Scalable Reliable Causal Multicast on the Edge

Válter Emanuel Trecitano da Costa Santos
valter.c.santos@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Edge computing is a paradigm where small datacenters and servers, located in the edge of the network, provide storage and computing services to smart devices. These devices need to offload tasks to the cloud, but cannot afford the high latency associated with the access to remote datacenters. Because edge devices may move and contact different edge servers, these servers need to coordinate to provide a consistent service to their clients. In this work we are interested in providing one of the most fundamental forms of consistency, namely causal consistency. For this purpose, we study ways of supporting causal multicast, among multiple groups of edge servers, in a reliable and scalable manner. Traditional techniques to enforce causality, such as maintaining vector clocks with one entry per edge server, may not scale in this setting. Thus we look for alternatives that are scalable, and still can provide a reliable service in face of transient failures of edge servers.

1 Introduction

The number of devices that are connected to the Internet keeps on growing, as more and more appliances have computing capacities. Many of these devices have sensors that produce large amounts of information that needs to be collected and processed. Also, most devices, such as smartphones, are becoming increasingly powerful and are today able to support novel applications such as augmented reality. Edge computing is a paradigm where small datacenters and servers, located in the edge of the network, provide storage and computing services to such smart devices. These devices need to offload tasks to the cloud, but cannot afford the high latency associated with the access to remote datacenters.

Because edge devices may move and contact different edge servers, these servers need to coordinate to provide a consistent service to their clients. In this work we are interested in providing one of the most fundamental forms of consistency, namely causal consistency. In particular, we are interested in mechanisms that can ease the deployment of a storage service, where different edge servers keep replicas of data items, that can be updated by different clients, maybe concurrently. Such updates need to be propagated and applied at different replicas in a way that prevents clients from observing violations of causality. It has been shown that causal consistency is the strongest consistency criteria that can be offered to applications without compromising availability (i.e., it does

not require the execution of consensus, that can be blocking)[1]. Also, causality is a key ingredient of many other stronger consistency criteria and, therefore, a fundamental building block to build distributed reliable systems.

Most systems that offer causal consistency resort to some form of reliable causal multicast, a primitive that is able to deliver messages in an order that respects causality. Note that, in an edge computing scenario, not all servers will keep replicas of all data items. Therefore, replicas would be organized in multiple groups, that can overlap only partially. When a data item is written, the update needs to be propagated to the group of nodes that maintain replicas of that item. However, because clients read and write different data items, causality needs to be preserved across groups. Classical techniques to implement causal order, such as the ones based on vector clocks[2] may not scale when applied to edge computing. Thus we aim at finding alternatives that are scalable, and still can provide a reliable service in face of transient failures of edge servers.

In order to implement reliable causal multicast one needs to address different complementary concerns. First data needs to be disseminated efficiently among group members. Second, dissemination needs to be reliable, which may require messages to be retransmitted when faults occur. Third, messages need to be tagged with metadata such that they can be delivered in an order that respects causality. Finally, the membership of each replica group can change dynamically. All these services need to be executed on top of some network that connects the edge replicas. Such network can be a clique, where each edge server is aware and can communicate with any other edge server directly, or an overlay network where each nodes is only aware of a limited number of other edge servers.

The literature is rich in systems that address several of these concerns, although very few systems address all concerns in a scalable and comprehensive manner. Among the relevant areas of related work we have identified the fields of overlay networks[3–6], reliable publish-subscribe systems[7–9], reliable causal multicast[2, 10] and causal storage systems[11–15]. In this report we survey some of the most relevant solutions from each of these fields and identify a set of mechanisms that can be useful to implement causal multicast on the edge. We then use these ideas to offer a comprehensive solution to the problem of supporting causal multicast in a reliable and scalable manner among edge servers.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. Section 3 explains in detail the participants in the edge. Section 4 presents the definition of *Reliable Causal Multicast* and how to implement it. In Section 5 we present all the background related with our work. Section 6 describes the proposed architecture to be implemented and Section 7 describes how we plan to evaluate our results. Finally, Section 8 presents the schedule of future work and Section 9 concludes the report.

2 Goals

The purpose of this work is to find scalable algorithms to implement scalable multicast among edge servers. More precisely:

> *Goals:* To devise a causal multicast protocol that is scalable, reliable, and can operate efficiently in a network with a large number of edge servers.

Any protocol that is able to ensure causal order in a fault-tolerant manner must resort to metadata, that is both stored at the endpoints and exchanged in the message headers. The most straightforward approach, in the line of systems such as [2, 12], would require nodes to maintain and exchange metadata whose size is linear with the number of servers and the number of replica groups. This can be an impairment to scalability. We aim at seeking algorithms that can implement causal multicast with a metadata size that is a function of the number of neighbours that each server has in the server network, that can be substantially smaller than the total number of nodes in the system.

The project is expected to produce the following results:

> *Expected results:* The work will produce i) a specification of a causal multicast protocol for the edge; ii) a prototype implementation, iii) an extensive experimental evaluation of its performance using different metrics such as latency, throughput, and metadata overhead.

3 System Model

We assume a model of edge computing as the one that is illustrated in Figure 1. In this model, we consider two main layers, namely the *cloud layer*, composed of large datacenters, and the *edge layer* composed of smaller servers, placed in the vicinity of end edge devices, and of the edge devices themselves.

In our work, we are not concerned on how the cloud layer is organized. We simply assume that cloud layer exhibits high-availability and has the capacity to store copies of objects that reside on the edge whenever needed. We also assume that the latency from the edge nodes to the datacenters may be relatively large, which motivates the need to keep replicas of data in the edge layer. Typically, the cloud layer is materialized by a relatively small number of large datacenters.

The edge layer can be divided into two main sublayers: the *user edge* and the *service edge* layers. The user edge layer is composed by the client devices, such as laptops, mobile phones, and IoT devices. These devices have low to medium capacity and need support from the service layer to store data and perform computation over data, which is replicated across several service layer nodes. The service edge layer is composed of application servers placed in private datacenters and in the premises on Internet Service Providers (ISPs), close to the edge, complemented by intelligent network components such as 5G towers. The nodes from the service layer have enough capacity to store a fraction, but not all, of the data stored at the cloud layer. They are expected to be available most of the time, but may suffer from transient unavailability periods.

Our work focuses on the service layer. In detail, we are interested in providing a reliable multicast communication primitive that allows edge servers to coordinate in order to provide a consistent view of the data that is stored in this layer

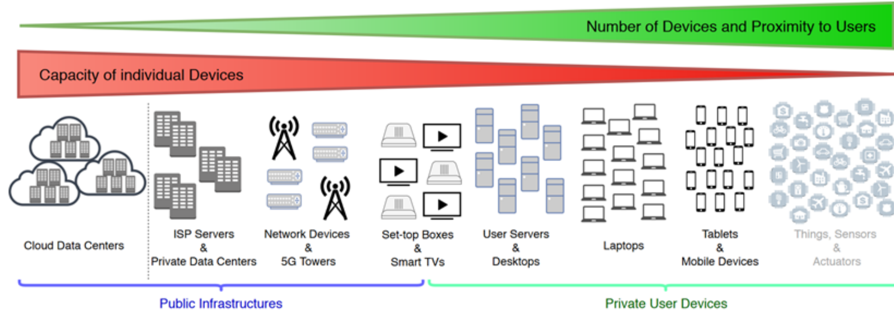

Fig. 1. The Edge

to the devices in the user edge. Because edge servers do not have the capacity to store all data, different servers will store different subsets, or *partitions* of the entire dataset. The group of servers that stores some partition i is denoted group G_i, and members of G_i will use a multicast primitive to coordinate in order to keep their replicas of data partition i consistent. As a result, multiple multicast groups will co-exist, and these groups may or may not overlap, or may overlap only partially.

4 Reliable Causal Multicast

In this section, we introduce the properties of the reliable causal multicast service we want to support and identify the main building blocks for implementing this service.

4.1 Properties

Reliable Causal Multicast is defined by the following definitions.

- *Asynchrony* Processes may broadcast messages concurrently.
- *Gapless Delivery* When a process P_1 delivers a message from another process P_2, all following messages from P_2 that P_1 is interested in receiving will be delivered by P_1.
- *No Duplication* No correct process delivers the same message more than once.
- *Causal Order* Messages may be received in any order, but are delivered in causal order, as defined by Lamport[16].

4.2 Building Blocks

As we have mentioned before, to implement reliable causal multicast it is necessary to combine different mechanisms that achieve complementary goals such

as efficient dissemination of data, fault masking or recovery, message ordering, and management of group membership. In detail, we can identify the following concerns, which are depicted in Figure 2.

