LoCaPS: Localized Causal Publish-Subscribe

(extended abstract of the MSc dissertation)

Filipa Salema Roseta Pedrosa
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—Publish-subscribe systems are a powerful abstraction to build distributed applications. This paper addresses the problem of reducing subscription latency in reliable publish-subscribe systems. In most systems that offer reliability guarantees, a subscriber needs to wait until its subscription has been propagated throughout the entire system, and known by all relevant publishers, before starting to receive events. Interestingly, this may happen even when a previously deployed subscription covers a new one. In this paper, we study the properties that need to be satisfied to reduce the subscription latency and propose a new publish-subscribe system that leverages causal order multicast to offer low subscription latency when these conditions are met.**

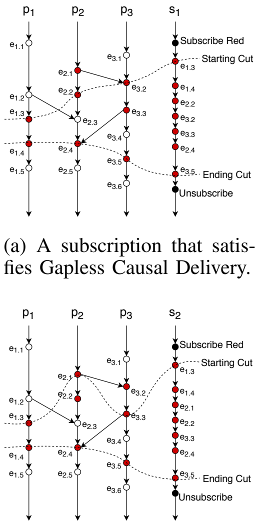
I. Introduction

The publish-subscribe abstraction [1] 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 may 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. In this paper, we are particularly interested in studying the *subscription latency* in reliable publish subscriber systems. This latency is the delay that occurs from the time a subscriber performs a subscription and the time it receives the first event from any publisher. This delay varies between different systems: it is a function of the number of steps the algorithm needs to execute, in the routing overlay, to deploy the state required to enforce the desired reliability guarantees.

We consider 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 [2], [3] is a well-known system that offers this type of reliability. GFD is defined for each publisher, regardless of the interactions between publishers. GCD is a stronger reliability criterion for publish-subscribe systems that avoids the anomalies that may result from missing cause-effect relations among events. These are created as a result of the interactions among different publishers. Examples of systems that offer GCD are [4], [5], [6], [7], [8]. In this paper, we answer 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 leveraging causality in the propagation of control messages among event brokers, we show that systems can guarantee GCD quickly. As soon as a subscription is known by all brokers *in a single path* from *one* of the publishers to the subscriber it becomes possible to enforce GCD. This property is substantially weaker than the property that is used by existing implementations, such as [9], [8], that 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 may exist 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 interesting question we address in this paper 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 [10], [11]. 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 the subscription latency for covered subscriptions. Finally, we introduce LoCaPS, a novel algorithm supporting GCD that leverages our findings. LoCaPS expands previous work on localized causal multicast [12] to implement efficient subscription protocols. These offer low subscription latency when suitable coverage relations are observed between a new subscription and previously deployed ones.

The rest of the paper is organized as follows. Section II presents the concepts related to our work and defines the GFD and GCD subscription semantics. In Section III we present two sufficient conditions to support these semantics. Section IV details a necessary (weaker) condition that provides lower subscription latency, and we present an algorithm based on said condition. In Section V, we discuss



(a) A subscription that satisfies Gapless Causal Delivery.

(b) A subscription that satisfies Gapless FIFO Delivery.

Figure 1. Subscription semantic examples.

how we can leverage already deployed subscriptions to reduce the subscription latency and present algorithms that support these optimizations. Section VI describes LoCaPS, a system that materializes our findings; we experimentally compare its performance with previous work in Section VII. Section VIII discusses related work and, finally, Section IX concludes the paper.

II. Subscription Semantics

In this section, we provide a precise characterization of Gapless FIFO Delivery (GFD) and Gapless Causal Delivery (GCD).

A. Definitions

Subscription semantics are defined using the concepts of *event graph*, *subscription history*, subscription *starting cut*, and subscription *ending cut*, that are introduced with the help of the examples depicted in Figure 1.

Assuming a publish-subscribe system, with multiple publishers and subscribers, where events may be causally related. We use the notion of causal order from Lamport [13]. If two events $e_{1.1}$ and $e_{1.2}$ produced by the same publisher p_1, where $e_{1.2}$ is produced after $e_{1.1}$, then $e_{1.2}$ may be causally dependent of $e_{1.1}$, denoted $e_{1.1} \rightarrow e_{1.2}$. Publishers may also subscribe to events from other publishers, creating potential cause-effect relations between events from different publishers. Again, we use Lamport's definition, and 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 defines a partial order on the events produced in the system, represented by an event graph where edges represent causal relations. The events may have different topics and/or contents: in the examples from Fig. 1, we have white and red content events.

The sequence of events delivered to a subscriber, associated with its subscription, defines the subscription event history. A special subscribe event bounds the subscription history, which is locally generated at the subscriber when it issues the subscription. Another special unsubscribe event is generated when the subscriber terminates the subscription. The set of events composed by the first event from each publisher that appears in the subscription history defines a cut in the event graph, which is the subscription *starting cut*. Similarly, the set of events composed by the last event from each publisher that appears in the subscription history defines a cut in the event graph, which is the subscription *ending cut*.

