Localized Reliable Causal Multicast

Válter Emanuel Trecitano da Costa Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

November 2019

Acknowledgments

First, I would like to thank my advisor Professor Luís Rodrigues for accepting me as his student and for the countless discussions we had along the year, which made this thesis possible.

I want to thank my friends that I met on the first year of college, namely, Diogo Antunes, Francisco Almeida, Miguel Martins and Ricardo Pereira. Enduring the hard years of IST with you made it much easier.

Throughout the duration of this thesis I resided in the best room of INESC, room 501, where I shared several meals and conversations with several people: my advisor's other sudents, Claúdio Correia and Taras Lykhenko; the other students in the room; Diogo Barradas, Maria Casimiro, Tiago Brito and new arrivals; Filipa Pedrosa and Luís Gomes. Meeting all these people made this year much more interesting.

Finally, I want to thank my mother, Maria João Santos, and my sister, Íris Santos, for supporting me throughout all these years.

To each and every one of you – Thank you.

Abstract

This thesis addresses the problem of offering reliable causal multicast in a setting where nodes are organized in an overlay network and use this network to disseminate information among each other. The use of overlay networks for this purpose is widely used when the number of nodes is large. For instance, many publish-subscribe system use an overlay of message brokers to support the exchange of information among publishers and subscribers. To the best of our knowledge, previous multicast algorithms for overlay networks either do not enforce causal order or, in order to do so, require nodes to keep metadata (for instance, sequence numbers) for all senders and are, therefore, inherently non-scalable. In this thesis we propose a novel localized algorithm to implement reliable causal multicast, where each node is only required to keep metadata regarding nodes in its neighbourhood (with a radius that is a function of the number of faults that need to be tolerated). Experimental results show that our algorithm can achieve significant improvements over non-localized alternatives, and can even outperform localized algorithms that do not offer causal order.

Keywords

Distributed Systems; Causal Order; Reliable Multicast; Localized Algorithms.

Resumo

Esta tese aborda o problema de oferecer difusão causal fiável em grupo em cenários onde os participantes estabelecem uma rede sobreposta e usam esta rede para trocar informação usando grupos de difusão. O uso de redes sobrepostas neste contexto é comum sempre que o número de participantes é elevado. Por exemplo, vários sistemas de edição-subscrição organizam os servidores que encaminham eventos numa rede sobreposta que é usada para propagar os eventos dos editores para os subscritores. Tanto quanto sabemos, o trabalho anterior que abordou o problema da difusão em grupo fiável neste contexto não oferece garantias de ordem causal ou para conseguir este objetivo, obriga todos os nós a manterem meta-informação (tipicamente, números de sequência) sobre as mensagens enviadas por todos os outros nós no sistema. Por este motivo, estas últimas soluções não possuem capacidade de escala. Nesta tese propomos um novo algoritmo localizado para suportar difusão em grupo fiável com ordem causal. Este algoritmo requer que cada participante mantenha meta-informação referente a apenas um subconjunto de participantes no sistema, aqueles que estão na sua vizinhança na rede sobreposta (o horizonte desta vizinhança é uma função do número de faltas que se pretende tolerar). Resultados experimentais mostram que o nosso algoritmo permite obter vantagens significativas em comparação com soluções não localizadas e, em certos cenários, tem mesmo melhor desempenho do que outros algoritmos localizados que, ao contrário do nosso, não suportam ordem causal.

Palavras Chave

Sistemas Distribuidos; Ordem Causal; Difusão em Grupo Fiável; Algoritmos Localizados.

Contents

1 Introduction **2**

1.1 Motivation . . . 3

1.2 Contributions . . . 4

1.3 Results . . . 4

1.4 Research History . . . 5

1.5 Thesis Outline . . . 5

2 Related Work **6**

2.1 Properties . . . 7

2.1.1 Properties . . . 7

2.1.2 Building Blocks . . . 7

2.2 Overlay Networks . . . 10

2.2.1 HyParView . . . 10

2.2.2 Plumtree . . . 11

2.2.3 Thicket . . . 12

2.3 Publish-Subscribe Systems . . . 13

2.3.1 Scribe . . . 13

2.3.2 Gryphon . . . 14

2.3.3 Delta-Neighbourhood . . . 16

2.3.4 VCube-PS . . . 20

2.4 Reliable Causal Multicast . . . 21

2.4.1 CBCAST . . . 21

2.4.2 Causal Barriers . . . 22

2.5 Comparison . . . 23

3 Local Causal Multicast **27**

3.1 Goals . . . 28

3.2 System Model . . . 28

3.2.1 Base Graph and Extended Graph . . . 28

3.2.2 Addressing Model . . . 29
3.2.3 Fault Model . . . 30
3.2.4 Localized Information and Safe Neighbourhood . . . 31
3.3 An example . . . 31
3.4 Message Identifiers . . . 32
3.5 Duplicate Detection and Merging Identifiers . . . 34
3.6 Safe Paths . . . 34
3.7 Causal Past and Causal Order . . . 36
3.8 Detailed Algorithm . . . 36
3.8.1 Overview . . . 36
3.8.2 Node Data Structures . . . 37
3.8.3 Message Control Fields . . . 38
3.8.4 Message Reception . . . 38
3.8.5 Message Forwarding . . . 38
3.8.6 Message Retransmission . . . 39
3.9 Additional Features of the Algorithm . . . 40
3.9.1 Garbage Collecting Messages . . . 40
3.9.2 Garbage Collecting the R_i Matrix . . . 40
3.9.3 Optimizing the Message Headers . . . 41
3.10 Correctness . . . 42
3.10.1 Safety . . . 42
3.10.2 Liveness . . . 43

4 Implementation **45**
4.1 Development Environment . . . 46
4.2 Framework . . . 46
4.3 Algorithms . . . 47
4.3.1 LoCaMu . . . 47
4.3.2 Delta-Neighbourhood . . . 48
4.3.3 CBCAST . . . 49
4.3.4 Causal Barriers . . . 49

5 Evaluation **51**
5.1 Goals . . . 52
5.2 Analytical Analysis . . . 52
5.2.1 Comparison of Localized Algorithms . . . 52
5.2.2 Comparison of Causality Algorithms . . . 53

5.2.3 Comparison of the overhead 53
5.3 Experimental Settings 54
5.4 LoCaMu vs Delta-Neighbourhood 56
5.5 LoCaMu vs Causal Multicast Algorithms 57
5.5.1 Single Group 58
5.5.2 Various Groups 59
5.6 Metadata Size Comparison 59
5.7 Discussion 61

6 Conclusion **63**
6.1 Conclusions 64
6.2 System Limitations and Future Work 64

List of Figures

2.1 Concerns of Reliable Causal Multicast 8

2.2 Concerns of Reliable Causal Multicast filled with the analyzed systems 24

3.1 Base graph and the corresponding extended graph ($f = 1$) 29

3.2 Message targets and active connections. 30

3.3 Scenario where causality is broken if node D delivers m_2 before m_1. 31

3.4 Scenario where a duplicate message is delivered. 32

3.5 Creation and propagation of identifiers ($f = 1$) 33

3.6 Duplicate detection and merging of identifiers ($f = 1$) 34

3.7 Invalid paths ($f = 1$) 35

3.8 Examples of a safe neighbourhood (blue circle, $f = 1$) 35

3.9 Forbidden message retransmission 39

3.10 Different retransmission scenarios ($f = 2$) 39

3.11 The existence of groups creates the need for a special mechanism to garbage collect the R matrix 41

3.12 D_m without (left) and with (right) compression. Without compression, the grey fields are not sent. With compression, each every field is sent, with the IDs occupying 4 bytes (letters) and the numbers 4 bytes. 41

4.1 Interaction between each component in the framework 47

5.1 LoCaMu vs Delta-Neighbourhood under different settings 56

5.2 Throughput comparison (left) in a system with 511 nodes with 1 group between the causal algorithms and the corresponding average metadata size per message (right). Bandwidth is 100 mbps (top) vs 10 mbps (bottom) 58

5.3 Throughput comparison (left) in a system with 511 nodes with 255 groups between the causal algorithms and the corresponding average metadata size per message (right). Bandwidth is 100 mbps (top) vs 10 mbps (bottom) 59

5.4 Metadata worst case and memory required (causal algorithms). Default settings: $f = 1$, node degree = 2, nodes = $65,536$ . . . 60

List of Tables

2.1 LoCaMu vs Related Work. The fault tolerance approaches are Virtual Reliable Nodes (VRN), which consists of replicating each node; On-line overlay reconfiguration (OR), which consists of reconfiguring the tree whenever a node crashes and Redundant Paths (RP), which allows each node to have a limited number of backup connections to other, nearby nodes. . . . 26

List of Algorithms

1 LoCaMu . . . 37

Acronyms

FIFO	First in, first out
IP	Internet Protocol
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
P2P	Peer-to-peer
DHT	Distributed Hash Table
SRT	Subscription Routing Table
VC	Vector Clock
KB	Kilobyte
MB	Megabyte

1 Introduction

Contents

1.1 Motivation 3
1.2 Contributions 4
1.3 Results 4
1.4 Research History 5
1.5 Thesis Outline 5

This thesis addresses the problem of offering reliable causal multicast in a setting where nodes are organized in an overlay network and use this network to disseminate information among each other. The use of overlay networks for this purpose is widely used when the number of nodes is large. Typically, the overlay is used to reduce the load imposed on individual nodes: a sender, instead of sending a copy of a message to a potentially large number of recipients, it just sends a copy of the message to a few neighbours which, in turn, propagate the message in the overlay network, distributing the load of message dissemination. Unfortunately, the use of overlay networks, *per se*, is not enough to ensure the scalability of the system. In order to detect message omissions, to discard message duplicates, and to ensure ordered delivery, many algorithms require participants to keep metadata (for instance, sequence numbers) for all nodes in the system and are, therefore, inherently non-scalable. In this thesis we are interested in developing and evaluating localized algorithms, i.e., algorithms where each node is only required to maintain information regarding a small number of neighbours, to implement reliable causal multicast.

1.1 Motivation

Reliable multicast is a classical problem in distributed systems, and a fundamental building block of distributed fault-tolerant systems [1–3]. Typically, reliable multicast protocols offer not only delivery guarantees but also ordering properties. Relevant ordering properties in this context are First in, first out (FIFO) order [4, 5] and causal order [1, 6, 7]. The problem is relevant for systems of all scales, from small-scale systems with less than a dozen replicas (such as a replicated server in a single data center) to large-scale systems with hundreds or thousands of participants (such as large scale publish-subscribe system). This thesis addresses the problem of ensuring reliability and causal order for large scale systems.

Techniques to implement reliable multicast with causal order for small-scale systems are now well studied, and several widely available systems, both academic and commercial, offer this service [1, 8, 9]. However, providing causal order for large scale systems is much more challenging. This happens because causal order protocols are required to maintain metadata (such as sequence numbers) to keep track of causal dependencies among messages. This metadata usually has the form of set of vector clocks [1] or a matrix clocks [6], that have an entry for each sender in the system. All these approaches are suitable for systems with small numbers of nodes but are inherently non-scalable. Some approaches offer causal order with less metadata, but are not able to deliver messages unless all nodes periodically send messages [10] or require nodes to be reliable and not fail (by replicating all nodes in the system) [7]. Again, none of the later approaches is practical in large-scale systems. This dissertation explores a different alternative, that requires each node to maintain metadata for a constant number of nodes in the system, regardless of the system scale. To the best of our knowledge this is the first localized algorithm to enforce causal order.