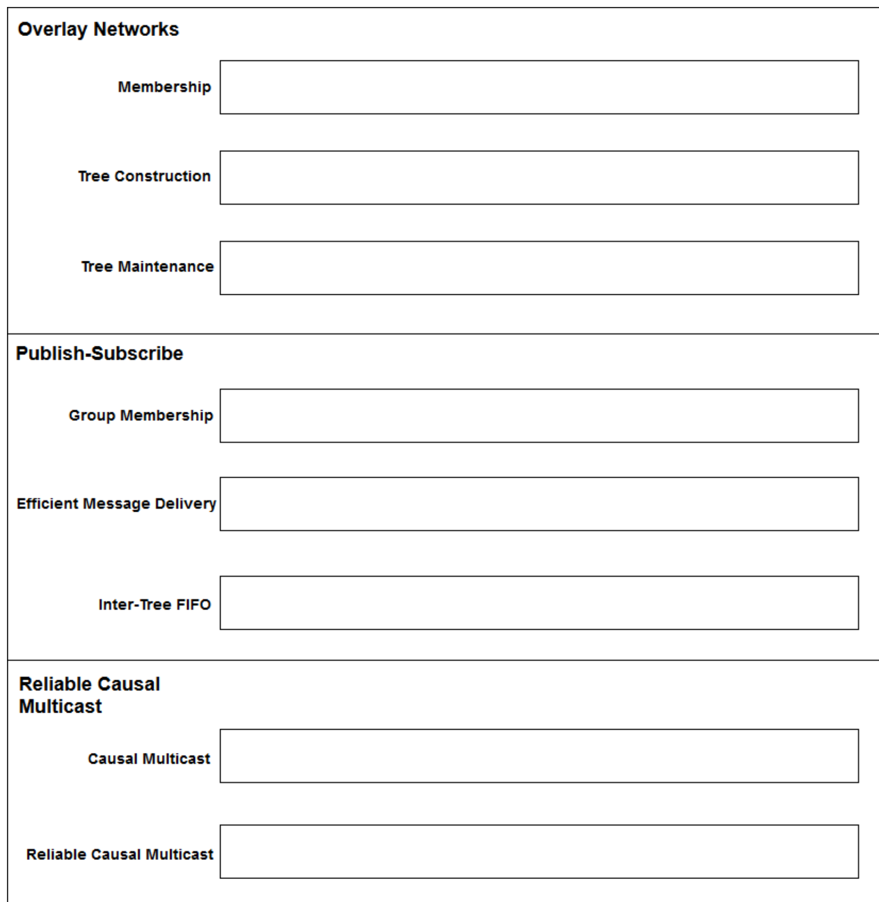

Fig. 2. Concerns of Reliable Causal Multicast

Overlay Membership addresses the problem of keeping all the nodes that participate in the network connected, by maintaining an overlay network that has at least one path between any two nodes. The simplest overlay is simply a clique, where every nodes knows every other node and can send messages directly to it. Maintaining a clique may not be scalable for large number of nodes. Alternative overlay structures, where each node is only required to maintain information regarding a small subset of the nodes in the system may scale better. Among the main classes of overlay networks it is useful to distinguish unstructured overlays,

using gossip protocols (such as HyParView[3]), and for structured overlays, such as Pastry[4].

Tree Construction addresses the problem of building a tree embedded in the underlying overlay. Trees offer the basis for implementing multicast efficiently.

Tree Maintenance addresses the problem of reconfiguring the tree when new nodes join, old nodes leaves, members of the tree crash, or links fail. A tree may also be reconfigured for improving performance, for instance when some links become congested and better paths become available.

Group Membership addresses the issue of upholding multiple application-level groups on top of a common underlying overlay. The membership of the groups is dynamic, and nodes can join or leave a group in run-time.

Efficient Message Delivery addresses the problem of propagating messages to the interested nodes as efficiently as possible. This usually means avoiding sending redundant messages to the same nodes and ensuring that only the nodes that are interested in the messages are required to participate in the multicast protocol.

Inter-Tree FIFO addresses the problem of ensuring that messages from the same sender are delivered in *First In First Out* order. This may require nodes to add metadata to the messages (such as sequence numbers) and to buffer out-of-order messages. Messages can be received out-of-order when different messages use different paths, or when messages are lost and later retransmitted.

Causal Multicast addresses the problem of ensuring that messages are delivered in an order that respects causality. As before, this may require nodes to add metadata to the messages (such as sequence numbers) and buffer out-of-order messages. The metadata required to enforce causal order can be significantly larger than the metatada required to ensure FIFO order. A significant challenge is to keep this metadata as small as possible, without inducing significant delays in the message delivery.

Reliable Causal Multicast addresses the problem of ensuring that messages are delivered reliably within each group. Informally, this usually means that if a member of the groups delivers a message, all members of the group also deliver that message.

In the literature, it is possible to find many works that address several of these concerns (but not all systems address all concerns). In the next section we describe some of the most relevant system from the related work that have influenced our design. When describing each system we will identify which concerns it addresses.

5 Related Work

5.1 Overlay Networks

We assume that every node in the system is able to exchange message any any other node using IP protocols, such as UDP or TCP. However, for scalability

reasons, it is often not desirable to maintain a full clique, since in that case every node needs to keep track of every other node in the system. Instead, it may be preferable to build an overlay network on top of IP, where each node is only required to become aware of a few neighbours. The construction of the overlay network can resort to a centralized component or be fully decentralized, also denoted as a *peer-to-peer* (P2P) network. P2P overlays can typically be classified into two main classes, namely: *structured* overlays, where nodes cooperate to maintain a distributed hash-table (DHT) and *unstructured* overlays, that put little constraints on how nodes establish neighbouring relations. The former have the advantage of supporting efficient routing but require the use of costly overlay construction and maintenance procedures while the later do not support routing but are cheaper to maintain.

5.1.1 HyParView

Objective HyParView[3] is an overlay construction and maintenance protocol that aims at building an unstructured overlay network where all links are bidirectional and each node has a small set of neighbours that represent a random sample of the entire network.

System model HyParView uses a fully decentralized P2P algorithm to build and maintain an unstructured overlay. The algorithms assumes that nodes are altruistic and only fail by crashing.

Algorithm In HyParView, each node maintains two distinct *partial views* of the system. The *passive view* offers a random sample of the entire network; it is maintained by having nodes to perform periodic gossip exchanges, to propagate information regarding existing nodes. The passive view is a building block to maintain a second view, called the *active view* that actually defines the HyParView overlay. A key property of HyParView is that the links defined the the active views are symmetric: if node n_i is the the active view of node n_j then node n_j is also in the active view of node n_i. The algorithms used to maintain both views ensure that, with high probability, the resulting overlay is connected, has an almost uniform degree distribution, small average diameter, and low clustering coefficient.

Concerns HyParView addresses the *Overlay Membership* concern.

5.1.2 Plumtree

Objective Plumtree[5] is an algorithm to create and maintain a spanning tree on top of an unstructured overlay network.

System Model Plumtree assumes that there is an underlying unstructured overlay that connects all nodes in the system and that this overlay, such as HyParView, is relatively stable (i.e., each node maintains its neighbours unless new nodes join, nodes leave or nodes fail) and where nodes have few neighbours. All nodes are altruistic and only fail by crashing.

Algorithm The Plumtree tree construction algorithm is based on a *broadcast and prune* strategy. The algorithm starts by eagerly sending the messages using f random neighbours in the underlying overlay. If a node receives a message multiple times, via different edges, it makes one of these edges as primary and the others as backup. Namely, it selects as primary the edge from which the message has been received first; this strategy tends to select edges on the shortest path as primaries. After all redundant edges have been turned into backups, the prune steps is finished and the remaining primary edges constitute a broadcast tree. Subsequent messages are sent using *eager push* on primary edges and using *lazy push* on backup edges, as follows:

- Eager push: Nodes send the message payload to the selected peers as soon as they receive it for the first time;
- Lazy Push: When a node receives a message for the first time, it sends the message id (but not the payload) to the selected peers. These messages are denoted "IHAVE" messages. If the peers have not received the message, they may make an explicit pull request.

Note that, at the start, each peer has two sets of peers: *eagerPushPeers* (EPP), which initially has f random peers, and *lazyPushPeers* (LPP), which is empty. Peers are moved from the first to the second set as a result of the broadcast and prune step described above.

Tree maintenance is performed as follows. When a node receives an IHAVE message from a backup edge without first receiving the corresponding payload from the primary edge, it starts a timer. If the timer expires before the payload is received, the primary edge is removed and the edge of the node who sent the IHAVE is promoted to primary.

The tree construction and maintenance algorithms use the message latency, with respect to the source of the messages, as a criteria to decide which edges are primary and which edges are backups. Therefore, the resulting tree is optimized for a single sender and may not provide good latency when used to propagate messages originating from other nodes. It is obviously possible to maintain multiple trees, one for each sender, but this may impose a significant signaling overhead on the system. The authors of Plumtree have also suggested a few strategies to tune a single tree to support multiple senders.