B. Semantics

We consider two different semantics that can be found in the literature, namely Gapless FIFO Delivery and Gapless Causal Delivery. We define them precisely 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 $H_i(S_i)$ to be the subscription history for subscriber s_i, then $H_i(S_i, p_j)$ is the set of events from $H_i(S_i)$ that have been published by publisher p_j. For each publisher p_j, we denote $e_{j.start}$ the event from that publisher that defines the starting cut, and $e_{j.end}$ the event from that publisher that defines the end cut of $H_i(S_i)$. Finally, let *matching*(G, S_i, p_j) be the set of events from the event graph G that have been published by p_j after $e_{j.start}$ and before $e_{j.end}$ that match the subscription S_i.

Definition 1. ***Gapless FIFO Delivery (GFD)****: There are no constraints on the set of events that belong to the starting cut and ending cut of a subscription S_i. Only $H_i(S_i, j) =$ 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.*

Definition 2. ***Gapless Causal Delivery (GCD)****: The subscription must satisfy the conditions of definition 1 plus the following property. Let j and k be two distinct publishers, let $e_{j.cause}$ be some event such that $e_{j.cause} \in H_i(S_i, p_j)$, and let $e_{k.effect} \in G : e_{j.cause} \rightarrow e_{k.effect} \rightarrow e_{k.end}$. Gapless Causal Delivery states that $e_{k.effect} \in H_i(S_i)$.*

Figure 1 illustrates the difference between GFD and GCD. The two subscription event histories have been obtained from the same event graph. In both cases, the subscriptions match red events only, however, the starting cut in each subscription is different. Subscription S_1, in Figure 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 1(b) satisfies GFD but not GCD: 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 $e_{2.1} \rightarrow e_{3.2} \rightarrow e_{3.3}$.

III. Sufficient Conditions to Enforce the Semantics

In this section, we focus on a particular class of implementations of publish-subscribe systems, 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 GFD and GCD.

A. 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*, while message brokers are called *servers*. Clients can only connect to one of the available servers at any given time. A server can attend to multiple clients.

Publishers produce events that need to be delivered to the 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 network, we consider the case where event flooding is avoided. Servers keep two different types of routing tables, namely: *subscription routing tables* and *event routing tables*. The former is used to forward subscriptions to publishers, and the later is used to forward events to subscribers. Publishers are required to send special *advertisement* messages to build the subscription routing tables. Advertisements are the only control messages flooded in the network. In the following, we describe the creation of the aforementioned routing tables for a publish-subscribe system with these characteristics. An advertisement includes a template of the type of events provided by the publisher. Advertisements are flooded in the broker network and are used to populate the subscription routing table at every server.

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 the identifiers of both the client and broker, as well as constraints for the matching events. Subscriptions are propagated to all publishers that match the subscription, along the paths established during advertisement propagation. When a subscription is propagated, the following steps are performed by the server: i) an entry to the subscriber, associated with the server edge, is added to the event routing table; ii) the subscription is propagated throughout the paths to matching publishers. When a *broker*$_k$ receives some event e from publisher p_i, via link l, it checks if there is one or more matching subscription S_i. This match is performed by verifying the local event routing table of *broker*$_k$. If there is no subscription that matches e, then the broker discards the event. If there is, let L be the set of downstream links that are in a path from *broker*$_k$ to s_i (L is stored in *broker*$_k$'s event routing table). For every link $l \in L$, the *broker*$_k$ adds l to a set named *link-matches*(k, e). The *broker*$_k$ does this for every subscription S_i that matches the event e. Finally, e is forwarded on all links in *link-matches*(k, e).

B. Subscription Stability

We now define a number of properties that are relevant to capture the sufficient conditions for achieving GFD and GCD. Let S_i be a subscription performed by subscriber s_i.

Definition 3. ***Link Stability**: Let broker$_a$ and broker$_b$ be two direct neighbours 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 broker$_a$ and broker$_b$.*

Definition 4. ***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).*

Definition 5. ***Publisher Stability**: Let p_j be a publisher and S_i a given subscription performed by subscriber s_i. 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).*

Definition 6. ***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).*

C. Stablity-Based Conditions

Using the definitions above, we can define two sufficient conditions to enforce GFD and GCD.

Condition 1. ***Sufficient condition for GFD**: 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 GFD it is sufficient that e has been sent by p_j after pub-stable(S_i, p_j).*

Condition 2. ***Sufficient condition for GCD**: 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).*

D. Evaluating Full Stability

The sufficient conditions expressed above are relevant because they are used by different publish-subscribe systems, to enforce both GFD and GCD (for instance, by [9], [14], [8]). An example of a system that offers GFD and that relies on full stability is described in [9], [14]. In this system, an acknowledge needs to be received from every broker in the path to publishers before the subscription returns. When a subscriber joins, its broker sends a subscription message to all neighbors in the path to the subscriber. The neighbors, in turn, send the subscription to their neighbors, until the subscription messages reach the publishers, where an acknowledgment is generated and then propagated back to the subscriber using the reverse path of the subscription. Each broker collects acknowledgments from all downstream brokers before propagating 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 broker.

It is also possible to find examples of systems that use full stability to implement GCD, such as VCube-PS [8]. Similarly to [9], [14] above, VCube-PS 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. It then waits for every broker to acknowledge the subscription. A subscriber s_i knows it is *F-stable*(S_i) as soon as it receives all acknowledgement messages.

