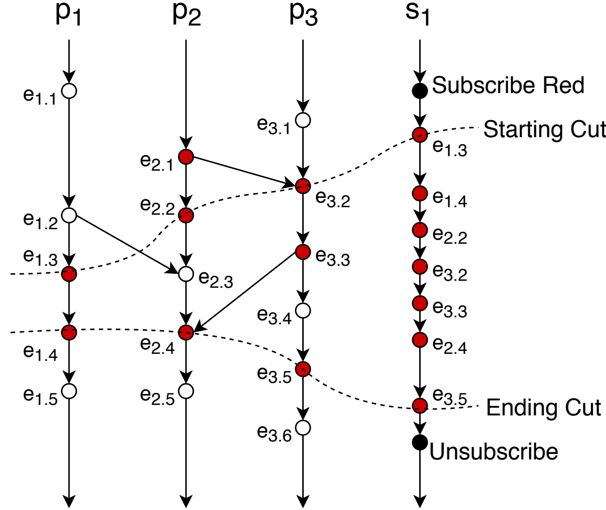


Figure 2.1: Message and execution flow in a Pub/Sub system.

after $e_{1.1}$, then $e_{1.2}$ will be causally dependent of $e_{1.1}$, denoted $e_{1.1} \rightarrow e_{1.2}$. Publishers can also subscribe to events from other publishers, creating potential cause-effect relations between events originated from such different sources. Again, we use Lamport's definition, whereas if publisher p_3 produces some event $e_{3.2}$ after delivering event $e_{2.1}$ from publisher p_2, we also say that $e_{2.1} \rightarrow e_{3.2}$. This definition specifies a partial order on the events produced in the system and can be represented by an event graph, where edges represent causal relations. The events can have different topics, contents, or both, depends if the system is topic-based or content-based. In the example of Figure 2.1, we have a topic-based publish-subscribe system, white and red topics and events.

We denote the subscription event history as the sequence of events that are delivered to a given subscriber. The subscription history starts with a special subscribe event, which is locally generated by the subscriber when it issues the subscription. Another special unsubscribe event is issued when the subscriber terminates the subscription. Figure 2.2 shows the subscription history associated with a subscription of red events by different subscribers.

The set of events composed by the first event from each publisher, appearing in the subscription history, defines a cut in the event graph, which is the subscription starting cut. In our example, the starting cut for the subscription of s_1 is defined by events $e_{1.3}$, $e_{2.2}$, and $e_{3.2}$. Similarly, the set of events composed by the last event from each publisher appearing in the subscription history defines a cut in the event graph, which is the subscription ending cut. In our example, the ending cut for the subscription of s_1 is defined by events $e_{1.4}$, $e_{2.4}$, and $e_{3.5}$.

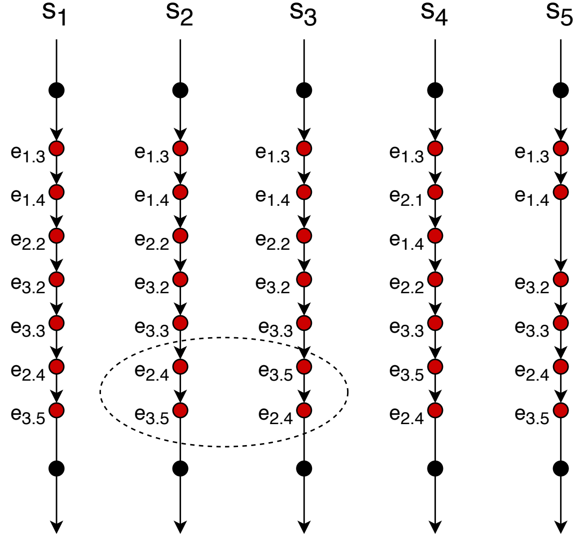


Figure 2.2: Examples of subscription event histories.

2.2 Properties of Publish-Subscribe Systems

In this section, we introduce the properties of publish-subscribe systems essential to understand how each scenario affects the observed latency.

2.2.1 Delivery Order

Publish-subscribe systems differ in the ordering properties of the subscription history. In particular, how the serial order of events in the subscription history relates to the partial order of the global event graph. The following ordering properties are relevant:

FIFO (First-In, First-Out) order [BSB+02, KJ11]:A system enforces FIFO order if, for any two events in subscription s, produced by the same publisher, the order by which these events appear in s is the same as the order by which these events occur in the event history. An example of a subscription event history that respects this ordering can be seen in Figure 2.1.

Causal order [dAADJ+19, NDA+14]: A system enforces causal order if the order of appearance for any two events in s, produced by the same or by different publishers, respects the partial order by which these events appear in the event history. The subscription history of s_4, depicted in Figure 2.2, does not respect this ordering. The history contains $e_{2.1}$. As such, it must also contain every event that is causally dependent on it. However, $e_{3.2}$ is missing.

The two ordering properties presented above are defined for a single subscription history. It is also possible to define ordering properties that relate multiple subscription histories, namely:

Total order [PLTP08, BBPQ12]: A system enforces total order if, for any two subscriptions s and s' (issued by different subscribers), and for any two events x and y such that $\{x, y\} \in s$ and

$\{x, y\} \in s'$, if x appears before y in s then x also appears before y in s'. Both the subscription history from s_1 and from s_2 in Figure 2.2 respect this ordering. Total ordering in violated in the subscription history of s_3 when compared to that of s_1, since it does not contain every event in the same order.

2.2.2 Reliability

We can define the reliability of a publish-subscribe system using the notion of event graph and subscription history. Let s be a given subscription history and E_s be the set of events from the event graph that simultaneously: i) match the subscription specification and; ii) belong to the subgraph of the event graph that is delimited by the starting and ending cut of s. We say that a publish-subscribe system offers *reliable delivery* [KJ09, CF04] if, for any subscription s, all events from E_s belong to the subscription history of s. For instance, using the example of Figure 2.1, the subscription history s_1 does not violate reliable delivery. All red events between the starting cut and the ending cut belong to its history. On the other hand, illustrated in Figure 2.2, the event history of s_5 violates reliable delivery, given that event $e_{2.2}$ is not included in the subscription history. In some works, reliable delivery can also be named *gapless delivery*.

The definition of reliable delivery above does not prevent the system from delivering one event more than once to a given subscriber. Most systems avoid this by providing *exactly once* delivery of each event, which is sometimes called *strong reliability* [BSB+02]. Conceptually, ensuring this type of delivery is simple: the interface can keep a log of the events that have been delivered to the subscriber. Brokers also filter out duplicate events that have already been forwarded. In practice, this could be a limitation if memory is constrained, unless it is possible to compress the event log efficiently.

Finally, we are also interested in a stronger form of reliability that we call *causal completeness* [dAADJ+19]. We say that a subscription history s is *causal incomplete* if there is an event x and two other events $before_x$ and $after_x$, such that $before_x \rightarrow x \rightarrow after_x$, and $before_x \in s$, $after_x \in s$, but $x \notin s$. Note that a history can be reliable without being causal complete. This can happen if the subscription starting cut is not consistent with causality.

Some systems do not enforce reliability and deliver events to subscribers in a best-effort manner [CRW01, CP05]. Others only ensure reliability with high probability [CMPC04, CMPC03]. In our work, we will mainly focus on systems that can offer strong reliability.

2.2.3 Other Relevant Features

The literature on publish-subscribe systems is extensive and detailed, thus, all relevant features that have been proposed or implemented are hardly covered in this report. Despite not being covered, for self-containment, we list in detail some of the most significant features.

Durability [BZA03]: Storage of events matching a given subscription in order to tolerate subscriber failures. By keeping those events, the system can send any missed events to a subscriber that reconnected, after being disconnected for a certain period.

Timeliness [EPB13]: Corresponds to the ability of enforcing either hard or soft real-time guarantees on event delivery. Ensuring that an event is delivered before some target deadline after being issued is a relevant property in several areas. As will be plain later in the report, our work is somewhat orthogonal to these aspects.

Configurable QoS [CAR05]: In some systems, the set of properties enforced on a given subscription can be selected by the subscriber when issuing the subscription.

Support for Subscriber Mobility [CDNF01]: Most publish-subscribe systems are only able to enforce reliability and ordering properties if a subscriber remains attached to the same broker. Supporting subscriber mobility allows a subscriber to disconnect from a broker and connect to a different one, while preserving a single causal history.

2.2.4 Network Overlays and Fault Tolerance

The networkk topology structure impacts both delivery guarantees and subscription latency. One possibility consists of organizing publishers and subscribers in a clique [NDA+14], such that publishers can connect directly to every subscriber without using any intermediary. This configuration can lead to scalability problems, considering that every subscriber must establish communication with every publisher. As such, we will only be focusing on publish-subscribe systems that use a distributed broker network.

The distributed broker network can be modeled as a graph, where each broker is a vertex, and the communication links are the edges between them. The broker topology can be organized according to different types of graphs. One of the most common strategies arranges brokers in a tree [ZSB04], which is an acyclic and directed graph where each vertex has, at most, one parent node but can have several child nodes. Other systems organize brokers in a DAG (Directed Acyclic Graph) [CF04], also an acyclic directed graph, but where each vertex can have several parent and child nodes. On the other hand, some systems use graphs in which the edges are undirected, allowing messages to be exchanged in both directions of the link. Finally, some systems also use cyclic or general graphs [CP05], which have several paths connecting any pair

of nodes.

Clients (that can be publishers, subscribers, or both) connect to one of the brokers on the network that they use as an access point to start publishing or receiving events. Some systems specify that clients can only access edge brokers [SDJ16]; in a tree topology, these are the root and leaf brokers, while in a DAG these are the brokers at the edge of the network. On the other hand, some systems allow clients to connect to any broker on the network [CMPC04].

Each topology, and each strategy to connect clients to brokers, has its advantages and disadvantages. In a tree-based architecture, many events need to be routed via root, which can become overloaded. Also, in a tree, every inner broker on the network is a single point of failure. Ensuring that the broker network is acyclic, as required in DAG architecture, can be difficult without centralized administration. Also, by construction, DAGs lack path redundancy. In a general graph architecture, redundant paths can be used as a fault tolerance mechanism, offering more flexibility when establishing connections. However, maintaining ordering and reliability guarantees when messages can travel through different paths is more challenging.

The fault-tolerant mechanisms implemented by different publish-subscribe systems are often dependent on the network topology. Commonly addressed issues are broker failures and link failures. The most common broker failures are crash failures that render a broker inoperative. The most common link failures are omissions (causing it to lose some messages at random) and link partitions (causing it to become inoperative until the connection is re-established).

Systems can either use redundant brokers or redundant paths to deal with failures. The topology can use the notion of virtual nodes [BRVR17, ZSB04] to mask the fault of a physical server: in said approach, a group of brokers represents a vertex of the graph and acts as a single, reliable broker. Thus, although at the logical level, there is only a virtual link connecting adjacent vertexes, at the physical level, several redundant paths are connecting the different replicas that implement these virtual nodes. Another alternative consists in using other brokers on the network as backups [SDJ16, KJ09] in case of failure of a given broker, creating additional edges, only used when failures happen. A general graph architecture having multiple paths among any two nodes inherently provides fault tolerance.

2.2.5 Event Routing

The broker network is required to deliver messages from publishers to interested subscribers. A simple approach for implementing this task consists of broadcasting all events to all brokers (using flooding, for instance) and then let each broker filter events relevant to its local subscribers. This strategy can be very inefficient, as bandwidth is wasted forwarding events to brokers that

are not interested in them.

A better strategy consists of only forwarding an event to brokers that are in the path to relevant subscribers. But in order to do so, brokers must be aware of the location of subscribers, creating routing tables that help them forwarding events through the right paths. This is possible if information regarding the subscriptions is propagated through the network, and brokers set up their routing tables accordingly. Subscriptions need to be broadcast in the overlay, but since they are generally much less frequent than event notifications, the relative cost of subscription broadcast can be small.

The cost of subscription broadcast can be avoided by forwarding subscription information only to brokers that are in the path from the publishers to the subscribers. This forwarding strategy is only possible if the location of the publishers is known. For this purpose, some systems also use *advertisements* [CRW01], which are messages used by publishers to announce their presence in the network. Announcements need to be broadcast in the network, but they are assumed to be even less frequent than subscriptions. Based on these advertisements, brokers can create routing tables to propagate subscriptions toward relevant publishers.

2.3 Subscription Latency

When a subscriber makes a new subscription, it needs to wait a certain amount of time before it starts receiving events associated with that subscription. We define the subscription latency as the time elapsed between the moment a subscriber issues a subscription request and the first event that matches the subscription being delivered. Naturally, we would like the subscription latency to be as small as possible. However, as we have seen, most publish-subscribe systems require routing tables to have the subscription on brokers that are in the path from the publisher to the subscriber. Before these routing tables are updated, event delivery cannot be ensured. Thus, in many cases, the subscription latency will be proportional to the end-to-end delay between the publisher and the subscriber. In the worst case, the subscription latency will be proportional to the diameter of the network.

There are many other aspects of the operation of the system that can also affect subscription latency. Consider, for instance, an overlay network that uses multiple paths for fault-tolerance. In this case, routing tables need to be set up in *all* routes, introducing additional delays. Systems that use a single path and rely on virtual nodes also introduce extra delays. This delay is due to the need of replicas of a given vertex to coordinate as to ensure they have consistent information regarding the routing table of their virtual node. In some cases, a consensus protocol needs to be run among the replicas of a vertex every time a subscription is propagated via that virtual

node.

The reliability of the event delivery service also has an impact on subscription latency. Consider, for instance, the case where multiple paths can be used to propagate an event from a publisher to a subscriber. Consider that the routing tables have already been set up in one of these routes (that we denote the *stable* path) but not in the other(s). In this case, depending on the way an event takes, it can be forwarded or dropped. A system that aims at offering gapless delivery needs to wait for all routes to be stable. However, a system that can tolerate event losses can start delivering events as soon as one path becomes stable.

Finally, the latency of a subscription is affected by previous subscriptions already in place. Consider the case where a subscriber makes a subscription that is covered by an already deployed one, made at the same broker. A subscription covers another if all messages matching one subscription also match the other. In this case, there is no need to update the routing tables, and the subscriber can start to be served immediately. Thus, the location and subscriptions of other subscribers are relevant aspects when analyzing subscription latency. Systems that do not support subscription covering require every broker on the network to have every issued subscription on their routing table.

2.3.1 Reducing Subscription Latency

There are a few techniques that can be used to reduce the latency experienced by subscribers. We discuss two relevant methods in this context.

The first technique consists of re-organizing the broker overlay to group subscribers that have similar subscriptions [LSB06]. Subscribers are brought closer to the publishers that are relevant to their subscriptions. This positioning reduces the path length from the publisher to the subscribers, reducing the time it takes to set up routing tables. Additionally, it increases the likelihood that a new subscription is already covered.

A second technique consists of using different event forwarding strategies when paths are either stable or unstable [CF04, ZSB04]. For instance, if a publisher becomes aware of a subscription, but it is not sure if all routes are stable, it can force events to be flooded in the network. This flooding prevents incorrect forwarding decisions while subscription information on the brokers is inconsistent, which will trade bandwidth for a lower subscription latency.

2.4 Summary

This chapter provided an overview of several important properties of publish-subscribe systems and discussed some of their advantages and limitations. In particular, we have addressed the

potential impact that the different guarantees offered by publish-subscribe systems can have on the subscription latency and discussed a number of techniques that can be used to reduce this latency.

Chapter 3

Related Work

In this chapter we survey several relevant publish-subscribe systems. In Section 3.1 we list relevant properties to consider when analyzing the solutions. Section 3.2 starts by describing a system that will be used by ours, as a causal multicast layer, and afterward analyzes the studied publish-subscribe systems.

3.1 Relevant Properties

In this chapter, we survey several relevant publish-subscribe systems. We start by describing LoCaMu[SR19], which will be used by our system as a causal multicast layer. In the analysis, we emphasize the following concerns:

- *Network Overlay and Fault Tolerance* Addresses how systems structure their overlays as well as the location for the access points. Addresses also the type of faults the systems tolerate and the techniques they use to achieve fault-tolerance.
- *Ordering* Ordering properties enforced by the system and the algorithms used to implement them.
- *Reliability* Level of reliability systems offer to their subscribers and which mechanisms they use to enforce it.
- *Subscription Starting Cut* How systems define the point when a subscriber can start receiving messages, depending on the guarantees they want to provide. The subscription latency is also an essential factor influenced by the level of service, topology, and relevant optimizations.
- *Routing Events* How systems forward events on the broker network, along with which metadata or storage is used by brokers to make forwarding decisions.