Overlay Dynamism Plumtree allows nodes to leave or join the overlay. If a node leaves, then it is simply removed from the membership. When a node joins the system, then it is added to the set of eagerPushPeers, being considered to become a part of the tree.

Concerns Plumtree addresses the *Tree Construction* concern, as it creates a tree overlay from a group of disconnected nodes by using gossip communication, and the *Tree Maintenance* concern as it handles nodes joining and leaving the system.

5.1.3 Thicket

Objective Thicket[6] is an algorithm to create and maintain *multiple* spanning trees on top of an unstructured overlay network. However, in opposition to algorithms that build a single tree, such as Plumtree, the construction of multiple trees is coordinated to ensure that, with high probability, each node is an interior node in only one tree and a leaf node in the remaining trees. This promotes a good load balancing among nodes that are part of more than one tree.

System Model This system, like Plumtree, assumes there is an underlying unstructured overlay that connects all nodes in the system and that this overlay, such as HyParView, is relatively stable and where nodes have few neighbours. All nodes are altruistic and only fail by crashing.

Algorithm The algorithm used by Thicket can be seen as an extension to the Plumtree algorithm. As in Plumtree each node also divides its neighbours in eager push and lazy push neighbours (here called *active peers* and *backup peers*). However, because the algorithm maintains multiple trees, Thicket maintains a separate active set for each tree.

The tree construction algorithm has been modified to promote load balancing for nodes that are part of multiple trees. The broadcast and prune procedure is changed as follows. As with Plumtree, the tree construction starts with a broadcast phase where each node send a message eagerly to f neighbours selected at random. However, if a node is already an interior node in another tree, it refuses to eagerly propagate the tree construction message and simply becomes a leaf. Since one or more nodes can refuse to participate in the broadcast phase, at the end of the prune phase it is likely that some nodes remain disconnected from the tree. This situation is detected thanks to the exchange of "SUMMARY" messages on the backup links (these messages play a role similar to that of IHAVE messages on Plumtree) .

If a node discovers, via a SUMMARY, that it is not part of a tree, it selects an edge to one of its upstream nodes to become a primary edge for that tree. To promote a good load balancing among nodes, each node keeps an estimate of the load of its neighbours (i.e., an estimate of the number of trees where that neighbour already plays the role of an interior node). It then selects the less loaded neighbour to become its source of eager push for the target tree. If an interior node crashes or leaves the network, the tree is repaired using a similar strategy.

Overlay Dynamism When a node n detects that another node p left the system, it simply removes that node from all active and backup sets. This may cause n to become disconnected from one or more trees, a scenario that is corrected by the tree repair algorithm briefly describe above. In case a node n detects a node p joining the system, then it adds p to the set of backup peers. This, in turn, will ensure that p will receive "SUMMARY" messages and will be able to active the tree repair mechanism to join all the active trees.

Concerns Thicket, like Plumtree, is an algorithm that addresses both *Tree Construction* and *Tree Maintenance* concerns.

5.2 Publish-Subscribe Systems

Publish-subscribe[17] (Pub/Sub) is a message passing paradigm that promotes decoupling between the producers of information (the *publishers*) and the consumers of information (the *subscribers*). When producing information, the publishers do not need to know the identity, number, or location of subscribers. Similarly, when consuming information, subscribers do not need to be aware of the identity, number, or location of publishers. This paradigm can be implemented in many different ways. One of the most common architectures to support publish-subscribe uses a network of intermediate brokers, that route messages from publishers to subscribers.

5.2.1 Scribe

Objective Scribe[18] is a Pub/Sub system built on top of a structured P2P overlay. It offers high scalability, efficient message propagation, and fault tolerance.

System Model Scribe implements what is known as *topic based* publish-subscribe. Publishers tag messages with a given *topicId* and subscribers use these topicIds to subscribe for events. Scribe builds and maintains a multicast tree for each different topicId. In Scribe, each node can play one or more of the following roles: publisher, subscriber, root of a multicast tree, or interior node of a multicast tree. Scribe is built on top of Pastry[4], a structured P2P DHT.

Algorithm Scribe builds a spanning tree on top of the underlying DHT for each topic. The tree is rooted in a *rendez-vous* node, the node with the identifier that is numerically closest to the topicId. To join a multicast tree, a subscriber uses the DHT to send a special "SUBSCRIPTION" to the rendez-vous point. This message is routed in the DHT and, each node in the path from the subscriber to the rendez-vous becomes an interior node in the tree. Reverse path forwarding is then used to route events from the rendez-vous to the subscribers. The algorithm buils a tree because "SUBSCRIPTION" messages do not need to travel up to the rendez-vous point: if they happen to be routed by a node that already belongs to the tree, that node registers the subscription locally and adds the downstream node to the list of tree branches (denoted as the *children table*). Publishers simply send events directly to the rendez-vous node that, in turn, uses the tree to send them to subscribers.

When a node wants to unsubscribe, it first checks if it does not have to forward the topic messages to other nodes by consulting its children table. If it does not, then it sends an "UNSUBSCRIPTION" message to its upstream node, who then performs the same procedure. This procedure is executed recursively until the "UNSUBSCRIPTION" message reaches a node that still has other entries in its children table.

Tree maintenance is performed with the help of "HEARTBEAT" messages. If a node suspects its parent has crashed, it then uses Pastry to re-subscribe to the same topic: Pastry will send the subscription message to the new parent, repairing the tree. Scribe tolerates faults of the rendez-vous node (the multicast tree root) by having its state replicated across the closest K nodes.

Overlay Dynamism New nodes can join the system by using Pastry. A new node then contacts a nearby node which gathers contact information for the new node. When a node leaves or fails, all nodes that knew the leaving node remove it from their contact list.

Concerns Scribe handles the *Tree Construction* and *Group Membership* concerns as it offers an algorithm to build a tree for each topic, where nodes can join or leave the topic group at will; *Tree Maintenance* concern as it allows nodes to join and leave the system and *Efficient Message Delivery* concern as messages are only propagated to the interested members of the overlay.

5.2.2 Gryphon

Objective Gryphon is a scalable content-based Pub/Sub system that has been developped by IBM. Different aspects of the system are described in different papers, including [9], [19] and [20]. In this subsection we focus on how the Gryphon ensures FIFO ordered exactly-once message delivery and how it handles subscriptions.

System Model The algorithm assumes that brokers are organized in an overlay that consists in a tree of logical nodes. The logical nodes that are leafs in the tree are materialized by a single broker. The logical nodes that are interior nodes in the tree are materialized by multiple "sibling" brokers. Publishers connect to the root broker (called the *pubend*) and subscribers connect to the leaf brokers (called the *subends*). When a message is forwarded to a logical node in the tree, it can be sent to any of the brokers associated with that logical node. Thus, a logical path from the root of the tree to a leaf logical node is supported by multiple redundant paths at the broker level. Global knowledge of the system is required, as each *pubend* needs to maintain state for each different *subend* and each broker may need to maintain state for each different *pubend* and *subend*.

Algorithms Gryphon uses an algorithm to ensure that messages are reliably delivered in a FIFO order, that we refer to as the *knowledge propagation* algorithm, and another algorithm to handle message subscriptions that we refer to as the *virtualtime* algorithm.

In a content-based publish-subscribe system, such as Gryphon, each subscriber may receive a different subset of the messages sent by the source (as a function of the content of those messages). This makes it hard to distinguish a message loss from a message that was filtered because it was not covered by a subscription. Gryphon solves this problem by requiring brokers that filter a

message to replace it by a *silence* token with the same sequence number as the filtered message. Using this strategy, sources can produce an ordered sequence of messages, and subscribers must receive a continuous stream of messages or silence tokens. Subscribers must acknowledge the reception of messages or silence tokens and should request the retransmission of messages (or silence tokens) if they observe a gap in the message stream. The downstream flow of messages and silence tokens is denoted by the authors as the *knowledge stream* and the upstream propagation of acknowledgements or retransmission requests as the *curiosity* stream. The authors propose a number of techniques to implement these streams efficiently and a technique that allows the system to identify when a given message as been received by all subscribers and no longer needs to be retransmitted.

The fact that multiple broker paths co-exists in a single logical path may create inconsistencies when subscriptions are forwarded from a leaf node to the root, given that the subscription information will not be propagated at the same pace in all broker paths. Thus, if two consecutive messages are propagated downstream using different broker paths, one message may use brokers that are aware of a new subscription and another message may use brokers that are not yet aware of that subscription. If care is not taken, one of these messages may be delivered to the new subscriber and the other may be dropped along the path, creating gaps in the message stream. To address this problem Gryphon relies on a *virtualtime* algorithm[9], that combines the following mechanisms. First, each subscription is assigned a logical time and each node keeps track of which subscriptions it is already aware. Second, each message is tagged by the sender with the logical time of the most recent subscription per *subend* known by the root. When a message is propagated downstream, two scenarios can happen. If the broker already knows all subscriptions known by the root it can use its own routing tables to decide if the message must be forwarded to each of its children. If the broker is outdated (i.e., it still misses some subscriptions) it floods the message to all children (this prevents messages from being prematurely dropped). As a result of this mechanism, when a subscriber sees the first message tagged with the corresponding *subend*'s virtual time greater or equal than its own subscription, it knows that it is safe to start consuming messages and that no gaps will be subsequently generated.