E. 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) is established but *before pub-stable*(S_i, p_k) is verified. 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 stronger than needed, and it is possible to define weaker necessary conditions that allow both GFD and GCD to be enforced more efficiently.

IV. A Necessary (and Sufficient) Condition to Enforce the Semantics

As noted above, although the sufficient conditions expressed in Section III are sufficient to enforce reliability, they may be more restrictive than necessary. In fact, Gryphon [3], [2] is able to enforce GFD based on a weaker guarantee, namely it starts delivering events as long as one (and only one) of the paths P_k to the subscriber s_i is *path-stable*(S_i, P_k). However, Gryphon does this at the cost of resorting to flooding, when brokers propagate events on non-stable paths. This raises the interesting question of knowing if it is possible to use weaker conditions while still avoiding event flooding. Below, we show that this is, in fact, possible.

A. Causality-Based Condition

Our work departs from the following observation: GCD requires events to be delivered in causal order, regardless of the algorithm used to identify the subscription starting cut. Thus, any algorithm that implements GCD must necessarily include mechanisms to keep track of causal dependencies and to ensure the delivery of messages in causal order. An algorithm to process subscriptions may then leverage causality to define a necessary and sufficient condition that is weaker than the sufficient conditions presented before. Namely:

Condition 3. ***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$_i$. Let e be some event from publisher p_j received by broker$_i$ 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, contrary to Conditions 1 and 2, Condition 3 does not require *all paths* from the publisher (Condition 1) to the subscriber or *all paths from all publishers* (Condition 2) to the subscriber to be stable; 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 larger subscription latencies than GFD. Another advantage of Condition 3 is that it is possible to derive simple algorithms that allow subscribers to check if the condition is met.

B. Leveraging Causality

We now describe the generic subscription propagation algorithm that leverages Condition 3. The algorithm, depicted in Algorithm 1, assumes a system with the characteristics described in Section III-A. When a publisher p_j receives a subscription S_i from subscriber s_i on a given path P_k, p_j knows that *path-stable*(S_i, P_k). Then p_j knows that Condition 3 has been met. To announce this , the publisher p_j issues a *marker*, $M^j_{S_i}$, to all subscribers that subscribe to p_j (line 14). When another publisher p_k receives the marker $M^j_{S_i}$, it will also become aware that the necessary and sufficient condition has been met. This publisher then also announces this fact by also publishing the marker $M^k_{S_i}$ to all its subscribers (line 21). Notice that, in this way, every publisher p_x that may send an event in the causal future of the marker $M^j_{S_i}$, also sends a marker $M^x_{S_i}$. The set of markers define the subscription starting cut (line 23). More precisely, the first event received from p_j, after receiving its marker $M^j_{S_i}$, belongs to the starting cut of the subscription (line 30). Events from any publisher p_j, sent before the marker $M^j_{S_i}$, are discarded by the subscriber s_i. Events sent after the marker are accepted (line 31).

The reader may notice the similarities between the algorithm above and the Chandy-Lamport [15] 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 producing, forwarding, and receiving events in the broker network overlay.

C. Correctness

We now prove the correctness of Condition 3 and of Algorithm 1.

