Subscription Latency in Publish-Subscribe Systems

Filipa Salema Roseta Pedrosa
filipa.s.r.pedrosa@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. 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, to relay events from publishers to consumers. These brokers require coordination to offer quality of service guarantees to message subscribers. Typical guarantees are reliable (gapless) FIFO delivery, reliable causal delivery, or even reliable totally ordered delivery of events. In this work, we study the impact of these guarantees on the latency of the subscription operation for different subscription models, in particular, for topic-based and content-based publish-subscribe systems. We aim at finding techniques to optimize the broker topology as well as the coordination tasks associated with the subscription process. Both can minimize the subscription latency, i.e., the time the subscriber needs to wait, after issuing a subscription request, to receive the first event of its subscribed stream.

Table of Contents

1 Introduction

The publish-subscribe abstraction[1] has emerged as a fundamental tool to build distributed systems that preserve strong decoupling among participants. Participants can be information producers or consumers. Producers of information are named *publishers* and produce *events*. An event is a data unit that can be modeled as a tuple containing multiple fields; in most publish-subscribe systems, one of these fields is a *topic*, typically a name in a hierarchical namespace, that characterizes the content of the information included in the event. For instance, an event can be $\langle /Nasdaq/XPTO, \$100\rangle$, where $XPTO$ represents some company trading on Nasdaq and \$100 is the trading value of their stock.

Consumers of information are named *subscribers*, which receive events they *subscribe* to. A participant 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. A participant alternatively can express constraints on the content of the event's fields, such as only receiving events where the stock value is lower than \$200. This type of system is called a content-based publish-subscribe system. A Publish-Subscribe system has two prominent features. On the one hand, publishers do not have to be aware of the identity or number of subscribers. On the other hand, subscribers do not need to know the number and identity of event producers.

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 the events are forwarded in the broker overlay, from the publishers to the interested subscribers. Brokers need to coordinate to offer quality of service guarantees to message subscribers. Typical guarantees are reliable (gapless) FIFO delivery, reliable causal delivery, or even reliable totally ordered delivery of events.

A subscriber may need to wait a certain amount of time after issuing a subscription request until it receives the first event of its subscribed stream. The amount of time required to process a subscription may depend on several factors: the type of subscription (topic-based or content-based), the quality of service requested by the subscriber or the topology of the broker network. In this work, we study how the interplay of these factors affects the subscription latency experienced by subscribers. Our ultimate goal is to find techniques to optimize the broker topology and the coordination tasks associated with the subscription process, which can minimize the subscription latency.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present an overview of the properties of Publish/Subscribe systems and in Section 4 we present all the background related to our work. Section 5 describes the proposed architecture to be implemented and Section 6 describes how we plan to evaluate our results.

Finally, Section 7 presents the schedule of future work and Section 8 concludes the report.

2 Goals

This work addresses the problem of reducing subscription latency in large-scale publish-subscribe systems. In detail:

> *Goals:* We aim at studying how parameters such as the quality of service requested by the subscribers, the broker network topology, or the subscription patterns affect the latency experienced by subscribers. From this study, we expect to get insights that help us find techniques to optimize the broker topology as well as the coordination tasks associated with the subscription process, which can minimize the subscription latency.

We will start by surveying the different subscription protocols that have been proposed in the literature. We define several criteria to help us compare the performance of these protocols, depending on the system's characteristics. Then we identify several key techniques that can be used to reduce the subscription latency. Examples of such techniques are event forwarding approaches that deliver messages respecting the requested quality of service, even in the presence of faults. Other examples include dynamic topology adaptation, to position subscribers closer to relevant publishers. The project will produce the following expected results.

> *Expected results:* The work will produce i) a set of metrics to compare the performance of different subscription protocols; ii) the design of a publish-subscribe system that minimizes the subscription latency; iii) an implementation of the system; iv) an extensive experimental evaluation of its performance using latency as a metric in different scenarios.

3 Properties of Publish-Subscribe Systems

The role of a publish-subscribe system is to deliver events produced by publishers to all interested subscribers. Different Pub/Sub systems ensure different properties on the history of events delivered to different 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 may have an impact on the time it takes for a subscriber to start receiving events after a subscription is made. In this section, we make an overview of the key properties provided by different Pub/Sub systems, of how these properties affect the subscription latency, and of the different techniques that can be used to reduce this latency.

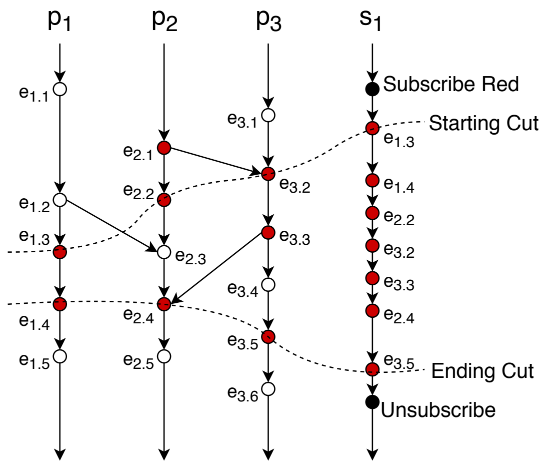

Fig. 1. Message and execution flow in a Pub/Sub system.

3.1 Event Graph and Subscription History

Before we discuss the properties of Pub/Sub systems in more detail, we introduce the concepts of *event graph*, *subscription history*, subscription *starting cut*, and subscription *ending cut*. These concepts are introduced with the help of the example depicted in Figure 1. We assume a Pub/Sub system, with multiple publishers, where each one produces a sequence of events.

Events may be causally related. We use the notion of causal order from Lamport[2], where 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 that are produced in the system, that can be represented by an event graph where edges represent causal relations. The events may have different topics, contents or both: in the example of Figure 1, we have white and red events.

We denote the subscription event history, for a given subscription at a given subscriber, as the sequence of events that are delivered to that subscriber and are associated with a given subscription; the subscription history is bound by a special subscribe event, that is locally generated at the subscriber when it issues the subscription. Another special unsubscribe event is generated when the subscriber terminates the subscription. Figure 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 that appears 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 $e_{1.2}$, $e_{2.2}$, and $e_{3.2}$. 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. In our example, the ending cut for the subscription of s_1 is defined by events $e_{1.4}$, $e_{2.4}$, and $e_{3.5}$.

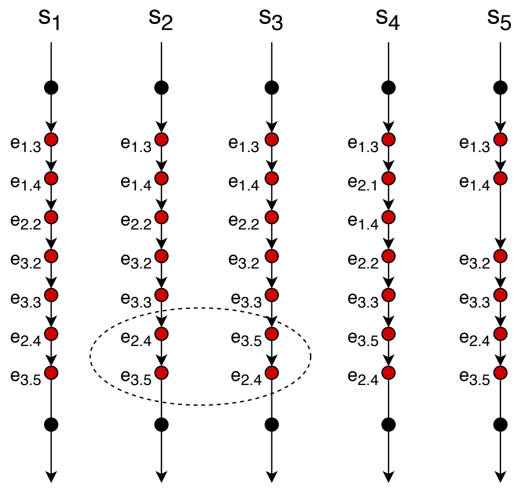

Fig. 2. Examples of subscription event histories.