This work assumes that the nodes in the system are organized in an overlay network, that can be modelled as an acyclic graph. The idea of organizing nodes in an overlay network to support multiple variants of group communication in large-scale systems has been extensively applied, particularly in the implementation of publish-subscribe systems, where the communication among publishers and subscribers is supported by an overlay of message brokers [4, 11]. In the overlay network, each node only maintains direct links with a small number of *neighbours* and should not be required to have information regarding the global system membership. An algorithm that requires each node to know only its k-neighbourhood (where k is the maximum distance between the node and any of its neighbours) is said to be *localized*. Previous work has proposed a localized algorithm for reliable propagation of information in publish-subscribe systems [5] but it only supports FIFO order and, furthermore, restricts the communication pattern, by preventing each publisher from having more than one message in transit at any given time. This dissertation's resulting algorithm, which has been named LoCaMu (Local Causal Multicast), is substantially more powerful because it offers causal order and allows multiple messages to be in transit, fully exploring message pipelining in the overlay. In LoCaMu, in order to tolerate f faults in a given part of the overlay, each node is required to maintain metadata for nodes in its $(2f + 1)$-neighbourhood.

LoCaMu has been implemented and simulations have been used to assess its performance against other algorithms that can offer causal order and also against [5], which does not enforce causal order but, as LoCaMu, uses a localized approach. Experimental results show that by keeping the metadata required to enforce causality localized, the system can scale without increasing the size of the metadata, even when several nodes are publishing messages concurrently, unlike the related work. Thus, LoCaMu can be successfully used in large scale systems where, due to the size of their metadata, previous works cannot be implemented in practice.

1.2 Contributions

This thesis describes the design, implementation, and evaluation of a reliable causal multicast algorithm. The main contribution of this dissertation is a novel reliable multicast algorithm that provides causal order across multiple groups based solely on localized information.

1.3 Results

The following results have been achieved from the work described in this dissertation:

- The specification of LoCaMu, a local causal multicast algorithm;

- An implementation of the proposed algorithm on the Peersim [12] simulator;
- An experimental evaluation of the algorithm and comparison with other algorithms that provide similar properties.

1.4 Research History

This work was developed in the context of the Cosmos research project, that aims at finding techniques to offer causal consistent storage for edge computing scenarios. Techniques to reliable multicast updates to large numbers of data replicas in causal order are expected to be a key component in the final COSMOS architecture.

In my work I have benefited from the useful feedback from the team members of COSMOS, both from INESC-ID Lisboa and from NOVA LINCS.

The work described in the thesis has been partially published in the following papers:

- V. Santos and L. Rodrigues. Difusão em Grupo Tolerante a Faltas com Ordem Causal Usando Informação Localizada. In *INForum 2019* [13].
- V. Santos and L. Rodrigues. Localized Reliable Causal Multicast. In *IEEE NCA 2019* [14].

This work was partially supported by FCT – Fundação para a Ciência e a Tecnologia, through projects UID/CEC/50021/2019 and COSMOS (funded by the OE with ref. PTDC/EEI-COM/29271/2017 and by Programa Operacional Regional de Lisboa and FEDER with ref. Lisboa-01-0145-FEDER-029271).

1.5 Thesis Outline

The rest of the document is organized as follows:

- Chapter 2 presents and discusses related work, categorizing several algorithms (namely publish-subscribe and causal multicast) under different properties related to this work;
- Chapter 3 describes the system model, assumptions and defines LoCaMu in detail;
- Chapter 4 addresses LoCaMu's implementation as well as some of the related work's algorithms;
- Chapter 5 presents the results of the evaluation performed on each algorithm and presents some discussion about the obtained results;
- Chapter 6 concludes this dissertation, summarizing the main points, presenting future work and final remarks.

2 Related Work

Contents

2.1 Properties 7
2.2 Overlay Networks 10
2.3 Publish-Subscribe Systems 13
2.4 Reliable Causal Multicast 21
2.5 Comparison 23

This chapter begins with Section 2.1 by introducing the properties that LoCaMu should offer and which mechanisms it will need. Then Sections 2.2, 2.3 and 2.4 present works that use some of the blocks of LoCaMu.

2.1 Properties

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.

2.1.1 Properties

Reliable Causal Multicast is defined by the following properties.

- *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.
- *Fault Tolerance*. The system should be able to remain operational in case some processes fail.
- *Causal Order*. Messages may be received in any order, but are delivered in causal order, as defined by Lamport [15].

In order to build a scalable algorithm, the following properties should be offered:

- *Fault Tolerance Without Full Replication*. The system should support fault tolerance without requiring each process to be replicated.
- *Message Pipelining*. Each process should be allowed to publish a new message without waiting for the previous message to be acknowledged by every target.
- *Localized*. Each process should only maintain state regarding a segment of processes of the whole system.

2.1.2 Building Blocks

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 2.1. These mechanisms are the following:

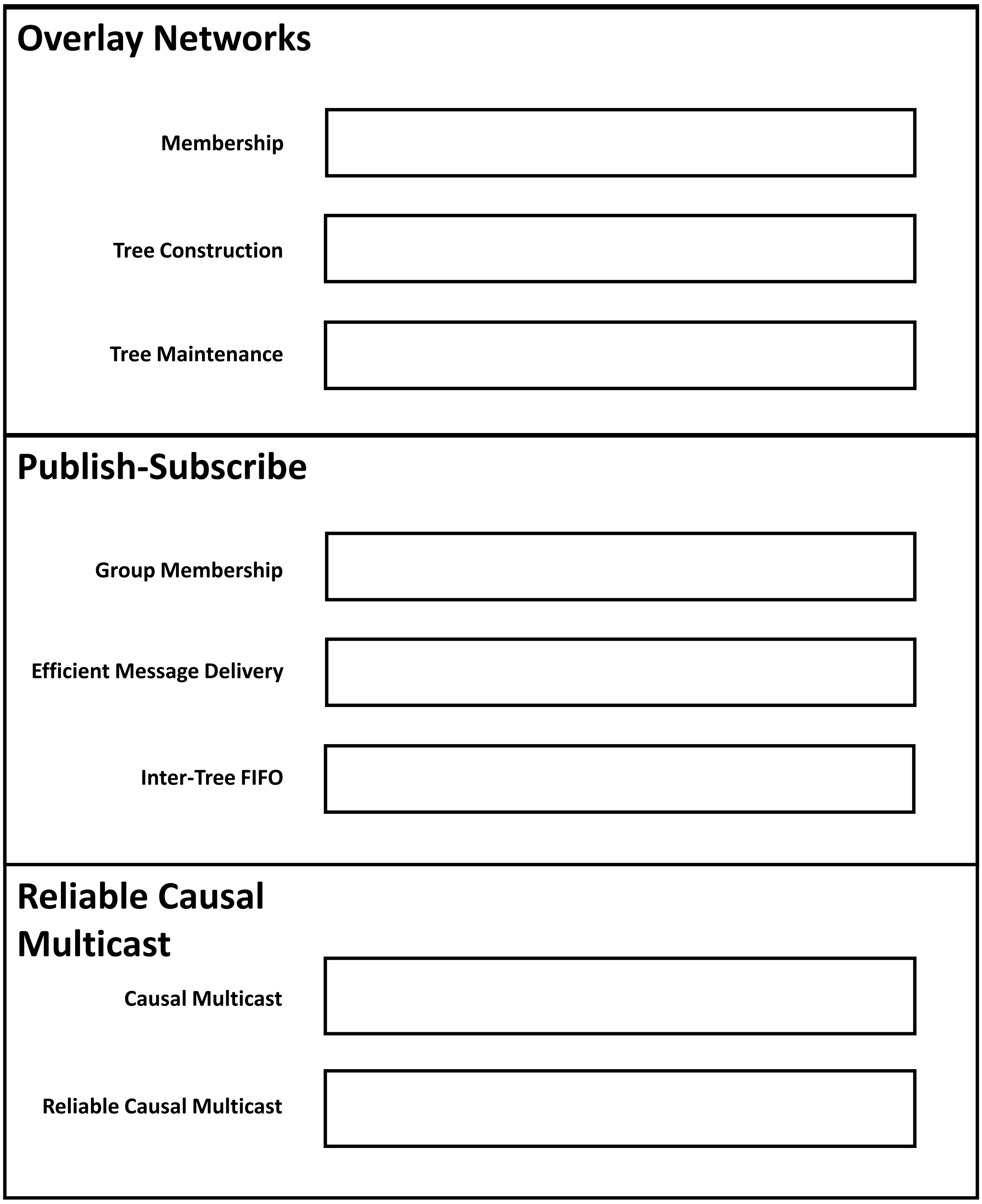


Figure 2.1: Concerns of Reliable Causal Multicast

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 them. 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 [16]), from structured overlays, such as Pastry [17].

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 metadata 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 systems that have influenced LoCaMu's design. Each system is described and its concerns are identified.

2.2 Overlay Networks

We assume that every node in the system is able to exchange messages with any other node using Internet Protocol (IP) protocols, such as User Datagram Protocol (UDP) or Transmission Control Protocol (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.

2.2.1 HyParView

Objective HyParView [16] 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 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 in 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.

2.2.2 Plumtree

Objective Plumtree [18] 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.

2.2.3 Thicket

Objective Thicket [19] 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.

2.3 Publish-Subscribe Systems

Publish-subscribe [20] (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.

2.3.1 Scribe

Objective Scribe [11] 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 [17], 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 builds 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.

2.3.2 Gryphon

Objective Gryphon is a scalable content-based Pub/Sub system that has been developed by IBM. Different aspects of the system are described in different papers, including [4, 21–23]. 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 [21], 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*

2.3.3 Delta-Neighbourhood

Objective Kazemzadeh and Jacobsen [5] 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 Delta-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 f 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 Delta-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 Delta-Neighbourhood algorithm tolerates f 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 f fault tolerance by having each node in the overlay maintain "knowledge" of $f + 1$ nodes in the neighbourhood, with knowledge implying each node knows the Subscription Routing Table (SRT)s (which are 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 $f + 1$ 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 f in a row. A partition barrier occurs when there are f 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 $2f + 1$ entries. 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 f jumps then the message is ignored, because if the message does not bypass more than f nodes in a given subpath of size $2f + 1$, 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 Delta-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 $f + 1$. However, when this happens, the broker is considered a permanent part of the system.

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

2.3.4 VCube-PS

Objective VCube-PS [24] 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 [6]. 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 [1], 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 *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.

2.4 Reliable Causal Multicast

In this section we go through two important works related to Reliable Causal Multicast.

2.4.1 CBCAST

System Model CBCAST's system [1] 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 [15].

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 Clock (VC)s 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.

2.4.2 Causal Barriers

System Model The Causal Barrier algorithm [6] 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 [6]): 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 x^{th} 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 y^{th} 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].

2.5 Comparison

LoCaMu handles **Efficient Message Delivery**, **Inter-Tree FIFO** and **Reliable Causal Multicast**. The construction blocks below **Efficient Message Delivery** are assumed to be accomplished using any of the algorithms present in the literature.

These systems are then compared, to illustrate how they can be used as building blocks below LoCaMu. Figure 2.2 shows where each algorithm fits into the different building blocks and Table 2.1 compares the properties of each algorithm.

Overlay Networks There are two types of systems. The first is a structured P2P system, (Scribe uses this type, running on top of Pastry) and the second is unstructured P2P (both Plumtree and Thicket use this type, both running on top of 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*.