3.2 Systems

In this section, we start by introducing the LoCaMu algorithm, which will be used as a base layer to provide causal delivery by our solution. Afterward, we present the studied publish-subscribe systems that vary widely in their characteristics, offered guarantees, and which topics they approach. Finally, we discuss and compare the analyzed solutions and how we could apply their properties to LoCaMu.

3.2.1 LoCaMu

The goal of the LoCaMu[SR19] system is to provide reliability and causal ordering guarantees using localized multicast on a distributed broker network. Subscription information is already pre-established in the routing tables, and each broker belongs to a multicast group.

Network Overlay and Fault Tolerance Brokers are arranged in an acyclic undirected graph and act as clients that can publish to groups. It uses the concept of neighborhood, which is the set of nodes at an established distance from a broker, to define the partial view each broker has of the network. The system tolerates f number of failures in a given neighborhood of size $2f+1$. This fault-tolerance mechanism uses redundant paths. Brokers create additional edges to the closest available neighbors on that path to bypass nodes that have failures. LoCaMu ensures that a broker forwards a message m to relevant available neighbors, and nodes that are temporarily unavailable will receive all events when they recover. Safety neighborhood and safe paths are also relevant concepts for this system. The safety neighborhood is defined by the $2f+1$ distance nodes in the base graph, and a safe path is any path of $2f+1$ nodes, where there are at most f faulty nodes. If a route has more than f faulty nodes, then messages cannot be forwarded through it.

Ordering To ensure causal ordering guarantees, LoCaMu tags events with metadata about the broker's neighborhood. The size of such metadata is limited because each node is only required to maintain metadata regarding messages sent or received by other nodes in their safety neighborhood. Nodes keep separate sequence numbers for each of their neighbors. Those assigned to the event are the ones corresponding to the brokers on the event's forwarding path. Brokers locally store the sequence numbers they have seen from their neighborhood, defining the node's past. This past is useful to tag events, such that they will carry information about the sender's causal history. By comparing the local causal past with the one attached to the event, nodes can verify if it is safe to forward messages. In practice, since causal order is transitive,

only the most recent sequence numbers need to be preserved in the node's past.

Reliability LoCaMu does not require a perfect failure detector. As such, nodes have different perceptions of which neighbors are available. This misleading perception can make the broker forward messages using a redundant path, creating duplicate messages. Duplicate events can be eliminated using the events' metadata and the node's causal past. Brokers verify if delivering an event respecting causal order is possible by comparing the local causal past with the event's metadata. If such delivery is not possible, the event is then buffered, and a request for retransmission of missed events is triggered.

Routing Events Brokers use routing tables to know which neighbors they have to forward a message to, such that it can be delivered to the members of the addressed group.

3.2.2 SIENA

SIENA[CRW01] is a content-based publish-subscribe system that focuses on giving expressiveness and flexibility to subscriptions. Another of its goals is to provide scalability by optimizing the subscription propagation process. It serves as a building block for many of the presented systems, which use techniques described in this solution.

Network Overlay and Fault Tolerance This system uses a general graph to structure the broker network. Clients can publish and subscribe by connecting to any broker. Using this graph structure provides multiple paths between any pair of brokers, requiring less coordination when a new broker joins the network.

Subscription Starting Cut SIENA uses subscription forwarding combined with advertisement forwarding to disseminate subscriptions on the network. Brokers maintain a structure called a *filters poset*. This set is a DAG of constrains, defining a partial order on the set of filters known by a broker. It supports subscription covering, represented in the covering relations in the filters poset. From the set, a root filter stands out as the most generic one, covering all other subscriptions.

Events start being delivered to a subscriber when its subscription reaches a broker with a connected publisher. The stability of multiple paths does not need to be verified, considering SIENA does not support any quality of service guarantees. As such, subscription latency is related to the distance from the subscriber to the event source, although this can be shortened significantly by subscription covering.

Routing Events Brokers use their filters poset to make forwarding decisions on events. If an event matches a filter, then it is forwarded to the nodefrom where the subscription originated. An additional algorithm is executed to maintain a minimal spanning tree for each publisher. The tree is used to avoid cycles when routing events, as well as to choose the shortest routing path.

3.2.3 Gryphon

The goal of Gryphon[ZSB04] is to build a content-based publish-subscribe system capable of offering strong reliability with FIFO message ordering, while maintaining high availability and scalability. The system is described in various papers, such as [BSB+02], [ASS+99] , and [BZA03], which characterize its different qualities.

Fault Tolerance and Overlay Network This system uses a tree topology for the broker network, where each intermediate node is a virtual one containing several redundant brokers. Publishers connect to the root node (publisher connecting broker), and subscribers connect to any leaf node (subscriber connecting broker).

Ordering To guarantee a FIFO message ordering, it uses a sequence number for each publisher. A publisher tags the events it publishes with sequence numbers, incremented for each event. Every subscribing broker must receive an ordered stream of events from a publisher. A silence token is forwarded downstream if an event happens to be filtered. The sequence number for the token is the same as the one in the filtered event.

Reliability The subscribing brokers start buffering events if sequence numbers are missing in the event stream from a publisher. To recover missed messages, the subscribing brokers send a curiosity message upstream stating the missed sequence numbers. Intermediate brokers receive these curiosity messages and can retransmit the events or tokens if they have them locally. Otherwise, they forward the received message to their parent broker. Consequently, this mechanism allows for an exactly-once reliable delivery.

Subscription Starting Cut Messages can travel along different paths, which can cause gaps in the delivery. To solve this challenge, each leaf broker, where subscribers can connect to, has a virtual time clock. For each new subscription or set of subscriptions, the leaf broker will increment this clock and assign the subscription a virtual *sst* (subscription starting time). This subscription needs to be propagated upstream to the publisher, creating a stable path for the

subscriber to start receiving events. The contents in which the subscriber is interested in and its starting time will be included in this message.

Other nodes will maintain a vector Vb with an entry for each subscriber broker. When a node receives a new subscription, it compares the sst with the entry in its Vb, corresponding to the subscriber broker. To accept this new subscription from node i and update its Vb it must obey the following constraint: $Vb[i] = sst - 1$. The subscription is propagated with its sst toward the publisher broker, the root of the tree topology, which means the experienced delay is equal to the network diameter. Timestamps serve to verify if a broker has a consistent subscription set and belongs to a stable path. Gryphon supports subscription covering by using a DAG to store subscription information. However, it still has to flood the subscription upstream toward the publishing broker, due to clock updates of leaf brokers.

Routing Events To route an event downstream, a vector Vm is used; this vector is attached to a published event by the publisher broker and can be equal to its current Vb. For an intermediate broker to detect if it belongs to a stable path, it must compare the received Vm with its Vb. For each entry i, if $Vb[i] \leq Vm[i]$, then it is safe to perform matching using the information in its routing table, forwarding the event through the links in the matching results. Otherwise, the broker floods the event downstream to ensure a gapless delivery. Upon receiving the event, the subscribing broker verifies if such event matches any of its subscriptions. If $Vm[subscribingbroker] \leq sst$, then it is safe to deliver the event.

3.2.4 δ-fault-tolerant

Kazemzadeh and Jacobsen [KJ09] [KJ11], propose a content-based publish-subscribe system named δ-fault-tolerant. The goal is to develop a system that is reliable and maintains availability when δ broker failures occur, including tolerance to network partitions.

Fault Tolerance and Overlay Network The system's overlay is an acyclic undirected graph where clients can connect to any broker. Each broker contains a partial view of the network, which includes brokers that are $\delta + 1$ hops away; enabling a broker to bypass up to δ unreachable neighbors by creating additional links.

Ordering To achieve FIFO ordering, each publisher can only have one event in transit at a time. As such, event propagation for each publisher is not concurrent. It is only possible to publish another event after the system confirms the previous has been delivered to every interested subscriber.

Reliability For strong reliability, the system uses end-to-end acknowledgments. Brokers forward the event after performing matching to select the links and wait for a confirmation that it has been delivered. Edge brokers receive the event and send an acknowledgment back to the sender. These acknowledgments are propagated back to the broker with the local publisher. When an acknowledgment arrives at this broker, it can confirm that every subscriber received the event and new messages can be published. Sequence numbers are added to each event by the brokers forwarding them. Brokers maintain $2\delta + 1$ sequence numbers, so there is at least one broker in common to compare sequence numbers. This mechanism is employed to detect duplicate events.

Subscription Starting Cut The safety condition, "a publication is delivered to a matching subscriber only if it is forwarded by brokers that are all aware of the client's subscription", has to be followed by the system to enforce the guarantees. The broker with a connected subscriber starts by flooding its subscription. When the subscription arrives at an edge broker, it sends an acknowledgment back to the sender. Brokers on the network will wait to receive a confirmation from the links they flooded the subscription. Afterward, a broker sends a confirmation to its upstream node, which attests to the reception by all its downstream neighbors. When the broker with a connected subscriber receives an acknowledgment, it can then start delivering events to its subscriber. In this system, the delay experienced by subscribers will be equal to the network diameter. After this process, every path in the network is stable for the subscription.

If there are partition islands, the broker bypasses the sequence of unreachable nodes with a new link. It sends the subscription through it, waiting for an acknowledgment from that link only. If a broker beyond the partition can connect to any node inside, then it can propagate the subscription there. It uses a tag on the event for the subscription to be disseminated, only between the nodes included in that *pid* (partition identifier, contains every broker on the partition and which broker detected its existence). In case there are partition barriers, the acknowledgments sent back must be tagged with a *pid*. The tag indicates that brokers beyond the partition in *pid* did not receive the subscription. The broker with a subscriber creates a new entry in its routing table when a confirmation is received. This entry includes the *pid* tagged in the subscription. That way, it knows that events coming from, or beyond, that partition cannot be safely delivered to the subscriber.

Routing Events There are two conditions to deliver a publication to a subscriber: if it matches the subscription's predicates; and if the safety condition is verified. When a broker receives an event, it checks if the event came from any partition known by it. If it did, the

event can be tagged with that *pid*. Next, it matches the event with the subscriptions in its routing table, to select active neighbors to forward the event. The tags in the events are used by brokers with connected subscribers, to verify the safety condition by comparing them with the subscription's stored tags. If there is at least an identical *pid*, then it is not safe to deliver the publication.

3.2.5 Dynamic Message Ordering for Publish-Subscribe

The main objective of Dynamic Message Ordering[BBPQ12] is to ensure that two subscribers interested in the same two or more topics will deliver the events respecting total order. There is no specified overlay for the network of the publish-subscribe system, which assumes a generic Event Notification Service (ENS). This service offers a topic-based interface for clients to publish events or subscribe to already pre-established topics. The system assumes the existence of specific brokers, called topic managers. These managers can be created statically or through a DHT (Distributed Hash Table) that selects these nodes dynamically.

Ordering To achieve total order, every topic manager contains every subscription with its topic and a sequence number. The system arranges topics using a precedence order, which can be predefined or changed dynamically. Every event is forwarded through the relevant managers respecting the precedence order so they can be arranged concerning other events.

To publish an event on the ENS, a publisher must first request a vector clock for that event. The sequencing group of topic T creates the timestamp. The group consists of T and preceding topics included in at least, two subscriptions, along with T. The timestamp has an entry for each topic manager in the sequencing group. The system delivers the event to each member of the sequencing group, according to the precedence order. Managers increment their sequence numbers and update their entry. After every topic manager has filled its entry, the event can be published on the ENS.

Reliability The system also offers reliable delivery by using message retransmission. Topic managers store partially filled-in vector clocks they have seen, so these can be forwarded again. Publishers solicit the creation of a new timestamp in case the request was lost.

Subscription Starting Cut To decide the starting cut for each topic, the subscriber must request the creation of a vector clock Vs. The system sends the request to all relevant topic managers. The subscriber sends an empty vector, with one entry for every topic, to the last topic manager, according to the precedence order. When the manager receives the request, it

increments its sequence number and fills the corresponding entry on the clock. It then updates its set of subscriptions with the new subscriber and every relevant topic. Afterward, it forwards the subscription to the next manager in the order. When the last topic manager fills its entry, the system sends the timestamp back to the subscriber. It will now use the timestamp as its local subscription clock.

When the subscriber is notified about a new event matching its subscription, it must compare and verify, for every entry i, that: $Vs[i] < timestamp[i]$. If the comparison holds, then the subscriber updates its local clock with the sequence numbers in the timestamp. Otherwise, it tags the event as being out-of-order and buffers it, to be delivered in order when possible.

3.2.6 XNET

One of the main goals of XNET[CF04] is to provide a reliable content-based publish-subscribe system. This system also aims at reducing traffic and minimizing the size of routing tables by integrating a routing protocol called XROUTE[CF03], which supports subscription covering.

Fault Tolerance and Overlay Network The system uses an acyclic undirected graph for the overlay, where each publisher or subscriber can connect to edge brokers. To ensure fault tolerance, it can use redundant paths. The system creates more than a single route from each pair of edge brokers, making a general graph overlay. Alternatively, it can use a redundant broker strategy, in which there are designated backup brokers used in case of failure of the direct neighbors.

Ordering To provide FIFO ordering, all links connecting brokers are TCP, which enforces this property. Additionally, each time a broker forwards a message through a link, it uses an increasing sequence number, unique to that link. Nodes store the highest sequence number they have received from each downstream broker plus the highest number they have sent to their upstream node.

Reliability To offer a reliable exactly-once delivery guarantee, end-to-end acknowledgments are used. Brokers store events until confirmation of reception has been sent back. The sequence numbers used for each link are useful for detecting duplicate messages.

Subscription Starting Cut XNET propagates subscriptions on the network using subscription forwarding along with three different strategies to prevent faults. The first is called Crash/Recover, where brokers buffer subscriptions until they receive an acknowledgment from

the upstream node to confirm reception. If a broker crashes, its downstream routers buffer the subscriptions. The system uses this strategy when failures are only transient. If a broker fails for a prolonged amount of time, there is a second strategy called Crash/Failover that uses backup nodes. The backup is on another location in the network, and brokers connect to their backups when their upstream routers fail. The chosen node must guarantee a valid routing path to the subscriber. When switching from the upstream broker to the backup one, the first broker again needs to send subscriptions to the latter. The new forwarding creates a routing path by registering subscriptions on the backup broker. The upstream broker of the crashed node must unsubscribe every entry to remove the previously established routing path.

Both schemes cause the system to be unavailable while recovering. There is a third strategy called Redundant Paths that maintains availability. In this strategy, each broker has at least one alternate route to publishers. Brokers replicate the routing tables on both paths. When one broker fails, brokers on the alternate route can still deliver the event to the subscriber. Brokers flood events to both alternate paths, such that at least the stable path forwards the event to the subscriber. In this system, the delay is equal to the network diameter, since clients connect only to the edge brokers.

Routing Events Brokers receive published events and match them against the patterns in their routing table. This matching decides the downstream links through which brokers must forward the event, only sending it if there is at least one interested subscriber.

3.2.7 Semi-Probabilistic Publish-Subscribe

Costa and Picco [CP05] propose Semi-Probabilistic Pub-Sub to address situations with a highly dynamic and reconfigurable broker network. The solution trades delivery guarantees for scalability and fault tolerance, supporting content-based publish-subscribe.

Fault Tolerance and Overlay Network The overlay topology is a general graph, and clients can connect to any broker of the network.

Subscription Starting Cut Subscriptions are not propagated to all brokers, since each of their routing tables can quickly become stale. Thus, each broker in the network only knows a limited portion of the subscriptions made. A parameter called subscription horizon ϕ determines the neighborhood size having the subscription in its routing table. The routing tables contain an additional field indicating the distance of that node from the broker that issued the subscription. Considering the system does not give any guarantees on the delivery of messages, there is no

need to verify path stability. The subscriber can start receiving events as soon as its subscription is deployed in its broker's vicinity.

Routing Events In this system, brokers do not forward events in a purely deterministic way. When there is no subscription information available, brokers make probabilistic routing decisions, sending the event to a random subset of the node's neighbors. This set is used to propagate events and constitutes a percentage of all the broker's neighbors, defined by a threshold τ. The selection of the forwarding set prioritizes neighbors with subscriptions matching the event. It also prioritizes the closest subscribers, while avoiding the event to be sent through a stale route. If the links do not reach the required percentage, the broker adds random links to forward the event.

3.2.8 GEPS

The goal of Gossip-Enhanced Pub-Sub[SDJ16] is to maintain a high delivery rate to subscribers while also providing high system availability. It supports content-based publish-subscribe and uses both advertisement forwarding and subscription coverage.