Theorem 1. *Condition 3 is a necessary condition.*

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: procedure PROCESS(subscription, $s_i, S_i$) at $b_k$
 9:     UPDATEEVENTROUTINGTABLE($s_i, S_i$)
10:     SUBSCRIPTIONFORWARD(subscription, $s_i, S_i$)
11:     if $\exists p_x \in pubs(b_k) : matches(p_x, S_i)$ then
12:         if $\neg markersent[p_x, s_i, S_i]$ then
13:             $markersent[p_x, s_i, S_i] \leftarrow$ true
14:             EVENTFORWARD(event, $p_x$, MARKER($s_i, S_i$))
15: procedure PROCESS(event, $p_j$, MARKER($s_i, S_i$)) at $b_k$
16:     $stable\text{-}paths \leftarrow stable\text{-}paths \cup \{(b_k, p_j, S_i)\}$
17:     EVENTFORWARD(event, $p_j$, MARKER($s_i, S_i$))
18:     if $\exists p_x \in pubs(b_k) : matches(p_x, S_i)$ then
19:         if $\neg markersent[p_x, s_i, S_i]$ then
20:             $markersent[p_x, s_i, S_i] \leftarrow$ true
21:             EVENTFORWARD(event, $p_x$, MARKER($s_i, S_i$))
22:     if $\exists s_i \in subsc(b_k) : starting\text{-}cut[S_i, p_j] = \perp$ then
23:         $starting\text{-}cut[S_i, p_j] \leftarrow$ MARKER
24: procedure PUBLISH($p_j$, EVENT($e$)) at $b_k$
25:     EVENTFORWARD(event, $p_j$, EVENT($e$))
26: procedure PROCESS(event, $p_j$, EVENT($e$)) at $b_k$
27:     EVENTFORWARD(event, $p_j$, EVENT($e$))
28:     if $\exists s_i \in subsc(b_k) : matches(e, S_i) \wedge starting\text{-}cut[S_i, p_j] \neq \perp$
    then
29:         if $starting\text{-}cut[S_i, p_j] =$ MARKER then
30:             $starting\text{-}cut[S_i, p_j] \leftarrow e$          ▷ $S_i$'s starting cut
31:         DELIVER($s_i, e$)
```

Proof: Assume that broker 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 $e_1 \rightarrow e_2$, $matches(e_1, S_i) \wedge matches(e_1, S_j)$, $matches(e_2, S_i) \wedge \neg matches(e_2, S_j)$. Assume that $F\text{-}stable(S_i)$ but that Condition 3 is not satisfied, i.e., there is no P_k from p_k to s_j, such that $path\text{-}stable(S_j, P_k)$. Thus, b_k that serves publisher p_k is not aware of subscription S_j. Because b_k is not aware of S_j it will drop e_2, thus accepting e_1 would violate both GFD and GCD. ■

Theorem 2. *Condition 3 is a sufficient condition.*

The proof is based on the correctness of Algorithm 1, which depends solely on Condition 3. Intuitively, the algorithm works because the marker M_{S_i} causally depends on S_i (i.e, $S_i \rightarrow M_S$). Thus, the marker is always delivered to any broker *after* S_i has been delivered to that broker. 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 implies that, when a broker processes an event in the future of the subscription starting cut, it is already aware of the subscription and will not drop the event.

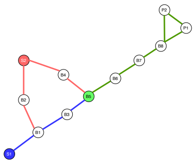


Figure 2. Single path subscription coverage.

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, in line 14, broker b_k that serves p_k sends marker $M^k_{S_i}$ as soon as P_k becomes stable. Therefore, we have $M^k_{S_i} \rightarrow e_1$. Assume that there is some event e_2 published by p_h, such that $e_1 \rightarrow e_2$. By transitivity of causality, we have $S_i \rightarrow M^k_{S_i} \rightarrow e_1 \rightarrow e_2$. From Algorithm 1, line 21, 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 $S_i \rightarrow M^k_{S_i} \rightarrow M^h_{S_i} \rightarrow e_2$. From Algorithm 1, line 23, 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 being aware of 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). ■

V. Leveraging Coverage

We now discuss how subscription coverage can help in further reducing the 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:

Definition 7. ***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 $matches(e, S_i)$.*

Definition 8. ***Link Coverage****: Let b_a and b_b be two direct neighbours 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 $link\text{-}covered(S_j, l_{ab})$.*

Definition 9. ***(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 $link\text{-}covered(S_j, l)$.*

A. 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:

Definition 10. ***Pivot Broker****: Let p_k be a publisher. Let s_i and s_j be two subscribers that perform subscription S_i (resp. 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$(p_k, s_i, s_j) = \{p_i, b_1, b_2, \dots, b_n\}$ 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 2. In this example, there is a subscription S_1, by subscriber s_1 that has already been deployed on the broker overlay (on paths from publishers p_1 and p_2 to s_1). These paths are represented by the blue and green links. 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), represented by the red and green links. The green portion of the paths is the portion common to both S_1 and S_2. 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 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) may 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, necessarily, be processed by a broker after S_j.

In the example of Figure 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 towards 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.

B. 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. Both the previously mentioned optimization in V-A and this new one can be performed in the same algorithm. Another subscription partially covers each of these paths.

Consider the example depicted in Figure 3. As before, there is a subscription S_1, by subscriber s_1 that has already been deployed on the broker overlay (on paths from publisher p_1 to s_1). Subscriber s_2 makes 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 $P_2 = \{p_1, b_1, b_2, b_4, b_6, b_8\}$. In the following we 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:

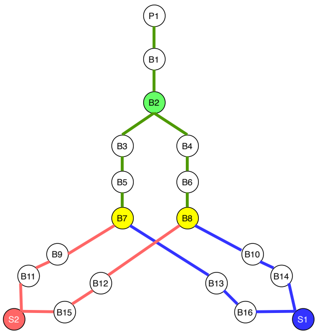


Figure 3. Multi-path subscription coverage.

Definition 11. ***Partial Pivot Set:*** *Let p_k be a publisher. Let s_i and s_j be two subscribers that perform some subscription S_i (resp. S_j) such that matches(p_k, S_i) and matches(p_k, S_j). Let $\mathcal{P}(p_k, s_i) = \{P_{i,1}, P_{i,2}, \dots, P_{i,n}\}$ be the set of paths from publisher p_k to subscriber s_i and $\mathcal{P}(p_k, s_j) = \{P_{j,1}, P_{j,2}, \dots, P_{j,n'}\}$ 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$(P_{i,x}, \mathcal{P}(p_k, s_j)) = \{p_1, b_1, b_2, \dots, b_n\}$ 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 3, *PPSet*$(p_1, S_1, S_2) = \{b_7, b_8\}$. One challenge in using partial pivots is that a partial pivot cannot proxy the publisher and send a marker on its behalf. This is because a partial pivot may only forward 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) can produce a partial marker for S_j on behalf of publisher p_k. The union of all partial markers from all partial pivots can be used 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 the 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 makes it harder to define the starting cut. When using a (full) marker, the first event e from publisher p_k, received after the marker 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 is no longer applies, and a more complex algorithm is needed to find a safe starting cut. We propose a novel 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 broker. Note that there is a different set of paths for each pivot broker. All events from p_k that are received via a given partial pivot, before a partial marker from that partial pivot is received, are discarded. All events from p_k that are received via a given partial pivot, after a partial marker from that partial pivot is received, are buffered. The receiver broker waits until all buffers have at least one event from p_k. When this condition is met, the most recent event e among all events in all buffers is selected to be part of the starting cut. From this point, all events in the future of e are accepted and all events in the past of e are discarded (including buffered events in the past of e).

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

C. Correctness

An extension to Algorithm 1, that includes both the single-prefix and the multi-prefix optimizations is presented in Algorithm 2. 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 V-B. Our proofs assume that Algorithm 2 is executed on top of a multicast layer that enforces causal order. 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 3. *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 subscrive s_j. We have 5 different cases:

case 1: $\not\exists S_1 : \mathit{PPSet}(p_1, S_1, S_j) \neq \emptyset \wedge \not\exists S_2 : \mathit{PPSet}(p_2, S_2, S_j) \neq \emptyset$. In this case, no optimization is triggered and the proof from Theorem 2 still applies.

case 2: $\not\exists S_1 : \mathit{PPSet}(p_1, S_1, S_j) \neq \emptyset \wedge \exists S_2 : \mathit{PPSet}(p_2, S_2, S_j) \neq \emptyset$. 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.

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