Publish-Subscribe There are some differences regarding how each system handles group membership. In VCube-PS, all nodes are aware of every subscription of every node, while in Scribe and

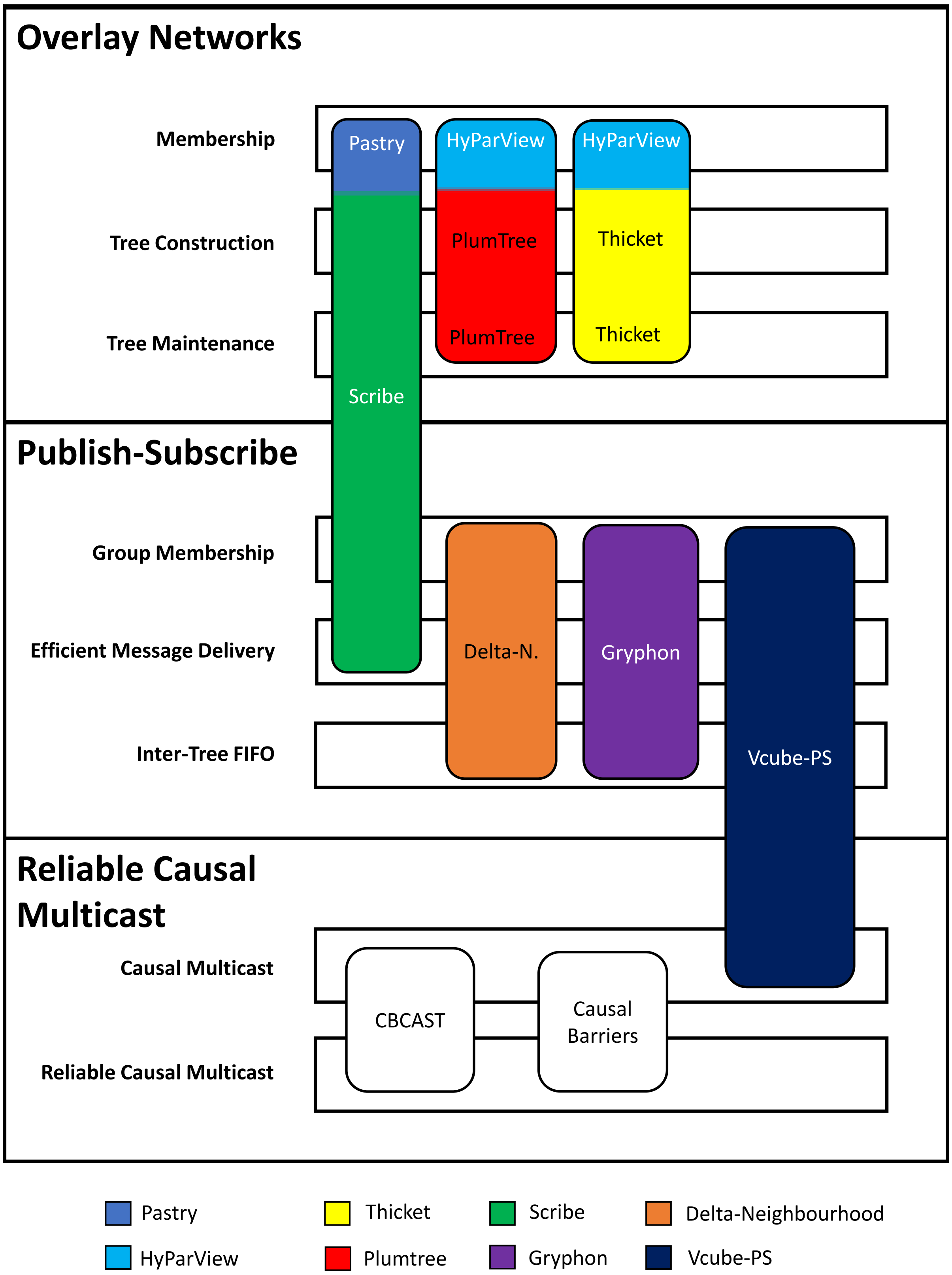


Figure 2.2: Concerns of Reliable Causal Multicast filled with the analyzed systems

Delta-Neighbourhood only the nodes on the 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 Delta-Neighbourhood and Gryphon deal with, namely message gaps, are dealt in two different ways, with Delta-Neighbourhood using *partition ids* and Gryphon using the concept of *virtual time* to detect when messages can start being delivered.

Scribe and VCube are topic-based pub-subs while Gryphon and Delta-Neighbourhood are content-based. Messages in Scribe need to go from the publisher to a special node (rendez-vous node) 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, Gryphon uses a spanning tree for each *pubend* and Delta-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. Delta-Neighbourhood and VCube-PS wait for each message to receive system-wide acknowledges.

Reliable Causal Multicast VCube-PS offers causal order by using Causal Barriers, however it does not offer causal order between different topics. Also, it does 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 a 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 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. 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.

Summary

This chapter introduced the properties of Reliable Causal Multicast and the properties required to create a scalable algorithm. There are three properties that should be offered when creating a scalable Reliable

Table 2.1: LoCaMu vs Related Work. The fault tolerance approaches are Virtual Reliable Nodes (VRN), which consists of replicating each node; On-line overlay reconfiguration (OR), which consists of reconfiguring the tree whenever a node crashes and Redundant Paths (RP), which allows each node to have a limited number of backup connections to other, nearby nodes.

Algorithm	Fault Tolerance Approach	Avoids Replicas	Parallel	Causality	Local
HyParView [16]	N/A	N/A	✓	✗	✗
Plumtree [18]	OR	✓	✓	✗	✗
Thicket [19]	OR	✓	✓	✗	✗
Scribe [11]	N/A	N/A	✓	✗	✓
Gryphon [4]	VRN	✗	✓	✗	✗
Delta Neighbourhood [5]	RP	✓	✗	✗	✓
VCube-PS	N/A	N/A	✓	✓	✗
CBCAST [1]	RP	✓	✓	✓	✗
Causal Barriers [6]	RP	✓	✓	✓	✗
LoCaMu	RP	✓	✓	✓	✓

Causal Multicast algorithm, namely *locality*, which requires each node to maintain state regarding a limited number of other nodes in the system; *Parallel messages*, which allows a node to publish messages continuously without waiting for each message to be acknowledged by every destination node in the system and *Fault Tolerance without replicas* which means that fault tolerance should be achieved without replicating each node, as this makes the algorithm too expensive to install in large systems. Given these requirements, several algorithms were studied, which addressed several mechanisms from connecting the nodes to create an overlay to offering FIFO and causality. Using some of the ideas present in these studied algorithms, a novel algorithm was created, which is presented in the next chapter.

3
Local Causal Multicast

Contents

3.1 Goals 28
3.2 System Model 28
3.3 An example 31
3.4 Message Identifiers 32
3.5 Duplicate Detection and Merging Identifiers 34
3.6 Safe Paths 34
3.7 Causal Past and Causal Order 36
3.8 Detailed Algorithm 36
3.9 Additional Features of the Algorithm 40
3.10 Correctness 42

This chapter introduces LoCaMu, the first Localized Reliable Causal Multicast algorithm. Section 3.1 starts the chapter by explaining the goals for the algorithm. Section 3.2 presents the system assumptions (node topology organization, addressing model and fault model). Section 3.3 illustrates the two types of scenarios that LoCaMu seeks to solve. Section 3.4 explains the main mechanism of LoCaMu which is used to ensure causal order and fault tolerance. Section 3.5 describes how duplicate messages are detected and handled. Section 3.6 points out an invariant used by LoCaMu. Section 3.7 addresses how causal order is offered. Section 3.8 explains in detail the algorithm. Section 3.9 provides some extra features that are not necessary for the correctness of the algorithm but improve performance. Finally, Section 3.10 proves that LoCaMu is correct.

3.1 Goals

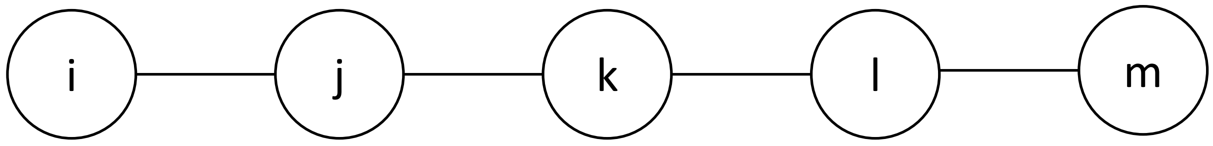
There are several causal multicast algorithms as seen in the Causal Multicast section of the Related Work chapter (2.4), however none of these offer all the properties present in 2.1 and all require the use of metadata that scales directly with the size of the system. When designing LoCaMu, there were several objectives in mind, such as creating the first algorithm capable of upholding simultaneously all the mentioned properties in Section 2.1 while being as scalable as possible. To make it as scalable as possible, it was necessary for the maximum amount of metadata a message carries to not depend on the size of the system. Using the *locality* property, this is automatically ensured, however, the remaining properties still need to be guaranteed. How these properties are kept will be explained in the rest of the chapter.

3.2 System Model

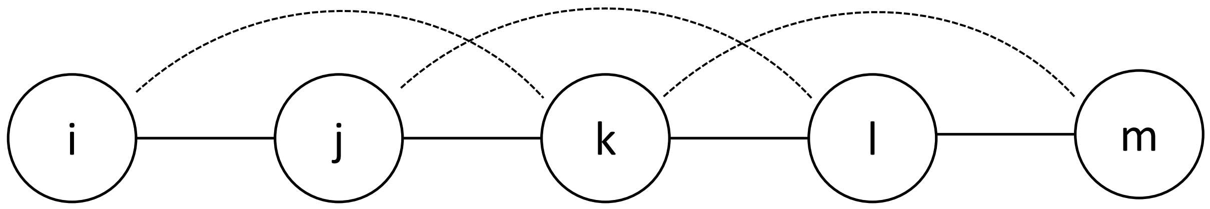
This section introduces the relevant information regarding the assumptions of the system model such as how nodes are organized, how they communicate and which faults are considered.

3.2.1 Base Graph and Extended Graph

It is assumed that nodes are organized in an overlay network that can be modelled by an undirected graph $G = (v, e)$, where vertices v represent the system nodes and edges e the communication channels between these nodes. This graph, which is referred to as the base graph, is acyclic and connected. Let v_i be a vertex of the graph. $\mathcal{V}(G, k, v_i)$ represents the set of neighbours of the vertex v_i which are at a distance no longer than k in the graph (that is, if $v_j \in \mathcal{V}(G, k, v_i)$, then the shortest path in the base graph between v_i and v_j has at most k edges). The vertices in $\mathcal{V}(G, 1, v_i)$ are denoted as direct neighbours of v_i (in the base graph). G has no redundant edges and the failure of any node partitions the graph.



(a) Base graph



(b) Extended graph

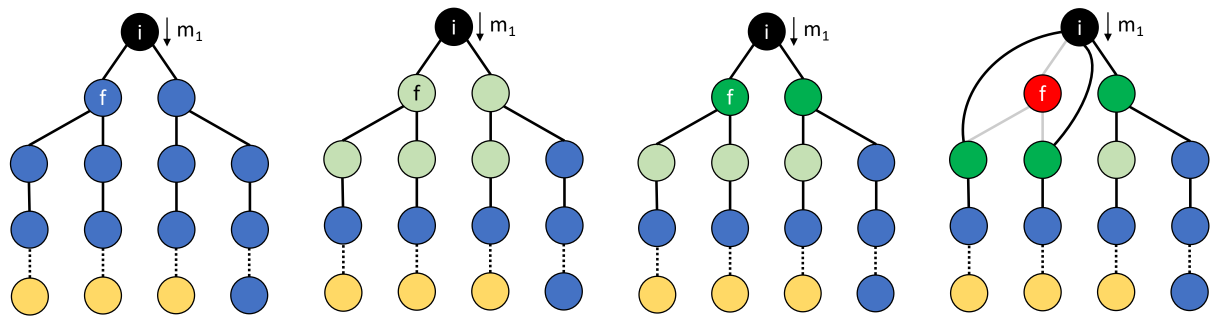
Figure 3.1: Base graph and the corresponding extended graph ($f = 1$)

In order to tolerate f failures in a given neighbourhood, the base graph G is augmented with additional edges, creating an extended graph G^f. The additional edges connect each vertex v_i with all vertices in $\mathcal{V}(G, f+1, v_i)$, that is, for each $v_j \in \mathcal{V}(G, f+1, v_i)$, an edge is added to the extended graph, connecting v_i and v_j. The additional edges in the extended graph create redundant paths that may be used if a path in the base graph becomes unavailable. Figures 3.1a and 3.1b illustrate a base graph and the corresponding extended graph for $f = 1$. When the value of f increases, additional redundant paths are created, but the previous redundant paths from the lower values of f are still used. In fault-free runs, the base graph is used to propagate the messages in a order that respects causality. If a node (such as f in Figure 3.2) fails, then the additional edges are used to propagate the message. Even when redundant paths are used, messages are routed along the paths defined by the base acyclic graph, such that cycles are avoided. When these redundant paths are used, messages may be duplicated or re-ordered. As it will be seen, LoCaMu maintains the required metadata to ensure that messages are reliably delivered in order even in runs where faults occur.