Fault Tolerance and Overlay Network The chosen topology is a tree-based graph where clients can only connect to edge brokers. When forwarding through direct links is not possible, it creates redundant paths using a similarity-based approach to bypass brokers with failures. This strategy was preferred instead of using a random method, selecting a random set of nodes from the network.

The main focus of GEPS is on fault tolerance by creating links able to bypass failures using the similarity metric. Each broker maintains a partial view of the system, which is a subset of the broker's siblings (brokers with the same depth in the network overlay). Sibling brokers use gossip to update their views, finding more similar brokers at their level or failed/recovering ones. In each round, brokers heartbeat their sets to alive siblings, in addition to any faulty nodes. This mechanism provides a distributed fault detection mechanism inside each sibling group. Brokers maintain two additional partial views, the parent view, which is the upstream node's sibling view, and the child view, which is a set of the downstream nodes' sibling views.

Subscription Starting Cut The system combines subscription forwarding with advertisement forwarding to create routing paths for events. When a stable path between the subscriber and a publisher exists, delivery can start. Therefore, the delay is equal to the network diameter. The system does not give any guarantee regarding message delivery, disregarding the need to

verify if paths are stable when defining a starting cut.

Routing Events Brokers also maintain a counter for each advertisement with the number of subscriptions in their routing table that match it. This counter serves to compare the similarity between two brokers, when creating sibling views. The method bases the metric on the subscription as well as advertisement knowledge of the brokers; the more they have in common, the more similar they are. Forwarding events to similar brokers means there is a higher chance that other subscribers with common interests will receive the events. Brokers forward events through the links having subscriptions matching the advertisement. In case direct links are not available, brokers use the child or parent partial views to gossip messages to the nodes in them. Brokers that receive gossip messages also route those messages using gossip in the same way.

3.2.9 JEDI

The purpose of JEDI[CDNF01] is to build a content-based publish-subscribe system with a distributed Event Dispatcher (ED), which delivers a published event to all interested subscribers. Dispatching servers (DS) act as the system's brokers. In this context, clients that connect to the ED are active objects (AO); they interact with each other by producing and consuming events.

Overlay Network The system organizes the DSs in a tree-based topology, and AOs can connect to any DS in the network.

Ordering To provide causal ordering, the system uses reliable TCP links. The event dissemination paths guarantee that if an AO delivers first event e_1 and then publishes e_2 (e_2 was caused by the generation of event e_1), then every subscriber must deliver e_1 before e_2.

Subscription Starting Cut It uses a hierarchical strategy to propagate subscriptions in the ED, and these are only propagated upwards on the tree. Propagating only to the root node avoids having to flood them to the entire network. The AO can start receiving events as soon as there is a stable path between it and a publisher. However, the subscription has to be propagated until the root of the tree.

Routing Events When a DS receives an event from a downstream node, then it forwards the event to its parent node. If the message came from upstream, then the DS forwards it to interested downstream nodes. If it has a connected AO with a subscription that matches the event, then it delivers the event to the AO. The event needs to be forwarded to the root DS

since this node knows all the system's existing subscriptions. Subscribers on other subtrees are unknown to intermediate DSs.

3.2.10 Sequencing Graph

The goal of Decentralized Message Ordering[LSB06] is to build a graph of forwarding brokers, called sequencers, that will order events. The system delivers events according to causal order across topics. Essential properties of the system are that all members of a group will deliver events in causal order if the publisher is part of the group; and that every subscriber can verify if an event is out of order, buffering it if needed.

Overlay Network The construction of the sequencing graph follows two criteria: the first is, there can only exist a single path that connects every sequencer associated with a double-overlapped group. These are groups that have more than two members in common. The second is that the final graph must be loop-free to avoid circular dependencies between events. When a message leaves the sequencing network, the sequencer forwards it to a tree-based overlay that connects a group's subscribers. The system assumes that each group's subscribers are globally known, forming a membership matrix. The system uses the global membership matrix to create a sequencer for each pair of double-overlapped groups. The new sequencer must be added to the graph, forming a path to the newly added group, following both described criteria.

Ordering The system's sequencing network determines the order of messages by having a sequence number for each double-overlapped set of groups. It also makes the paths to those groups intersect in a sequencer that will be associated with the overlap. When a sequencer receives a message, it checks if the event is addressed to one of the groups it represents. Then it increments its sequence number and assigns it to the event, forwarding it to the next sequencer in the path to the group. The links between the sequencers provide FIFO order, and the graph is acyclic.

Reliability To offer reliability, it uses end-to-end acknowledgments between sequencers. Sequencers wait for confirmation of the reception of events by their neighboring brokers.

3.2.11 Epidemic Algorithms for Publish-Subscribe

The main objective of Epidemic Algorithms [CMPC03][CMPC04] is to offer reliability, minimizing loss of events in a highly reconfigurable scenario. This content-based publish-subscribe system uses epidemic algorithms to provide a probabilistic reliable delivery and scalability.

Fault Tolerance and Overlay Network This system assumes an acyclic undirected graph overlay for the network, where clients can connect to any broker. The system uses a base gossip[EG02] algorithm to tolerate faults caused by unreliable links and a highly reconfigurable scenario. The authors propose three solutions, which use that algorithm differently. We will focus on one category, which contains two of said solutions. This category is the pull-negative approach, in which brokers gossip about events they have missed. Brokers send gossip messages soliciting the transmission of missed events from other nodes.

Ordering Gossip schemes guarantee a FIFO ordering of messages per-publisher, using sequence numbers incremented at the publisher. In the pull approaches, events are ordered by using a sequence number for each pattern at the publishing source. When publishing an event, the broker must go through its entire routing table. The node then selects expressions matching the event, tagging it with the corresponding patterns' sequence numbers.

Reliability All of the gossip strategies provide probabilistic reliability guarantees, which allow brokers to recover lost messages. In subscriber-based pull, each broker disseminates a gossip message by selecting a random pattern from its local subscriptions. The message contains the sequence numbers of events the broker knows to have missed. The gossip message is routed as a regular event with that specific pattern, forwarded only to a random subset of interested neighbors. The brokers in the network cache events they have seen. When a broker receives a gossip message, it can send the requested events, if it has them stored. Brokers must forward gossip messages toward subscribers of the same pattern, so that the likelihood of recovering events is much higher.

In the publisher-based pull method, brokers tag events with the route they have followed until reaching a subscriber. A broker creates a gossip message by selecting a publisher, instead of a pattern. This message contains every event it has missed from that particular publisher. The broker then disseminates said message using a route the broker is aware of leading to the publisher. This choice increases the likelihood of reaching a broker that cached the event.

Subscription Starting Cut Brokers flood the subscription on the network, establishing routes for published events to follow. A subscriber can start delivering messages as soon as there is a stable path between it and a publisher. When the subscription reaches the publisher, it starts tagging its events with the sequence number relative to the subscription's pattern. At that point, the subscriber can commence delivering events from it.

Routing Events When a broker receives an event, it is matched with the routing table subscriptions and then forwarded to the resulting links.

3.2.12 VCube-PS

VCube-PS [dAADJ+19, dAAD+17] is a topic-based publish-subscribe system that uses dynamically built spanning trees to propagate events. Brokers deliver subscriptions and published events according to causal order.

Fault Tolerance and Overlay Network Brokers are organized in a hypercube-like topology and create hierarchical spanning trees when they want to broadcast a message to a group. This tree is rooted in the message's publisher. The sender also creates the tree with all brokers that subscribe to a topic. Nodes are assumed to be available, and only leave or join groups dynamically.

Ordering This system implements causal broadcast by applying Causal Barriers [PRS96], which only keeps track of the message's direct dependencies. This causal broadcast primitive is more suitable for dealing with dynamic membership. VCube-PS guarantees per-source FIFO reception order, and subscribers will never receive previously published messages.

Reliability This solution uses end-to-end acknowledgments of published messages to ensure reliable delivery. A source node can only broadcast a new message in the spanning tree after every member of it has confirmed the reception of the previously published event.

Subscription Starting Cut When a node subscribes to a topic, it propagates its subscription to every member of the network, using a spanning tree. Brokers update their views of the group after receiving either a subscription or unsubscription. The system can start delivering messages to a subscriber once every broker on the tree has acknowledged the reception of its subscription. Thus, the latency in this system will be equal to the network diameter for every subscription.

3.2.13 P2PPS

The goal of P2PPS [NDA+14] is to create a publish-subscribe system based on the peer-to-peer architecture where events are delivered by peers respecting their causal dependencies. A peer waits for notifications of only events for which it subscribes, and it can subscribe to groups.

Ordering P2PPS uses a hybrid clock to provide causal order, as it would be impossible to adapt a vector clock to a scalable group due to message length. Peers order some pairs of messages unnecessarily in the linear clock protocol, due to concurrency issues. As such, this type of clock reduces unnecessary ordering by mixing linear and physical clocks. The physical one shows the precise time by synchronizing with a time server.

Subscription Starting Cut The system notifies brokers when an event is published. Nodes can only deliver events to subscribers once a condition is verified: the published event must have the subscription in its causal past. For that condition to be checked, the publisher must have received the subscription beforehand. Since the system uses a generic publish-subscribe abstraction, the latency values cannot be inferred.

3.2.14 Publish-Subscribe Middleware with DSM

The goal of Publish-Subscribe Middleware with DSM [PLTP08] is to use this abstraction with an architecture based on distributed shared memory to maintain event ordering. It uses a topic-based generic publish-subscribe middleware where brokers synchronize with each other using shared logical clocks. This system provides total and causal order of messages to its subscribers.

Ordering This solution uses vector clocks to enforce a total and causal ordering for delivery. The shared memory computes the timestamp as a consistent global snapshot of a distributed system. Each group has an associated logical clock to tag messages, which will maintain causal relations between all events in that group. This group clock can be accessed via DSM, such that brokers can keep a consistent view of the timestamp's values. The DSM orders events within a group. As such, every member will deliver them in the same order.

Subscription Starting Cut The DSM also stores a hash map that associates groups with subscriptions. When a broker issues a subscription, the DSM receives it and inserts it in the hashmap. Thus, brokers know which subscribers will receive which events, and how to route messages to them. A subscriber can start receiving events as soon as the published events have a higher group clock value. This system uses a generic publish-subscribe abstraction. Thus the latency values are not explicit.

3.2.15 Comparison

In this section, we analyze how the techniques can be applied to LoCaMu, which will be the building block for our work. Additionally, we will make a detailed comparison of the systems. We

will be comparing how the several systems provide a set of guarantees to subscribers, exploring different techniques for different levels of service. When possible, we also want to consider how it verifies path stability when delivering events to subscribers, which directly correlates with the starting cut for a subscription. Finally, it is also of importance if systems use any type of technique to shorten the delay required to start delivering events. Not every system characterizes its behavior on every relevant topic. As such, in Figure 3.1, we have the concerns that the given systems approach, with only four systems addressing every subject.

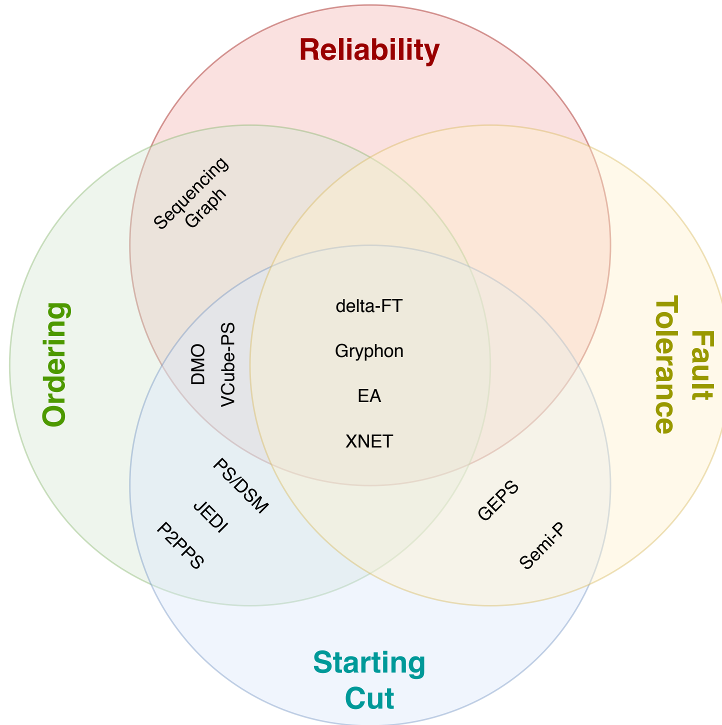


Figure 3.1: Addressed concerns by respective system.

Fault Tolerance Most presented systems consider a failure model encompassing node and link failures, mainly using one of three techniques to tolerate these faults. Redundant Paths are used in XNET and Semi-Probabilistic by creating multiple routes between brokers. δ-fault-tolerant and GEPS also use this strategy, creating additional links on failure. The former uses a topology-based approach, while the latter uses a similarity-based one. Gryphon uses redundant brokers by having virtual nodes in the topology, while XNET uses this technique by having backup brokers. Epidemic Algorithms uses gossiping to recover undelivered events.

Ordering Most systems provide one of three levels of ordering by using sequence numbers, which is a counter incremented for each event. Others additionally use TCP for the communication links. Systems like Gryphon, XNET, and Epidemic Algorithms use sequence numbers

to ensure FIFO ordering for messages. δ-fault-tolerant also provides this type of delivery by not allowing parallel events for each publisher. Sequencing Graph and PS/DSM use sequence numbers to provide causal ordering, while JEDI uses TCP as well as a single path between every broker on the network. P2PPS uses a mix of sequence numbers with physical time to enforce causal delivery. Dynamic Message Ordering and PS/DSM uses sequence numbers to provide a total ordering for events.

Reliability In terms of reliability guarantees, the systems divide themselves mainly into two levels offering either strong reliability or probabilistic/best-effort reliability. The ones that provide exactly once reliable delivery by using message retransmissions are Gryphon and Dynamic Message Ordering. This strategy requires brokers to store forwarded events and to know paths in the network to recover missed events. Others offer this guarantee by using end-to-end acknowledgments, such as δ-fault-tolerant, XNET, Sequencing Graph, and VCube-PS. In this case, brokers store messages until the nodes that must receive them acknowledge their reception. Epidemic Algorithms offer probabilistic reliability by using several gossip strategies.

Subscription Starting Cut In most systems, the latency to start delivering either depends on the network diameter or the distance to the closest publisher. In ones like Gryphon, which disseminates subscriptions toward the root broker, δ-fault-tolerant, VCube-PS, and XNET, which flood subscriptions and GEPS, which uses advertisement forwarding, the latency depends on the network diameter. On the other hand, some systems can start delivery as soon as the subscription reaches a publisher. Epidemic Algorithms that flood the subscription, and JEDI that uses a hierarchical strategy, have this type of latency. Semi-Probabilistic propagates the subscriptions in the vicinity of a broker. In Dynamic Message Ordering, P2PPS, and PS/DSM, the latency does not depend on a network element and cannot be explicitly inferred.

XNET and GEPS both support subscription coverage, which can influence the subscription latency depending on the subscriber population. This technique can make latency equal to the network diameter in a worst-case scenario when the subscription is not covered by any other. Gryphon and XNET use event flooding to compensate for unstable paths on the network. Sequencing Graph reconfigures the topology according to a global membership matrix.

In Table 3.1, we summarize the essential characteristics of those systems. If a system does not address a specific concern, then it is not specified (N/A).

As we can observe, systems that focus on offering stronger guarantees have higher latency. This latency is proportional to the network diameter each time a subscriber joins the system. Others can make the latency depend only on a neighborhood. However, to achieve this goal,

Systems	Latency	Reliability	Ordering	Fault Tolerance
Gryphon	Network Diameter	Strong	FIFO	RB
δ-FT	Network Diameter	Strong	FIFO	RP
XNET	Local	Strong	FIFO	RP/RB
Epidemic Algorithms	Publisher	Probabilistic	FIFO	Gossip
Semi-Probabilistic	Local	N/A	N/A	RP
GEPS	Local	N/A	N/A	RP
JEDI	Publisher	N/A	Causal	N/A
P2PPS	Network Diameter	N/A	Causal	N/A
PS/DSM	Update DSM	N/A	Causal	N/A
Sequencing Graph	N/A	Strong	Causal	N/A
VCube-PS	Network Diameter	Strong	Causal	N/A
Dynamic Ordering	Topic Managers	Strong	Total	N/A

Table 3.1: Systems' guarantees and strategies for each topic.

they do not provide delivery guarantees to the subscriber.