3.2 Delivery Order

Pub/Sub 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: A Pub/Sub 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 appear in the event history. An example of a subscription event history that respects this ordering can be seen in Figure 1.

Causal order: A Pub/Sub 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, 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: A Pub/Sub system enforces total order if, for any two subscriptions s and s' (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 s_2 from Figure 2 respect this ordering. Total ordering is violated in the history from s_3 since it does not contain every event by the same order as in s_1.

3.3 Reliability

We can define the reliability of a Pub/Sub 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 Pub/Sub system offers *reliable delivery* if, for any subscription s, all events from E_s belong to the subscription history of s. For instance, using the example of Figure 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, s_5's event history violates reliable delivery, given that event $e_{2.2}$ is not included in the subscription history. In some works, reliable delivery is also named *gapless delivery.*

The definition of reliable delivery above does not prevent the Pub/Sub system from delivering one event more than once to a given subscriber. Most systems avoid this by providing *exactly once* delivery of each event, what is sometimes called *strong reliability*[3]. 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 may 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.* 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 $\mathit{before}_x \to x \to \mathit{after}_x$, and $\mathit{before}_x \in s$, $\mathit{after}_x \in s$, 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. We denote the property of ensuring causal completeness, *strong causal reliability.*

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

3.4 Other Relevant Features

The literature on Pub/Sub systems is extensive and detailed, thus, all relevant features that have been proposed or implemented are hardly covered in this report. Still, for self-containment, we list a few of the most important features not covered in detail.

Durability:[4] A quality that some systems provide, which consists of storing events that match a subscription to tolerate failures of the subscriber. By storing those events, the system can send any missed events to a subscriber that reconnected after being disconnected for a certain period.

Timeliness:[5] The ability to enforce either hard or soft real-time guarantees on event delivery is a relevant property in many areas. For instance, ensuring that an event is delivered before some target deadline after it is produced. As will be clear later in the report, our work is somewhat orthogonal to these aspects.

Configurable QoS:[6] In some systems, the set of properties enforced on a given subscription may be selected by the subscriber when issuing the subscription.

Support for Subscriber Mobility:[7] Most Pub/Sub 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.

3.5 Network Overlays and Fault Tolerance

The structure of the network topology impacts delivery guarantees and subscription latency. One possibility consists of organizing publishers and subscribers in a clique, such that publishers can connect directly to every subscriber without using any type of 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 Pub/Sub 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 organizes brokers in a tree, an acyclic and directed graph where each vertex has at most one parent node but may have several child nodes. Other systems organize brokers in a DAG (Directed Acyclic Graph), 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, 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;

in a tree topology these are the root and leaf brokers, and 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.

Each topology, and each strategy to connect clients to brokers, has its advantages and disadvantages. In a tree-based architecture, many events may need to be routed via the 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 a centralized administration. Also, by construction, DAGs lack path redundancy. In a general graph architecture, the 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 make a broker inoperative. The most common link failures are omissions (that cause the link to lose some messages at random) and link partitions (that cause the link to become inoperative until the partition is healed).

Systems can either use redundant brokers or redundant paths to deal with failures. To mask the failure of a physical server, the topology can use the notion of virtual nodes: in this approach, each vertex of the graph is implemented by a group of brokers, that act as a single, reliable, broker. Thus, although at the logical level, there is only a virtual link connected 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 in case of failure of a given broker, creating additional edges that are only used when failures happen. A general graph architecture, that has multiple paths among any two nodes, inherently provides fault tolerance.

3.6 Event Routing

The broker network is used to deliver events from publishers to the interested subscribers. A simple approach for implementing this task consists of broadcasting all events to all brokers (for instance, using flooding) and then let each broker filter the events that are relevant to its local subscribers. This strategy may be very inefficient, as bandwidth is spent forwarding events to brokers that are not interested in them.

A better strategy consists of only forwarding an event to brokers that that are in the path to the relevant subscribers. But in other to do so, brokers must be aware of the location of subscribers, creating routing tables that help in forwarding events in the right paths. This is possible if information regarding the subscriptions is propagated in the network and brokers set up their routing tables accordingly. Subscriptions need to be broadcast in the network 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 the subscription information only to the brokers that are in the path from the publishers to the subscribers. This is only possible if the location of the publishers is known. For this purpose, some systems also use *advertisements*, special 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 the advertisements, brokers can create routing tables to propagate subscriptions towards relevant publishers.

3.7 Subscription Latency

When a subscriber makes a new subscription, it may need to wait a certain amount of time before it starts receiving the 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 associated with that subscription being delivered. Naturally, we would like the subscription latency to be as small as possible. However, as we have seen, most Pub/Sub 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 Pub/Sub system that may also affect the subscription latency. Consider, for instance, an overlay network that uses multiple paths for fault-tolerance. In this case, routing tables may need to be set up in *all* paths, introducing additional delays. Systems that use a single path and rely on virtual nodes, also introduce additional delays. This is due to replicas of a given vertex needing to coordinate to ensure that they have consistent information regarding the routing table of their virtual node. In some cases, a consensus protocol may need to 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 paths (that we denote the *stable* path) but not in the other(s). In this case, depending on the route an event takes, it may be forwarded or dropped. A system that aims at offering gapless delivery may need to wait for all paths to be stable. However, a system that can tolerate event losses may start delivering the 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 a previous subscription made at the same broker. A subscription covers another if all messages matching one 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 the subscription latency. Systems that do not support subscription covering require every broker on the network to have every issued subscription on their routing table.

3.8 Reducing the Subscription Latency

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

The first technique consists in re-organizing the broker overlay to group subscribers that have similar subscriptions. The subscribers are brought closer to the publishers that are relevant to their subscriptions. This reduces the path length from the publisher to the subscribers, reducing the time it takes to set up the routing tables. Also, it makes it more likely that a new subscription is covered by a previous subscription.

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

4 Existing Systems

In this section, we will survey several relevant Pub/Sub systems. In the analysis, we will emphasize on the following concerns:

– *Network Overlay and Fault Tolerance* Addresses how the 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* Addresses the ordering properties enforced by the system and the algorithms used to implement them.
– *Reliability* Addresses the level of reliability they offer to their subscribers and which mechanisms they use to enforce it.
– *Subscription Starting Cut* Addresses how the 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 important factor influenced by the level of service, topology and any relevant optimizations to reduce the latency.
– *Routing Events* Addresses the problem of how systems route events on the broker network along with which metadata or storage is used by brokers to make forwarding decisions.

4.1 Systems

LoCaMu The goal of the LoCaMu[8] 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 joins a group using a subscription.