3.2.2 Addressing Model

Nodes are assumed to be organized into groups that are used to define the destination address of a message; each message being addressed to a single group. The communication uses a publish-subscribe model, where information about group members is propagated in the extended graph through *subscription* messages. This allows each node to maintain a routing table that, for each group, indicates which neighbours participate in the forwarding of the messages. The size of this table depends on the number of groups and not on the number of nodes in the system.

Given a message m, which has a recipient group $g(m)$, a given node i can query its routing table to identify the sub-set of its neighbours in the extended graph that are involved in routing m. This sub-set is



(a) Base graph (blue) and m_1 recipients (yellow)

(b) Targets for m_1, $\mathcal{T}(m_1, i)$ (light green)

(c) Active connections for m_1 (dark green)

(d) Active connections for m_1 (dark green) after a node disconnects (red node)

Figure 3.2: Message targets and active connections.

called the *targets* of m in i, denoted $\mathcal{T}(m, i)$. This is illustrated in Figure 3.2, where Figure 3.2a depicts the base graph and the group members for message m_1 that needs to be forwarded by node i and Figure 3.2b shows the set of target nodes $\mathcal{T}(m_1, i)$ for that message for $f = 1$.

3.2.3 Fault Model

Nodes are assumed to be capable of failing and subsequently recover, and their state is kept in persistent memory. The system is asynchronous and a faulty node may remain unavailable for an arbitrary amount of time. Finally, it is assumed that at any instant of time, in any neighbourhood of size $2f + 1$ in the base graph, there are at most f nodes unavailable. A node that never fails is denoted as correct.

Each node of the system is assumed to have access to an eventually perfect failure-detector [25,26], that reports if its neighbours are available or unavailable. This detector ensures that if a neighbour becomes unavailable, the node is eventually notified of this fact (and can avoid sending messages in vain while the faulty node remains unavailable). Similarly, when the neighbour recovers, the node is also notified (so that it can use that neighbour again).

Using the output of the failure detector, every node i maintains a set of *activeConnections* as follows. Every time the set of faulty nodes in its $(f + 1)$-neighbourhood changes, i recomputes the shortest path to every other node in $\mathcal{V}(G, f + 1, v_i)$. Then, the nodes that are both the first non-failed node and hop on any of these paths are added to the *activeConnections* set. When propagating a message in the network, each node only forwards messages to the nodes maintained in this set. This limits the amount of redundant messages that circulate in the network. Figure 3.2c shows the set of active connections at the black node when its direct neighbours in the base graph are active and Figure 3.2d shows how this set is updated when one of these direct neighbours (in this case, node f) becomes unavailable.

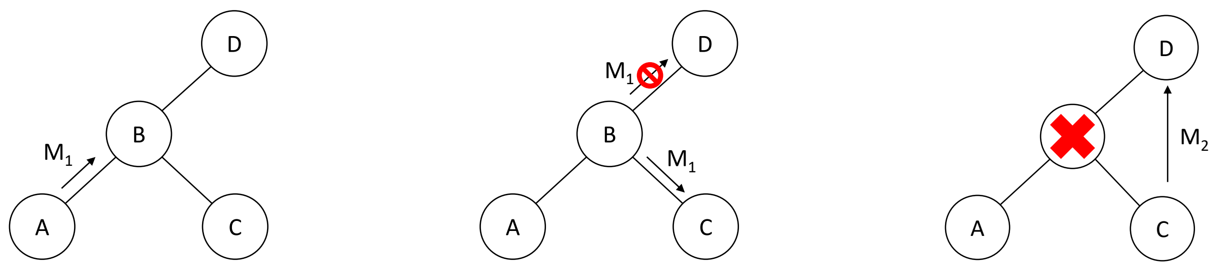
LoCaMu is fault tolerant in the sense that a message m sent to a set of vertices V is delivered to all correct nodes $v_j \in V$ (that is, all recipients who are not or that will not fail), regardless of the failure of other nodes or the eventual recovery of nodes that failed. LoCaMu also ensures that nodes that are temporarily unavailable will end up receiving all messages; of course, this only happens when they recover.

3.2.4 Localized Information and Safe Neighbourhood

LoCaMu is a localized algorithm, where each node is only required to maintain metadata regarding messages sent or received by other nodes into some *safety neighbourhood*. For this reason, when a message is sent and tagged with metadata, it only needs to be tagged with information regarding nodes that belong to the safety neighbourhood of the recipient. In the case of LoCaMu, the safety neighbourhood includes all nodes that are at most $2f + 1$ hops away in the base graph. More precisely, the safety neighbourhood of node i, denoted $\mathcal{V}^S(i)$ is $\mathcal{V}^S(i) = \mathcal{V}(G, 2f + 1, i)$. The next subsections provide an intuitive rationale for the size of the safety neighbourhood in LoCaMu and how it relates to the possible paths that can be followed by messages.

3.3 An example

To help the reader have a better understanding of the problem given the system model, two scenarios will be illustrated with the objective of explaining how causal order can be broken or duplicate messages be delivered if there is no algorithm to prevent these issues.



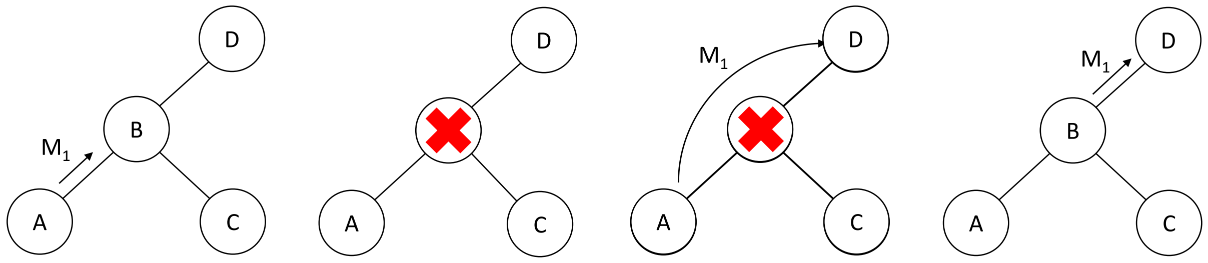
(a) Node A forwards m_1 whose targets are nodes B, C and D

(b) Node B forwards m_1 to C successfully and D unsuccessfully

(c) Node B is down, therefore C forwards m_2, which causally depends on m_1, directly to D.

Figure 3.3: Scenario where causality is broken if node D delivers m_2 before m_1.

The first scenario (Figure 3.3) depicts a situation where causality is broken. It starts with node A publishing m_1, which nodes B, C and D are interested in. When A calculates the targets, it only forwards the message to B, as this node is the first common node on the paths to the other targets and the node is alive (as explained in Section 3.2.3). B then receives the message and promptly delivers and forwards successfully to C but has a problem when sending to D and crashes, resulting in D missing the message. Finally, node C delivers m_1 and publishes a new message m_2, whose target is D. As node B is down, C forwards m_2 directly to D, which will deliver it as it does not know that there is a missing message. When D eventually receives m_1, it will break causal order.



(a) Node A forwards m_1 whose targets are nodes B and D **(b)** Node B receives and delivers m_1 but crashes before forwarding it to D **(c)** Node A retransmits m_1 to node D, who then delivers the message **(d)** Node B recovers and forwards m_1 to D, who then delivers the duplicate message

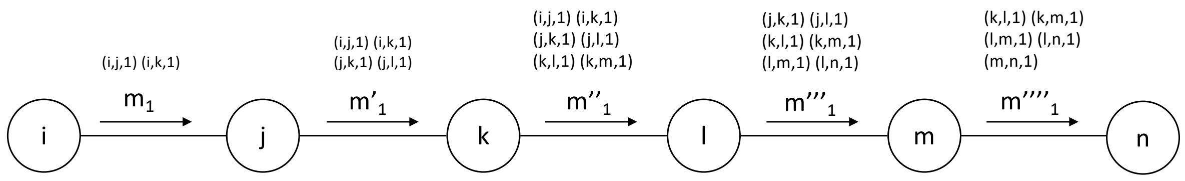
Figure 3.4: Scenario where a duplicate message is delivered.

The second scenario (Figure 3.4) represents an eventual recovery of messages where a duplicate message is received and delivered. Node A publishes a message m_1 whose targets are B and D and promptly forwards it to B, who then crashes before forwarding the message to D. Using a simple mechanism to detect if there are any missing messages (such as a message counter per source to detect if there are any missing messages, meaning D is counting how many messages it has received from A), D can detect if there is a missing message from A and request it, which happens in Figure 3.4c. Then, if C recovers and forwards m_1 to D, D will have received m_1 twice but be unable to detect that the message is a duplicate.

The reader may notice that there are several possible solutions to both of these problems, but as stated in the Goals section, it is important to solve these situations in the most efficient manner. The main mechanism for solving both these problems with now be presented.

3.4 Message Identifiers

Non-localized algorithms typically assign an unique identifier to a message when the message is generated. This unique identifier is then preserved as the message is propagated in the network. The most common way to generate such unique identifiers is to use a tuple that includes the unique identifier of the source node and a sequence number generated at the source. A fundamental drawback of this solution is that, to perform tasks such as detecting duplicates or enforcing order properties, nodes in the system need to keep track of sequence numbers generated by every other node in the system. These approaches are, therefore, not localized (and not scalable).



(i,j,1) (i,k,1)
m_1
(i,j,1) (i,k,1) (j,k,1) (j,l,1)
m'_1
(i,j,1) (i,k,1) (j,k,1) (j,l,1) (k,l,1) (k,m,1)
m''_1
(j,k,1) (j,l,1) (k,l,1) (k,m,1) (l,m,1) (l,n,1)
m'''_1
(k,l,1) (k,m,1) (l,m,1) (l,n,1) (m,n,1)
m''''_1
i
j
k
l
m
n

Figure 3.5: Creation and propagation of identifiers ($f = 1$)

LoCaMu is localized as nodes are only required to keep track of identifiers generated by nodes in their safe neighbourhood. Thus, if a message is generated by a remote node, outside the safe neighbourhood, the identifier generated by the source cannot be used; it needs to be replaced by some identifier that is meaningful locally. In consequence, in LoCaMu messages do not have a single identifier that is preserved across the system. Instead, messages are assigned multiple identifiers as they are forwarded in the network, where each of these identifiers is only valid in a given neighbourhood. Still, LoCaMu is able to keep track of how these identifiers are related, such that it is able to eliminate duplicates and to deliver messages in the right order.