```
procedure PROCESS(subscription, $s_i, S_i$) at $b_k$
    UPDATEEVENTROUTINGTABLE($s_i, S_i$))
    SUBSCRIPTIONFORWARD(subscription, $s_i, S_i$)
    // publisher $p_x$ sends the marker (default)
    if $\exists p_x \in \mathit{pubs}(b_k) \cap \mathit{pubs}(b_k) : \mathit{matches}(p_x, S_i)$ then
        if $\neg \mathit{markersent}[p_x, s_i, S_i)]$ then
            $\mathit{markersent}[p_x, s_i, S_i] \leftarrow$ true
            EVENTFORWARD(event, $p_x$, MARKER($s_i, S_i$))
    // broker $b_k$ sends marker on behalf of $p_x$ (proxy marker)
    if $\exists p_x, S_x : b_k = \mathit{pivot}(p_x, S_x, S_i)$ then
        if $\mathit{covers}(S_x, S_i) \wedge \{(b_k, p_x, S_x)\} \in$ stable-paths then
            if $\neg \mathit{markersent}[p_x, s_i, S_i)]$ then
                $\mathit{markersent}[p_x, s_i, S_i] \leftarrow$ true
                EVENTFORWARD(event, $p_x$, MARKER($s_i, S_i$))
    // broker $b_k$ sends partial marker on behalf of $p_x$
    if $\exists p_x, S_x : b_k = \mathit{PPSet}(p_x, S_x, S_i)$ then
        if $\mathit{covers}(S_x, S_i) \wedge \{(b_k, p_x, S_x)\} \in$ stable-paths then
            if $\neg \mathit{pmarkersent}[p_x, s_i, S_i, S_x)]$ then
                $\mathit{pmarkersent}[p_x, s_i, S_i, S_x] \leftarrow$ true
                EVENTFORWARD(event, $p_x$, PMARKER($b_k, s_i, S_i, S_x$))
procedure PROCESS(event, $p_j$, PMARKER($b_x, s_i, S_i, S_x$)) at $b_k$
    $\mathit{pmarkers} \leftarrow \mathit{pmarkers} \cup \{(p_j, b_x, s_i, S_i, S_x)\}$
    EVENTFORWARD(event, $p_j$, PMARKER($b_x, s_i, S_i, S_x$))
    if $\exists p_x \in \mathit{subsc}(b_k) \cap \mathit{pubs}(b_k) : \mathit{matches}(p_x, S_i)$ then
        if $\neg \mathit{markersent}[p_x, s_i, S_i)]$ then
            $\mathit{markersent}[p_x, s_i, S_i] \leftarrow$ true
            EVENTFORWARD(event, $p_x$, MARKER($s_i, S_i$))
procedure PROCESS(event-message, $p_j$, EVENT($e$)) at $b_k$
    EVENTFORWARD(event-message, $p_j$, EVENT($e$))
    if $\exists s_i \in \mathit{subsc}(b_k)$ then
        if starting-cut$[S_i, p_j] \neq \perp$ then
            if starting-cut$[S_i, p_j] =$ MARKER then
                starting-cut$[S_i, p_j] \leftarrow e$
                DELIVER($s_i, e$)
            else if starting-cut$[S_i, p_j] \rightarrow e$ then
                DELIVER($s_i, e$)
        else if $\exists b_x \in \mathit{path}(e) : \{(p_j, b_x, s_i, S_i)\} \in \mathit{pmarkers}$ then
            $\mathit{buffer}[S_i, p_j, b_x] = \mathit{buffer}[S_i, p_j, b_x] \cup \{e\}$
            if $\exists S_j : \mathit{covers}(S_j, S_i) \wedge \forall b_x \in \mathit{PPSet}(p_j, S_i, S_j) : \mathit{buffer}[S_i, p_j, b_x] \neq \emptyset$ then
                $e \leftarrow$ MOSTRECENT($\mathit{buffer}[S_i, p_j, b_x], \forall b_x \in \mathit{PPSet}(p_j, S_i, S_j)$))
                starting-cut$[p_j, S_i] \leftarrow e$
                DELIVER($s_i, e$)
```

case 3: $\exists S_1 : \mathit{PPSet}(p_1, S_1, S_j) \neq \emptyset \wedge \not\exists S_2 : \mathit{PPSet}(p_2, S_2, S_j) \neq \emptyset$. If s_j delivers e_1 then, either p_1 sends $M^1_{S_j} \rightarrow e_1$ (no optimization was triggered) or every $b_x \in \mathit{PPSet}(p_1, S_1, S_j)$ sends a partial marker $\mathit{PM}^{1,k}_{S_j} \rightarrow e_1$, on behalf of p_1. Note that every path from p_1 to p_2 includes some $b_x \in \mathit{PPSet}(p_1, S_1, S_j)$ (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 $\mathit{PPSet}(p_1, S_1, S_j)$ which would be a contradiction). From causality, p_2 will receive $M^{1,x}_{S_j}$ from some $b_x \in \mathit{PPSet}(p_1, S_1, S_j)$ 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: $\exists S_1 : \mathit{PPSet}(p_1, S_1, S_j) \neq \emptyset \wedge \exists S_2 : \mathit{PPSet}(p_2, S_1, S_j) \neq \emptyset \wedge \mathit{PPSet}(p_1, S_1, S_j) \neq \mathit{PPSet}(p_2, S_1, S_j)$. The reasoning from case 3 also applies, and the proof is the same.