Network Overlay and Fault Tolerance The brokers are arranged in an acyclic undirected graph and act as clients that can publish and subscribe to groups. It uses the concept of neighborhood, which is the set of nodes that are 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 is accomplished by using redundant paths where brokers create additional edges to the closest available neighbors on that path to bypass the nodes that have failures.

Ordering To ensure causal ordering guarantees, LoCaMu tags events with metadata about the broker's neighborhood. Nodes keep separate sequence numbers for each neighbor they may forward events to. The ones assigned to the event are the ones corresponding to the brokers on the event's forwarding path. Brokers store locally the sequence numbers they have seen from their neighborhood, which defines the node's past. This is used to tag events such that they will carry information about the sender's causal past. By comparing the local causal past with the one attached to the event, nodes can verify if it is safe to forward messages.

Reliability Brokers verify if delivering an event respecting causal order is possible by comparing the local causal past with the event's metadata. If not, then the event is 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.

SIENA The SIENA[9] system is a content-based Pub/Sub 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. Subscribers use filters, which are a set of constraints on event notifications, to specify the events they are interested in receiving. In this approach, brokers store advertisements on their tables and only forward subscriptions towards event sources that match those subscriptions. Advertisements set paths for subscriptions, and these, in turn, set the paths for events to be forwarded. Brokers maintain a structure called a filters poset. This poset is a DAG of constrains, defining a partial order on the set of filters a broker knows of. It supports subscription covering, that defines the covering relations in the filters poset. A root filter covers all other subscriptions and is the most generic one.

The 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 that SIENA does not support any type of quality of service guarantees. As such, the subscription latency is related to the distance from the subscriber to the event source, although this can be shortened by using 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 node where the subscription came from. An additional algorithm is used 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.

Gryphon The goal of Gryphon[10] is to build a 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 [3], [11] , and [4], which characterize its different qualities. It is a content-based Pub/Sub system in which subscribers use a set of conjunctions to specify interest in events.

Fault Tolerance and Overlay Network This system uses a tree topology for the broker network, where each intermediate node is a virtual node that contains 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 the sequence numbers, incremented for each event. Every subscribing broker must receive an ordered stream of events from a publisher. If the event happens to be filtered by a broker on the network, then a silence token with the sequence number replaces the event and is sent downstream instead.

Reliability The subscribing brokers start buffering events if sequence numbers are missing in the event stream from a publisher. To recover the missed events or silence tokens the subscribing brokers send a curiosity message upstream with the lost 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. This mechanism allows for an exactly once reliable delivery since it can detect duplicate or missed events, enabling brokers to retransmit the events that were lost.

Subscription Starting Cut The main challenge to be solved is deciding a starting cut. Messages can travel along different paths, which can cause gaps in the delivery. To solve this, each leaf broker, where subscribers can connect to, has a virtual time clock. For each new subscription or set of subscriptions, it 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 on the system 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 for the broker where the subscription came from. 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 towards the publisher broker, the root of the tree topology, meaning that the experienced delay is equal to the network diameter.

Only one of the redundant brokers needs to forward the subscription upstream towards the publisher. As such, the rest of the redundant brokers on the intermediate node are on unstable paths and can make incorrect forwarding decisions. The Vb vectors kept by each broker are what determines path stability from a publisher to a subscriber. The timestamps are used 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 the increment in the clock of the 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 results. Otherwise, the broker has to flood the event to all its downstream nodes to guarantee that there is no gap in the delivery. When the subscribing broker finally receives the event it verifies if it matches any of its subscriptions. If it does, then it must compare its entry in Vm, with the sst for the subscriber. If $Vm[subscribingbroker] \leq sst$ then it is safe to deliver the event. It means that the subscription has reached the publisher connecting broker, creating at least one stable path between the subscriber and the publisher.

δ-fault-tolerant Kazemzadeh and Jacobsen [12] [13], propose δ-fault-tolerant. The first goal is to develop a system that is reliable and maintains availability when δ broker failures occur. In the second paper, this is expanded to include tolerance to partitions in the network with the same service guarantees. In this context, δ refers to the concurrent broker or link failures that can happen. It is a content-based Pub/Sub system where subscriptions specify a set of predicates. The subscriptions are stored on the brokers' routing tables and have to be known to the whole network.

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

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 that 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 the event has been delivered. The edge brokers receive the event and send an acknowledgment back to the link where it came from. 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 events can be published. Sequence numbers are added to each event by brokers that forward them. Brokers maintain $2\delta + 1$ sequence numbers, so there is at least one common broker to compare sequence numbers. This mechanism is used to detect duplicate events, preventing them from being forwarded multiple times through the same link.

Subscription Starting Cut The problem with having partitioned brokers is that these create inconsistent subscription sets. The subscription did not reach nodes on the partition, introducing gaps in the delivery. A safety condition has to be followed by the system to avoid this situation, "a publication is delivered to a matching subscriber only if it is forwarded by brokers that are all aware of the client's subscription". There is an important distinction between two types of partitions: the partition island and the partition barrier. In the first type, the broker cannot reach a sequence of neighbors, but it can bypass them with a link to another available broker on its partial view. In the second type, the broker cannot create any new link to bypass the partition.

The broker with a connected subscriber starts by flooding its subscription. It will wait for a confirmation that every broker on the network received the subscription. When the subscription arrives at an edge broker it sends an acknowledgment back to the link where the subscription came from. Brokers on the network will wait to receive acknowledgments from the links they flooded the subscription to. Afterward, they send a confirmation to its upstream node that the subscription has been received by all its downstream neighbors. When the broker with a connected subscriber receives an acknowledgment then it can start delivering events to its subscriber. In this system, the delay experienced by the subscribers will be equal to the network diameter. After this process is finished all the paths on the network are stable.

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 that is beyond the partition can connect to any node in it then it can propagate the subscription inside the partition. It uses a tag on the event for the subscription to be propagated 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 that are 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 the local subscriber stores the subscription in the routing table, plus the several tagged *pid*, when this acknowledgment is received. 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 not violated. When a broker receives an event it starts by checking if the event is duplicate. If not then the broker stores the event, which will be removed once all its downstream active links have confirmed its reception. Secondly, the broker verifies if the event came from any partition known by it. If it did, then 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 to. The tags in the events are used by the brokers with connected subscribers to verify the safety condition. In case the event came from or beyond a partition that did not know about the subscription then the delivery to the subscriber cannot be made. The broker knows this locally by comparing the tags on the event with the tags that are stored in the routing table for the subscription. If there is at least an identical *pid* then it is not safe to deliver the publication.

Dynamic Message Ordering for Pub/Sub The main objective of Dynamic Message Ordering[14] is to ensure that two subscribers that are 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 Pub/Sub system. It assumes a generic Event Notification Service (ENS). This ENS offers a topic-based interface for the clients to publish events or subscribe to already pre-established topics.