The way message identifiers are created, propagated, and replaced, is key to the operation of LoCaMu. To better understand the process of propagating message identifiers, an example is used. Figure 3.5 shows a network that consists of a line, and illustrates the propagation of a message that is propagated from node i to node n. In this example the number of tolerated faults is $f = 1$ and, therefore, the set of targets for the message are the next two nodes in the line (i.e, $\mathcal{T}(m, i) = \{j, k\}$, $\mathcal{T}(m, j) = \{k, l\}$, etc). Note that in this case, the safety neighbourhood of node m no longer includes node i (given that the distance from i to l is more than $2f + 1$), thus m will not process metadata produced by node i. This results in each message only having identifiers from at most $2f + 1$ different nodes.

Each node keeps a separate sequence number for each target and the tuples *(source, target, sequence number)* are used as messages identifiers. Note that, in the example, when message m is generated, it is assigned two identifiers, namely $(i, j, 1)$ and $(i, k, 1)$, one identifier for each link that can be used to propagate the message. To avoid redundant messages in the network, the message is not sent immediately by both links. In this example, the message is just propagated in the base graph (and edges from the extended graph are just used if faults occur). Then, when the message is propagated by node j it gets assigned two new identifiers, $(j, k, 1)$ and $(j, l, 1)$, and now carries a total of 4 different identifiers. This process is repeated every time a message is forwarded in the network. However, at each step, each sender only forwards identifiers that belong to the safe neighbourhood of the recipient. Thus, when node l forwards message m to node m, it no longer includes the identifiers generated by node i (similarly, when m forwards the message it no longer includes the identifiers generated by node j).

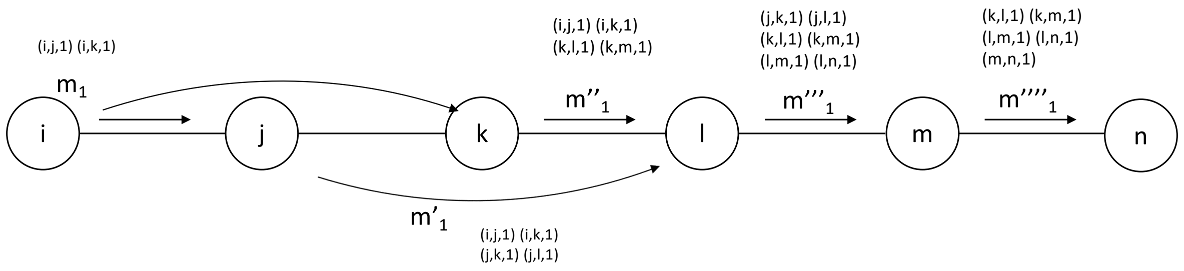


Figure 3.6: Duplicate detection and merging of identifiers ($f = 1$)

3.5 Duplicate Detection and Merging Identifiers

As noted above, LoCaMu does not require a perfect failure detector. Thus, at any given time, different nodes may make a different assessment on the correctness of a given neighbour. Also, nodes may falsely suspect that a neighbour is down and trigger the propagation of the message via the redundant edges defined by the extended graph. This may cause different copies of the message to be propagated by different paths, as illustrated in Figure 3.6 where two copies of message m_1 are received by node l via different paths, namely, the copy m_1' is received via the path that uses vertices $\{i, j, l\}$ and copy m_1'' via $\{i, k, l\}$. In this case, m_1' does not carry the identifiers added by node k and m_1'' does not carry the identifiers added by node j. Note that, in this example, node j can still detect that m_1' and m_1'' are duplicates of the same original message, given that there is at least one common identifier carried by both copies (more precisely, identifiers $(i, j, 1)$ and $(i, k, 1)$ are common to both copies). Also, when node l propagates the message by sending m_1''', it can tag the copy with all the identifiers known to it (the set of known identifiers results from "merging" the set of identifiers carried by m_1' and m_1'').

This example also shows why node l needs to maintain metadata produced by node i (and thus, i needs to be in the safe neighbourhood of l). If this was not the case, in this run, where a single node is suspected as being failed, l would not be able to detect that m_1' and m_1'' are duplicates of the same original message.

3.6 Safe Paths

The correctness of LoCaMu is rooted on the *Safe Paths Invariant*, which is defined as follows:

Invariant 1. *Any path of $2f + 1$ nodes is considered safe if there are at most f faulty nodes. Otherwise it is unsafe.*

This means that if two nodes (say i and l in the example of Figure 3.7) are at $2f + 1$ hops away in the base graph, it should be possible to propagate a message from i to l in a path that contains at least $f + 1$ neighbours. Using the example of Figure 3.6, a copy of m_1 arrives to l via a path that uses $\{i, k, l\}$

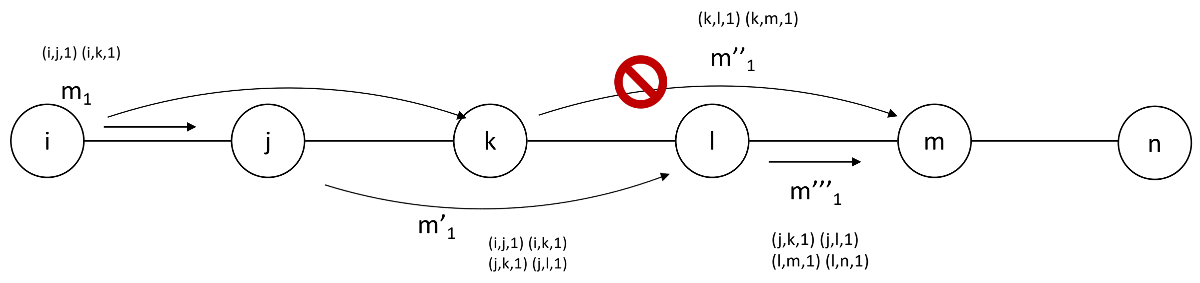
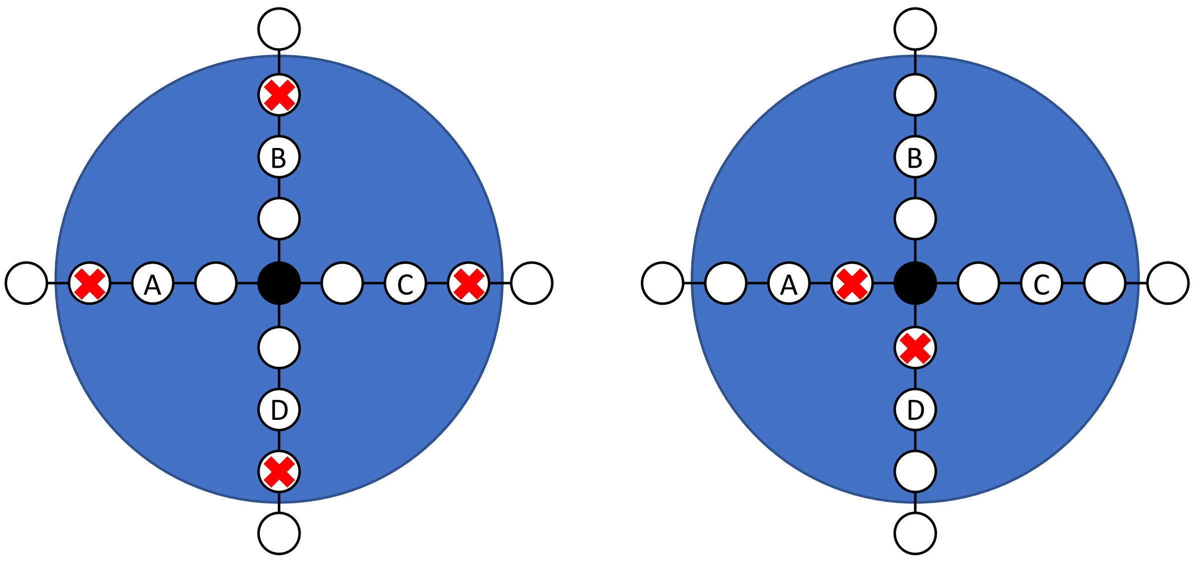

(i,j,1) (i,k,1)
m_1
(k,l,1) (k,m,1)
m''_1
i
j
k
l
m
n
m'_1
(i,j,1) (i,k,1) (j,k,1) (j,l,1)
m'''_1
(j,k,1) (j,l,1) (l,m,1) (l,n,1)

Figure 3.7: Invalid paths ($f = 1$)



(a) Safe neighbourhood where every path is a safe path

(b) Safe neighbourhood where there is one path that is not safe (between nodes A and D)

Figure 3.8: Examples of a safe neighbourhood (blue circle, $f = 1$)

(skipping j) and another copy arrives to l via a path that uses $\{i, j, l\}$ (skipping k). These paths are said to be *safe*.

Using an example, it is shown that safe paths are not only possible but required to ensure several properties of the algorithm, such as performing duplicate detection. Consider the example of Figure 3.7. Node i suspects node j is down and sends m_1 directly to k. Assume that node k suspects node l and decides to send a copy of m_1, m_1'' directly to node m. The reader will notice that for such path to be used, there are two nodes from the safety neighbourhood of m that have been suspected, thus one of these suspicions is false (recall that the has $f = 1$). The example also shows that, if an unsafe path is used, node m may not be able to detect duplicates (in this example m_1'' and m_1''' do not share identifiers). Note also that node l can still send m_1'' to l as that path is safe. In the forwarding algorithm of LoCaMu, it is ensured that only safe paths are used.

A safe neighbourhood can have several safe paths and several unsafe paths. There is no formula for calculating how many faults a safe neighbourhood can tolerate, as the number of neighbours a node can have is dependant on the overlay. Therefore, to give a visual representation of faults and safe paths inside a safe neighbourhood, Figure 3.8 is used. In Figure 3.8a, there are 4 faults, but every path inside the safe neighbourhood is safe. In Figure 3.8b there are only 2 faults, but there is one unsafe path, which is between nodes A and D, as this path of 3 nodes contains 2 faults, effectively breaking invariant 1 (every other path is safe, for example between nodes A and B).

3.7 Causal Past and Causal Order

To enforce causal order every node keeps a log (simply called the node's *past*) of the identifiers of all the messages it has received and processed in the past. The algorithm enforces that messages transmitted via a given edge are delivered in FIFO order. Thus, in practice, because causal order is transitive, only the most recent identifier generated by any node needs to be preserved in the node's past. Also, nodes are only required to keep in the past information from nodes in their safe neighbourhood. When messages are sent they are tagged with the casual past of the sender. By comparing the causal past of a message with its own causal past, a recipient can check if the message can be processed without violating causal order. If some message m cannot be immediately processed when it is received because some other messages in its causal past are missing, the message is stored in a buffer, until it can be processed.