case 5: $\exists S_1 : \mathit{PPSet}(p_1, S_1, S_j) \neq \emptyset \wedge \exists S_2 : \mathit{PPSet}(p_2, S_1, S_j) \neq \emptyset \wedge \mathit{PPSet}(p_1, S_1, S_j) = \mathit{PPSet}(p_2, S_1, S_j)$. 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 $b_x \in \mathit{PPSet}(p_1, S_1, S_j)$ sends a partial marker $\mathit{PM}^{1,x}_{S_j} \rightarrow e_1$, on behalf of p_1. However, because $\mathit{PPSet}(p_1, S_1, S_j) = \mathit{PPSet}(p_2, S_1, S_j)$, every b_x that sends $\mathit{PM}^{1,x}_{S_j}$ also sends immediately $\mathit{PM}^{2,x}_{S_j}$ on behalf of p_2. Thus, because $e_1 \rightarrow e_2$, when e_2 is received by b_j a partial makrker $\mathit{PM}^{2,x}_{S_j}$ has lready is received from every $b_x \in \mathit{PPSet}(p_2, S_2, S_j)$ (toghether with $\mathit{PM}^{1,x}_{S_j}$), and thus, if e_1 is delivered, e_2 is also delivered. ■

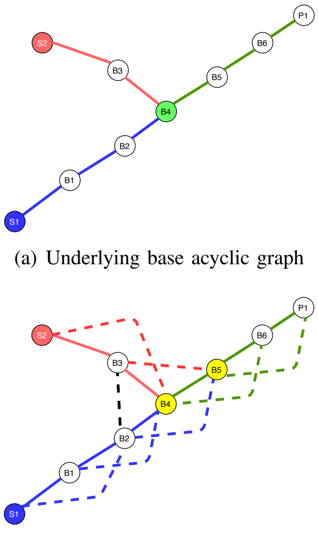
VI. An Implementation

In the previous section, we have proposed several optimizations that allow us to reduce subscription latency. Unfortunately, these optimizations are hard to implement in general topologies. In fact, in the general case, it may be 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 a marker directly from the publisher. This may explain why most implementations that offer GFD or GCD semantics fail to leverage the coverage property to speed up the subscription process. When suitable overlays are used, 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 that it is, in fact, possible.

We describe a new system, named LoCaPS, that leverages the necessary and sufficient condition presented in section IV and the optimizations from sections V-A and V-B to build an efficient publish-subscribe system. Our system is built on top of LoCaMu [12], a causal multicast substrate for publish-subscribe systems. LoCaPS also offers low subscription latency in favorable conditions. We start by describing LoCaMu and then describe how we can exploit the properties of LoCaMu for applying the optimizations proposed in Section V.

A. LoCaMu

LoCaMu [12] is a causal multicast substrate for publish-subscribe systems. LoCaMu does not support dynamic subscriptions: in its current form, all subscriptions must be statically deployed in the network before it starts operating. LoCaPS extends LoCaMu with dynamic (low latency) subscriptions. The main feature of LoCaMu is that it works 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 feature of LoCaMu that is relevant to LoCaPS is its particular broker topology, which is inherited from [14].



(a) Underlying base acyclic graph

(b) Extended graph ($f = 1$)

Figure 4. LoCaMu broker network topology.

The topology of the broker network used by LoCaMu is presented in Figure 4. LoCaMu organizes the brokers in an undirected acyclic graph, as depicted in Figure 4(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. To achieve fault-tolerance the underlying graph is augmented with additional links, that allow a path to circumvent f failed nodes. This is illustrated in Figure 4(b) for the case of $f = 1$. As such, LoCaMu can offer reliable causal delivery, and tolerate f faulty nodes in each local neighborhood.

B. LoCaPS

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, the optimization described in Section V-A cannot be applied. In the extended graph, *PPSet*s always have $f + 1$ members, this simplifies the use of the optimizations described in Section V-B.

Let S_j be some subscription issued by subscriber s_j that is covered by some subscription S_i issued by subscriber s_i. Let p_k be a publisher that matches both S_i and S_j. In LoCaPS, $\mathit{PPSet}(p_k, S_i, S_j)$ consists of both $\mathit{pivot}(p_k, S_i, S_j)$ on the underlying acyclic graph (in Figure 4(a), broker b_4), and the next f nodes on the underlying acyclic graph on the path to the publisher (in Figure 4(b), broker b_5). Thus LoCaPS implements Algorithm 1 on top of LoCaMu, using the rules above to define the *PPSet*. Two different scenarios can define the first event, from a given publisher p_k, to belong to the starting cut of a given subscription. Either by a full marker sent by the pivot broker directly or by $f + 1$ partial markers from a given *PPSet*. After a subscriber receives the markers, as described in Algorithm 1, LoCaPS can start to deliver events to it without violanting the GCD semantic.