The system assumes the existence of a special broker, called topic manager, for each topic. This manager can be created statically or through a DHT (Distributed Hash Table) that selects these nodes dynamically. Considering that no additional assumptions are made there is an additional challenge in providing order. Specifically, two events can follow distinct paths through the ENS and be delivered out-of-order to the subscribers.

Ordering The system ensures TNO (Total Notification Order), meaning that if event e_1 is delivered to a subscriber before event e_2 then no subscriber will deliver e_1 before e_2. Every topic manager contains every subscription with its topic and a sequence number. All topics in the system are ordered by a precedence relation, which can be predefined or changed dynamically. Every event is forwarded through the relevant topics respecting the precedence order, so they can be ordered with respect to other events.

To publish an event on the ENS a publisher must first request a vector clock for that event. Its creation is carried out by the sequencing group of the topic T. The group consists of the topics that precede T and are included in at least two subscriptions, along with T. The vector has an entry for each of the topic managers in the group. The event is forwarded first to the manager of the topic the event is being published to. The manager increments the topic's sequence number and adds it to the corresponding vector clock entry. The event is then forwarded, according to the precedence order, to each member of the sequencing group. 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 the partially filled-in vector clocks they have seen, so these can be forwarded again. Publishers can solicit the creation of the timestamp again in case the request was lost or a timer on the publisher has expired.

Subscription Starting Cut To decide the starting cut for each topic, the subscriber must request the creation of a vector clock. This request is sent to all topic managers for the subscription's topics. 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 topic it is subscribed to. Afterward, it forwards the subscription to the next manager in the order. When the last topic manager fills its entry, the timestamp is sent back to the subscriber. It will now use the timestamp as its local subscription clock. The subscriber must wait until its request goes through every relevant topic manager to start delivering events.

When the subscriber is notified about a new event matching its subscription, it must compare and verify for every entry i that: $localclock[i] < timestamp[i]$. 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 until it can be delivered in order. This ensures that the subscriber can only deliver events that went through topic managers that know of its subscription's existence.

XNET One of the main goals of XNET[15] is to provide reliability in the delivery, ensuring that the shared state of the system is consistent at all times. The shared state is defined as the knowledge each broker on the network has of each subscription. This has to be consistent with the subscriber population, such that the routing paths reflect the registered subscriptions. Another goal is to reduce traffic and minimize the size of the routing tables by integrating a routing protocol called XROUTE[16], also developed by the same authors.

This system supports content-based Pub/Sub by allowing publishers to generate events with an XML structure. Subscribers can specify predicates for the events with a subscription language that uses the XML standard. It uses XROUTE to propagate events, which supports subscription covering. Brokers aggregate subscriptions to eliminate redundant knowledge and improve routing performance.

Fault Tolerance and Overlay Network The system uses an acyclic undirected graph for the overlay, where each publisher or subscriber can connect to the edge brokers. To ensure fault tolerance, it can use redundant paths. The system creates more than a single path 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 that can be used in case of failure of the direct neighbors.

Ordering To provide FIFO order to the delivery all the links connecting the brokers are TCP, which enforces this type of order. 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, it uses end-to-end acknowledgments. Brokers store the events until confirmation of reception has been sent back. The sequence numbers used for each link are used to also detect duplicate messages.

Subscription Starting Cut XNET propagates the subscriptions on the network using subscription forwarding along with three different strategies to prevent faults. The first is called Crash/Recover, where brokers buffer advertisements; in this context, these correspond to a subscription or unsubscription. They buffer those until they receive an acknowledgment from the upstream node to confirm it has updated its routing table with the new subscription. If a broker crashes, all of the advertisements that were supposed to be sent to it are buffered on its downstream routers. The Crash/Recover strategy is used when failures are only transient.

The downtime of a broker can be very long, which can cause buffers to overflow or to recover very slowly. When the broker eventually recovers, it can create a bottleneck when processing all of the downstream advertisements. For these situations, there is a second strategy called Crash/Failover that uses backup nodes. This can be a node that is on another location on the network, to which a broker will connect to when its upstream router fails. The chosen node must guarantee a valid routing path to the subscriber. When switching from the upstream broker to the backup one, it needs to send advertisements to the backup again. This is to create a new routing path by registering the subscriptions on the backup broker. The upstream broker of the crashed node must advertise unsubscription of downstream subscriptions. This removes the previously established routing path.

Both schemes cause the system to be unavailable while recovering. To maintain availability there is a third strategy called Redundant Paths. In this strategy, it is assumed that each broker has at least one alternate route to the publishers. The routing tables are replicated on both routes. When one broker on a route fails the brokers on the other path can still deliver the event to the subscriber. The nodes on this available route maintain information that is consistent with the consumer population. The subscriber can start delivering events once its subscription reaches a publisher, creating one stable path. Events are flooded to both alternate routes, 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 is done to decide the downstream links through which they must forward the event, only sending it if there is at least one interested subscriber.

Semi-Probabilistic Pub/Sub Costa and Picco [17] 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. Routing events in a deterministic way, using a tree-based topology and global subscription knowledge brings an additional overhead. It is inefficient in these scenarios and only provides a single route between any pair of brokers on the network. The system supports content-based Pub/Sub by allowing subscribers to use predicates on the published events, providing added expressiveness.

Fault Tolerance and Overlay Network The topology of the overlay network is a general graph and clients can connect to any broker of the network. This type of graph inherently provides multiple routes between any two brokers.

Subscription Starting Cut The system floods the subscriptions to establish routes that are followed by events. However, subscriptions should not be propagated to all brokers since each of their routing tables can quickly become stale. To solve this issue each broker of the network only knows a limited portion of the subscriptions made. This is determined by a parameter called subscription horizon ϕ. The parameter determines the size of the neighborhood that has 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 that 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 set up in the vicinity of the broker it is connected to.

Routing Events The subscriptions are only known in a limited portion of the network. Therefore, events are not forwarded 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 the events and is a percentage of all the neighbors the broker has defined by a threshold τ. The selection of the forwarding set prioritizes neighbors with subscriptions matching the event. It also prioritizes the closest subscribers, avoiding sending the event through a stale route. If the set does not reach the required percentage, the broker adds random links to forward the event. Brokers recognize duplicate events by storing identifiers of the ones they have forwarded. This avoids the creation of forwarding loops since events are not forwarded through the same links twice.

GEPS The goal of Gossip-Enhanced Pub/Sub[18] is to maintain a high delivery rate to the subscribers while also providing high system availability. A tree-based topology does not offer path redundancy, so there is a need to create extra links around failed brokers. For scalability purposes, these must be created avoiding global knowledge of the network structure or the existing subscriptions. It supports content-based Pub/Sub where clients can subscribe using predicates that will be matched against events. It also uses advertisement forwarding and subscription covering.