Overlay Dynamism The analyzed papers do not describe the algorithms used to add and remove brokers from the overlay.

Concerns The *knowledge* algorithm handles the *Efficient Message Delivery* and *Inter-Tree FIFO* concerns. The *virtualtime* algorithm handles the *group membership concern*

5.2.3 Fault-Tolerant Δ-neighbourhood.

Objective Kazemzadeh and Jacobsen[8] propose a technique to ensure the fault-tolerant implementation of a content-based Pub/Sub on an overlay of brokers,

where each broker is only required to maintain information regarding other brokers in its Δ-neighbourhood.

System Model Brokers are connected in a overlay in the form of a shared tree. However, the system assumes that links can be arbitrarily slow and nodes can become temporarily disconnected from other nodes in the tree. To tolerate these faults without rebuilding the tree, the algorithm allows messages to "jump" over faulty nodes when they are propagated in the tree. For instance, when a node is propagating a message downstream, if one of its children appears to be disconnected, the node can send the message directly to its grand-children, bypassing the faulty node. As a result, messages can be propagated using different paths. The algorithm ensure reliable, ordered, delivery of messages despite the fact that each broker only needs to contact and maintain information about neighbours on the tree that are at most Δ hops away.

Algorithm As noted above, when a message is propagated in the tree that constitutes the broker overlay, it may bypass faulty nodes. As a result, nodes may see only a subset of the messages and may miss important information. In particular, nodes in the tree may miss subscription information. The algorithm ensures that when a node recovers it eventually becomes up-to-date but, during transient periods, it may operate on outdated information and not be able to route events correctly. To address this problem, the Δ-neighbourhood enforces the following invariant: *"a publication is delivered to a matching subscriber only if it is forwarded by brokers that are all aware of the client's subscription"*.

The purpose of enforcing this invariant is to avoid scenarios such as the one described below:

- There are two subscribers, Sub_1 and Sub_2;
- Each subscriber propagates a subscription message, Sub_1 propagates s_1 while Sub_2 propagates s_2;
- s_1 is received by every broker in the path, but s_2 is not received by a broker B;
- A publisher publishes 3 messages m_1, m_2 and m_3, with m_1 and m_3 matching s_1 and m_1, m_2 and m_3 matching s_2
- B then receives m_1 and since it matches s_1, it forwards m_1 (Sub_1 and Sub_2 both receive m_1)
- B then receives m_2 but since m_2 does not match s_1, it does not forward the publication.
- Finally the broker B receives m_3 and since it matches s_1, it forwards m_3, with both Sub_1 and Sub_2 receiving m_1 and m_3, however Sub_2 did not receive receive m_2, resulting in a message gap.

The Δ-neighbourhood algorithm tolerates $\boldsymbol{\delta}$ concurrent faults, a fault being either a broker crash or a link failure. It includes three sub-protocols to address the following events: i) subscription propagation; ii) event forwarding and iii) broker recovery. They achieve $\boldsymbol{\delta}$ fault tolerance by having each node in the overlay maintain "knowledge" of Δ=$\boldsymbol{\delta}$+1 nodes in the neighbourhood, with knowledge

implying each node knows the Subscription Routing Tables (SRT, used to decide to which nodes to forward a message) of the neighbour nodes, as well keep track of how many messages were sent by each node (a variable called *brokerVal*) in a larger neighbour radius.

If a node A can not communicate with the next node B, then it communicates directly with the node (or nodes) after B, say C, being able to communicate directly to up $\boldsymbol{\delta}+1$ nodes in a given path.

Messages that need to be propagated in the tree are queued in a FIFO queue. Only one message may from each publisher be in-transit at any given point in time. After sending one message, the next message is not sent before the previous message has been fully acknowledged. At a given intermediate node in the tree, a message is just acknowledged upstream after all downstream nodes have acknowledged the message. A node that has no children to propagate a given message (either because the nodes is a leaf node or because its children do not have a matching subscription) can acknowledge a message immediately.

The paper introduces two concepts related to the occurrence of faults in the tree, namely *partition islands* and *partition barriers*. A partition island is a sequence of brokers that are not reachable (either due to crashing or having a link failure) by a broker but can be bypassed as they are less than $\boldsymbol{\delta}$ in a row. A partition barrier occurs when there are $\boldsymbol{\delta}$ or more unreachable brokers in a row and therefore can not be bypassed. When a node detects a partition island or barrier, then it becomes a Partition Detector (PD) and adds the partition's nodes (called *pid*, partition id) to its Partition Table (PT) and propagates the partition information to its neighbours.

The first sub-protocol, named subscription propagation, controls how a subscription is propagated downstream to the relevant brokers until it reaches the publisher. Each subscription message carries a predicate (which is used for matching), a subpath of brokers along the propagation path of the subscription (which is used for forwarding) and a sequence of numbers, which is used for duplicate detection, and when the subscription is accepted at the publisher, a confirmation message is sent upstream to the subscriber. There are three possible scenarios that can happen when propagating a subscription:

- Every broker on the path accepts the subscription, so every broker will be aware of the subscription and the confirmation message does not have any tags;
- Some brokers were unable to accept the subscription (partition islands), but the publisher accepted it. This means that some brokers downstream of the partition islands received the subscription, so they will send the subscription to the partition islands before forwarding them publications (when they can communicate with the islands) and the confirmation message does not have any tags;
- The subscription did not reach the publisher because of a partition barrier. In this case, the broker that detected the partition barrier stops attempting to forward the subscription downstream, tags the confirmation message with the *pids* of the partition barrier and sends the confirmation upstream, to

the subscriber. These tags will be used for resolving whether the subscriber should accept publications or not.

The second sub-protocol controls event forwarding. The following steps are executed when a publication p arrives at a broker B:

1. *Queuing step:* duplicate messages are detected and new messages are put on a FIFO queue.
2. *Barrier checking step:* B checks if p's sender is on any partition barrier known to B by checking its PT. If it is, then B tags the message with the corresponding *pids*. This tag is used by the subscribers, whose subscription was possibly confirmed with the same tags.
3. *Matching step:* B computes the subscribers that match p and obtains the next routes to forward the publication.
4. *Routing step:* B sends p to each path obtained in the matching step, and awaits an acknowledge from each path the message is sent to.
5. *Cleanup step:* p is discarded after receiving all acknowledges and an acknowledge is sent to whichever nodes sent p to B.

In case B can deliver p, then it verifies if it is safe to deliver the publication. For this, it compares the tags p carries with the tags of the subscription. If there is a shared tag, then it is unsafe to deliver p, as the publication may have been forwarded by a broker that did not know of the subscription. Otherwise it is safe to deliver.

In order to detect duplicate messages, each message carries a vector of metadata. This vector has a maximum size of $2\boldsymbol{\delta}+1$. Each entry of the vector is a pair (brokerId, brokerVal), where brokerVal is the number of new messages sent by the broker brokerId. This vector is updated as the message is propagated through the path to the subscribers. If a broker jumps over another broker when propagating the message, then it inserts a special null value in the vector, representing a jump and if it has to retransmit the message, the broker does not increase its own brokerVal, as it is not a new message that is being sent. As previously mentioned, the brokerVal of each neighbour is the second type of information that is kept by each node. A message is detected as a duplicate if for any entry of the vector, the receiving broker sees a brokerVal that is not higher than the corresponding broker's knowledge of said brokerId's brokerVal. If there are more than $\boldsymbol{\delta}$ jumps then the message is ignored, because if the message does not bypass more than $\boldsymbol{\delta}$ nodes in a given subpath of size $2\boldsymbol{\delta}+1$, then the message is always forwarded by a majority, which means a node will always see all messages and therefore be able to detect duplicates.

The third sub-protocol, broker recovery, is executed when a broker that was on a partition manages to regain connection to the rest of the network and now needs to recover the missing subscriptions. There are two types of recovery procedures, the first being a full recovery, for when the broker crashed and its SRT is empty and the second is partial recovery, for when the broker lost communication with the rest of the network and its SRT may be out of sync.

The partial recovery consists of the recovering broker (R) connecting to a stable broker (S), where R sends a summary of its SRT to S, S sends the missing subscriptions in R, R propagates the missing subscriptions to parts of the network that were partitioned and S removes the *pid* of the partitions that it is a PD from its PT and notifies its neighbours.