3.8 Detailed Algorithm

3.8.1 Overview

Messages carry two header fields: a set of identifiers (as described above) and a causal past. When a message is received it is not processed until two conditions are met: i) the message is being processed in FIFO order with regard to other messages from the same sender (message identifiers are used to make this check) and ii) the message is being processed in causal order (the causal past of the message is used to make this check). When a message m becomes ready to be processed one checks if the message is being received for the first time or if another copy of the same message has been previously received via another path (again, message identifiers allow to detect duplicates, as illustrated before). If the message is a copy, its identifiers are merged with the identifiers of the previous copy. After a message is processed, if the message needs to be forwarded (i.e, if the routing table indicates that there are recipients downstream), the message is scheduled for retransmission. Messages are

Algorithm 1 LoCaMu

```
▷ Retransmits a message to the relevant nodes
function (RE)TRANSMIT($m$) at node $i$
    for $k \in \mathcal{T}(m, i) \cap$ activeConnections do
        if PATHISSAFE($m$, $k$) then
            SEND($m$, $k$)
▷ Forwards a message to the interested nodes
function FORWARD($m$) at node $i$
    $D_m \leftarrow P_i$
    for $k \in \mathcal{T}(m, i)$ do
        $I_m[i][k] \leftarrow D_m[i][k] + 1$
        $D_m[i][k] \leftarrow D_m[i][k] + 1$
    (RE)TRANSMIT($m$)
    sent $\leftarrow$ sent $\cup \{m\}$ ▷ For future retransmissions
▷ Updates the Past and Received matrices of a node using message $m$
function UPDATENODESTATE($m$) at node $i$
    for $k, l \in I_m \cap R_i$ do
        $R_i[k][l] \leftarrow R_i[k][l] \cup I_m[k][l]$
    for $k, l \in \mathcal{V}^S(i)$ do
        $P_i[k][l] \leftarrow$ MAX$(P_i[k][l], D_m[k][l], I_m[k][l])$
▷ Updates any missing identifiers in a sent message
function MERGEIDENTIFIERS($m$) at node $i$
    if $\exists m' \in$ sent $: \exists k, l : I_m[k][l] = I_{m'}[k][l]$ then
        for $k, l \in \mathcal{V}^S(i)$ do
            $I_{m'}[k][l] \leftarrow$ MAX$(I'_m[k][l], I_m[k][l])$
▷ Checks if any buffered messages can be processed
function UNBUFFER at node $i$
    for $m \in$ buffer do
        if $\forall k : D_m[k][j] \leq P_i[k][j]$ then
            buffer $\leftarrow$ buffer $\setminus \{m\}$
            RECEIVE($m$)
▷ Receives a message from FORWARD($m$)
function RECEIVE($m$) at node $i$ sent by node $j$
    if $\exists k : D_m[k][i] > P_i[k][i] \vee \exists l : l < I_m[j][i] \wedge l \notin R_i[j][i])$ then
        ▷ Message out of order
        buffer $\leftarrow$buffer $\cup \{m\}$
    else
        UPDATENODESTATE($m$)
        if $\exists k, l : I_m[k][l] \in R_i[k][l]$ then
            ▷ Duplicate copy
            MERGEIDENTIFIERS($m$)
            TRYSENDUNSENTMESSAGES()
        else
            ▷ First copy
            if $i$ belongs to $g(m)$ then DELIVER($m$)
            FORWARD($m$)
        UNBUFFER()
▷ A node generates a message
function PUBLISH($m$)
    FORWARD($m$)
```

only retransmitted via safe paths. In the next paragraphs a more detailed description of the algorithm is provided. Pseudo-code is provided in Algorithm 1.

3.8.2 Node Data Structures

Each node maintains three data structures, listed below:

- The first is a matrix of sequence numbers called *Past*, denoted by P_i, with one entry for each pair of nodes in $\mathcal{V}^S(i)$. The matrix size is given by $\mathcal{V}^{S^2}(i)$. P_i captures the causal past of the i node. Consider that the position $P_i[j][k] = s$ $(s \neq 0)$. This means that the state of i depends causally on a message sent from node j to node k with the sequence number s.

- The second is a matrix of *received* identifiers, R_I, whose size is $\mathcal{V}^{S^2}(i)$. Each entry $R_i[j][k]$ is an ordered set that contains the identifiers of all messages sent by node j to node k that have already been processed by node i. The objective of R_i is to detect duplicate messages, since in faulty runs, the same message can be received via different paths. While every entry is a set, each set is regularly garbage-collected, as explained in Section 3.9.2

- The third is the *sent* set that contains all buffered messages.

3.8.3 Message Control Fields

Each message contains two control fields, listed below:

- The first, called *identifiers* (denoted as I_m), is an array of up to $2f+1$ vectors, containing the known sequence numbers that were assigned to m by each node in any given path. The size of this structure is highly variable, as it depends on the targets of a given message.

- The second, called *dependencies* (denoted by D_m) carries the causal past of m known by the forwarder; D_m is of size $|\mathcal{V}^S(i)|^2$.

3.8.4 Message Reception

When a node i receives m, which was sent by j, i performs the steps presented in the RECEIVE function in Algorithm 1. In short, the node verifies if all causal dependencies have been satisfied (that is, the messages in the past have already been delivered). If that is the case, then i updates its P_i and R_i and then, if the message is not a duplicate, delivers m if i is interested in it and *forwards* m. If it is a duplicate and the original copy was forwarded then the original copy's identifiers are updated with the duplicate's identifiers. Note that, after the identifiers have been merged, some paths for the original message may become safe, which means that the original message may be sent to a given path if it originally was not sent due to the lack of identifiers.

3.8.5 Message Forwarding

When a node i wants to forward a message m for the first time, the following manipulations are made to its own state and to the metadata of message m: First, i puts $D_m = P_i$. Then i queries its routing table to get the targets of m, $\mathcal{T}(m,i)$. Then, the values of $P_i[i][j]\ \forall j \in \mathcal{T}(m,i)$ are incremented by one and a new vector with the changed entries is added to I_m. m is then sent to the targets present on *active-Connections* (i.e., to $\mathcal{T}(m,i) \cap$ *activeConnections*), provided the resulting path is safe, as described in Section 3.6

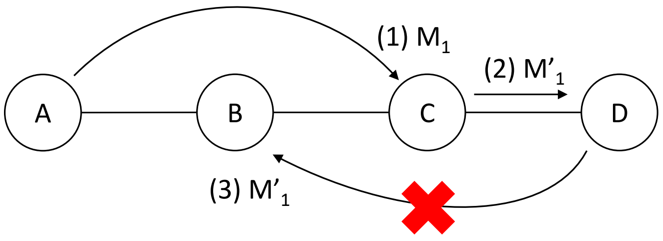


Figure 3.9: Forbidden message retransmission

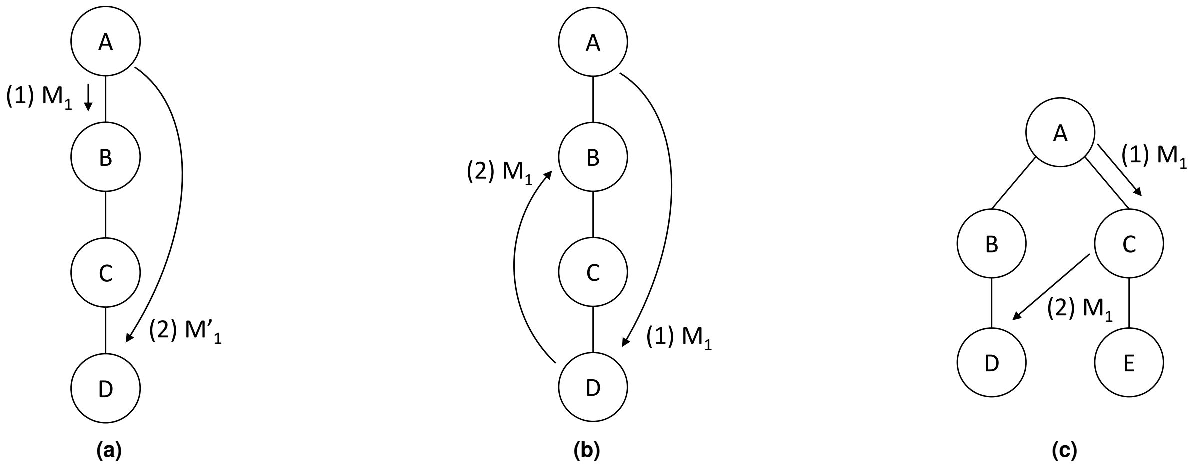


Figure 3.10: Different retransmission scenarios ($f = 2$)

3.8.6 Message Retransmission

When a node j notices that there are missing messages from a node i, it sends a *Missing Messages Request* accompanied by the number of the last received message from said node (essentially, a negative acknowledge). Afterwards, i sends the missing messages including those that caused any of these messages.

A message retransmission is allowed depending on whether the message is safe to send or the receiver is present on the last vector of I_m. This is because if the receiver is indeed present, then the message can be seen as being sent directly from the original forwarder of the message (who put its targets on the last vector of I_m) to the receiver, provided this is safe.

Figure 3.9 shows an instance where the retransmission of a message is forbidden. This is because node D is retransmitting a message whose I_m's last vector only contains node D. Therefore, even though node B needs to deliver the data content of m_1, it cannot receive m_1', as it will not be able to process the metadata correctly. Instead, it needs to receive the message from node A.

Figure 3.10 depicts some examples of message retransmissions.

- Figure 3.10a shows a scenario where node A forwards m_1 towards the expected fault-free path (to node B), eventually being required to retransmit the message directly to D as the communication links of A to B and A to C are broken. Here, A sends the original message directly to D, provided it is safe.

- Figure 3.10b begins with the communication links A to B and A to C broken, therefore A needs to forward m_1 directly to D. Then, if D needs to forward a message directly to B, this message will be causally dependent on m_1, as B is also a target m_1. D can retransmit m_1 directly to B, as the message still contains the original metadata put by node A.
- Figure 3.10c represents the scenario where a message goes to one path (namely A, C, E) but then needs to be retransmitted to another other path (A, B, D). Here, node C receives m_1 from A and then needs to send it to D. Since the message contains the metadata from A, the retransmission is as if it is node A sending m_1 to node D, therefore it is accepted.

3.9 Additional Features of the Algorithm

This section provides extra features which are not required for the algorithm to work correctly, but are useful for when deploying the system in a large scale system. These features are garbage collection (Section 3.9.1 and Section 3.9.2) and compression of metadata (Section 3.9.3).

3.9.1 Garbage Collecting Messages

Nodes keep a *sent* set with copies of messages that have processed and forwarded previously. This set is required given that, to ensure reliable delivery, messages may need to be retransmitted in the future. A message that has been already received by all neighbours is said to be *stable* and can be garbage collected from the *sent* set. LoCaMu embodies a stability tracking mechanism, that runs in background, where nodes periodically exchange information with their neighbours regarding the messages they have already processed in order to detect which messages are stable.

3.9.2 Garbage Collecting the R_i Matrix

Each entry of the *received* identifiers matrix, R_i, is an ordered set that will grow indefinitely unless unnecessary entries are removed. There are two phases to clear each set. Firstly, all entries from a gapless prefix in the history of received messages are replaced by the last message in the prefix. This means that if $R_i[j][k] = \{0, 1, 2, 3, 4, 5, 6, 8, 9\}$ then it is just stored as $R_i[j][k] = \{6, 8, 9\}$. However, this mechanism by itself will not always work. As shown in Figure 3.11, because of the existence of groups, some sets will never be allowed to purged (Node D may receive a message with $I_m[A][B] = 1$ and

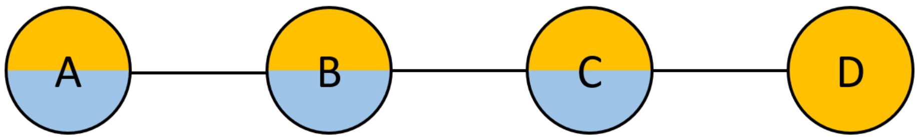


Figure 3.11: The existence of groups creates the need for a special mechanism to garbage collect the R matrix

	A	B	C
A	-	2	3
B	3	-	3
C	3	3	-

A	B
C:2	A:1
D:2	

Figure 3.12: D_m without (left) and with (right) compression. Without compression, the grey fields are not sent. With compression, each every field is sent, with the IDs occupying 4 bytes (letters) and the numbers 4 bytes.

the next message may have $I_m[A][B] = 3$, meaning that node D will never receive a message with $I_m[A][B] = 2$, as the message with this identifier was sent to the other group). To solve this issue, every node sends a matrix equal to its *received* identifiers matrix, but where each set only contains the lowest value, to every active connection in timed intervals. By having the nodes trade their matrices, it is possible to solve the issue by comparing this lowest value and replacing the lowest value of their own sets with the value present on the received matrix, provided it is higher.