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

The main focus of GEPS is on fault tolerance, by creating links to bypass failures using the similarity metric. To achieve this, 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 periodically update their sibling views, and find other similar brokers at their depth or to discover failed/recovering siblings. In each gossip round brokers heartbeat their views to their alive siblings along with the nodes perceived as failed or recovering. This 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. Publishers send advertisements about the events they will publish; these are flooded on the network using an advertisement forwarding strategy. Subscriptions are forwarded further if they match any advertisement the broker is aware of. The subscription must reach an edge broker with a publisher whose advertisement matches it. When a stable path between the subscriber and a publisher is created then event delivery can start. Therefore, the delay is equal to the network diameter. The system does not give any guarantee regarding message delivery so there is no 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 its routing table that match it. This counter serves to compare the similarity between two brokers when creating sibling views. The similarity is based 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 that have subscriptions matching the advertisement. In case the direct links are not available to forward messages, the brokers use the child or parent partial views to gossip the messages to the nodes in them. Brokers that receive gossip messages also route those messages using gossip to their views.

JEDI The purpose of JEDI[7] is to build a Pub/Sub system with a distributed Event Dispatcher (ED), which delivers a published event to all interested subscribers. Its components are called dispatching servers (DS). In this context, the clients that connect to the ED are called active objects (AO); they interact with each other by producing and consuming events. It supports content-based Pub/Sub by allowing AOs to subscribe to a specific event or an event pattern. Patterns are regular expressions that express constraints on event notifications.

Overlay Network The system's DSs are organized in a tree-based topology and AOs can connect to any DS in the network. The links between components are also TCP.

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. The events generated by the same publisher are delivered to the subscribers respecting FIFO order.

Subscription Starting Cut To propagate subscriptions in the ED it uses a hierarchical strategy in which subscriptions are only propagated upwards on the tree. They are sent until they reach the root DS, to avoid having to propagate them to the entire network. The AO can start receiving events as soon as there is a stable path between it and a publisher. Although, 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 event 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 of all the system's existing subscriptions. Subscribers on other subtrees are unknown to intermediate DSs.

Sequencing Graph The goal of Decentralized Message Ordering[19] is to build a graph of forwarding brokers, called sequencers, that will order events. The events are delivered respecting causal order across topics. A topic defines a group of brokers that publish and subscribe to it. The challenge is for events to be delivered in the same order by the subscribers of double-overlapped groups. These are groups that have more than two common members. The common subscribers by themselves can make inconsistent ordering decisions when delivering causally related events. Although, messages to unrelated groups may be delivered in any order by its subscribers. The key 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 the event 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. The second is that the final graph must be loop-free to avoid circular dependencies between events. When a message leaves the sequencing network it is forwarded to a tree-based graph that connects a group's subscribers. The system assumes that each group's subscribers are globally known, forming a membership matrix. The global membership matrix is used 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 message is addressed to one of the groups it represents. Then it increments its sequence number and assigns it to the message, forwarding it to the next sequencer in the path to the group. The links between the sequencers provide FIFO ordering and the graph is acyclic. This ensures that the order of arrival of two messages at one node on the graph will be preserved.

Reliability To offer reliability, it uses end-to-end acknowledgments between sequencers. When a sequencer forwards an event to another it buffers the event, waiting for an acknowledgment from the sequencer to which it sent the message.

Epidemic Algorithms for Pub/Sub The main objective of Epidemic Algorithms [20][21] is to offer reliability, minimizing loss of events in a highly reconfigurable scenario. The system uses epidemic algorithms to provide a probabilistic reliable delivery and scalability since those are resilient to reconfiguration. Content-based Pub/Sub is supported where subscriptions contain expressions and the brokers perform matching of events to deliver them to the subscribers. Using gossip protocols in this type of setting to prevent message loss is challenging. Considering that events can match multiple subscriptions, which can make detecting missed messages a difficult task.

Fault Tolerance and Overlay Network This system assumes an acyclic undirected graph overlay for the network, where clients can connect to any broker. The links between brokers are assumed to be unreliable, causing messages to be lost. To tolerate this type of fault, caused by unreliable links and a highly reconfigurable scenario, the system uses different gossip[22] techniques. Three solutions are proposed with different gossip strategies in mind. We will be focusing on one category which contains two of the solutions. This is the pull-negative approach, in which brokers gossip about the events they have missed. Brokers send the gossip messages soliciting the transmission of missed events from other nodes.

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

Reliability All of the gossip strategies provide probabilistic reliability guarantees. The strategies allow brokers to recover lost messages and deliver those to the subscribers. Two different pull-approaches can be used in combination. 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 it has missed. The gossip message is routed as a regular event with that specific pattern, forwarded only to a random subset of interested neighbors. Brokers in the network cache events they have forwarded. When a broker receives a gossip message it can send the requested events, if it has them stored. The gossip messages must be forwarded towards subscribers of the same pattern so the likelihood of recovering events is much higher.

In the publisher-based pull method, the events are also tagged with the route they have followed until reaching a subscriber. A broker creates a gossip message by selecting a publisher, instead of a pattern. The message contains every event it has missed from that particular publisher. It is then disseminated using a route the broker is aware of that leads to the publisher. This route is chosen to increase the likelihood of reaching a broker that cached the event.

Subscription Starting Cut Brokers flood the subscription on the network, establishing the routes for the 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, the events start being tagged with the sequence number relative to the subscription's pattern. At that point, the subscription is known to the publisher and the subscriber can start delivering the events from it.

Routing Events When a broker receives an event it matches the event with the subscriptions on the routing table. It then forwards the event to the resulting links.

4.2 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 different systems provide a set of guarantees to the subscribers, exploring different techniques for different levels of service. If the system does provide any type of guarantees we want to consider how it verifies path stability when delivering events to the subscribers. This directly correlates with the starting cut for a subscription; when the system verifies that all paths are stable then it can define a cut in the event graph. 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 we have the concerns that the given systems approach, with only four systems addressing every important topic. We will be comparing the systems per section. In Table 1 we summarize the most important characteristics of the systems. If the system does not address a specific concern then it is not specified (N/A).

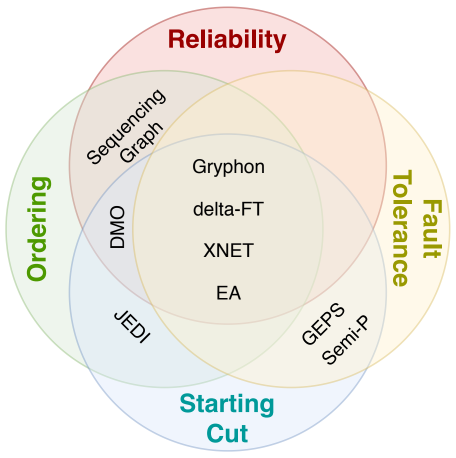

Fig. 3. Addressed concerns by respective system.

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

Table 1. Systems' guarantees and strategies for each topic.