The full recovery consists of running the partial recovery protocol for every neighbour, as the broker crashed and is now establishing new connections to every neighbour.

Overlay Dynamism The Δ-neighbourhood algorithm does not focus in allowing new brokers join the system, merely stating brokers can join the system with the help of a registry service and then obtain knowledge of its neighbourhoods within a distance of Δ. However, when this happens, the broker is considered a permanent part of the system.

Concerns The Δ-neighbourhood algorithm addresses the *Group Membership*, *Efficient Message Delivery* and *Inter-Tree FIFO* concerns.

5.2.4 Sequencing Graph

Objective The *Sequencing Graph* algorithm[21] aims at providing a scalable and decentralized method of causally ordering messages across (potentially overlapping) groups of subscribers in a a topic-based Pub/Sub system.

System Model This system model consists of having subscribers join groups that represent topics, with the option to join or leave groups (by subscribing or unsubscribing to the corresponding topic), send messages to any group (even if they are not part of that group) and receive messages. For a subscriber to send a message, the message is sent to a graph of *sequencers*, which act as special brokers. An interesting note is that while the previous analyzed Pub/Sub systems use trees to connect nodes, this system uses a graph of sequencers with some invariants, namely a single path must connect sequencers associated with each group and the graph must be loop free.

Algorithm In this ordered message delivery system, for a subscriber to send a message, the message first needs to be sent to the sequencing network, where it then goes through the sequencing network collecting sequence numbers and then it leaves the network and is sent to the destination group. The focus of this work is within the second part of this process, namely how messages travel through the sequencing network.

Causal ordering is provided when senders subscribe to groups that they send messages to. These messages are not ordered across the system, as they may be delivered in any order to unrelated groups. This approach does not depend on the size of the destination group and in the worst case depends on the number of groups. The ordering protocol consists of having several sequencers, which are connected in graph overlay, assign sequence numbers to messages to the groups.

Groups may have members that are overlapped in other groups (referred to as double-overlapped) and these members must see the common messages by the same order. Each group has a specific sequencer and for each double overlap between groups there is a *sequencer* which handles all messages for both groups.

A sequencer can be seen as a broker, as its state consists of having a sequence number for its overlapped groups, a group-local sequence number for the group that is directly connected to the sequencer, information regarding the sequencers in the network on the path for each group and a buffer to store messages.

When a node sends a message to the sequencing network, the group-local sequencer attributes the message a group-local sequence number and if the destination group has no double overlaps, then the message can be delivered immediately. However, if there is a double overlap at the present sequencer then the current sequence number for the double overlap is added and the message is distributed to the next sequencer (in case it exists). As messages for double overlapped groups must go through the same sequencer, causal order is guaranteed.

Overlay Dynamism Sequencers can be added and removed as long as they follow two invariants: i) A single path must connect the sequencers associated with each group and ii) The sequencers graph must be loop free. The first one, there can only be one path connecting the sequencers to each group, is easy to guarantee. However, the second, the sequencer graph must be loop free, is harder and the authors decided it was difficult to uphold with only local knowledge and therefore use a full vision of the current sequencing graph.

Concerns As the name implies, the Sequencing Graph uses a graph instead of a tree. However the graph shares some of the properties of a tree (a single path connecting the sequencers with each group, undirected and loop-free). It addresses the *Tree Construction*, *Tree Maintenance*, *Efficient Message Delivery*, *Inter-Tree FIFO* and *Causality* concerns.

5.2.5 VCube-PS

Objective VCube-PS[22] is a topic-based Pub/Sub system that enforces causal order and propagates messages by dynamically building trees for each message.

System Model VCube-PS is built on top of an hypercube-like topology, where there is no network partitioning, nodes do not fail and links are reliable, meaning messages can not be lost, corrupted or duplicated, with messages of the same topic respecting causal order. Nodes on the overlay are grouped in clusters, where the neighbours of a node i are defined as the first fault-free node of each cluster.

Algorithm VCube-PS is different from the previous publish-subscribe systems as there are no brokers, with every node in this system being aware of the subscriptions of every other node. As such, we will explain how nodes subscribe to topics, how dynamic trees are created to propagate messages, the properties

of their message delivery and finally comment on how causality is maintained in the system.

There are three types of messages, namely *subscription*, *unsubscription* and *publish* messages, with the first two being sent to every node and the latter being associated with a topic and sent only to the nodes in the same topic. As mentioned in the System Model section, nodes have a neighbourhood and when a message is to be propagated, a dynamic tree is built using that information, with each tree being constructed by using the relevant neighbours of each node (all neighbours in the case of a subscription or unsubscription or the neighbours that are part of a topic, in case of a publish message), starting at the root and the tree will eventually have all nodes of the system (in case of a subscription or unsubscription message) or all nodes that are part of a topic.

VCube-PS provides causal order per topic, by using Causal Barriers[10]. This algorithm will be explained more in depth further in the report, but in summary, the algorithm uses direct dependencies on the messages instead of the nodes' identifiers, because it is more suitable for group dynamics where nodes can join or leave a group and it does not require all nodes of a group to have the same view of the group, like in CBCAST[2], another algorithm that provides causality and that will also be discussed. An example of this in VCube is considering there are two nodes, P_1 and P_2 in a topic t. P_1 sends m_1 to the members of t, which currently is only to P_2. In the middle of this, a node P_3 joins t. P_2 receives m_1 and broadcasts m_2 to t with a dependency on m_1. P_3 receives m_2 but then does not know when to deliver it, because it depends on the first message sent by P_1 (m_1), which P_3 does not know if the message was directed at itself as well. To solve this issue, nodes have a have a *Per-source FIFO Reception Order*, meaning messages published by the same publisher are received by the same order they are produced, which is assured by having the publisher only publish a new message after receiving all the acknowledgements for the previous message it previously published. If P_3 receives another message from P_1 in this topic then it knows that it will never receive m_1, as P_1 received the acknowledges for all destinations of $m1$, and therefore can deliver m_2.

Overlay Dynamism No assumptions are made about new nodes joining or leaving the system, other than nodes do not fail.

Concerns VCube-PS handles the *Group Membership*, *Efficient Message Delivery*, *Inter-Tree FIFO* and *Causality* concerns.

5.3 Reliable Causal Multicast

In this section we go through two important works related to Reliable Causal Multicast, which has been defined previously.

5.3.1 CBCAST

System Model CBCAST's system[2] is composed of N processes, where a process may belong to one or several groups of processes. Processes multicast messages

to groups. Processes may leave or join groups dynamically. Vector clocks are used to causally order messages, as is defined by Lamport[16].

Algorithm CBCAST operates under a *virtual synchrony* execution model, where messages are sent to groups and every recipient of the group is in an identical group view (informally, this means every process in a group has the same knowledge of which processes belong to the group) when the message arrives and messages are delivered fault-tolerantly, meaning all operational destinations eventually receive a message if it is sent.

This protocol uses vector clocks (VC) to order messages, with an entry of the vector clock corresponding to a process of a group, and with one VC per group. As it can be inferred, this will lead to a vast amount of metadata overhead in each message, however it is not always necessary to transmit the full vector timestamps, so these vectors can be compressed by only sending the entries that changed since the last sent message, e.g. if a process sends m_1 with the entire vector timestamp and then immediately after sends m_2, m_2 only needs to carry the entries of the vector that changed. However, this compression requires extra data to represent which fields changed, so in some cases the compression may cause more overhead than without compression.

To synchronize the views of the group when there is a change in membership, the concept "flushing" is used, where members send a message that contains the new view. For a process to send a flush message, it first waits for its multicasts to be stable (meaning its multicasts reached every destination). After a process sends a flush, it will accept and deliver messages but will not start multicasts. A member of the group, called *flush coordinator*, receives flushes from all members of the group and then sends its own flush message to all members. When a member receives the flush message from the coordinator then it means the system is stable and it can start multicasting. To support virtual synchrony when failures occur, a k-resilient protocol is used that delays communication outside of a group until all causally previous messages are k-stable. For example, if a process P_i has sent or received multicasts in group G_1, it will delay multicasting to a group G_2 until g_1's multicasts are stable.

5.3.2 Causal Barriers

System Model The Causal Barrier algorithm[10] ensures that messages are delivered according to causal order in a system composed of N processes that can communicate directly with each other. Unlike CBCAST, messages are not constrained to be sent to groups of processes that need to be explicitly created. Instead, any subset of the N processes can be selected as a multicast address for any message.

Algorithm Causal barriers is an algorithm that causally orders messages by using direct dependencies of messages instead of node dependencies.

An example of this algorithm is as follows (taken from the paper): Message M_1 is sent from P_1 to P_2, P_3 and P_4. P_2 receives M_1 and then sends M_2 to P_3,

P_4 and P_4. P_3 receives M_1 and M_2 and then sends M_3 to P_4. P_4 cannot deliver M_3 before M_2 and before M_2 it needs to first deliver M_1. Therefore, to guarantee causality M_3 only needs to carry information about its' direct dependency, M_2, not requiring to carry information about its transitive dependency on M_1 with respect to P_4.