VII. Evaluation

In this section, we evaluate the performance of LoCaPS, in terms of subscription latency. We compare the observed latency by the subscribers with two of the systems from the related work that most resemble ours, namely the δ-fault-tolerant [14] (Delta) and Gryphon systems [2], [3]. The Delta system ensures a GFD to the subscribers and also uses the concept of neighborhood and localized information. The Gryphon system provides a GFD to the 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 the network diameter, the distance to the publishers, and the size of the neighborhood. Another parameter that we also consider in the evaluation is the likelihood that a new subscription is covered by other subscriptions, previously deployed on the system. To perform the evaluation we use the Peersim [16] simulator with an extension that simulates network latency. We consider an average latency of $50ms$ between brokers, which approximates thee average latency between google datacenters in North America. We consider that the time required to process the messages on the brokers is negligible. We have chosen a binary tree as out network topology. Subscribers are placed on the leaves, to not impair the Delta system, where the latency always depends on the network diameter. In the Gryphon system, subscribers must always be placed on the tree leaves and the publishers on the tree root. The network diameter is set at 18 in a network with 512 brokers, unless stated otherwise.

Figure 5(a) illustrates the average latency observed by subscribers from both systems when the network diameter is increased. In this scenario, the publishers are always at a 5 hop distance from subscribers. In Figure 5(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. 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.

Figure 5(c) illustrates the average subscription latency in the three systems when there is a publisher on the graph root and stable subscriptions in the systems. We set $f = 1$. 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. Subscription latency in LoCaPS decreases as the coverage probability increases. Figure 5(d) illustrates the average latency reached by LoCaPS with the parameters mentioned in the scenario for Figure 5(c). 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 is because the latency is proportional to the value of f.

Figure 5(e) 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. In the case of LoCaPS, we observe that the latency increases with the neighborhood size. In Figure 5(f), we use the same parameters as in Figure 5(e). 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.

VIII. Related Work

It is possible to find in the literature systems that focus on decreasing the subscription latency. However, most provide a best-effort semantic to their subscribers, which is characterized by offering no guarantees. One example of this approach is the Semi-Probabilistic Pub-Sub [17] system. In this system, the subscription is only known by a subscriber's neighborhood. Therefore, the latency depends on the neighborhood size.

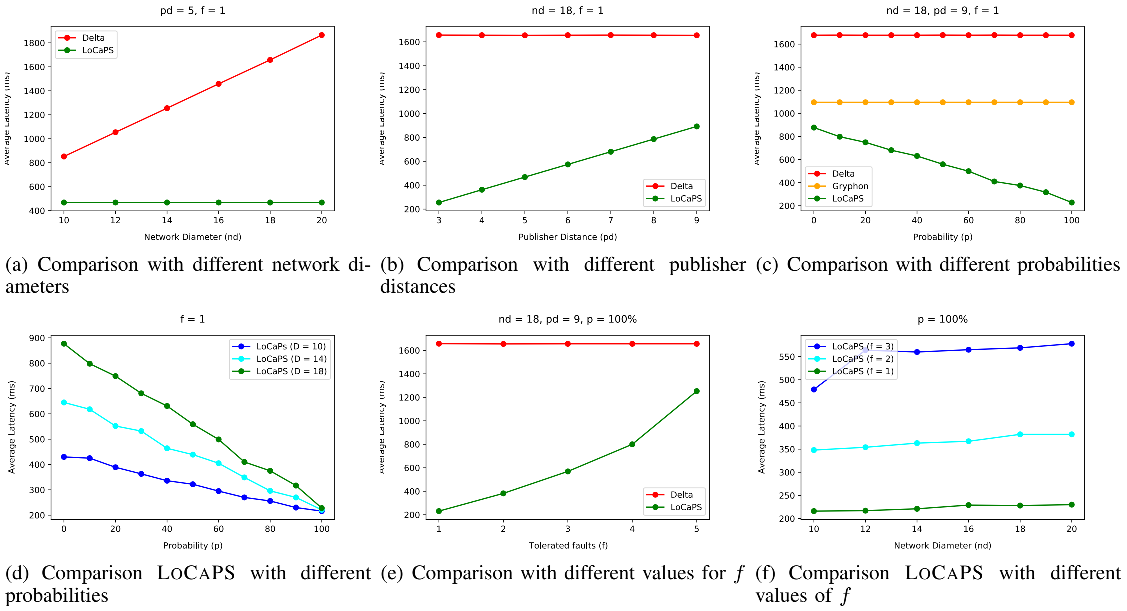
One way to reduce latency is to leverage subscription coverage. However, most systems that use coverage, such as the SIENA [10] and GEPS [11], provide only a best-effort semantics for the subscriptions.

Gryphon [2], [3] is a well-known system that provides a GFD semantic. Similar to LoCaPS, Gryphon also relies on causality to ensure reliability. In comparison to our work, the most relevant limitation of Gryphon is that it cannot leverage subscription coverage to reduce the subscription latency. In Gryphon, reliability is only guaranteed if the subscription is acknowledged by the publisher, regardless of the existence of other covering subscriptions. The same limitation applies to the δ-fault-tolerant system [14].