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 approach. 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, a counter that is 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 ordering by not allowing parallel events for each publisher. Sequencing Graph uses sequence numbers to provide causal ordering, while JEDI uses TCP as well as a single path between every broker on the network. Dynamic Message Ordering 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 offer 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 and Sequencing Graph. In this case, brokers store messages until the nodes that must receive the message acknowledge its 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 systems like Gryphon, which disseminates subscriptions toward the root broker, δ-fault-tolerant and XNET, which flood the subscriptions and GEPS, which uses advertisement forwarding, the latency depends on the diameter of the network. 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, a subscriber must wait for its subscription to go through every relevant topic manager.

There is also a difference in the experienced delay between systems that support topic-based versus content-based Pub/Sub. In the former, there is an established group to disseminate the subscription, determined by the topic, as in Sequencing Graph and Dynamic Message Ordering. Meanwhile, in content-based, there is not a clear notion of a group; therefore the subscription usually needs to be propagated to the entire network, such as in Gryphon, δ-fault-tolerant, XNET, Epidemic Algorithms, Semi-Probabilistic, and GEPS.

XNET and GEPS both support subscription covering, which can influence the subscription latency depending on the subscriber population. This technique can make the 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.

Issues with current approaches for LoCaMu Keeping in mind the characteristics and qualities of LoCaMu described earlier, we will analyze how the several subscription techniques from the studied systems can be applied to it. JEDI is similar to LoCaMu; however, it does not consider the fault tolerance and reliability aspects.

Gryphon and GEPS have a tree-based topology in which the subscriptions are not flooded to the entire network, only until the root node. On the other hand, LoCaMu uses an acyclic undirected graph for the overlay, as such the subscriptions will have to be flooded to every broker. Although like JEDI, the system allows clients to connect to any broker. Thus, events from a set of publishers can start being delivered without waiting for the subscription to reach the whole network.

There are several options to verify path stability on the network, ensuring the subscription event history obeys the guarantees provided by LoCaMu. One option can be the strategy in δ-fault-tolerant that floods the subscription on the whole network before starting delivery. This solution causes the latency to grow with the network diameter. We want to avoid these types of solutions, as to guarantee a smaller latency to the subscribers.

XNET uses a Crash/Recovery scheme which can be applied to LoCaMu. Although, the system has to wait for every broker on a path to acknowledge a subscription. It has to confirm if a path is stable to prevent brokers from making incorrect forwarding decisions. This may make the subscription latency grow.

In Epidemic Algorithms, publishers have sequence numbers for each unique subscription. This leads to a huge overhead when publishing an event. If an event matches almost every subscription on the system then the metadata will have one entry for each matching.

Using virtual time as in Gryphon does not work as-is for LoCaMu due to their different characteristics. In Gryphon, we have vector clocks with an entry for each leaf node, which is the access point for subscribers of the system. In LoCaMu, a client can connect to any broker on the network, which requires using a virtual timestamp with an entry for every node. This method brings a huge overhead to event forwarding with timestamps that contain one entry for each broker on the system.

Optimizations can be applied to the system to shorten the subscription latency. Flooding the events while paths are unstable is a viable strategy in Gryphon or XNET where the clients only connect to the edge brokers and both provide FIFO order. On the other hand, LoCaMu allows access points on every node of the network. As such, if events were flooded the strong causal reliability property may not be ensured.

Another possibility consists of using subscription covering, XNET and GEPS take advantage of this technique. Although, in LoCaMu it is not possible to use this technique since it bypasses brokers with failures. This means that there may be inconsistencies in subscription knowledge on a given path, making each broker's covering relations inconsistent.

Another technique to reduce the delay is to reconfigure the topology dynamically according to the subscription population. This is used in Sequencing Graph and is more suitable for LoCaMu.

Propagating the subscriptions locally on a selected horizon is the best optimization, as in Semi-Probabilistic Pub/Sub. However, with probabilistic routing decisions, the system may not be able to provide a strong causal reliability guarantee.

Dynamic Message Ordering provides total order for messages, which is not offered by the LoCaMu system. As such, this property will not be supported, as it requires a graph of brokers to order every event from the publishers.

Conclusion To support any type of ordering or reliability guarantees there is a need to verify stability on the routing paths on the network. When a broker receives an event it has to make forwarding decisions with the matching results. These can be inconsistent with other brokers, due to the differences in the subscription sets, violating the guarantees provided by the system. The subscriber may have to wait for the subscription to be flooded on the whole network to start receiving events, such as in XNET or δ-fault-tolerant. However, we want to minimize the delivery delay while using flooding, thus the latency depending on the network diameter is to be avoided. To make the latency depend on the distance to the publisher and verify path stability we can use virtual timestamps as in Gryphon. However, the metadata must not contain the global subscription state of every broker to avoid adding major overhead to event routing.

5 Architecture

There are three main objectives that the architecture will have to fulfill. The first is to provide different levels of service to subscribers during the joining process. Each level has a well-defined set of delivery guarantees. When a subscriber joins using a subscribe event, it can specify what type of guarantees it wants for the message delivery. The system will ensure that events will be delivered to the subscribers according to the level of service they requested.

The second is to verify the path conditions necessary to start delivering the events, to define a subscription starting cut. The system must be able to perform a cut on the event graph for the subscription history while preserving the guarantees for delivery. Each broker will have enough information available to avoid introducing gaps in the delivery or to incorrectly order messages. The conditions and definition of a starting cut will be different depending on the chosen level of service, resulting in different algorithms for each.

The final objective is to reconfigure the topology in a way that brokers with connected subscribers are brought closer to brokers with connected publishers. This shortens the path between a subscriber and event sources that match its subscription. This will cause the delivery delay to be smaller, due to the smaller number of hops an event has to go through. The system starts with a pre-established topology that will be changed dynamically to better suit the subscriber population that exists at a given time.

The mechanisms to enable the system have to be scalable, by not depending on the number of nodes on the network. Availability is also a desirable quality, considering that a broker's forwarding decision has to be made locally without requiring any type of consensus protocols.

The LoCaMu[8] system, which already provides topology and guarantees, will be the building block for our subscription semantics. During the joining process, a subscriber can select weaker delivery guarantees than the ones LoCaMu offers. Although after the subscription is known in the whole network, its history will respect strong causal reliability. This is due to LoCaMu already providing this guarantee when the subscriptions are set up on every broker.

5.1 Subscriber Join Semantics

The system must enable different joining semantics for each level of service a subscriber might choose. Each level defines a different set of guarantees and constraints on the subscription event history. The distinct semantics will also signify that brokers have to make different forwarding decisions, to preserve the requested guarantees by the subscriber. There can be two different subscribers that request different sets of guarantees for the same subscription. The brokers must then decide which events will be forwarded and be contained in each subscription event history without violating any guarantee. For each service the system will have to verify different path stability conditions before starting delivery to the subscribers, defining the subscription starting cut.

When a subscriber connects to the system it must specify which level of service it wants to receive along with the subscription predicates. The system will flood the subscription on the whole network. Brokers keep routing tables with subscriptions, the corresponding level of service required for each, and where they came from. The tables are used to perform matching of the subscription predicates with the events that arrive at the broker.