Each process i has the following data structures:

- A counter $sent_i$ which counts the number of unique messages it sent to other processes;
- A N*N matrix called $Delivered_i$ to track dependency information. This tracks P_i's knowledge of the latest messages delivered to other processes. $Delivered_i$[j, k] = x would mean that P_i knows that all messages with sequence number equal or less than x from P_j where delivered to P_k.
- A vector CB of length N which stores direct dependency information. Each entry of this vector is a set of tuples, in the form of (process, counter). The number of tuples is bounded by N (meaning there is a message dependency from each other node), but is usually less than that. An example of an entry in the vector would be if a tuple (k, x) $\in CB_i$[j]. This means if P_i sent P_j a message, the next message sent could only be delivered after P_j receives the xth message from P_k.

If a process P_i receives a message *M*, it may also receive information saying which other processes received the same message (not necessary for the message to carry this extra metadata if it is known by other means which other processes will receive the message). Messages sent by P_i to those processes are then causally dependant on *M*, therefore the set of CB_i[k], where k is every other process that received *M*, is updated by adding the set ($sender_M$, senderCounter) and transitive dependencies are deleted.

The $Delivered_i$ matrix is used for garbage collection. For example, given $Delivered_i$[l, k] = y, then P_i knows that the yth message from P_l to P_k has been delivered, so if there is a set (l, x) $\in CB_i$[k] such that x < y, then P_i knows this constraint has already been fulfilled and can safely delete the set from CB_i[k].

5.4 Causally Consistent Storage

We now address replicated storage system that keep different data items, that can be read and written, and where clients observe a state that is always consistent with causality. We assume that the storage system is composed of multiple datacenters and each datacenter is materialized by a set of nodes. Inside each datacenter, different data items can be stored in different nodes (each set of items is denoted a *partition*). Datacenters may replicate all partitions (a scenario known as *full replication*) or just a subset of the partitions (a scenario known as *partial replication*). Clients execute read and write operations by contacting one of the datacenters and accessing the copy of the data item maintained at that datacenter. When a client makes an update, the update needs to be subsequently propagated to other datacenters.

5.4.1 Gentlerain

Description Gentlerain[11] is a consistent geo-replicated data store using causal order, where servers contain partitions that are replicated in other servers and if one partition is updated, the update is propagated to the servers that contain the replicated partitions.

Causality Each server maintains a version vector for each partition, and this vector has an entry for each data center the partition is replicated in. Each entry of these vectors is a timestamp and the *global stable time* (GST) is the lower bound of the minimum element of all partitions within the server.

Updates are timestamped with the physical clock of the originating server and remote updates become visible when the update's timestamp is older than the *global stable time*. If a partition is not updated, then the GST will not increase. In order to prevent this, partitions send heartbeats with their server's physical time.

Fault tolerance Servers are assumed to be replicated with strongly consistent replicas, so server failures are discarted; network failures are also assumed to be handled as datacenters have redundancy in their networks. Datacenters are not assumed to be immune to partitions and if this does happen then all remote updates stop being processed, as the GST stops advancing. In order to deal with this, they discard the partitioned datacenter from the system.

5.4.2 Cure

Description Cure[12] is a protocol for highly available transactions between datacenters. The system contains geo-replicated key-value stores that are replicated across D datacenters and each datacenter has N partitions.

Causality Each partition has two vectors, both of size D. where the first one tracks the number of updates received by the replicated partition in each datacenter and the second, which provides a globally stable consistent snapshot (GSS).

Remote updates carry a timestamp that is a vector clock with one entry per datacenter and when the timestamp is lower or equal to the GSS vector, then the remote update can be made visible.

Fault tolerance While if there is a failure in Gentlerain the entire system stops being able to make remote updates visible; in Cure, since there are vector clocks, updates from healthy DCs can be made visible, however updates from disconnected DC remain frozen until they recover.

5.4.3 Saturn

Description Saturn[14] is a distributed metadata service for Causal Consistency. In Saturn there are several datacenters that have data replicated across other

datacenters and in order to maintain causal order, each time data is updated in one datacenter, a *label* is generated and sent to the Saturn network to be propagated to the datacenters that replicate that data. Each Saturn node may have several datacenters attached.

Causality The Saturn network maintains causality with these updates by having each Saturn node connected in a tree overlay. Since nodes communicate with FIFO channels, causality is naturally maintained.

Each update has a constant metadata size, and unlike Gentlerain, there is no need for a global synchronization of the system, relying instead on the tree topology.

Fault tolerance The possibility of network partitions is taken into account, which can cause the tree topology to become disconnected. The authors do not focus in fault tolerance, using a fail-stop model in their implementation.

5.5 Comparison

Our system will need to deal with the **Group Membership** concern, as nodes should nodes should be able to join and leave groups (data partitions), **Tree Maintenance**, as it is desirable for service layer nodes to join and leave, **Efficient Message Delivery**, as messages should only be propagated to the interested members, **Inter-Tree FIFO**, as messages from one node should always be received in a FIFO order to another node in the system and finally **Reliable Causal Multicast** as messages need to be causally ordered while supporting faults.

We will now compare systems per section.

Overlay Networks In this section we analyzed two types of systems. The first is structured Peer to Peer system, with Scribe (running on top of Pastry) and the second is unstructured peer to peer, with Plumtree and Thicket (both using HyParView). Scribe's trees are built by using Pastry's routing, whereas Plumtree's and Thicket's are built by using gossip, with Thicket creating several trees.

Plumtree and Thicket both allow nodes to join and they eventually are made part of the tree (or trees). In Scribe, nodes may join the system by first using Pastry to become part of the overlay and then by using Scribe to subscribe to a topic and join a tree.

When nodes leave, the tree/trees become disconnected and the repair mechanism in Plumtree will create cycles (which they detect by receiving a message twice) and change the overlay in unwanted ways, which also happens in Thicket. In Scribe, if a node leaves, the tree is repaired using Pastry, at the cost of sacrificing *Inter-Tree FIFO*.

In the Sequencing Graph, when the sequencers join the system they have full information of the current sequencing graph, in order to prevent cycles in the graph and when a sequencer is removed, its parent is informed to connect to its children. Requiring the full information of the sequencing graph is not a desirable approach, however the logic for removing a node is useful.

Overlay Networks

Membership: Pastry, HyParView, HyParView

Tree Construction: Scribe, Thicket, Plumtree, Sequencer

Tree Maintenance: Scribe, Thicket, Plumtree, Sequencer

Publish-Subscribe

Group Membership: Scribe, Hans, Gryphon, VCube-PS

Efficient Message Delivery: Scribe, Hans, Gryphon, Sequencer, VCube-PS

Inter-Tree FIFO: Hans, Gryphon, Sequencer, VCube-PS

Reliable Causal Multicast

Causal Multicast: Sequencer, VCube-PS

Reliable Causal Multicast: CBCAST, Causal Barriers

Causally Consistent Systems: GentleRain, Cure, Saturn

Fig. 3. Concerns of Reliable Causal Multicast filled with the analyzed systems

Publish-Subscribe There are some differences in how each system handles group membership. In VCube-PS, all nodes are aware of every subscription of every node, while in Scribe and Δ-neighbourhood only the nodes on path from the subscriber to the publisher are aware of the subscription and in Gryphon the nodes in the redundant paths between a subscriber and the publish eventually receive the subscription. An unique aspect that both Δ-neighbourhood and Gryphon deal with, namely message gaps, are dealt in two different ways, with Δ-neighbourhood using *partition ids* and Gryphon using the concept of *virtual time* to detect when messages can start being delivered.

Scribe, VCube and Sequencing Graph are topic-based pub-subs while Gryphon and Δ-neighbourhood are content-based. Messages in Scribe need to go from the publisher to a special node (rendez-vous) who then spreads the message to the subscribers via a tree. This can require global knowledge if a node is a publisher that has topics in every node. In VCube messages are spread through dynamically built trees, Sequencing Graph uses a graph, Gryphon uses a spanning tree for each *pubend* and Δ-neighbourhood uses a single tree.

Scribe does not guarantee FIFO order by default. Gryphon guarantees Inter-Tree FIFO by having a stream of messages per publisher node where messages are continuously sent without having to wait for the previous message to be acknowledged. Sequencing Graph uses FIFO channels between nodes, however has no fault-tolerance. Δ-neighbourhood and VCube-PS wait for each message to receive system-wide acknowledges.

Reliable Causal Multicast Sequencing Graph provides causal order by having messages that are shared between elements of different groups go through a specific *sequencer*. VCube-PS offers causal order by using Causal Barriers, however it does not offer causal order between different topics. Also, both systems do not take into account node failures.