3.3 Summary

This chapter analyzed several publish-subscribe systems with vastly different characteristics in terms of reliability, ordering, and fault tolerance. If subscription latency was a concern for these, we also considered optimizations they used to reduce it. In the following, we aim at identifying the exact conditions that allow subscription latency to be reduced.

Chapter 4

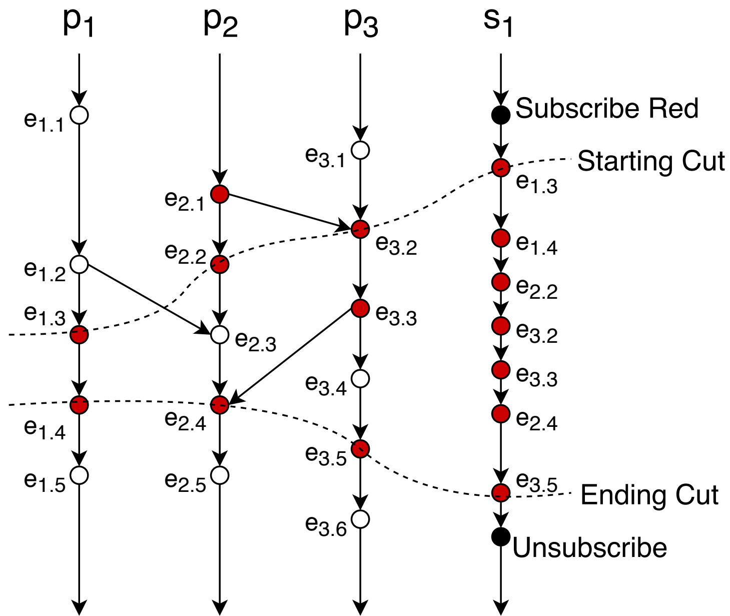
Subscription Semantics

This chapter introduces our causal subscription algorithms. Section 4.1 provides a precise characterization of Gapless FIFO Delivery (GFD) and Gapless Causal Delivery (GCD). Section 4.2 presents the system assumptions and model, as well as the sufficient conditions required to enforce both semantics. Section 4.3 illustrates, in detail, the algorithm to provide GCD using a weaker requirement, with correctness proofs. Section 4.4 describes how subscription coverage can optimize the observed latency. Section 4.5 explains in detail the optimized algorithm to enforce GCD and proves its correctness.

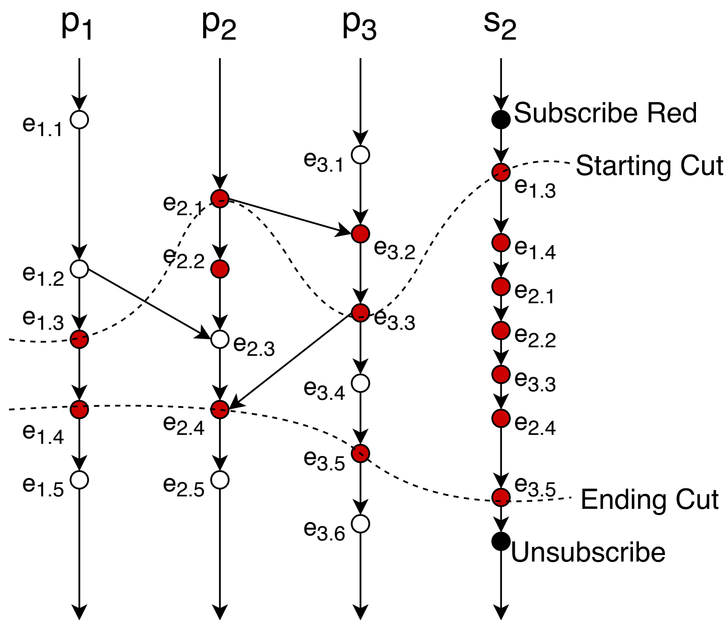
4.1 Semantics

We consider two different semantics that can be found in the literature, namely Gapless FIFO Delivery and Gapless Causal Delivery. We define them based on the notion of starting cut and ending cut. Let G be an event graph, and let S_i be a subscription performed by some subscriber s_i. Considering $H(S_i)$ to be the subscription history for subscriber s_i, then $H(S_i, p_j)$ is the set of events from $H(S_i)$ that have been published by p_j. For each publisher p_j, we denote $e_{j.start}$, the event from that publisher that defines the starting cut of $H(S_i)$. Additionally, we define $e_{j.end}$ as the event from that publisher defining the ending cut of $H(S_i)$. Necessarily, we have that $e_{j.start} \rightarrow e_{j.end}$. An event e matches S_i if $matches(e, S_i)$, and a publisher p_j's advertisement matches a subscription if $matches(p_j, S_i)$. Finally, let $matching(G, S_i, p_j)$ be the set of events published by p_j after $e_{j.start}$ and before $e_{j.end}$ that match the subscription S_i.

Gapless FIFO Delivery (GFD): There are no constraints on the set of events from G that belong to the starting cut and ending cut of a subscription S_i. Only $H(S_i, p_j) = matching(G, S_i, p_j)$ needs to be verified, i.e, all matching events between $e_{j.start}$ and $e_{j.end}$ need to be included in the subscription history.



(a) A subscription that satisfies Gapless Causal Delivery.



(b) A subscription that satisfies Gapless FIFO Delivery.

Figure 4.1: Subscription semantic examples.

Gapless Causal Delivery (GCD): The subscription must satisfy the conditions of definition 4.1 plus the following property: Let j and k be two distinct publishers, let $e_{j.cause}$ be some event such that $e_{j.cause} \in H(S_i, p_j)$, and let $e_{k.effect} : e_{j.cause} \rightarrow e_{k.effect} \rightarrow e_{k.end}$. Gapless Causal Delivery states that $e_{k.effect} \in H(S_i)$.

Figure 4.1 illustrates the difference between Gapless FIFO delivery and Gapless Causal delivery. The two subscription event histories have been obtained from exactly the same event graph. In both cases, subscriptions match red events only, however, the starting cut in each subscription is different. Subscription S_1, in Figure 4.1(a) satisfies GCD: the starting cut is defined by events $\{e_{1.3}e_{2.2}, e_{3.2}\}$ and all events that are in the causal future of the starting cut are included in the event history. The subscription s_2, in Figure 4.1(b) satisfies only GFD: the starting cut is defined by events $\{e_{1.3}e_{2.1}, e_{3.3}\}$, but event $e_{3.2}$ is not included in the event history of s_2 even if $e_{2.1} \rightarrow e_{3.2} \rightarrow e_{3.3}$.

Considering a subscriber will eventually stop processing events, and it can stop whenever it desires, in the rest of this work, we do not discuss unsubscription in detail. Instead, we focus on the difficult challenge of defining an appropriate subscription starting cut.

4.2 Sufficient Conditions for Semantics Enforcement

In this section, we focus on a particular class of publish-subscribe implementations, arguably the most common, based on the use of a network of event brokers. For these systems, we identify the properties that need to be satisfied to offer Gapless FIFO Delivery and Gapless Causal Delivery.

4.2.1 System Model

We consider a content-based publish-subscribe system which supports three types of participants: *publishers*, *subscribers*, and *event brokers*. Publishers and subscribers are also denoted *clients*, and event brokers are named *servers*. Clients may connect to any server, but exclusively to one. A server can attend multiple clients.

Publishers produce events that need to be delivered to interested subscribers. To route events from publishers to subscribers, servers are organized in a network, that can be modeled by a general (cyclic) undirected graph. While some implementations use flooding to propagate events in the broker network, in this paper we consider the case where event flooding is avoided. Subscribers consume events by using subscriptions, which are only issued once by each client.

Servers keep two different types of routing tables to avoid the use of event flooding, namely: *subscription routing tables* and *event routing tables*. Servers use the former to forward subscriptions from subscribers to publishers, and the latter to send events from publishers to subscribers.

Publishers are required to send special *advertisement* messages to build the subscription routing tables. Advertisements are the only messages flooded in the network. We now describe the creation and contents of both routing tables for a system with these characteristics.

Advertisement Propagation and Subscription Routing Table

An advertisement includes a template of the type of events a publisher produces. Advertisements are flooded in the broker network and used to populate the subscription routing table at every server. To simplify the explanation, we assume that advertisements collect the path they have followed when servers propagate them in the network. This collection allows servers to discard advertisements from edges that would create routing loops. When a server receives an advertisement, without creating a loop (i.e., the message does not include the receiving server in its path), the following steps are performed:

- It adds an entry to the publisher to its subscription routing table, associated with the ingress server's edge;

- It propagates the advertisement to all edges except to the ingress edge.

Brokers drop advertisements that would originate a loop in the subscription routing table. When a broker receives this type of message, it creates an entry in its subscription routing table. This entry contains the publisher's identifier and event template, as well as which neighbors sent it. When the advertisement propagation is complete, all servers know the publisher exists, what type of events it produces, and all possible paths, through neighbors, to that publisher.

Subscription Propagation and Event Routing Table

The event routing tables of message brokers are populated using special *subscription messages*, generated when a client issues a new subscription. The subscription message includes both client and broker identifiers, as well as constraints for the matching events. Servers send subscriptions to all publishers with an advertisement that matches it. A subscriber can receive messages from a publisher if it produces events that match its subscription. Instead of flooding subscriptions, brokers propagate them along the paths established during the advertisement propagation procedure, explained above. When a server forwards a subscription, the following steps are performed:

- It adds an entry to the subscriber to its event routing table. This entry contains the subscriber identifier, its subscription constraints, and which neighbor sent it;

- It propagates the subscription to all edges that belong to paths to the subscriber (defined by the subscription routing tables).

Event Routing

When a broker b_k receives some event e from publisher p_j, via link l, it checks if there is one or more matching subscriptions S_i, such that *matches*(e, S_i). This match is performed by verifying the local event routing table of b_k. If there is no subscription matching e, then the broker discards the event. If there is, let L be the set of downstream links that are in a path from b_k to s_i. These links refer to all links that connect b_k to other brokers, except for the one through which b_k received the message. The L link set is stored in b_k's event routing table. For every link $l \in L$, b_k adds l to a set named *link-matches*(k, e). Broker b_k does this for every subscription S_i that matches the event e. Finally, e is forwarded on all links in *link-matches*(k, e).

4.2.2 Subscription Stability

We now define several properties relevant to capture the sufficient conditions for achieving Gapless FIFO delivery and Gapless Causal delivery. Let S_i be a subscription performed by subscriber s_i. We say that a subscription is known to a broker when it creates an entry to its event routing table for the subscription. Additionally, we use the term stability to express a property that does not change over time since the broker overlay is not dynamic. In the following, we present novel definitions that help us better understand how to achieve the requirements.

Link Stability: Let brokers b_a and b_b be two neighbors in the broker network and *link*$_{ab}$ be the network link connecting these two brokers. We say that a subscription S_i is *link stable* on *link*$_{ab}$, denoted *link-stable*(S_i, l_{ab}), if S_i is known both by b_a and b_b.

Path Stability: Let p_j be a publisher and let P_k be a path in the broker network, connecting p_j to s_i. We say that a subscription S_i is *path stable*, denoted *path-stable*(S_i, P_k), iff, for every link $l \in P_k$, we have *link-stable*(S_i, l).

Publisher Stability: We say that S_i is stable regarding publisher p_j, denoted *pub-stable*(S_i, p_j), iff, for every path P_k connecting p_j to s_i, we have *path-stable*(S_i, P_k).

Full Stability: We say that S_i is *fully stable*, denoted *F-stable*(S_i), iff, for every publisher p_j that matches S_i, we have *pub-stable*(S_i, p_j).

4.2.3 Stability-Based Conditions

Using the definitions above, we can define two sufficient conditions to enforce Gapless FIFO Delivery and Gapless Causal Delivery.

Sufficient condition for enforcing Gapless FIFO Delivery: Let S_i be a subscription performed by subscriber s_i that is attached to broker b_i. Let e be some event from publisher p_j received by b_i such that $matches(e, S_i)$. To deliver e to s_i without risking violating GFD it is sufficient that e has been sent by p_j after *pub-stable*(S_i, p_j).

Sufficient condition for enforcing Gapless Causal Delivery: Let S_i be a subscription performed by subscriber s_i that is attached to broker b_i. Let e be some event from publisher p_j received by b_i such that e matches S_i. To deliver e to s_i without risking violating GCD it is sufficient that e has been sent by p_j after *F-stable*(S_i).

4.2.4 Evaluating Full Stability

The sufficient conditions expressed above are relevant because they are used by different publish-subscribe systems to enforce both Gapless FIFO Delivery and Gapless Causal Delivery. We now briefly describe the algorithms these systems use to evaluate the moment when a subscription has reached full stability.

An example of a system that offers GFD and relies on full stability is described in [KJ09, KJ11]. This system uses an acknowledgment collection method to verify if a subscription has achieved full stability. As such, the subscriber's server needs to receive confirmation from all paths between publishers and the subscriber. When a subscriber joins, its server sends a subscription message to all its neighbors. These brokers, in turn, send the subscription to their neighbors, until the subscription messages reach the publishers. These generate an acknowledgment message and propagate it back to the subscriber using the reverse path of the subscription. Each broker collects acknowledgments from all downstream brokers before forwarding one upstream, making the acknowledgment collection process to operate as an aggregation tree. A subscriber s_i knows it is *F-stable*(S_i) in the system once its server receives the aggregated acknowledgments from its server.

It is also possible to find examples of systems that use full stability to implement GCD, such as VCube-PS [dAADJ$^+$19]. Similarly to [KJ09, KJ11] above, this system also uses acknowledgment messages to detect full stability. To forward a subscription, the client's broker creates a spanning tree, which contains every node on the network, and uses it to disseminate the subscription to every broker. The server then waits for every broker to send an acknowledgment for the subscription. A subscriber s_i knows it is *F-stable*(S_i) as soon as it receives the aggregated acknowledgments.

4.2.5 Impact on Latency

These sufficient conditions, and their use in existing systems, suggest that the subscription latency for GFD can be smaller than the subscription latency for GCD. For instance, consider two different publishers p_j and p_k that match some subscription S_i. Assume that some event e is sent by publisher p_j after *pub-stable*(S_i, p_j), but *before* *pub-stable*(S_i, p_k). According to the conditions above, a subscriber S_i would be able to deliver e under GFD, but not under GCD. However, as we show next, these conditions are stricter than needed, and it is possible to define weaker necessary conditions that allow a system to enforce both semantics with a shorter subscription latency.

4.3 A Necessary (and Sufficient) Condition for Semantics Enforcement

As noted above, although the sufficient conditions expressed in Section 4.2 are enough to enforce gapless delivery, they are stronger than strictly needed. The Gryphon[ZSB04] system enforces Gapless FIFO Delivery with a weaker guarantee. Namely, it starts delivering events as long as one (and only one) of the paths P_k from a publisher to the subscriber s_i is *path-stable*(S_i, P_k). As mentioned in the Related Work chapter, Gryphon uses logical clocks to verify when at least one path is stable, by having the publisher update its clock when it receives a subscription. However, Gryphon enforces GDF with only one stable path at the cost of resorting to flooding, when brokers propagate events on non-stable paths. This issue raises the interesting question of knowing if it is possible to use weaker conditions while still avoiding resorting to event flooding. Below, we show that this is, in fact, possible.

4.3.1 Causality-Based Condition

Our work departs from the following observation: Gapless Causal Delivery requires events to be delivered in causal order, regardless of the algorithm used to identify the subscription starting cut. Any system that provides GCD must necessarily include an algorithm to keep track of causal dependencies and to ensure the delivery of messages in causal order. However, it also requires an additional mechanism to explicitly identify the starting and ending cuts, as causal delivery does not guarantee a subscription history will respect GCD. An algorithm to process subscriptions can then leverage causality to define a necessary and sufficient condition that is weaker than the sufficient conditions presented before, concretely:

Necessary and sufficient condition for enforcing both Gapless FIFO Delivery and Gapless Causal Delivery: Let S_i be a subscription performed by subscriber s_i that is attached to broker b_i. Let e be some event from publisher p_j received by b_i such that e matches S_i. For e to be delivered to s_i without risking violating GFD or GCD it is necessary and sufficient that e has been sent by p_j after there is *some* path P_k from p_j to s_i, such that *path-stable*(S_i, P_k).