The system must make available three distinct levels of service to the subscribers during the joining process. In the best-effort level, the subscription event history does not need to obey any constraint, as the system will only provide best-effort unordered delivery. A broker with a local subscriber can start delivering events to it as soon as the join process starts, as the subscription only needs to be in the local routing table. Brokers do not need to verify path stability when forwarding an event. They only need to verify if it matches any subscription on their routing tables. As such, the starting cut for each publisher can be on any event that arrives at a broker with a subscriber and matches its subscription. As illustrated in Figure 4, C receives event $e_{1.1}$ from A, due to a failure on B. It will now decide if it should forward the event to F since it matches the subscription from s_1. If s_1 requested a best-effort level for event delivery then C can send $e_{1.1}$ to F for it to then be delivered.

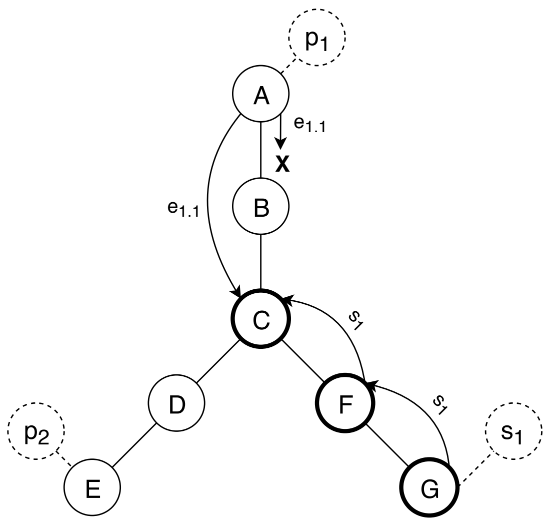

Fig. 4. Example of subscription propagation and event forwarding.

In the FIFO level, the system must ensure strong reliability with FIFO ordering guarantees for the subscribers. The subscription event history has to obey a strong reliability property, so the system has to verify path stability between a subscriber and a publisher. A broker with a connected subscriber has to wait until there is a stable path between itself and a publisher to start delivering events from it. Brokers need to verify that their downstream and upstream neighbors that belong to this path know of the subscription for events to be forwarded. After this process of defining the starting cut for that publisher is finished then the events from that source can start being delivered. As an example, in Figure 4, A and B do not know about s_1's subscription yet, as such B will not forward events to C. In case B fails, if s_1 requested the FIFO level of service, then it is not safe for C to forward the event to F yet.

In the causal level of service, the system provides strong causal reliability guarantees to the subscribers. This level implies that the subscription event history has to be causal complete and obey strong causal reliability. There is an additional challenge in causality, as the cause-effect relations between events must be preserved in the subscription event history. These occur when a publisher subscribes to events from other publishers. Keeping this in mind, a broker with a local subscriber can start delivering events from a publisher when two conditions are met. The path between this publisher and the subscriber must be stable. Additionally, every path between this publisher and all other publishers it subscribed to must also be stable. Brokers have to verify if all their relevant upstream and downstream neighbors know the subscription before forwarding an event. After the conditions are met the events can start being delivered to the subscriber. Using Figure 4 as an example, p_1 subscribed to content published by p_2 and s_1 subscribed to content from both publishers, requesting a causal level. When C receives event $e_{1.1}$ from A it cannot forward it to F. Neither the brokers on the path to p_1 nor the path to p_2 know the subscription.

5.2 Subscription Starting Cut

There are several issues to keep in mind when defining a subscription starting cut and also reducing the experienced latency by the subscribers. Bypassing neighbors when disseminating subscriptions introduces inconsistencies on the routing tables of brokers on a path. Even though the subscription information reached the publisher it does not mean the path between it and the subscriber is stable. For the subscriber to not wait for the paths to be stable the system must be able to compensate for differences in subscription information.

Using local information along with metadata carried by events the brokers can assess their neighbor's subscription state, verifying if a path is stable. Afterward, it makes forwarding decisions according to the state of the path, as to not introduce gaps or an incorrect ordering in the event history. If a broker belongs to an unstable path it can use metadata to compensate for its lack of subscription information, enabling it to make forwarding decisions.

The main idea to define a starting cut and shorten the subscription delay is to use virtual time as in Gryphon[10]. The virtual timestamps symbolize how many subscriptions a broker knows of. However, it is not scalable to apply this algorithm as-is to our setting. To reduce the size of the vectors, we use ideas from δ-fault-tolerant[12], which defines the concept of the neighborhood as a partial view of the system. By applying virtual time, we remove the delay of propagating the subscriptions to the entire network to start delivering messages. Brokers can compensate for inconsistencies in their routing tables with local knowledge of neighborhood subscriptions and the virtual time attached to events.

Each broker will maintain a vector clock, each entry being a pair (Node:Value). The Node corresponds to a broker on its neighborhood along with a Value denoting the number of subscriptions they have forwarded. The goal is for a broker to locally verify path stability and the conditions to be met for a given level of service requested by downstream subscribers. The broker cannot rely on its local timestamp only to make forwarding decisions since it might not be aware of the system's current subscription state. As such, the events must carry metadata, which corresponds to the vector timestamp of the broker that forwarded it. This metadata conveys information about the sender and the broker's neighborhood.

When a broker receives an event it compares the received vector timestamp with the one it has locally. This is to verify if the path is stable by confirming if either the upstream or downstream path has a consistent subscription set. The event is forwarded downstream if it notices it is lacking subscription information. Otherwise, it can use its routing table to perform matching and decide if an event should be sent downstream or not. If forwarding an event causes the constraints on a subscription's event history to be violated then the broker must not forward it downstream. The size of the neighborhood to keep subscription information is $\delta + 2$ hops. This allows timestamp comparison for the entry of at least the next direct neighbor on the path.

5.3 Overlay Reconfiguration

The main idea to accelerate the delivery is to reconfigure the overlay, by grouping together subscribers who share similar interests. These neighborhoods must be created closer to event sources that will match their subscription content.

We will use ideas from Sequencing Graph[19] along with GEPS[18] to reconfigure the overlay. Sequencing Graph uses a global membership matrix to create a graph to order the events addressed to different groups. To maintain a global membership matrix can bring a huge overhead to the system, thus instead brokers can use their partial views to infer memberships. The reconfiguration must follow a constraint, that is also applied in Sequencing Graph, which is to maintain an acyclic graph. This avoids circular dependencies between events. LoCaMu also relies on this topology constraint to causally order messages, as such after reconfiguration the network must remain loop-free. Additionally, the resulting overlay must still maintain consistent routing paths using the partial membership view.