CBCAST and Causal Barriers are two algorithms that provide causality in group communication, albeit by using different kinds of metadata with several differences. Whereas CBCAST guarantees causality by having the sender of a message attach a vector clock for each existing group it is aware of, Causal Barriers uses direct dependencies on messages. Another difference is that in CBCAST messages are sent to specific groups and every process of the group must receive the message in the same "view" of the group, meaning if process sends a message to a group G_i and then a new process joins the group, every member needs to be sure they received every previously sent message in the previous view of the group before starting sending new messages to the new view of the group. In Causal Barriers, however, messages can be sent to one or several processes, with dependencies being tracked by knowing which processes received which messages messages. This either requires the nodes to know in some way who the recipients of the message are (in VCube-PS, this is known by having the message be sent to every subscriber of a topic, and every node knows who are the current subscribers for a topic) or additional metadata needs to be attached with each message, indicating who the recipients are.

There is also the difference in the amount of data that needs to be stored. Considering a system with N processes, in CBCAST a process only needs to store data related to the groups it belongs, which can be very small (if a process only belongs to one group of 2 processes) or very huge (if a process can indirectly interact with all possible group combinations of N processes). In Causal Barriers each process needs to always store a $N \times N$ matrix, to track how many messages a process knows each process received from another process.

	Type of Metadata	Uses Extra Metadata	Amount of Data Stored
CBCAST	Vector Clock(s)	No	Variable, from 2 to *
Causal Barriers	Message dependencies	Possibly	N x N

Table 1. Causality Protocols comparison

$$* = \sum_{r=2}^{n} \frac{n!}{r!(n-r)!}$$

Causally Consistent Storage In GentleRain, if there is one datacenter in a network partition then the entire system stops processing remote updates. In Cure, if there is one datacenter in a network partition then every remote update that is not related to that datacenter is able to be processed, however updates from failed datacenters remain frozen. In Saturn, if there is a leaf in the saturn tree that stops responding then the tree becomes disconnected and remote updates will not reach all datacenters through the tree. However, remote updates can still be processed as labels are piggybacked with the data itself.

Conclusion As our system is meant to be used on the edge, global knowledge is not scalable. Therefore, each node in our system will need to only use local knowledge. It is more interesting to use an unstructured peer to peer overlay, where nodes can join any part of the system without the restrictions of managing a DHT. As analyzed, it is possible to create a tree using gossip, which is enough for our objective. To deliver messages, Δ-neighbourhood is the most suitable work as it focuses purely on local knowledge. However, it has the constraint of requiring every subscriber to acknowledge each message before sending a new message. Gryphon, on the other hand, does not require this, at the cost of requiring global knowledge. Our ideal solution would be to mix these two solutions in order to forward messages while using local knowledge and not having to wait for all interested nodes to acknowledge.

Regarding the causality concern, our system cannot follow Sequencing Graph's ideas as we desire to support failures and the graph (in our case, tree) overlay would be changing very often (as subscribers join different groups and require new sequencers to be created), which is something that would be more expensive on our system. On the other hand, VCube offers per topic causal ordering, which

is a weaker property than the one we desire (inter topic causality) and in Vcube each node only publishes a new message after receiving acknowledges from all interested parties, which is a slower form of publishing messages, similar to Δ-neighbourhood, than the one we desire. However, by adapting Causal Barriers into our problem, this may be a good solution.

6 Architecture

Before discussing the architecture, let us recap our system model and the objective. We focus on the service layer of the edge, where edge servers are located and user devices are connected to these servers, interacting over *partitions* of data. As these servers (which we will call nodes) do not have the capacity to store all data, each node will store different partitions of data and when a partition i is updated in a node, all nodes that replicate that partition need to receive the same update, called *remote update*, obeying causal order between the nodes of the same group. Therefore, a group of nodes that replicate the partition i is denoted G_i. However, as the number of nodes and different groups can be vast, it is unfeasible for each node to be able to communicate directly with every other member in every group the node belongs to while guaranteeing causal order, which means these remote updates will need to travel through intermediary members, not belonging to the update's group, in order to reach all destination nodes while keeping the metadata for ensuring causal consistency small and the system tolerant to node failures.

Taking into account all analyzed systems and our concerns, we will address each concern and how our system will handle it, either by using an already existing system or by modifying one to better suit our requirements.

Membership Since our system is located on the edge, it is important that nodes are logically connected to physically close nodes, therefore an unstructured P2P overlay offers more freedom to achieve this, such as HyParView[3].

Tree Construction Taking into consideration that we chose an unstructured P2P approach for the $Membership$ concern, it is necessary to choose a protocol that creates a spanning tree on top of an unstructured overlay network. As shown by Plumtree[5] and Thicket[6], there are already existing protocols that handle this. As such, we do not create a new algorithm to handle this concern, instead using an existing one, choosing Plumtree as it is desirable to maintain a single tree.

Tree Maintenance Although the analyzed systems that handle the *Tree Construction* concern also handle *Tree Maintenance*, we will need to handle this concern, as the logic for the arrival and departure of nodes impacts directly our algorithm for *Reliable Causal Multicast*. This concern will be explored in detail in section 6.3.

Group Membership In our desired system, nodes can join a group G_i by replicating the i data partition. When this happens, all existing members of that group will need to know about the new member. However, as nodes will be separated in the tree, each node will receive the notification about a new member at different times. This can be seen as a subscriber joining a new topic and propagating its subscription across the system to every publisher of the given topic.

Efficient Message Delivery Basing on the previous concerns, we will have a single tree connecting every node in the system (similar to the Δ-neighbourhood), with remote updates published on this tree eventually reaching every destination. This tree will be rooted around a datacenter, which will reliably store most or all data partitions.

Inter-Tree FIFO Our work presents two requirements regarding this concern, namely each node should only need to use local knowledge of its neighbourhood (as in Δ-neighbourhood[8]) and a node that is the source of updates should be able to publish updates without waiting for each update to receive acknowledges from every interested destination (as in Gryphon[19]), and as previously mentioned in the conclusion of the related works, there is no existing work (to the best of our knowledge) that solves both requirements. Therefore we will create a new algorithm in order to solve the Inter-Tree FIFO concern with our requirements. This algorithm is explained later in section 6.1

Causal Multicast and Reliable Causal Multicast For these two concerns both CBCAST[2] and Causal Barriers[10] work, with both using additional metadata for different reasons. We adopt Causal Barriers, explaining in detail how this is applied in our algorithm in section 6.2.

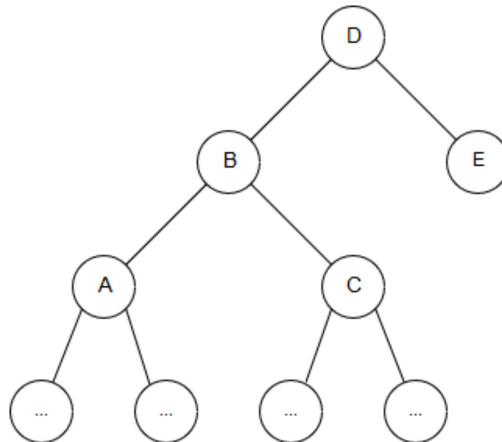

Fig. 4. Tree overlay

After analyzing each concern, we can divide our solution in three stages:

The first stage consists of providing *reliable delivery*, which in our system means nodes receive gapless ordered streams of messages from each node in a tree overlay in the face of node faults (section 6.1). The second consists of adding causality to the reliable delivery, providing Reliable Causal Delivery (section 6.2). The third and final stage consists of allowing nodes to leave or join the tree (section 6.3).

We will now discuss each stage in detail.

6.1 Reliable Delivery

As mentioned, it is our objective to have a system using local knowledge, like in Δ-neighbourhood[8], while not waiting for acknowledges across the system, like in Gryphon[19]. Therefore our solution uses ideas from both algorithms.

We want to tolerate $\boldsymbol{\delta}$ consecutive node failures on a given subpath of the tree overlay, so each node in our system will have knowledge of the Subscription Routing Tables of every node in a $\boldsymbol{\delta}+1$ radius, which will be the node's neighbourhood. Using figure 4 as an example overlay, if A wants to send a message that will be forwarded to D by B, then A sends the message to B, who then forwards the message to D. However, if B becomes unreachable (due to a network partition), then A contacts D directly and sends the message there instead, jumping over B. This is possible because of the neighbourhood knowledge.

This becomes more complex when A sends multiple messages to B which will be forwarded to D. If A sends a message m_1 to B who then forwards it to D and then B is detected as being in a partition and A sends m_2 directly to D, it is possible D did not receive the first message as B might have failed before sending the message to D, resulting in an unordered stream of messages.

Therefore, it is required for some kind of metadata to be attached to each message so that D is able to detect if there is any message missing.