Note that, Conditions 4.2.3 and 4.2.3 require *all paths* from the publisher to the subscriber or *all paths from all publishers* to be stable, respectively. Contrary to those, Condition 4.3.1 does not impose such strict requirements. Instead, *a single* path needs to be stable. This condition is valid for both GFD and GCD, which indicates that GCD does not necessarily impose higher subscription latencies than GFD. Another advantage of Condition 4.3.1 is that it is possible to derive simple algorithms that allow subscribers to verify if their subscriptions have achieved the condition.

4.3.2 Leveraging Causality

We now describe the generic subscription propagation algorithm that leverages Condition 4.3.1. The algorithm, depicted in Algorithm 1, assumes a system with the characteristics described in Section 4.2.1. We abstract from the underlying message forwarding substrate used to propagate messages on the broker network. Our subscription algorithm is independent of the method used to propagate subscription and event messages in the broker network as long as it is reliable and enforces causal order. We reiterate that the availability of a causal reliability substrate is required to enforce Gapless Causal Delivery, regardless of the algorithm used to manage subscriptions. All the described procedures are executed locally by brokers.

From the previously mentioned Algorithm 1, when a client s_i issues subscription S_i (lines 4 - 8), its broker b_i adds S_i to its local connected subscribers. It then sets the starting cut of S_i to *null* and proceeds to forward S_i to its neighbors by calling the subscriptionForward method. It sends S_i only to brokers in the path to a publisher whose advertisements match the subscription.

When a broker b_k receives a subscription S_i (lines 9 - 18) from s_i on a given path P_k, it starts by creating an entry for it in its event routing table. This is executed by calling method updateEventRoutingTable. Afterward, it sends the subscription to its neighbors in a path to a relevant publisher by executing the subscriptionForward procedure. If b_k has a publisher client p_j then it knows that *path-stable*(S_i, P_k). Then p_j knows that Condition 4.3.1 has been achieved. To announce this, p_j issues a *marker*, $M^j_{S_i}$, if it has not already done so. This marker is sent to all subscribers that subscribed to p_j by calling the eventForward method.

When another publisher p_k receives marker $M^j_{S_i}$ (lines 19 - 31), it also becomes aware that

S_i has achieved the necessary and sufficient condition. This publisher then announces this fact by publishing the marker $M^k_{S_i}$ to all its subscribers by calling the eventForward method. Notice that every publisher p_x that sends an event, in the causal future of $M^j_{S_i}$, also sends $M^x_{S_i}$ by executing the eventForward procedure. When broker b_i receives one for its local subscriber s_i, it sets S_i's starting cut to this marker. The set from all publishers defines the subscription starting cut. Brokers use these to know which events to deliver, since the events will be in the future of the markers in the starting cut ($M_{S_i} \rightarrow e$).

When broker b_i receives an event from publisher p_j (lines 35 - 43) it decides whether to deliver the event to s_i or not. First, b_i forwards the event to neighbors in the path to interested subscribers by calling the eventForward procedure. Events from p_j, sent before the marker $M^j_{S_i}$, are discarded by b_i. The first event received from p_j, after receiving its marker $M^j_{S_i}$, defines the starting cut of the subscription. All events sent after the marker are accepted.

The reader may notice the similarities between our subscription algorithm above and the Chandy-Lamport [CL85] algorithm to compute distributed snapshots. A subscription starting cut is nothing more than a causally-consistent cut of the distributed execution. This cut is defined by the sequence of actions on events in the broker overlay, such as producing, forwarding, and receiving events.

4.3.3 Correctness

We now prove the correctness of Condition 4.3.1 and of Algorithm 1.

Theorem. Condition 4.3.1 is a necessary condition.

Proof. Assume that b_i serves two subscribers, s_i and s_j. Subscriber s_j has previously issued a subscription S_j, and publisher p_k produces events that match S_j. Assume that s_i makes a subscription S_i and that b_i needs to define the starting cut for that subscription. Assume two events e_1 and e_2 such that $e_1 \rightarrow e_2$, $\textit{matches}(e_1, S_i) \wedge \textit{matches}(e_1, S_j)$, $\textit{matches}(e_2, S_i) \wedge \neg\textit{matches}(e_2, S_j)$. Assume that $\textit{F-stable}(S_i)$ but that Condition 4.3.1 is not satisfied, i.e., there is no P_k from p_k to s_j, such that $\textit{path-stable}(S_j, P_k)$. Thus, b_k that serves publisher p_k does not know of subscription S_j. Because b_k does not know S_j it will drop e_2, thus accepting e_1 would violate both GFD and GCD. □

Theorem. Condition 4.3.1 is a sufficient condition.

The proof, based on the correctness of Algorithm 1, depends solely on Condition 4.3.1. Intuitively, the algorithm works because marker M_{S_i} causally depends on S_i (i.e, $S_i \rightarrow M_{S_i}$). Thus, the marker is always delivered to any broker *after* S_i has been delivered to that broker.

Algorithm 1 Subscription Algorithm

1: $subsc(b_k)$ ▷ local subscribers attached to broker b_k
2: $pubs(b_k)$ ▷ local publishers attached to broker b_k
3:
4: **procedure** SUBSCRIBE(s_i, S_i) at b_k
5: $subs(b_k) \leftarrow subs(b_k) \cup \{s_i\}$
6: $starting\text{-}cut[S_i, p_j] \leftarrow \perp, \forall p_j : matches(p_j, S_i)$
7: SUBSCRIPTIONFORWARD(subscription, s_i, S_i)
8: **end procedure**
9: **procedure** PROCESS(subscription, s_i, S_i) at b_k
10: UPDATEEVENTROUTINGTABLE(s_i, S_i)
11: SUBSCRIPTIONFORWARD(subscription, s_i, S_i)
12: **if** $\exists p_x \in pubs(b_k) : matches(p_x, S_i)$ **then**
13: **if** $\neg markersent[p_x, s_i, S_i)]$ **then**
14: $markersent[p_x, s_i, S_i] \leftarrow$ **true**
15: EVENTFORWARD(event, p_x, MARKER(s_i, S_i))
16: **end if**
17: **end if**
18: **end procedure**
19: **procedure** PROCESS(event, p_j, MARKER(s_i, S_i)) at b_k
20: $stable\text{-}paths \leftarrow stable\text{-}paths \cup \{(b_k, p_j, S_i)\}$
21: EVENTFORWARD(event, p_j, MARKER(s_i, S_i))
22: **if** $\exists p_x \in pubs(b_k) : matches(p_x, S_i)$ **then**
23: **if** $\neg markersent[p_x, s_i, S_i)]$ **then**
24: $markersent[p_x, s_i, S_i] \leftarrow$ **true**
25: EVENTFORWARD(event, p_x, MARKER(s_i, S_i))
26: **end if**
27: **end if**
28: **if** $\exists s_i \in subsc(b_k) : starting\text{-}cut[S_i, p_j] = \perp$ **then**
29: $starting\text{-}cut[S_i, p_j] \leftarrow$ MARKER
30: **end if**
31: **end procedure**
32: **procedure** PUBLISH(p_j, EVENT(e)) at b_k
33: EVENTFORWARD(event, p_j, EVENT(e))
34: **end procedure**
35: **procedure** PROCESS(event, p_j, EVENT(e)) at b_k
36: EVENTFORWARD(event, p_j, EVENT(e))
37: **if** $\exists s_i \in subsc(b_k) : matches(e, S_i) \wedge starting\text{-}cut[S_i, p_j] \neq \perp$ **then**
38: **if** $starting\text{-}cut[S_i, p_j] =$ MARKER **then**
39: $starting\text{-}cut[S_i, p_j] \leftarrow e$ ▷ S_i's starting cut
40: **end if**
41: DELIVER(s_i, e)
42: **end if**
43: **end procedure**

Any event e sent after the marker is transitively also in the future of S_i, and delivered to any broker after S_i. This transitivity implies that, when a broker processes an event in the future of the subscription starting cut, it already knows the subscription and will not drop the event.

Proof. The proof is by contradiction. Let e_1 be some event published by p_k such that $matches(e_1, S_i)$. Assume also that e_1 is sent by p_k after there is some path P_k between p_k and s_i, such that $path\text{-}stable(S_i, P_k)$. From Algorithm 1, broker b_k that serves p_k sends marker $M^k_{S_i}$ as soon as P_k becomes stable. Therefore, we have $M^k_{S_i} \to e_1$. Assume that there is some event e_2 published by p_h, such that $e_1 \to e_2$. By transitivity of causality, we have $S_i \to M^k_{S_i} \to e_1 \to e_2$. Additionally, from Algorithm 1, broker b_h that servers p_h sends marker $M^h_{S_i}$ as soon as it receives $M^k_{S_i}$. Thus, we have $S_i \to M^k_{S_i} \to M^h_{S_i} \to e_2$. If e_2 is received by b_i after $M^h_{S_i}$, then e_2 is delivered to s_i. For e_2 not to be delivered to s_i we would require: i) a broker to receive e_2 before knowing S_i, and failing to forward e_2 (a contradiction, as this violates causality) or ii) b_i that serves s_i to process e_2 before $M^h_{S_i}$ (again, a contradiction as this would also violate causality). □

4.4 Leveraging Coverage

We now discuss how subscription coverage can help in further reducing subscription latency. Let S_i and S_j be two subscriptions performed by subscribers s_i and s_j, respectively. Before proceeding, we introduce the following auxiliary definitions:

Subscription Coverage: We say that a subscription S_i *covers* subscription S_j denoted $covers(S_i, S_j)$, iff for every event $e : matches(e, S_j)$ then $matches(e, S_i)$.

Link Coverage: Let b_a and b_b be two neighbors in the broker network and l_{ab} be the link connecting these two brokers. We say that a subscription S_j is *link covered* on l_{ab}, if there is a subscription S_i such that $covers(S_i, S_j)$ and $link\text{-}stable(S_i, l_{ab})$. We denote this by $link\text{-}covered(S_j, l_{ab})$.

(Sub-)Path Coverage: Let P_k be a path in the broker network connecting brokers b_i and b_j. We say that a subscription S_j is $path\text{-}covered(S_j, P_k)$, iff for every link $l \in P_k$, we have $link\text{-}covered(S_j, l)$.

We now show how subscription coverage can be used by systems to reduce subscription latency. We consider first the case where there is a single common prefix among the paths used by both the covering and covered subscriptions. This situation is illustrated in Figure 4.2. Then we discuss how the result can be generalized for the case where there are multiple prefixes.

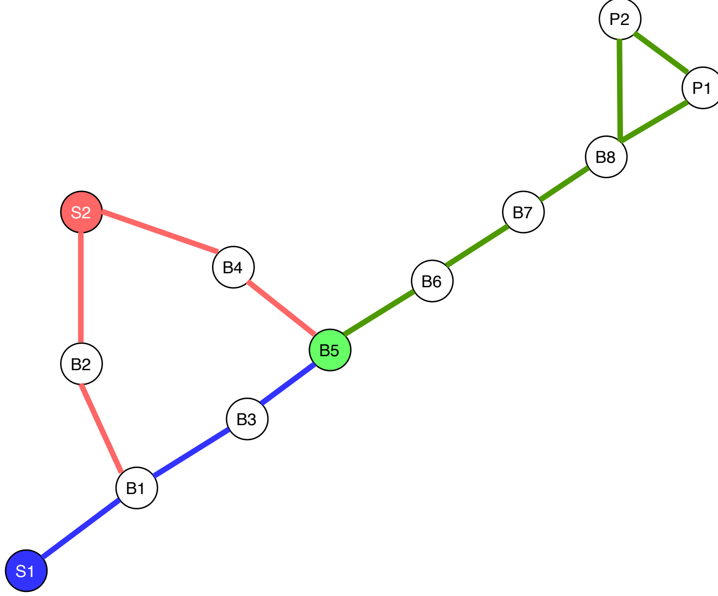


Figure 4.2: Single path subscription coverage.

4.4.1 Single-Prefix Coverage

We first propose an optimization to Algorithm 1 that allows a system to reduce the subscription latency when there is a single common prefix among covered and covering subscriptions. This optimization is based on the concept of *pivot broker*, that is defined as follows:

Pivot Broker: Let p_k be a publisher. Let s_i and s_j be two subscribers that perform subscription S_i (respectively S_j) such that $matches(p_k, S_i)$ and $matches(p_k, S_j)$. Let $\mathcal{P}(p_k, s_i)$ be the set of paths from publisher p_k to subscriber s_i and $\mathcal{P}(p_k, s_j)$ be the set of paths from publisher p_k to subscriber s_j. Let $LCP(p_k, s_i, s_j) = \{p_i, b_1, b_2, \ldots, b_n\}$ be the longest common prefix among all paths in $\mathcal{P}(p_k, s_i)$ and $\mathcal{P}(p_k, s_j)$. We call broker b_n, which is the last broker in the LCP, the *pivot broker*, denoted $pivot(p_k, S_i, S_j)$.

Consider the example depicted in Figure 4.2 of a publish-subscribe broker overlay with two subscribers, s_1 and s_2, and two publishers, p_1 and p_2. The blue links are link-stable for subscription S_1, and the red links for S_2 respectively. Green links are link-stable for both subscriptions. In this example, s_1 has already been deployed on the broker network (on paths from publishers p_1 and p_2 to s_1). Subscriber s_2 makes a new subscription S_2 such that $covers(S_1, S_2)$. There are two paths from p_1 to s_2 (one via b_2 and the other via b_4). In this case, the $LCP(p_1, s_1, s_2) = \{p_1, b_8, b_7, b_6, b_5\}$ and the pivot broker $pivot(p_1, S_1, S_2)$ is broker b_5.

Pivot as a proxy for publisher: Assume that subscription S_j is already covered by some other subscription S_i on $LCP(p_1, s_i, s_j)$. In this case, the pivot broker $b_k = pivot(p_k, S_i, S_j)$ can generate a marker on behalf of p_k and soon as it receives S_j. Note that, any event that is forwarded by b_k, after the marker has been sent, will have S_j in its causal past and will,

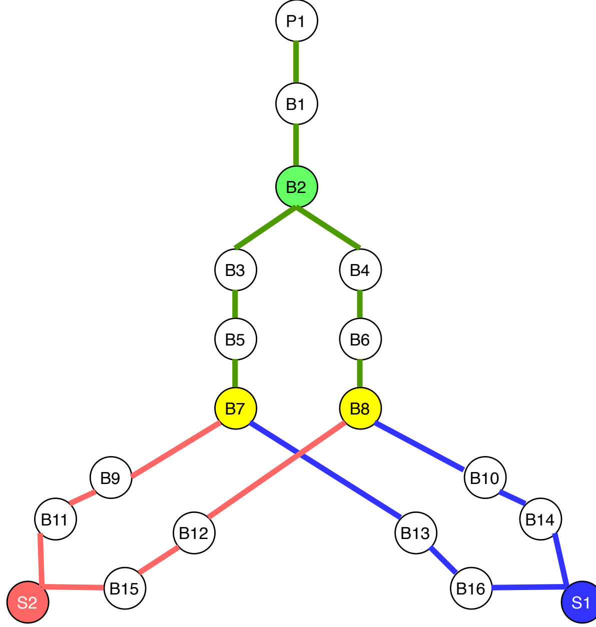


Figure 4.3: Multi-path subscription coverage.

necessarily, be processed by a broker after S_j.

In the example of Figure 4.2, the subscription latency for subscriber s_2 is significantly reduced by this optimization. With Algorithm 1 the marker is produced by publisher p_1. This requires the subscription to make 6 hops toward the publisher and the marker another 6 hops on the reverse path. The propagation amounts to a total of 12 hops before the subscriber starts receiving events. With the optimization, the marker is generated by the pivot broker b_5, taking a total of just 4 hops to be received. Note that, also in this example, similar reasoning can be applied to publisher p_2. Thus, the pivot broker b_5 would send markers on behalf of both p_1 and p_2.

4.4.2 Multi-Prefix Coverage

We now consider a more complex optimization that aims at addressing the case where a subscriber has partially disjoint paths to the publisher. The same algorithm can perform both the previously mentioned optimization in 4.4.1 and the new one. Another subscription partially covers each of these paths.