P2PPS [7] is a P2P (Peer-to-Peer) publish-subscribe system that offers GCD. Unlike our system, this work does not address the problem of reliability when considering dynamic subscriptions. The system described in [6] also supports GCD but requires the use of an external Distributed Shared Memory system to share information among brokers. As such, scalability is limited. Finally, like LoCaPS, VCube-PS [8] also exploits the use of a causal broadcast primitive [5] to offer GCD. However, unlike our system, they are not able to exploit subscription coverage, and all subscriptions need to be propagated and acknowledged by the entire network.

IX. Conclusion

In this paper, we have studied the necessary and sufficient conditions that need to be met to offer different reliability semantics to subscribers, namely Gapless FIFO delivery and Gapless Causal delivery. Our conditions 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. In particular, when a subscription is covered by another already deployed in the system. An experimental evaluation



(a) Comparison with different network diameters

(b) Comparison with different publisher distances

(c) Comparison with different probabilities

(d) Comparison LoCaPS with different probabilities

(e) Comparison with different values for f

(f) Comparison LoCaPS with different values of f

Figure 5. Performance of LoCaPS vs. performance of the Delta and Gryphon algorithms.

of LoCaPS shows that it can achieve significantly better performance than previous state-of-the-art solutions.

Acknowledgments

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.

References

[1] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, jun 2003.

[2] Y. Zhao, D. Sturman, and S. Bhola, "Subscription propagation in highly-available publish/subscribe middleware," in *Middleware*, Toronto, Canada, oct 2004.

[3] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *DSN*, Bethesda (MD), USA, jun 2002.

[4] G. Cugola, E. Di Nitto, and A. Fuggetta, "The jedi event-based infrastructure and its application to the development of the opss wfms," *IEEE Transactions on Software Engineering*, vol. 27, no. 9, sep 2001.

[5] R. Prakash, M. Raynal, and M. Singhal, "An efficient causal ordering algorithm for mobile computing environments," in *ICDCS*, Hong Kong, may 1996.

[6] C. Pereira, D. Lobato, C. Teixeira, and M. Pimentel, "Achieving causal and total ordering in publish/subscribe middleware with DSM," in *MW4SOC*, Leuven, Belgium, dec 2008.

[7] H. Nakayama, D. Duolikun, A. Aikebaiery, T. Enokidoz, and M. Takizaw, "Causal order of application events is p2p publish/subscribe systems," in *NBiS*, Vienna, Austria, sep 2014.

[8] J. de Araujo, L. Arantes, E. Duarte Jr, L. Rodrigues, and P. Sens, "Vcube-ps: A causal broadcast topic-based publish/subscribe system," *Journal of Parallel and Distributed Computing*, jul 2018.

[9] R. Kazemzadeh and A. Jacobsen, H, "Reliable and highly available distributed publish/subscribe service," in *SRDS*, Niagara Falls (NY), USA, sep 2009.

[10] A. Carzaniga, D. Rosenblum, and A. Wolf, "Design and evaluation of a wide-area event notification service," *ACM TOCS*, vol. 19, no. 3, aug 2001.

[11] P. Salehi, C. Doblander, and H.-A. Jacobsen, "Highly-available content-based publish/subscribe via gossiping," in *DEBS*, Irvine (CA), USA, jun 2016.

[12] V. Santos and L. Rodrigues, "Localized reliable causal multicast," in *NCA*, Cambridge (MA), USA, sep 2019.

[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, jul 1978.

[14] R. Kazemzadeh and H.-A. Jacobsen, "Partition-tolerant distributed publish/subscribe systems," in *SRDS*, Madrid, Spain, oct 2011.

[15] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM TOCS*, vol. 3, no. 1, p. 63–75, Feb. 1985.

[16] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *P2P*, Seattle (WA), USA, sep 2009.

[17] P. Costa and G. Picco, "Semi-probabilistic content-based publish-subscribe," in *ICDCS*, Columbus (OH), USA, jun 2005, pp. 575–585.