3.9.3 Optimizing the Message Headers

If all messages were to transport the entire *Dependencies* field, the metadata overhead would be very taxing on the protocol performance. A common optimization that is used in the literature [1] when sending a message m is to compare D_m with the last sent message's (m') dependencies (D'_m) and only send the different entries. This optimization has two costs: Firstly, each node needs to keep track of the *Dependencies* control field of the last message sent to and received from every other node in its $(f+1)$-neighbourhood and secondly, each entry occupies 8 bytes instead of 4 bytes. This is because if there are no optimizations, every entry of D_m is constant and known, therefore it can be optimized to not need the use of identifiers. However, if these entries are dynamic (meaning the first entry can correspond to the value of a node and in the next message the first entry can correspond to the value of another node), it is required to use identifiers. These identifiers are used as follows: one identifier for the "from" column and then each entry of the column has two values: a node identifier, corresponding to the target and an integer value. Figure 3.12 illustrates this.

Two things should be noted, namely I_m cannot be optimized, as every entry is always necessary and message retransmissions should always be sent without optimizations.

3.10 Correctness

3.10.1 Safety

Let some node i be the recipient of two messages, m_1 and m_2, such that $m_1 \to m_2$. We want to show that LoCaMu ensures that m_1 is always processed by i before m_2. We first consider the case where all messages involved in the causal dependencies are sent by nodes in the safety neighbourhood of i.

Lemma 1. *Let $m_1 \to m_2$ be two messages delivered by process i. If m_1 and m_2 are sent by the same node $j \in \mathcal{V}^S(i)$, then i processes m_2 after m_1.*

Proof. When m_1 is sent by process j the identifiers of m_1 are added to the causal past of node j (line 19). When m_2 is sent, the casual past of j is added to the header of m_2 (line 8). When m_2 is received by node i it is not processed before all messages with identifiers included in the header of m_2 have been processed locally (line 28). □

Lemma 2. *Let $m_1 \to m_2$ be two messages delivered by process i. Let m_1 be sent by some node $j \in \mathcal{V}^S(i)$ and received by some other node $k \in \mathcal{V}^S(i)$ ($j \neq k$). Let m_2 be sent by node k. Then i processes m_2 after m_1.*

Proof. When m_1 is sent by process j, the identifiers of m_1 are added to the causal past of node j (line 8). When m_1 is delivered by node k, the identifiers of m_1 are added to the causal past of node k (line 10). When m_2 is sent, the casual past of k is added to the header of m_2 (line 8). When m_2 is received by node i it is not processed before all messages with identifiers included in the header of m_2 have been processed locally (line 28). □

Lemma 3. *Let $m_1 \to m_2$ be two messages delivered by process i. Assume there is a chain of messages $m_1 \to m_a \to m_b \to \ldots \to m_2$ and all messages in the chain have been sent by nodes inside the safety neighbourhood of i. Then i processes m_2 after m_1.*

Proof. The detailed proof is omitted due to lack of space. It is trivially obtained by induction using the previous two lemmas. □

We now prove that causal order is preserved even if the causal dependency $m_1 \to m_2$ is created by some node outside the safety neighbourhood of node i. The proof relies on the fact that the base graph is acyclic and that messages are always propagated via safe paths.

Theorem 4. *Let $m_1 \to m_2$ be two messages delivered by process i. Then i processes m_2 after m_1.*

Proof. The proof is by contradiction. Assume that $m_1 \rightarrow m_2$ but node i delivers m_2 before m_1, violating causality. Let *chain*$_{m1} = m_{root1} \ldots \rightarrow m_{m1} \rightarrow m_{l1} \rightarrow m_{k1} \rightarrow m_{j1} \rightarrow m_1$ and *chain*$_{m2} = m_{root2} \ldots \rightarrow m_{m2} \rightarrow m_{l2} \rightarrow m_{k2} \rightarrow m_{j2} \rightarrow m_2$ be the causal chains of m_1 and m_2 respectively, where m_{root1} and m_{root2} have been sent by the same node *origin* and $m_{root1} \rightarrow m_{root2}$. Because the base graph is acyclic and because messages are always propagated using safe paths, if $m_1 \rightarrow m_2$, the paths used by both chains cannot be disjoint in the safe neighbourhood of i. Thus, there is at least one node x in the safe neighbourhood of i and a pair of messages $m_{a1} \in$ *chain*$_{m1}$ and $m_{b2} \in$ *chain*$_{m2}$ such that m_{a1} and m_{b2} are sent by x. By the lemmas above, if node x has processed m_{a1} before m_{b2}, then node i would have delivered m_1 before m_2. Therefore, node x must have processed m_{b2} before m_{a1}. This argument can be used recursively, to show that there must be some node y, in the safe neighbourhood of node x, and a pair of messages $m_{c1} \in$ *chain*$_{m1}$: $m_{c1} \rightarrow ma1$ and $m_{d2} \in$ *chain*$_{m2}$: $m_{d2} \rightarrow mb2$ that have been sent by node y. Again, node y must have processed m_{c1} before m_{d2}. Because the base graph is acyclic, the recursion eventually ends at node *origin*, and *origin* should have processed m_{root2} before m_{root1}, a contradiction. □

3.10.2 Liveness

Liveness is proved in two steps: firstly it is proven that messages from any publisher will eventually reach every destination node and secondly it is proven that every message will eventually be delivered.

For any message, there is a source node and several destination nodes. Between each source and destination nodes there may be several intermediary nodes, who must also receive and deliver the message. Therefore it is necessary to prove two additional scenarios, namely that for any message, it will be received and delivered by the interested nodes in the initial neighbourhood of the sender and then it will be received and delivered by the interested nodes in the outside neighbourhoods. Therefore it is necessary to first prove that every message will be received and delivered by the interested nodes in the neighbourhood of the publisher and then the messages will be received by the interested nodes in the outside neighbourhoods.

Lemma 5. *Every message is received by every destination node inside the publisher's neighbourhood.*

Proof. To prove that messages are received in their initial neighbourhood, recall that there are direct links between nodes at f+1 nodes of distance. Considering two such nodes A and B, both nodes will have an entry $P[A][B]$ and $P[B][A]$. Therefore they each keep count of how many messages have been exchanged between each other and can detect any missing messages by using the identifiers $I_m[A][B]$ and $I_m[B][A]$. Node B will always deliver these messages, unless there are some missing dependencies. □

Lemma 6. *In a given neighbourhood, every received message is eventually delivered by every node.*

Proof. It may happen in the system that a node A may multicast a message M_1 to a node B and then multicast a message M_2 to a node C. If node C now publishes a message M_3 to B and M_1 was not received by B (Either because there were problems in the connection or node A crashed before ensuring it was delivered) then B will not be able to deliver M_3, as it depends on M_1, which only A contains. However, it is assumed that node crashes are transient and when a node recovers, it is able to forward at least one message. Therefore A will eventually recover and forward M_1 to B, who will then deliver M_1 and M_2. □

Lemma 7. *Every message is received by every destination node.*

Proof. It is guaranteed that messages between nodes at $f + 1$ nodes of distance are always received and delivered and therefore, due to the intersection of neighbourhoods, a message will eventually reach the destination nodes present in every neighbourhood. □

Theorem 8. *The algorithm ensures that every message is eventually delivered to its destination node(s).*

The proof is similar to Causal Barriers' [27]'s liveness proof. A message M received by a node P_i can be delivered if both conditions present in lines 33 and 38 are false (meaning that the messages dependencies have been fulfilled and the message is not a duplicate). Consider all messages that have not been delivered by the node P_i. The $happened - before$ relation can be used to do a partial ordering of the messages that are yet to be delivered. Let M' be one of the messages in this partial order that has no message dependencies and is not a duplicate message. As M' has been delivered upon being received, the condition $\exists k \in P_i[k][i] < D'_M[k, i]$ must be true. This means that there is a message M_x that caused M' but that was not delivered in P_i. This violates the assumption that among the messages that are yet to be delivered to P_i, M' does not have a predecessor.

Summary

This chapter addressed LoCaMu, the first localized algorithm that enforces causal order and provides fault tolerance. This is achieved through several components: Firstly, each node only maintains state regarding a given neighbourhood, whose size is directly connected to the number of consecutive faults to be supported, f. Secondly, fault tolerance is provided by using redundant paths when nodes crash. Thirdly, by using the *identifiers* mechanism, duplicate and missing messages are detected. Finally, by having each node use a matrix that contains entries related to its safety neighbourhood, causal order is guaranteed.

4 Implementation

Contents

4.1 Development Environment 46
4.2 Framework 46
4.3 Algorithms 47

This chapter describes the implementation of LoCaMu and of the related systems that will be used in the evaluation (namely Delta-Neighbourhood, CBCAST, and Causal Barriers). In order to make the evaluation fair, all solutions use the same framework, with unique code being related to each algorithm. Section 4.1 presents the technologies that were used to implement the algorithms. Section 4.2 explains the details behind the framework used by every algorithm and Section 4.3 presents relevant details concerning the implementation of each algorithm.

4.1 Development Environment

The prototype was implemented using the Java programming language (OpenJDK 11 [28]) on Windows 10. As Java is a Virtual Machine based language, the choice of operating system is not relevant for the development. The Peersim [12] simulator was used as a base program for the LoCaMu implementation; Maven [29] was used in order to compile the software and JUnit [30] was used for unit testing.

4.2 Framework

To make the comparisons among the related algorithms as fair as possible, a single framework was created to include all common code. This framework offers by default four common components, that are used by every algorithm:

- The *Message Publisher*, which generates messages that belong to a group and that will be received by nodes belonging to the same group;
- The *Causality Handler*, that detects if the algorithm is well implemented by verifying if causality is not broken when delivering a message;
- The *Fault Detector*, which detects what neighbours are down and connects to the new neighbours;
- The *Statistics* component, that is responsible for collecting statistics (such as how long a node was down, longest amount of time it was down without recovering, amongst others);

The framework requires the following components to be implemented when a new algorithm is added to the system:

- The *Message Storage*, which decides if a message should be accepted or buffered (in case of causal algorithms).
- The specific *Front-End*, whose job is sending and receiving messages to and from other nodes.

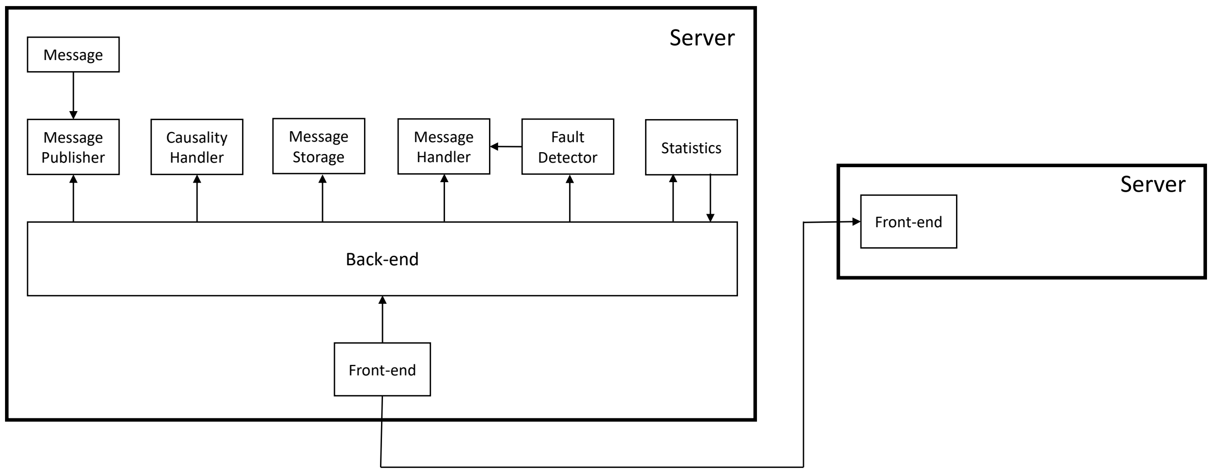


Figure 4.1: Interaction between each component in the framework

- The *Connection Handler*, being responsible for deciding which neighbours should receive a given message, storing the original message (as to ensure fault tolerance) and in some cases, modify the message in order to optimize the metadata;
- The algorithm-specific *Message*, which holds the metadata that each message will carry and any methods required to access or modify the metadata.
- The *Back-End*, which is responsible for holding and connecting every necessary component;

Figure 4.1 illustrates how all of these components are connected.