Consider the example depicted in Figure 4.3. As before, we have a broker network with two subscribers, s_1 and s_2, and a publisher p_1. Subscription S_1 is link-stable in the blue links, and red links represent link-stability for S_2. Green links are link-stable for both subscriptions, and yellow brokers are the first brokers to have both subscriptions. Subscription S_1 has already been deployed on the overlay (on paths from publisher p_1 to s_1). Then s_2 issues a new subscription S_2 that is covered by S_1. In this case the $LCP(p_1, s_1, s_2) = \{p_1, b_1, b_2\}$ and the pivot broker

pivot(p_1, S_1, S_2) is broker b_2. Using the optimization described in the previous section, broker b_2 could send a marker on behalf of publisher p_1 for subscription S_2. However, in this case we can observe that S_2 is already covered by S_1 on the following path prefixes: $P_1 = \{p_1, b_1, b_2, b_3, b_5, b_7\}$ and $P_2 = \{p_1, b_1, b_2, b_4, b_6, b_8\}$. We now discuss how brokers b_7 and b_8 can cooperate to reduce the latency of S_2. Our optimization is based on the concept of a *partial pivot set*, defined as follows:

Partial Pivot Set: Let p_k be a publisher. Let s_i and s_j be two subscribers that perform some subscription S_i (respectively S_j) such that *matches*(p_k, S_i) and *matches*(p_k, S_j). Let $\mathcal{P}(p_k, s_i) = \{P_{i,1}, P_{i,2}, \ldots, P_{i,n}\}$ be the set of paths from publisher p_k to subscriber s_i and $\mathcal{P}(p_k, s_j) = \{P_{j,1}, P_{j,2}, \ldots, P_{j,n'}\}$ be the set of paths from publisher p_k to subscriber s_j. Let *pairwise-lcp*($P_{i,x}, P_{j,y}$) be the longest common prefix to $P_{i,x} \in \mathcal{P}(p_k, s_i)$ and $P_{j,y} \in \mathcal{P}(p_k, s_j)$. Let *max-pairwise-lcp*($P_{i,x}, \mathcal{P}(p_k, s_j)$) $= \{p_1, b_1, b_2, \ldots, b_n\}$ be the longest *pairwise-lcp*($P_{i,x}, P_{j,y}$) for every $P_{j,y} \in \mathcal{P}(p_k, s_j)$. We define b_n to be the *best partial pivot* for path $P_{i,x} \in \mathcal{P}(p_k, s_i)$ with regard to S_j. We define the *Partial Pivot Set* for S_i with regard to S_j, denoted *PPSet*(p_k, S_i, S_j), the set of the best partial pivots for all $P_{i,x} \in \mathcal{P}(p_k, s_i)$.

In the example depicted in Figure 4.3, *PPSet*(p_1, S_1, S_2) $= \{b_7, b_8\}$ defines the partial pivot set. One challenge in using partial pivots is that a partial pivot cannot proxy the publisher and send a marker on its behalf. This issue arises due to a partial pivot only forwarding a subset of the events produced by the publisher. We overcome this challenge by using *partial makers*. Each partial pivot in *PPSet*(p_k, S_i, S_j) produces a partial marker for S_j on behalf of publisher p_k. Brokers use the union of all partial markers from all partial pivots as evidence that subscription S_j is covered in all paths that belong to $\mathcal{P}(p_k, s_i)$. This results in the following optimization:

Partial pivot proxy for publisher: Assume a subscription S_i that is already *F-stable*(S_i). Consider a new subscription S_j that is covered by S_i. Each partial pivot in *PPSet*(p_k, S_i, S_j) produces a partial marker for S_j on behalf of publisher p_k. Subscriber s_j uses the partial markers to define the first event from p_k that belongs to S_i's starting cut.

Another challenge in using partial pivots is that it complicates the definition of the starting cut. When using a full marker, the first event e from publisher p_k, received after the one from p_k, belongs to the starting cut of the subscription. Furthermore, all events in the future of e are guaranteed to be received by brokers after e. With partial markers, this no longer applies, and a more complex algorithm is needed to find a safe starting cut. We propose the following algorithm to select the first event from p_k to belong to the starting cut of the subscription:

- The subscription broker keeps a buffer for each set of paths that lead to each partial pivot. Note that there is a different set of routes for each pivot broker.

- All events from p_k, received via a given partial pivot, before a partial marker from that partial pivot is received, are discarded.

- All events from p_k received via a given partial pivot, after a partial marker from that partial pivot is received, are buffered.

- The receiving broker waits until all buffers have at least one event from p_k. When its buffers verify this condition, it selects the most recent event e among them to be part of the starting cut. After this selection, it can deliver all events in the future of e. Additionally, all messages in the past of e are discarded, including buffered ones in e's past.

- Another publisher that receives a partial marker for a given subscription sends its full one for that subscription as in Algorithm 1. After a subscriber receives a full marker, it can start delivering events from all publishers.

4.5 Subscription Coverage Optimized Algorithm

In this section we present our subscription algorithm, updated with both the single-prefix and the multi-prefix optimizations from Section 4.4. Algorithm 2 illustrates an extension to Algorithm 1. We now describe the parts of the optimized algorithm that differ from our base one.

When a broker b_k receives a subscription S_i (lines 1 - 29) from subscriber s_i, it can only send a marker or partial marker if:

- There exists a local publisher p_k, which has not sent a marker $M^j_{S_i}$ for S_i yet. This step is the same as in Algorithm 1, and serves to signal that Condition 4.3.1 has been achieved.
- There exists a subscription S_j such that b_k is the pivot broker for both S_j and S_i, regarding p_k. Additionally, S_j must cover S_i, and it must have a stable path to p_k in the network. Broker b_k can then send a full marker $M^k_{S_i}$ on behalf of publisher p_k, if it hasn't sent one yet.
- There exists a subscription S_j such that b_k is the partial pivot broker for both S_j and S_i regarding p_k. Additionally, S_j must cover S_i and it must have a stable path to p_k in the network. Broker b_k can then send a partial marker $PM^{1,n}_{S_i}$ on behalf of publisher p_k, if it hasn't sent one yet.

When a broker b_k receives a partial marker $PM^{1,n}_{S_i}$ (lines 30 - 39) and has a local publisher b_k it can forward a full marker $M^j_{S_i}$ for S_i, if it hasn't sent one yet.

When broker b_i, which has s_i as a local subscriber, receives an event e from publisher p_k, it has to decide whether or not to deliver e to s_i. The decision process, described in lines 40 - 59, is as follows:

- **Starting Cut Unknown**: Subscriber s_i is joining the system and has not received any marker or partial marker. In this state, all events are discarded.
- **Starting Cut with Marker**: Either subscriber s_i has received a marker $M^k_{S_i}$ from p_k, or from a pivot broker on behalf of p_k. In this state, the next event e_k to be processed by b_i will be delivered. Event e will define the starting cut for S_i.
- **Starting Cut with Event**: Subscriber s_i can now deliver all events from every publisher that are in the future of the starting cut's event.
- **Starting Cut with Partial Marker**: Subscriber s_i received a partial marker from a partial pivot broker b_x on behalf of publisher p_k on path P_x. In this state, the next event e_x from p_k forwarded by b_x on path P_x will be stored in its respective buffer.
- **Starting Cut with Events in Every Buffer**: Subscriber s_i has received all partial markers from every broker b_x in the partial pivot set of S_i and S_j regarding p_k. Additionally, S_j must cover S_i. An event was also forwarded by every broker in this set and stored in its respective buffer. The most recent event e_x, according to causal order, among every buffer, will define the starting cut.

4.5.1 Correctness

We now prove the correctness of these optimizations. The reader should note that the single-prefix optimization is just a particular case of the multi-prefix optimization, i.e., when a partial pivot set has a single broker, this broker is a pivot broker. Therefore, we only prove the more general optimization described in Section 4.4.2. Our proofs assume the system executes Algorithm 2 on top of a multicast layer that enforces causal order. As described later, this is required to enforce GCD, regardless of the subscription algorithm.

We want to show that the optimized subscription coverage algorithm can safely enforce a GCD semantic for subscribers. To do so, we consider that we are using a system that guarantees causal message delivery.

Theorem. Let e_1 and e_2 be two events that match S_j, such that e_1 is delivered by s_j and $e_1 \rightarrow e_2$. Then, when using Algorithm 2, e_2 is necessarily delivered to s_j.

Proof. Let p_1 be the publisher of e_1 and p_2 be the publisher of e_2. Let b_j be the broker that serves subscriber s_j. We have 5 different cases:

case 1: $\not\exists S_1 : PPSet(p_1, S_1, S_j) \neq \emptyset \wedge \not\exists S_2 : PPSet(p_2, S_2, S_j) \neq \emptyset$. In this case, no optimization is triggered and the proof from Theorem 4.3.3 still applies.

Algorithm 2 Optimized Algorithm (only parts that differ from Algorithm 1)

```
1: procedure PROCESS(subscription, s_i, S_i) at b_k
2:     UPDATEEVENTROUTINGTABLE(s_i, S_i))
3:     SUBSCRIPTIONFORWARD(subscription, s_i, S_i)
4:     // publisher p_x sends the marker (default)
5:     if ∃p_x ∈ pubs(b_k) ∩ pubs(b_k) : matches(p_x, S_i) then
6:         if ¬markersent[p_x, s_i, S_i)] then
7:             markersent[p_x, s_i, S_i] ← true
8:             EVENTFORWARD(event, p_x, MARKER(s_i, S_i))
9:         end if
10:    end if
11:    // broker b_k sends marker on behalf of p_x (proxy marker)
12:    if ∃p_x, S_x : b_k = pivot(p_x, S_x, S_i) then
13:        if covers(S_x, S_i) ∧ {(b_k, p_x, S_x)} ∈ stable-paths then
14:            if ¬markersent[p_x, s_i, S_i)] then
15:                markersent[p_x, s_i, S_i] ← true
16:                EVENTFORWARD(event, p_x, MARKER(s_i, S_i))
17:            end if
18:        end if
19:    end if
20:    // broker b_k sends partial marker on behalf of p_x
21:    if ∃p_x, S_x : b_k = PPSet(p_x, S_x, S_i) then
22:        if covers(S_x, S_i) ∧ {(b_k, p_x, S_x)} ∈ stable-paths then
23:            if ¬pmarkersent[p_x, s_i, S_i, S_x)] then
24:                pmarkersent[p_x, s_i, S_i, S_x] ← true
25:                EVENTFORWARD(event, p_x, PMARKER(b_k, s_i, S_i, S_x))
26:            end if
27:        end if
28:    end if
29: end procedure
30: procedure PROCESS(event, p_j, PMARKER(b_x, s_i, S_i, S_x)) at b_k
31:    pmarkers ← pmarkers ∪ {(p_j, b_x, s_i, S_i, S_x)}
32:    EVENTFORWARD(event, p_j, PMARKER(b_x, s_i, S_i, S_x))
33:    if ∃p_x ∈ subsc(b_k) ∩ pubs(b_k) : matches(p_x, S_i) then
34:        if ¬markersent[p_x, s_i, S_i)] then
35:            markersent[p_x, s_i, S_i] ← true
36:            EVENTFORWARD(event, p_x, MARKER(s_i, S_i))
37:        end if
38:    end if
39: end procedure
40: procedure PROCESS(event-message, p_j, EVENT(e)) at b_k
41:    EVENTFORWARD(event-message, p_j, EVENT(e))
42:    if ∃s_i ∈ subsc(b_k) then
43:        if starting-cut[S_i, p_j] ≠ ⊥ then
44:            if starting-cut[S_i, p_j] = MARKER then
45:                starting-cut[S_i, p_j] ← e
46:                DELIVER(s_i, e)
47:            else if starting-cut[S_i, p_j] → e then
48:                DELIVER(s_i, e)
49:            end if
50:        else if ∃b_x ∈ path(e) : {(p_j, b_x, s_i, S_i)} ∈ pmarkers then
51:            buffer[S_i, p_j, b_x] = buffer[S_i, p_j, b_x] ∪ {e}
52:            if ∃S_j : covers(S_j, S_i) ∧ ∀b_x ∈ PPSet(p_j, S_i, S_j) : buffer[S_i, p_j, b_x] ≠ ∅ then
53:                e ← MOSTRECENT(buffer[S_i, p_j, b_x], ∀b_x ∈ PPSet(p_j, S_i, S_j))
54:                starting-cut[p_j, S_i] ← e
55:                DELIVER(s_i, e)
56:            end if
57:        end if
58:    end if
59: end procedure
```

case 2: $\nexists S_1 : PPSet(p_1, S_1, S_j) \neq \emptyset \wedge \exists S_2 : PPSet(p_2, S_2, S_j) \neq \emptyset$. Since there is no S_1 for which a set of partial pivot brokers for p_1 exists, then it needs to send a marker $M^1_{S_j}$ explicitly before event e_1, for e_1 to be delivered by b_j to s_j ($M^1_{S_j} \rightarrow e_1$). From causal order, p_2 will receive $M^1_{S_j}$ before receiving e_1, and will therefore send an explicit $M^2_{S_j}$ before sending e_2. Also from causality, b_j will receive $M^2_{S_j}$ before e_2 and will deliver e_2 ro s_j.

case 3: $\exists S_1 : PPSet(p_1, S_1, S_j) \neq \emptyset \wedge \nexists S_2 : PPSet(p_2, S_2, S_j) \neq \emptyset$. If s_j delivers e_1 then, either p_1 sends $M^1_{S_j} \rightarrow e_1$ (no optimization was triggered) or every $b_x \in PPSet(p_1, S_1, S_j)$ sends a partial marker $PM^{1,k}_{S_j} \rightarrow e_1$, on behalf of p_1. Note that every path from p_1 to p_2 includes some $b_x \in PPSet(p_1, S_1, S_j)$ (if another path would exist, there there would be a path from s_j to p_1, passing via p_2, that will not include members of $PPSet(p_1, S_1, S_j)$ which would be a contradiction). From causality, p_2 will receive $M^{1,x}_{S_j}$ from some $b_x \in PPSet(p_1, S_1, S_j)$ before receiving e_1, and will send $M^2_{S_j} \rightarrow e_2$ before sending e_2. Also from causality, b_j will receive $M^2_{S_j}$ before e_2 and will deliver e_2 to s_j.

case 4: $\exists S_1 : PPSet(p_1, S_1, S_j) \neq \emptyset \wedge \exists S_2 : PPSet(p_2, S_1, S_j) \neq \emptyset \wedge PPSet(p_1, S_1, S_j) \neq PPSet(p_2, S_1, S_j)$. The reasoning from case 3 also applies, with the proof being the same.

case 5: $\exists S_1 : PPSet(p_1, S_1, S_j) \neq \emptyset \wedge \exists S_2 : PPSet(p_2, S_1, S_j) \neq \emptyset \wedge PPSet(p_1, S_1, S_j) = PPSet(p_2, S_1, S_j)$. If b_j delivers e_1 to s_j then $M^1_{S_j} \rightarrow e_1$, either because p_1 sends $M^1_{S_j}$ explicitly or because every $b_x \in PPSet(p_1, S_1, S_j)$ sends a partial marker $PM^{1,x}_{S_j} \rightarrow e_1$, on behalf of p_1. However, because $PPSet(p_1, S_1, S_j) = PPSet(p_2, S_1, S_j)$, every b_x that sends $PM^{1,x}_{S_j}$ also sends immediately $PM^{2,x}_{S_j}$ on behalf of p_2. Thus, because $e_1 \rightarrow e_2$, when e_2 is received by b_j a partial marker $PM^{2,x}_{S_j}$ has already been received from every $b_x \in PPSet(p_2, S_2, S_j)$ (together with $PM^{1,x}_{S_j}$), and thus, if e_1 is delivered, e_2 is also delivered. □

4.6 Summary

This chapter addressed our GCD subscription algorithms, which both enforce Gapless Causal Delivery and reduce latency by using coverage. Firstly, we defined the sufficient condition and characteristics required by publish-subscribe systems to provide GCD. Then we provided a less strict requirement when assuming a causal multicast layer and described our base subscription algorithm. Secondly, we considered how subscription coverage improves the latency and how systems that enforce GCD can apply that method. Finally, we defined an optimized subscription algorithm, which leverages the proposed necessary and sufficient condition with coverage to provide strong delivery guarantees with low latency.

Chapter 5

LoCaPS

This chapter introduces LoCaPS, the first Localized Causal Publish-Subscribe system. Section 5.1 starts the chapter by explaining the implementation objectives. Section 5.2 presents LoCaMu, the base system used as a causal multicast layer, and how LoCaPS applies the proposed optimized subscription algorithm. Finally, Section 5.3 describes the relevant details concerning the implementation of LoCaPS, as well as the used technologies and framework.