To solve this, each time a node forwards a message to a path, it attaches two values to the message: An ID of the path where the message is supposed to travel and the number of messages sent to that path by the node. Each node would be required to store a list of path IDs and the number of messages received per path. When a node receives a message, it compares the metadata entries of the message to the metadata that it has stored. For each path on the message, it compares the corresponding number on the message with the number it has stored:

- If all entries have a number higher by one compared to the entries that are stored, then the message is ordered;
- If at least one entry on the message has a lower or equal number than the one that is stored, then the message is a duplicate;
- If at least one entry on the message has a number higher by more than one than the one that is stored, then there is a missing message.

The following scenarios will use the path itself as the path ID to better explain what is put on each message.

In the described scenario and with $\boldsymbol{\delta}$=1, A would send m_1 with a value "ABD:1" to B and if B failed afterwards, A would send m_2 with a value "ABD:2" to D. If D did not receive the first message, then it would notice that a message is missing and would either wait for m_1 to arrive or ask A directly for the missing message.

The size of this path is $\boldsymbol{\delta}$+2. This value comes from the node at the start of the path (1), the number of intermediate nodes that can fail ($\boldsymbol{\delta}$) and the node at the end for the path (1). We consider the message has a vector of metadata, with each entry being a pair (Path:Value), corresponding to a path and the number of messages sent to said path, and nodes can add or remove one entry to the vector. Since every node will be attaching metadata to the message, there has to be a limit to the size. This limit is $\mathbf{2}\boldsymbol{\delta}$+1, for the same reason as in Δ-neighbourhood[8], namely it is required for retransmissions of the same message to pass through a common node in order to detect duplicate messages.

This solution is a tradeoff between the size of metadata each message carries and amount of metadata that each node needs to store, with a focus on smaller message metadata and bigger metadata stored.

Since a node may need to forward the same message to different paths, having the metadata for a single path will not suffice to guarantee our reliable delivery to all routes. Therefore, the message will need to have additional metadata entries, one for each path. These additional entries may then be removed when the message is forwarded to the different paths. For example, in Figure 4, if A sends m_1 to both D and C through B, A puts [ABD:1, ABC:1] on the message. When B receives m_1, B sends m_1 with [ABD:1, BD:1] to D and m_1 with [ABC:1, BC:1] to C. However, not removing these entries is useful, as shown in the next subsection.

6.2 Reliable Causal Delivery

We start this discussion with the following scenario using figure 4. A sends message m_1 to both D and C through B. B forwards m_1 to C but crashes before sending it to D. C wants to send a message m_2 to D but cannot communicate with B, therefore it sends m_2 directly to D. D will receive m_2 before m_1, and since it does not know that there was a causal dependency between messages, causality will be broken.

Stabilization tactics (either by adding a single entry or vector clock to track dependencies, like in GentleRain or Cure) do not work here as if B does not recover, stabilization will never occur therefore D will not know if there was any messages sent by B that caused other messages.

However, by using Causal Barriers[10], C can send m_2 directly to D and say that it depends on m_1. D will see that it did not receive m_1 and either wait to receive it or ask for it directly.

In the previous subsection, it is mentioned that when a message is forwarded to several paths, the message needs to carry each path as additional metadata and when the message is forwarded to each path, the additional metadata may be removed. However, by using Causal Barriers this metadata will not be removed

as it is required so that each node knows which other nodes received the same message. This is the required extra metadata as indicated on Table 1.

6.3 Overlay Dynamism

We assume nodes may join or leave the tree. For a node to join, it will need to synchronize with its neighbourhood. As such, when a node joins the tree, it will communicate with every neighbour to announce its presence, generate path IDs that include itself and distribute them to the neighbours. For a node to leave, if it is a leaf then it can send a special message to its neighbours so that they remove the path entries related to the leaving node, otherwise it also needs to tell its children to communicate directly to the parent node, so that they can update their neighbourhood.

7 Evaluation

The system will be evaluated based on the impact of metadata with various values for $\boldsymbol{\delta}$ to analyze the impact on throughput based on the amount of consecutive faults supported. The metadata used to support causality will also be evaluated, testing the system with and without this additional metadata for various values of $\boldsymbol{\delta}$.

As our objective is to create a system that is scalable to the scales of the edge, we will compare our system with an implementation of Gryphon (that is topic-based instead of content-based), Δ-neighbourhood and a clique, where every node communicates directly with every node. However, Gryphon and Δ-neighbourhood were not designed to support causal order, so these two systems will be compared to our system without the use of causal order metadata.

Latency will be measured, namely the time difference between sending messages without any jumps on the path and with jumps and also how long it takes one message to reach its destination based on various values of $\boldsymbol{\delta}$. For causality, different timeout values will be used when waiting for messages to arrive before asking for the messages directly, in order to obtain the best balanced value.

Membership dynamism will be evaluated based on the value of $\boldsymbol{\delta}$ to compare the delay between synchronizing with the neighbourhood and starting to propagate messages.

8 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.

9 Conclusions

Edge computing is a new paradigm where small datacenters or servers are located in the edge of the network to provide services for clients. Since these servers share data, it is required to discover a way to multicast updates between servers, in a reliable and causal way that is also scalable to handle the edge. In this work we discussed the motivation for requiring reliable causal multicast, what it is, where it is to be applied, the requirements for it and several works related to one or several of the requirements. We presented a scalable algorithm in order to support reliable causal multicast on the edge. Finally, we presented our evaluation methods and the schedule of future work.

Acknowledgments

We are grateful to Manuel Bravo for the fruitful discussions and comments during the preparation of this report. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) and Feder through the projects with references PTDC/EEI-COM/29271/2017 (Cosmos) and UID/ CEC/ 50021/ 2019.

References

1. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing (mar 1995)
2. Birman, K., Schiper, A., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Transactions on Computer Systems (aug 1991)
3. Leitao, J., Pereira, J., Rodrigues, L.: Hyparview: A membership protocol for reliable gossip-based broadcast. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. (jun 2007)
4. T., A.R., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware. (jan 2001)
5. Leitao, J., Pereira, J., Rodrigues, L.: Epidemic broadcast trees. In: 26th IEEE International Symposium on Reliable Distributed Systems. (oct 2007)
6. Ferreira, M., Leitao, J., Rodrigues, L.: Thicket: A protocol for building and maintaining multiple trees in a p2p overlay. In: 2010 29th IEEE Symposium on Reliable Distributed Systems. (oct 2010)
7. Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R.E., Sturman, D.C.: An efficient multicast protocol for content-based publish-subscribe systems. In: Proceedings. 19th IEEE International Conference on Distributed Computing Systems. (jun 1999)
8. Kazemzadeh, R.S., Jacobsen, H.: Partition-tolerant distributed publish/subscribe systems. In: IEEE 30th International Symposium on Reliable Distributed Systems. (oct 2011)
9. Zhao, Y., Sturman, D.C., Bhola, S.: Subscription propagation in highly-available publish/subscribe middleware. In: Middleware. (oct 2004)
10. Prakash, R., Raynal, M., Singhal, M.: An efficient causal ordering algorithm for mobile computing environments. IEEE 32nd International Conference on Distributed Computing Systems (may 1996)

11. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. Proceedings of the 5th ACM Symposium on Cloud Computing (nov 2014)
12. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: IEEE 36th International Conference on Distributed Computing Systems. (jun 2016)
13. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. Proceedings of the 23rd ACM Symposium on Operating Systems Principles (oct 2011)
14. Bravo, M., Rodrigues, L., Roy, P.V.: Saturn: a distributed metadata service for causal consistency. In: EuroSys. (apr 2017)
15. Afonso, N., Bravo, M., Rodrigues, L.: Armazenamento de dados com coerência causal na periferia da rede. In: Actas do décimo Simpósio de Informática (Inforum). (sep 2018)
16. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM 21 (jul 1978)
17. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Comput. Surv. (jun 2003)
18. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: Scribe: a large-scale and decentralized publish-subscribe infrastructure. IEEE Journal on Selected Areas in Communications (jan 2002)
19. Bhola, S., Strom, R., Bagchi, S., Zhao, Y., Auerbach, J.: Exactly-once delivery in a content-based publish-subscribe system. In: Proceedings International Conference on Dependable Systems and Networks. (jun 2002)
20. Aguilera, M., Strom, R., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. Symposium on Principles of Distributed Computing (jan 2003)
21. Lumezanu, C., Spring, N., Bhattacharjee, B.: Decentralized message ordering for publish/subscribe systems. In: Middleware. (nov 2006)
22. d. Araujo, J.P., Arantes, L., Duarte, E.P., Rodrigues, L.A., Sens, P.: A publish/subscribe system using causal broadcast over dynamically built spanning trees. In: 29th International Symposium on Computer Architecture and High Performance Computing. (oct 2017)