From GEPS we will take the idea of similarity between brokers to create neighborhoods of similar subscribers. GEPS uses similarities between a sibling group of brokers to build the broker's partial view. The similarity metric will be used to find similar neighborhoods of brokers. Publishers must advertise the content of the events they will publish on the network and brokers keep these advertisements on their routing tables. Using this mechanism, brokers know the paths in their neighborhood to reach an event source and which subscriptions match their content. Advertisements and localized subscription information enable the system to use the similarity metric, creating neighborhoods of subscribers with similar interests. With the use of advertisements, the neighborhoods can also be brought increasingly closer to event sources.

Using a dynamic topology will affect the operation of LoCaMu, as it uses localized neighborhood information to provide reliable causal delivery guarantees. A simple approach to consider is to have different versioned graphs. The system starts with a pre-established broker network and for each reconfiguration creates a new graph with a different version. Brokers can only deliver messages from a new overlay once all the messages from the previous one have been delivered to all relevant subscribers. When all events have been delivered, brokers begin using the connections defined in the new graph to forward events. With this approach, the algorithm from LoCaMu can be used as-is to offer reliable causal delivery.

6 Evaluation

Our main goal is to analyze how the proposed system and optimizations affect subscription latency. To do so, we will use a simulated environment for the system to be tested. To evaluate we will consider two different topics: 1) experienced latency by the subscribers; 2) the impact of reconfiguring the network overlay.

6.1 Subscription Latency

Our system will offer several different quality of service guarantees to the subscribers, as such, we want to measure latency on the different levels. The subscription latency can be defined as the time between a subscriber joining using a subscribe event and the first event that is delivered to it by the system. The requested level of service, as well as the current subscriber population connected to the system, will influence the latency value. Therefore it is also important to analyze how different population scenarios affect the latency. We will consider distinct cases, ranging from few subscribers to a dense population along with identical subscriptions to dissimilar ones.

Our solution will be compared to an implementation of Gryphon and δ-fault-tolerant regarding subscription latency. Both systems only support gapless FIFO message delivery, so we will only use the FIFO level described in our system to measure and compare the delay. We will be taking into account different subscriber population scenarios when comparing the systems.

6.2 Overlay Reconfiguration

Since our system uses a dynamic topology we will be analyzing the impact it has on the subscription latency when compared to a static network. We want to analyze the overhead of reconfiguring the network considering the latency improvements it may provide. We will use as a baseline the described approach, which is very slow to move brokers to a new overlay, to analyze how improvements to it can influence the latency.

We will be taking into account two different scenarios. One with similar subscriptions where brokers can be easily grouped to form a cluster, and another with distinct subscriptions. The scenarios will be used to assess if a dynamic topology will bring any benefits to the delivery latency, even in an unfavorable situation.

7 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.

– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15 Deliver the MSc dissertation.

8 Conclusions

In this report 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 start receiving events associated with a given subscription. We aim at designing a system that supports multiple subscription semantics, including guarantees of causal completeness, a property that most systems do not offer. Furthermore, we aim at studying techniques that can reduce the subscription latency. In this context, we have identified some possible strategies to reduce the subscription latency. The first includes the use of a more conservative event forwarding strategy while a subscription is propagated in the network, that can compensate for possible inconsistencies in the routing tables of different brokers. The second consist in adapting the overlay, to better fit the subscriber population at a given time. The report also discusses the different metrics that we plan to use to evaluate the resulting system.

Acknowledgments We are grateful to Nivia Quental for the fruitful discussions and comments during the preparation of this report.

References

1. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys, 35(2) (jun 2003)
2. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7) (jul 1978)
3. Bhola, S., Strom, R., Bagchi, S., Zhao, Y., Auerbach, J.: Exactly-once delivery in a content-based publish-subscribe system. In: International Conference on Dependable Systems and Networks. (jun 2002)
4. Bhola, S., Zhao, Y., Auerbach, J.: Scalably supporting durable subscriptions in a publish/subscribe system. In: International Conference on Dependable Systems and Networks. (jun 2003)
5. Esposito, C., Platania, M., Beraldi, R.: Reliable and timely event notification for publish/subscribe services over the internet. IEEE/ACM Transactions on Networking, 22(1) (feb 2013)
6. Carvalho, N., Araujo, F., Rodrigues, L.: Scalable qos-based event routing in publish-subscribe systems. In: IEEE 4th International Symposium on Network Computing and Applications. (jul 2005)
7. Cugola, G., Di Nitto, E., Fuggetta, A.: The jedi event-based infrastructure and its application to the development of the opss wfms. IEEE Transactions on Software Engineering, 27(9) (sep 2001)

8. Santos, V., Rodrigues, L.: Localized reliable causal multicast. In: IEEE 18th International Symposium on Network Computing and Applications. (sep 2019)
9. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems, 19(3) (aug 2001)
10. Zhao, Y., Sturman, D., Bhola, S.: Subscription propagation in highly-available publish/subscribe middleware. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. (oct 2004)
11. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: Symposium on Principles of Distributed Computing. (may 1999)
12. Kazemzadeh, R.S., Jacobsen, H.A.: Reliable and highly available distributed publish/subscribe service. In: IEEE 28th International Symposium on Reliable Distributed Systems. (sep 2009)
13. Kazemzadeh, R.S., Jacobsen, H.A.: Partition-tolerant distributed publish/subscribe systems. In: IEEE 30th International Symposium on Reliable Distributed Systems. (oct 2011)
14. Baldoni, R., Bonomi, S., Platania, M., Querzoni, L.: Dynamic message ordering for topic-based publish/subscribe systems. In: IEEE 26th International Parallel and Distributed Processing Symposium. (may 2012)
15. Chand, R., Felber, P.: Xnet: a reliable content-based publish/subscribe system. In: IEEE 23rd International Symposium on Reliable Distributed Systems. (oct 2004)
16. Chand, R., Felber, P.: A scalable protocol for content-based routing in overlay networks. In: IEEE 2nd International Symposium on Network Computing and Applications. (apr 2003)
17. Costa, P., Picco, G.P.: Semi-probabilistic content-based publish-subscribe. In: IEEE 25th International Conference on Distributed Computing Systems. (jun 2005)
18. Salehi, P., Doblander, C., Jacobsen, H.A.: Highly-available content-based publish/subscribe via gossiping. In: ACM 10th International Conference on Distributed and Event-based Systems. (jun 2016)
19. Lumezanu, C., Spring, N., Bhattacharjee, B.: Decentralized message ordering for publish/subscribe systems. In: ACM/IFIP/USENIX International Conference on Middleware. (nov 2006)
20. Costa, P., Migliavacca, M., Picco, G.P., Cugola, G.: Introducing reliability in content-based publish-subscribe through epidemic algorithms. In: 2nd International Workshop on Distributed Event-based Systems. (jun 2003)
21. Costa, P., Migliavacca, M., Picco, G.P., Cugola, G.: Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In: IEEE 24th International Conference on Distributed Computing Systems. (mar 2004)
22. Eugster, P.T., Guerraoui, R.: Probabilistic multicast. In: International Conference on Dependable Systems and Networks. (jun 2002)