5.1 Goals

Several publish-subscribe systems enforce a GCD semantic, as previously seen. However, none of the studied algorithms approach every relevant topic mentioned in Section 3.2.15, required to provide a GCD semantic, such as fault tolerance. Additionally, none consider optimizations to subscription latency. When designing LoCaPS, there were several objectives in mind. The first was to use a causal multicast layer capable of upholding three of the mentioned topics (ordering, reliability, and fault tolerance). The second was to tackle the subscription starting cut topic. Thus, we presented an algorithm in Section 4.3, which leverages Condition 4.3.1 to enforce GCD, while using the mentioned multicast layer. Finally, to reduce the subscription latency, we implement the optimized algorithm, presented in Section 4.5, which uses subscription coverage in conjunction with the condition. In the rest of the chapter, we will describe the causal multicast layer and how the implementation provides GCD with low latency.

5.2 LoCaPS

In the previous chapter, we have proposed several optimizations that allow us to reduce subscription latency. In particular, systems achieve this reduction when using covering subscriptions already deployed in the network. Unfortunately, these optimizations are hard to implement in

general topologies. In fact, in those overlays, it is hard, or even impossible, for a broker to know if it is a pivot broker. In the case of a subscriber, it is hard to identify the size of the *PPSet* when it receives a partial marker. Without this information, the subscriber cannot start delivering events earlier and needs to receive an acknowledgment directly from the publisher. This issue can explain why most implementations that offer Gapless FIFO Delivery or Gapless Causal Delivery fail to leverage the coverage property to speed up the subscription process.

However, not all publish-subscribe systems use arbitrary broker topologies. When systems use suitable overlays, it is possible to leverage our findings to derive an efficient implementation that can offer low subscription latency. In this section, we give a concrete example that shows it is, in fact, possible.

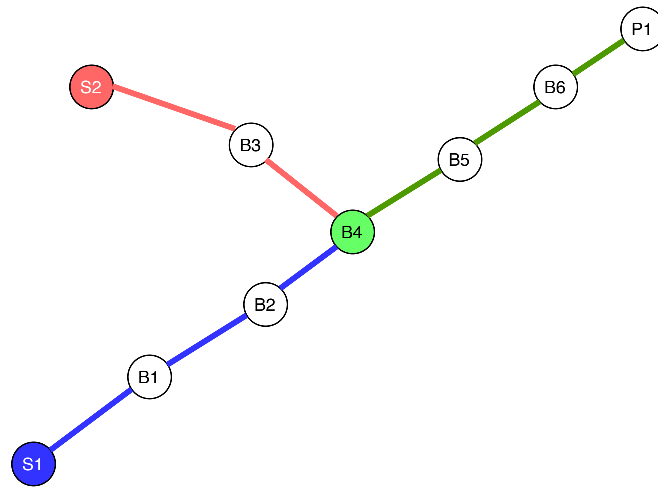
We describe a new system, named LoCaPS, that leverages the necessary and sufficient condition presented in Section 4.3 and the optimizations from Sections 4.4 to build an efficient publish-subscribe system. LoCaPS also offers low subscription latency in favorable conditions and is built on top of LoCaMu [SR19], a causal multicast substrate for publish-subscribe systems. We start by describing LoCaMu and then describe how we can exploit the properties of LoCaMu for applying the proposed optimized subscription algorithm from Section 4.5.

5.2.1 LoCaMu

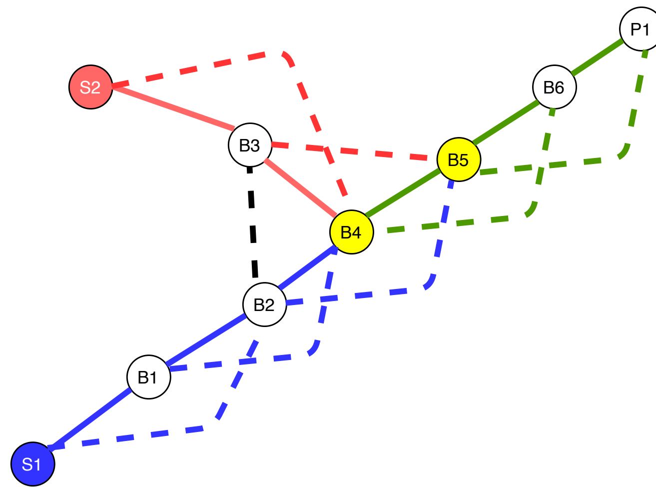
LoCaMu [SR19] is a causal multicast substrate for publish-subscribe systems. LoCaMu only supports static subscriptions: in its current form, all subscriptions must be statically deployed in the network before it starts operating. LoCaPS extends LoCaMu with dynamic subscriptions that have low latency.

The main feature of LoCaMu is that it works by using localized information, i.e., each node only needs to maintain metadata regarding a set of nodes in its neighborhood. Although the localized feature of LoCaMu is interesting, it is not the most relevant to the work described in this paper. The most prominent characteristic of LoCaMu that is relevant to our system is its particular broker topology, which is inherited from [KJ11].

Figure 5.1 presents the topology of the broker network used by LoCaMu. This system organizes brokers in an undirected acyclic graph, as depicted in Figure 5.1(a). In this underlying base graph, there is a single path connecting a subscriber to a publisher. The underlying graph is not fault-tolerant: if a broker fails, the broker topology becomes disconnected. The system augments the underlying graph with additional links to achieve fault-tolerance. These extra links allow a path to circumvent f failed nodes. This is illustrated in Figure 5.1(b) for the case of $f = 1$. As such, LoCaMu can offer reliable causal delivery and tolerate f faulty nodes in each



(a) Underlying acyclic graph



(b) Extended graph

Figure 5.1: LoCaMu's underlying acyclic graph and extended graph.

local neighborhood.

To summarize, the LoCaMu system has several properties that we use in the implementation of LoCaPS, namely:

- LoCaMu delivers messages according to causal order.
- LoCaMu organizes the broker overlay in an acyclic undirected graph. The underlying acyclic graph is augmented with additional links to create an (extended) fault-tolerant topology. Although the fault-tolerant network can contain cycles, LoCaMu's message propagation algorithm ensures this graph operates as if brokers used the underlying acyclic graph.
- LoCaMu offers reliable delivery and ensures brokers eventually deliver all messages in causal order, even if one is temporarily disconnected.

While LoCaMu only considers static subscriptions the system installs at deployment time, we assume dynamic subscriptions. Using LoCaMu as a causal multicast layer, we can build a publish-subscribe system that offers a GCD subscription semantic. Brokers order the subscriptions they send with metadata concerning other subscriptions and events. This system uses a multi-prefix approach, with several redundant paths between a publisher and a subscriber. As we have seen, solely adding metadata for the brokers to causally order subscriptions does not enforce a GCD semantic. The subscription history can have gaps, since a starting cut was not clearly defined. As such, we will describe how LoCaPS applies the optimized algorithm in conjunction with LoCaMu.

5.2.2 LoCaPS Algorithm

Looking at LoCaMu's topology, it is possible to make the following key observations:

- In the extended graph, for fault-tolerant reasons, there are multiple paths from a publisher to a subscriber. As such, LoCaPS cannot apply the optimization described in Section 4.4.1 for overlays with a single-path prefix.
- In the extended graph, the partial pivot set always has $f+1$ members. This fact simplifies the use of the optimizations described in Section 4.4.2.

LoCaPS implements the optimized subscription algorithm, detailed in Section 4.5, on top of LoCaMu using the rules above to define the *PPSet*. We describe, in Algorithm 3, how brokers discover if they are part of this set, and therefore can send a partial marker. We use Figure 5.1 to illustrate how the discovery is processed. Consider two subscribers, s_1 and s_2, and a publisher,

Algorithm 3 LoCaPS Algorithm to find *PPSet*

```
1: procedure PPSET(p_x, S_x, S_i) at b_k
2:     if b_k = pivot(p_x, S_x, S_i) ∨ b_k ∈ pivot(p_x, S_x, S_i) + f then
3:         PPSet(p_x, S_x, S_i) ← b_k    ▷ If b_k is the pivot broker or belongs to the f next brokers
4:     end if
5: end procedure
```

p_1. Blue links and red links are link-stable for subscription S_1 and S_2 respectively. Green links are link-stable for both, and black links for none. In LoCaPS, $PPSet(p_1, S_1, S_2)$ consists of the following brokers:

- Broker $b_k = pivot(p_1, S_1, S_2)$, which is the pivot broker in the underlying acyclic graph. In the case of Figure 5.1(a), this broker is b_4.
- The next f brokers on the underlying acyclic graph on the path to the publisher. In the case of Figure 5.1(b), only b_5 belongs to the f set.

Brokers know if they are either the pivot broker or part of the next f brokers due to their neighborhood knowledge. For example, in Figure 5.1, b_4's neighbours are b_2, b_3, and b_5, and this broker knows that it has a link stable connection to b_2 and b_3 with subscription S_1 and S_2 respectively. Two different scenarios can define the first event, from publisher p_1, to belong to the starting cut of either S_1 or S_2. Either a full marker is sent by p_1 directly, or a partial marker is sent by every broker in $PPSet(p_1, S_1, S_2)$. After subscribers receive the markers, as described in Algorithm 2, the system can start delivering events to them without violating the GCD semantic.

With LoCaPS, we can optimize the subscription latency, making it depend on a localized portion and not on the distance to the publisher. In terms of subscription latency values, if a subscription is not covered, then it will be proportional to the distance to the publisher. As we have seen, using LoCaMu as a substrate for causal delivery ensures the necessary and sufficient condition to enforce GCD semantic. However, the latency differs when we have covered subscriptions. In this case, the latency will be proportional to the portion of the path that is not covered. This dependency results from brokers forwarding the subscription until the pivot and the f next brokers.

5.3 Implementation

In this section, we describe the implementation of LoCaPS used in the evaluation. The same framework described here is used by all the implemented algorithms used for evaluation purposes. Additionally, we implemented and executed a specific code for each.

5.3.1 Development Environment

We implemented the prototype using the Java programming language (OpenJDK 11). As Java is a Virtual Machine based language, the choice of an operating system is not relevant for the development. We also used the Peersim [MJ09] simulator as a base program for the LoCaPS implementation, as well as the LoCaMu system. Lastly, we used Maven for compiling and executing the software.

5.3.2 Framework

We created a single framework to include all the classes used by all algorithms. This framework enabled us to make comparisons among related algorithms as fair as possible. This framework offers by default three components that are used by every algorithm: Message Publisher, Causality Handler, and Fault Detector.

The framework requires each algorithm to implement the Message Storage, Front-End, Back-End, and Subscription Handler components. The last one is responsible for managing both the subscription and event routing tables and performing matching operations on subscriptions and events.

5.3.3 LoCaPS

LoCaPS uses the LoCaMu system, which has been implemented in the same framework and works as a base causal multicast layer. Thus, our implementation required the creation of the Subscription Handler component and the message's structure. The message types include events, advertisements, and subscriptions. Each message contains the parameters required by LoCaMu. Plus, the constraints associated, in case of subscriptions and announcements, or the contents, in case of events.

Brokers store in the subscription routing table the advertisement's publisher identifier, the template for produced events, and which of the broker's neighbors forwarded it (this information is accessible through LoCaMu's localized information in the message). Brokers use the event routing table to store subscriptions. The entry for this structure contains the subscriber's identifier, the constraints used for matching, and the traveled path of the subscription.

When a broker receives an event or subscription, it uses its Subscription Handler component to perform matching and find the set of neighbors to forward the messages. For subscriptions, it goes through every entry in the subscription routing table and selects the ones with publishers that produce events matching the subscription. After that, it collects all neighbors in the matching entries to forward the subscriptions. For events, it goes through every entry in the

event routing table and selects the ones to which the event's content matches the constraints. Afterward, the broker sends the event to all neighbors in the set of selected entries.

5.4 Summary

This chapter addressed LoCaPS, the first localized causal publish-subscribe system that provides Gapless Causal Delivery with low subscription latency. This system leverages the necessary and sufficient condition to enforce GCD with subscription coverage. It uses LoCaMu as a causal multicast layer to have a causal delivery guarantee and ensure subscriptions achieve the requirement. By using the LoCaMu system, which has a particular topology, the brokers of LoCaPS can discover the relevant PPSets. Thus, LoCaPS provides an implementation of the optimized subscription algorithm. Finally, we described how the prototype was implemented first by stating what was the development environment and presenting the used framework. Lastly, we disclosed specific details regarding our implementation.

Chapter 6

Evaluation

This chapter presents the evaluation of LoCaPS, based on a prototype built for a simulator. The main goal of this evaluation is to analyze how the characteristics of the LoCaPS system impact the observed latency when compared to two of the mentioned works in the Related Work chapter. Section 6.1 describes the goals and what we will test from the systems in the evaluation. Section 6.2 describes the system settings, offering information about the system overlay, client positioning, and coverage conditions. Section 6.3 analyzes how both considered conditions impact latency. Section 6.4 compares LoCaPS with the studied systems and analyzes how coverage impacts the latency. Section 6.5 studies how localized information can impact latency when used with subscription coverage. Finally, Section 6.6 discusses the main differences between each algorithm and their respective observed latency tendencies.

6.1 Goals

We evaluate LoCaPS to answer the following questions:

- What is the impact on the latency of using the necessary and sufficient condition to enforce Gapless Causal Delivery?
- What is the impact of subscription coverage on the observed latency?
- How do the f and network diameter values affect the subscription latency in localized systems?

6.2 Experimental Settings

In this section, we evaluate the performance of LoCaPS in terms of subscription latency. We compare the latency, observed by subscribers, with two of the presented systems that most re-

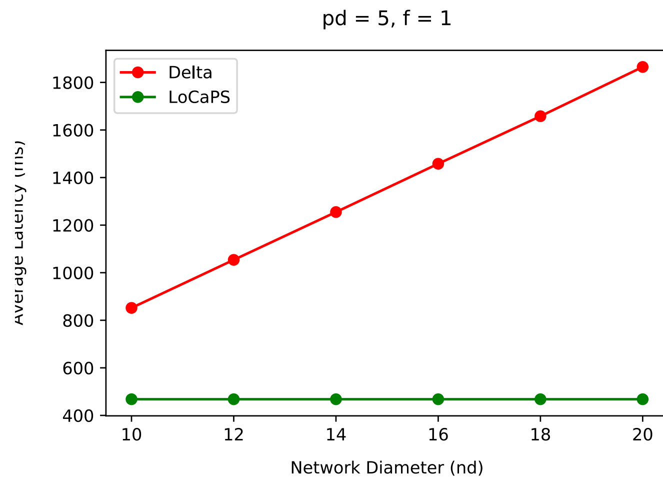
semble ours, namely the δ-fault-tolerant [KJ11] (Delta) and Gryphon systems [ZSB04, BSB+02]. The Delta system ensures Gapless FIFO Delivery to subscribers and also uses the concept of neighborhood and localized information. The Gryphon system provides GFD to subscribers as well and uses logical clocks to define the subscription starting cut. We evaluate how these systems behave when we vary different system characteristics. The parameters to modify include network diameter, distance to the publishers, and size of the neighborhood. Another parameter that we also consider in the evaluation process is the likelihood that a previously deployed subscription covers a new one joining the system.

To perform the evaluation, we use the PeerSim [MJ09] simulator with an extension that simulates network latency. We consider an average latency of $50ms$ between brokers, which approximates the average latency between Google data centers in North America. Additionally, the time required to process the messages on the brokers is negligible. We have chosen a binary tree as our network topology. In the Delta system, the latency always depends on the network diameter. As such, subscribers are always placed on the leaves to provide a fair evaluation. The Gryphon system positions its subscribers on the tree leaves and the publishers on the root. The network diameter is set at 18 in a network with 512 brokers, unless stated otherwise.

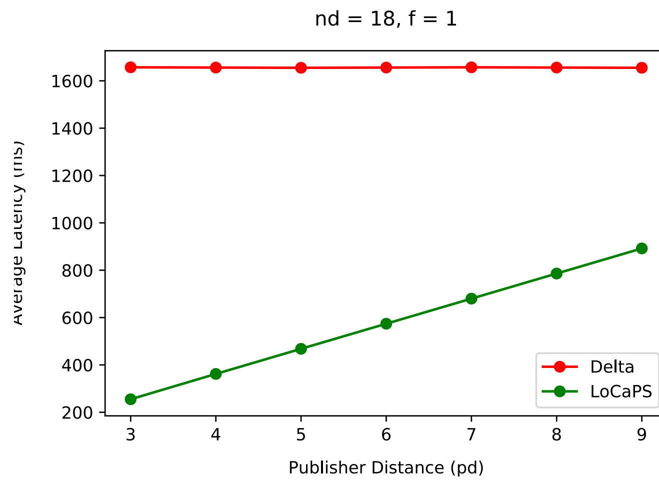
6.3 Analyzing the Sufficient and Necessary and Sufficient Conditions

In this section, we analyze two systems, LoCaPS and Delta, which use respectively the necessary and sufficient condition and full-stability to enforce their semantics. Figure 6.1(a) illustrates the average latency observed by subscribers from both the LoCaPS and Delta systems as the network diameter increases. In this scenario, the publishers are always at a five hop distance from subscribers. In Figure 6.1(b), the distance to publishers varies. As we would expect, on LoCaPS, the latency increases with the distance to the publisher, not increasing with network diameter. LoCaPS considers only Condition 4.3.1 to enforce GCD. Thus, we can expect that the subscription latency will depend on the distance to the publisher.

In the Delta system, the latency will be proportional only to the network diameter and does not depend on the publisher's location, increasing the latency with a bigger diameter. This system uses full-stability to enforce GFD, which requires brokers to flood a subscription to the entire network. As expected, the observed latency is also higher when compared to the one observed by subscribers from LoCaPS. By using full markers from publishers, it is possible to start delivering events sooner, which reduces subscription latency.

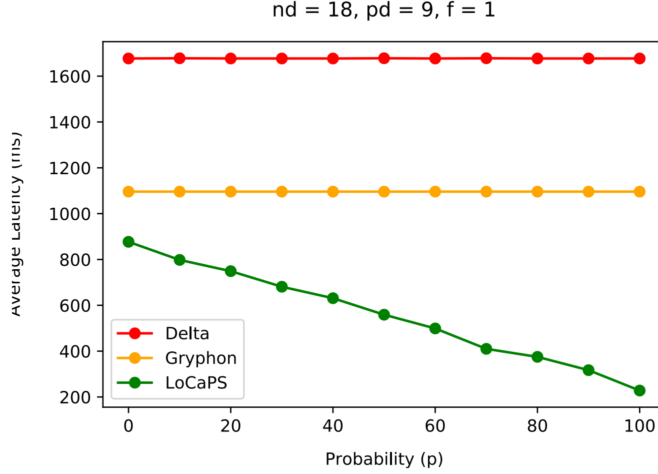


(a) Comparison with different network diameters

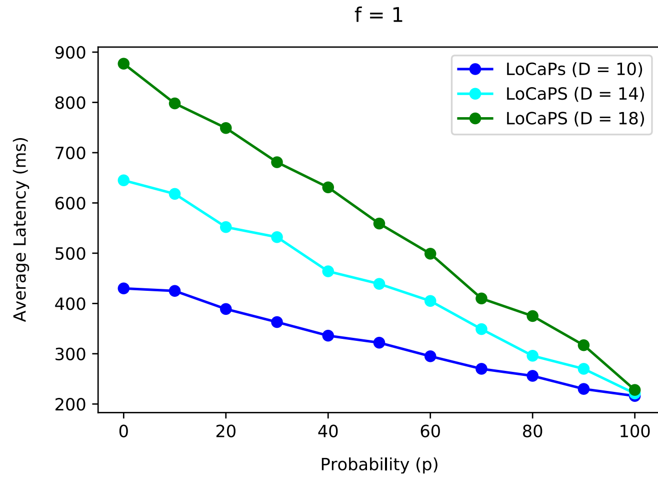


(b) Comparison with different publisher distances

Figure 6.1: LoCaPS vs Delta under different settings



(a) Comparison with different probabilities

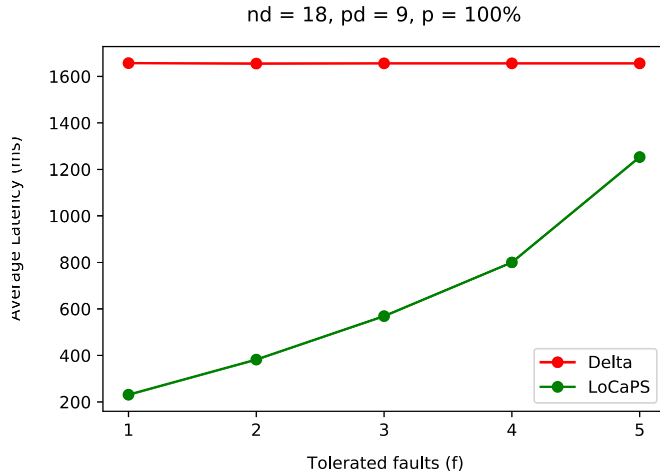


(b) Comparison with different probabilities and network diameters for LoCaPS

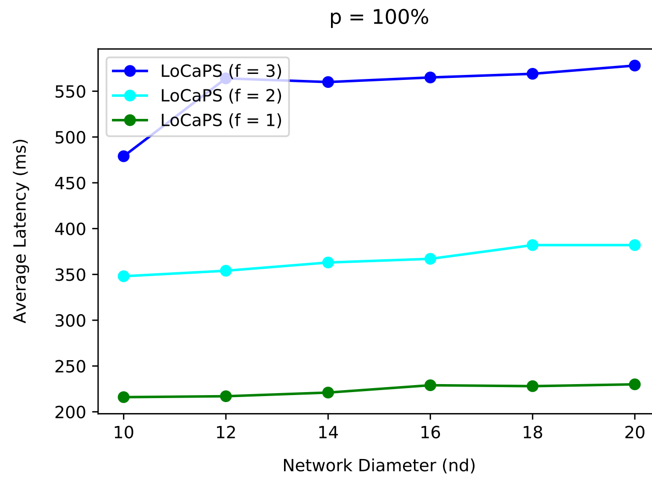
Figure 6.2: LoCaPS vs the Delta and Gryphon algorithms

6.4 Analyzing Subscription Coverage

Figure 6.2(a) illustrates the average subscription latency for the three systems when there is a publisher on the graph root and stable subscriptions in the systems, whereas $f = 1$ is set. We vary the probability that a new subscription on the tree leaves is already covered. In this case, the latency of the Delta and Gryphon systems is constant, since these do not use subscription coverage to decrease latency. Thus, in these systems, a subscription will have to be propagated the same number of steps every time a new subscriber joins, causing a constant observed value. Subscription latency in LoCaPS decreases as the coverage probability increases. When a new subscriber joins the system, its subscription will not have to be propagated to a publisher if it is covered by a deployed one. Thus, we observe a steady decrease in latency, as brokers only propagate the subscription to partial pivot nodes. In this case, these are in the local



(a) Comparison with different values for f



(b) Comparison with different values of f and network diameter for LoCaPS

Figure 6.3: LoCaPS vs Delta under different neighborhood scenarios

neighborhood of the subscriber's server.

Figure 6.2(b) illustrates the average latency reached by LoCaPS with the parameters mentioned in the scenario for Figure 6.2(a). Once again, we vary the coverage probability as well as the network diameter. In this case, we also note that the latency will tend to the same value, regardless of the network diameter. This tendency is due to the latency being proportional to f, in case another subscription covers the new one. The latency depends on f due to the partial pivot broker set being in the local neighborhood of the subscriber's server. This value is independent of the network diameter, as such we observe the same latency values when we have full coverage probability.

6.5 Analyzing Localized Algorithms

Figure 6.3(a) presents the average subscription latency when we vary the neighborhood size of a node, defined by the fault tolerance value f. In this case, there is a publisher on the root, and every new subscription is already covered. For the Delta system, the latency is constant for the different neighborhood sizes, since this only depends on the network diameter. Although this system uses localized information, it does not leverage it with subscription coverage to reduce latency. In the case of LoCaPS, we observe that the latency increases with the neighborhood size. As we have observed, the latency will depend on the f value, which dictates neighborhood size, and therefore the position of the partial pivot brokers.

In Figure 6.3(b), we use the same parameters as in Figure 6.3(a) and focus on the LoCaPS system. In this scenario, we also vary the network diameter. We can observe that the latency only depends on the broker's neighborhood size, in case the subscription is already covered. As expected, as the size of the neighborhood increases, so does the latency. It will remain constant independently of the network diameter since a new subscription is always covered.

6.6 Discussion

There are two main points relevant when discussing the evaluation. The first one is regarding the application of the *sufficient* and *necessary and sufficient* conditions to enforce GCD and GFD on the algorithms. The Delta system relies on full-stability to provide guarantees, which has the highest impact on latency, by making it depend on the network diameter. Both LoCaPS and Gryphon verify the necessary and sufficient condition to enforce semantics, making latency proportional only on the distance to the publisher. However, Gryphon only provides GFD and does so at the expense of using event flooding.

The second essential topic to analyze is how the subscription coverage optimization can impact the latency. The Delta system does not consider this optimization. As such, the subscription latency is always proportional to the network diameter. Although Gryphon uses this technique, it does not use it to reduce latency, but to reduce the routing table size. Thus, the observed latency is dependent on the distance to the publisher for every subscription. LoCaPS leverages this method to offer low latency when already deployed subscriptions cover new ones. Although both Delta and LoCaPS use localized information for their algorithms, only LoCaPS uses this to its advantage when it comes to reducing latency. In our system, the latency will depend solely on the f value, due to the algorithm used to find the partial pivot sets.

6.7 Summary

This chapter presented the evaluation of LoCaPS. By using a simulator, we made three sets of comparisons. The first compares the performance of full-stability versus using the necessary and sufficient condition. The conclusion is that systems that rely on full-stability will have a higher subscription latency. The second set compares different algorithms regarding subscription coverage. The general outcome is that the subscription coverage optimization considerably reduces observed latency if applied in conjunction with the weaker Gapless Causal Delivery requirement. Finally, we compare localized publish-subscribe systems and their subscribers' observed latency. We observe that, when used with subscription coverage, this information facilitates the implementation of the optimized algorithm and reduces the latency.

Chapter 7

Conclusions

In this thesis, we have addressed the problem of implementing the publish-subscribe paradigm in large scale-systems using a distributed broker network. We gave particular emphasis to the semantics of the subscribe operation and how these semantics affect the subscription latency, i.e., the time a subscriber needs to wait before it starts receiving events associated with a given subscription. We have studied the necessary and sufficient conditions that systems need to achieve to offer different reliability semantics to subscribers, namely Gapless FIFO delivery and Gapless Causal delivery. Our requirements are weaker than those typically used in previous systems. These have allowed us to implement LoCaPS, a reliable causal publish-subscribe system that offers low subscription latency, namely, when an already deployed subscription covers a new one. An experimental evaluation of LoCaPS shows that it can achieve significantly better performance than previous state-of-the-art solutions.

7.1 Future Work

Several aspects are worth exploring in the future. Currently, LoCaPS only considers static overlays. Support for dynamic overlays, meaning the network graph and broker position could change by having new nodes joining and old ones leaving the system, is something that definitely should be added to the system. By positioning relevant publishers closer to interested subscribers, we could observe further improvements in latency.

Due to a combination of time constraints and lack of a suitable test-bed, able to support the deployment of very large overlays, we have resorted to simulations to evaluate LoCaPS. The deployment and evaluation of LoCaPS in a real deployment is an interesting avenue for future work.

Bibliography

[ASS+99] Marcos K Aguilera, Robert E Strom, Daniel C Sturman, Mark Astley, and Tushar D Chandra. Matching events in a content-based subscription system. In *ACM 18th Symposium on Principles of Distributed Computing*, pages 53–61, Atlanta (GA), USA, 1999.

[BBPQ12] Roberto Baldoni, Silvia Bonomi, Marco Platania, and Leonardo Querzoni. Dynamic message ordering for topic-based publish/subscribe systems. In *IEEE 26th International Parallel and Distributed Processing Symposium*, pages 909–920, Shanghai, China, 2012.

[BRVR17] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: A distributed metadata service for causal consistency. In *12th European Conference on Computer Systems*, pages 111–126, Belgrade, Serbia, 2017.

[BSB+02] Sumeer Bhola, Robert Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *IEEE International Conference on Dependable Systems and Networks*, pages 7–16, Bethesda (MD), USA, 2002.

[BZA03] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably supporting durable subscriptions in a publish/subscribe system. In *IEEE International Conference on Dependable Systems and Networks*, pages 57–66, San Francisco (CA), USA, 2003.

[CAR05] Nuno Carvalho, Filipe Araujo, and Luis Rodrigues. Scalable qos-based event routing in publish-subscribe systems. In *IEEE 4th International Symposium on Network Computing and Applications*, pages 101–108, Cambridge (MA), USA, 2005.

[CDNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-

based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[CF03] Raphaël Chand and Pascal Felber. A scalable protocol for content-based routing in overlay networks. In *IEEE 2nd International Symposium on Network Computing and Applications*, pages 123–130, Cambridge (MA), USA, 2003.

[CF04] Raphael Chand and Pascal Felber. Xnet: a reliable content-based publish/subscribe system. In *IEEE 23rd International Symposium on Reliable Distributed Systems*, pages 264–273, Florianpolis, Brazil, 2004.

[CL85] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[CMPC03] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In *2nd International Workshop on Distributed Event-based Systems*, pages 1–8, San Diego (CA), USA, 2003.

[CMPC04] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In *IEEE 24th International Conference on Distributed Computing Systems*, pages 552–561, Tokyo, Japan, 2004.

[CP05] Paolo Costa and Gian Pietro Picco. Semi-probabilistic content-based publish-subscribe. In *IEEE 25th International Conference on Distributed Computing Systems*, pages 575–585, Columbus (OH), USA, 2005.

[CRW01] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[dAAD+17] Joao Paulo de Araujo, Luciana Arantes, Elias P Duarte, Luiz A Rodrigues, and Pierre Sens. A publish/subscribe system using causal broadcast over dynamically built spanning trees. In *IEEE 29th International Symposium on Computer Architecture and High Performance Computing*, pages 161–168, São Paulo, Brazil, 2017.

[dAADJ+19] João Paulo de Araujo, Luciana Arantes, Elias P Duarte Jr, Luiz A Rodrigues, and Pierre Sens. Vcube-ps: A causal broadcast topic-based publish/subscribe system. *Elsevier Journal of Parallel and Distributed Computing*, 125:18–30, 2019.

[EFGK03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[EG02] Patrick Th Eugster and Rachid Guerraoui. Probabilistic multicast. In *International Conference on Dependable Systems and Networks*, pages 313–322, Bethesda (MD), USA, 2002.

[EPB13] Christian Esposito, Marco Platania, and Roberto Beraldi. Reliable and timely event notification for publish/subscribe services over the internet. *IEEE/ACM Transactions on Networking*, 22(1):230–243, 2013.

[KJ09] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Reliable and highly available distributed publish/subscribe service. In *IEEE 28th International Symposium on Reliable Distributed Systems*, pages 41–50, Niagara Falls (NY), USA, 2009.

[KJ11] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Partition-tolerant distributed publish/subscribe systems. In *IEEE 30th International Symposium on Reliable Distributed Systems*, pages 101–110, Madrid, Spain, 2011.

[Lam19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. pages 179–196, 2019.

[LSB06] Cristian Lumezanu, Neil Spring, and Bobby Bhattacharjee. Decentralized message ordering for publish/subscribe systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 162–179, Melbourne, Australia, 2006.

[MJ09] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *9th International Conference on Peer-to-Peer*, pages 99–100, Seattle (WA), USA, 2009.

[NDA+14] Hiroki Nakayama, Dilawaer Duolikun, Ailixier Aikebaiery, Tomoya Enokidoz, and Makoto Takizaw. Causal order of application events is p2p publish/subscribe systems. In *IEEE 17th International Conference on Network-Based Information Systems*, pages 444–449, Vienna, Austria, 2014.

[PLTP08] Cássio Pereira, Daniel Lobato, César Teixeira, and Maria Pimentel. Achieving causal and total ordering in publish/subscribe middleware with DSM. In *3rd Workshop on Middleware for Service Oriented Computing*, pages 61–66, Leuven, Belgium, 2008.

[PRS96] Ravi Prakash, Michel Raynal, and Mukesh Singhal. An efficient causal ordering algorithm for mobile computing environments. In *IEEE 6th International Conference on Distributed Computing Systems*, pages 744–751, Hong Kong, 1996.

[SDJ16] Pooya Salehi, Christoph Doblander, and Hans-Arno Jacobsen. Highly-available content-based publish/subscribe via gossiping. In *ACM 10th International Conference on Distributed and Event-based Systems*, pages 93–104, Irvine (CA), USA, 2016.

[SR19] Válter Santos and Luís Rodrigues. Localized reliable causal multicast. In *IEEE 18th International Symposium on Network Computing and Applications*, pages 1–10, Cambridge (MA), USA, 2019.

[ZSB04] Yuanyuan Zhao, Daniel Sturman, and Sumeer Bhola. Subscription propagation in highly-available publish/subscribe middleware. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 274–293, Toronto, Canada, 2004.