4.3 Algorithms

This section describes how each algorithm was implemented. As LoCaMu does not specify how groups may change dynamically, subscriptions or group membership changes were not implemented in any of the algorithms. Additionally, in order to create a fair comparison, every algorithm is used in a tree overlay. This is nevertheless necessary as when running large-scale systems, it is unrealistic for a node to be connected and to be able to send a message to every other node, as seen in [10], where the authors compare the cost of broadcasting messages in a clique and in a tree overlay.

4.3.1 LoCaMu

LoCaMu has been implemented mostly as specified, with some differences. This is regarding how messages are detected as being safe to send or not. Instead of having each node count the number of identifiers a message has and if they are enough for the given travelled path, a system similar to Delta

Neighbourhood's is used, where if there is a jump then an "empty" entry is added to a message's I_m. Another difference is regarding the mechanism for garbage collecting the R_i matrix. Instead of creating a special type of message to propagate the matrix copies, the relevant portions of these copies are piggybacked on normal messages. It should be noted that this extra data is not accounted for when calculating how long a message will take to reach the destination. This means this extra data has no impact on the obtained results.

Message Implementation

The I_m field is represented using a List of Pairs, where a Pair is composed of a value (a forwarder node's ID) and a map of key-values, where the key is a target node's ID and the value is the corresponding message number. The *dependencies* matrices are represented using primitive arrays. The unoptimized version of the matrix consists of with one entry per pair of nodes in a given neighbourhood where every value is either zero or positive and the size is calculated by multiplying the number of entries by 4. The optimized version still uses a matrix with the same size as the unoptimized version, but the entries whose values did not change have a value of -1. The size of each message is given by: $(DiffCol \times 4) + (DiffEntry \times 8)$, where:

- DiffCol: Number of changed columns that have at least one entry with a value other than -1;
- DiffEntry: Number of entries whose value is different than -1.

Every message that is sent to a given node is optimized.

4.3.2 Delta-Neighbourhood

As partitions and subscriptions are not considered in the evaluation, the features related with the partition detector and subscriptions were not implemented. Otherwise, the algorithm was implemented as specified. Effectively, this means that each message uses 8 bytes per each entry on the metadata vector. A jump is represented with the identifier as -1 and value as -1, costing 8 bytes.

Message Implementation

As the amount of metadata present in this algorithm is much smaller than the others, there was no programming effort in optimizing the data structures. Therefore, the *sequence vector* is represented by a List of Pairs. The metadata size is equal to $|$*sequence vector*$| \times 8$.

4.3.3 CBCAST

CBCAST's algorithm was implemented as specified on Chapters 5.1 and 6.1 of [1] and uses the optimizations present on Chapters 5.4 and 6.3 of the same paper. As group membership is not addressed in this dissertation, the *flush* mechanism was not implemented. The optimization works by comparing the metadata of the last sent message and sends the different values, having the penalty of having to identify which were the altered entries. For example, if m_1 has the entries $\{G_1[0,1]$, $G_2[2,2]\}$ and m_2 has $\{G_1[0,2]$ and $G_2[2,2]\}$, then m_2 is sent with only $G_1[1:1]$ (first value indicates the index, second value indicates the value).

Message Implementation

Each message that is sent to any given node is optimized, containing two structures: a vector v and a matrix ma. Each entry $v[i]$ holds a group ID and each line $ma[i]$ contains a vector of key-values, where the key is the node ID and the value is the corresponding clock value. As such, the metadata size is equal to:

$$\sum_{n=1}^{numberChangedGroups} 4 + (\textit{changedEntries} * 4)$$

This means that m_2, containing $G_1[1:1]$, uses 12 bytes.

4.3.4 Causal Barriers

Causal Barrier's implementation fully followed the specification as presented on Chapter 5.2 [6]. Unlike the previous algorithms, there are no missing features. A useful property of this algorithm is that for any given message m, all targets of that message will receive the same copy of the message, unlike CBCAST and LoCaMu, where, because of the optimizations, each target may receive the message with different metadata. As the evaluation uses a simulator, it is possible to only create a single message object, which is then shared between all targets, saving a noticeable amount of memory.

Message Implementation

The $targets$ structure uses a Java HashSet and the $CausalBarriers$ structure uses a Map of Java HashMaps (essentially being a matrix whose number of lines and columns can increase and decrease). The reason that these structures are used is that while they require more memory and processing power, they are more suited for the algorithm (which uses the comparison of matrices to find common entries and remove entries of matrices) Additionally, although these data structures use more memory than traditional primitive arrays, as each published message is only created once (because the targets will all receive the same message object), the amount of memory is reduced, making the trade-off between programming complexity and memory worth it. When a message is sent to other node, the amount of metadata is given by $|\mathit{Targets}| * 8 + |\mathit{list}| * 8 : \ \forall \ \mathit{list} \in \mathit{CausalBarriers}$.

Summary

This chapter described how the prototypes were implemented, first by stating what was the development environment, then explaining how to create a prototype based on the used framework (pointing out which code is common to all prototypes and their use and which code needs to be implemented and why) and finally disclosing specific details regarding the implementation of each of the four prototypes.

5 Evaluation

Contents

5.1 Goals 52

5.2 Analytical Analysis 52

5.3 Experimental Settings 54

5.4 LoCaMu vs Delta-Neighbourhood 56

5.5 LoCaMu vs Causal Multicast Algorithms 57

5.6 Metadata Size Comparison 59

5.7 Discussion 61

This chapter presents the evaluation of LoCaMu, based on a prototype built for a simulator. The main goal of this evaluation is to verify if LoCaMu does in fact scale better than some of the mentioned works in the Related Work. Section 5.1 describes what will be tested in the evaluation. Section 5.3 describes the system settings, offering information about the system overlay, groups, the metrics, bandwidth, amongst others. Section 5.4 compares LoCaMu with Delta Neighbourhood, Section 5.5 compare LoCaMu with CBCAST and Causal Barriers when there is a single group of communication and several groups of communication. Section 5.6 compares both the biggest amount of memory required by a node and biggest amount of metadata a message may have in LoCaMu, CBCAST and Causal Barriers. Finally, Section 5.7 discussions the trade-offs present in both locality algorithms and the main differences in the causality algorithms.

5.1 Goals

LoCaMu is evaluated in order to answer the following questions:

- What is the performance of LoCaMu when compared to a localized algorithms (even if it only offers FIFO order)?
- What is the performance of LoCaMu when compared to other algorithms that enforce causal order?
- How does LoCaMu compare with previous works in term of signaling and memory overhead?

5.2 Analytical Analysis

In this section it is predicted what will be the answers for the three questions, based on how each algorithm works.

5.2.1 Comparison of Localized Algorithms

Delta-Neighbourhood requires much less metadata than LoCaMu, however a new message can only be published once the previous message has been acknowledged by every target in the system. In turn, LoCaMu's messages can be continuously published without this restriction. This means that individually, a publisher in LoCaMu will have a much higher throughput than in Delta-Neighbourhood, however the bandwidth can be saturated much more quickly in LoCaMu. In a scenario where the number of publishers is increasing, this means two things: first, if the bandwidth is limited then the throughput of Delta-Neighbourhood will eventually be superior to LoCaMu's. If the bandwidth is unlimited then LoCaMu's throughput will always be superior to Delta-Neighbourhood's. An important difference between both algorithms is that Delta uses metadata regarding the path that a message has gone through and LoCaMu uses metadata regarding both the path that a message has gone through and the paths it will go through. This means that in Delta-Neighbourhood the breadth of the tree has no impact, but in LoCaMu it does and therefore given a higher value of f and a higher breadth of the tree, LoCaMu's throughput will suffer much more than Delta-Neighbourhood.

5.2.2 Comparison of Causality Algorithms

There is an important component that was not discussed in this dissertation which is how nodes join groups. In CBCAST, every node of a group must be aware of a node joining or leaving the group. In Causal Barriers, this is not required. The trade-off here is group dynamism (the changing of group members) is very fast in Causal Barriers and slow in CBCAST, but in turn message optimization is much better in CBCAST than in Causal Barriers. This means that if concurrency is low, the message size in CBCAST will be very small while in Causal Barriers it will be high. LoCaMu does not deal with group dynamism, but it can optimize the *Past* part of a message as well as CBCAST but not the *Identifiers*. This means that each message in LoCaMu will require more metadata than CBCAST when there is low concurrency (as the *Identifiers* will not be compressed), but when there is a high amount of concurrency then LoCaMu will eventually require less metadata than CBCAST, because of how each message only keeps information about a given neighbourhood. Causal Barriers will always require more metadata than any of the other protocols, as its optimizations are poorer. In sum, CBCAST will require less metadata than LoCaMu but will eventually be outscaled.

5.2.3 Comparison of the overhead

CBCAST and Causal Barriers do not depend on the breadth of the overlay and the number of tolerated faults, so by increasing these variables the metadata of a message will be the same. However, they do scale with the system size. LoCaMu is the opposite of this. Due to using locality, the system size does not impact the size of a message, but the breadth of the overlay and the number of tolerated faults does. It is important to note that LoCaMu's message metadata size increases exponentially with the breadth and the number of faults, so for high values of both variables, the message size can be bigger than CBCAST's or Causal Barriers'. Therefore, for a large enough system, LoCaMu will have the best results.

5.3 Experimental Settings

LoCaMu is compared with CBCAST [1] and Causal Barriers [6] (that ensure causal order but are not localized) and with Delta-Neighbourhood [5] (that is localized but only only ensures FIFO). Nodes are organized in an overlay network with the topology of a binary tree. It is assumed that there is a multicast group for each subtree of the overlay. This means that there is a large group that contains all nodes, then two smaller groups with the nodes on the left subtree and the nodes of the right subtree, respectively, and then 4 smaller groups the result from further dividing the tree, and so forth. This is a simple setup that ensures that the experiments combine groups of all sizes (up to groups of just two members), while showcasing the advantages of locality. Simulations were used to perform the evaluation because this allowed us to experiment large overlays that would be otherwise impossible to test. For this the Peersim [12] simulator was used, with extensions to simulate edges with realistic latency and bandwidth constraints. In the experiments, each node has a limited bandwidth that is shared among all its edges. All links have an average latency of 1ms (note that the latency only affects the throughput of [5], not the throughput of LoCaMu, [1], or [6]).

Regarding the performance metrics, *maximum individual throughput* was the most used metric, which considers a single publisher, and the *maximum aggregated throughput*, which considers several different publishers. By definition, aggregated throughput is the total number of messages delivered to all nodes in the overlay in a second. All the algorithms used in the evaluation are able to capture causality with the same degree of accuracy. Thus, the differences in performance, if any, will be mainly caused by the amount of metadata that messages need to carry. This metadata consumes bandwidth and affects the latency of message propagation. Since the overlay is a tree, the maximum aggregated throughput is limited by the bandwidth of the root node. Therefore, it is expected that algorithms that use less metadata can achieve higher aggregated throughput than algorithms that use more metadata. However, algorithms such as Delta-Neighbourhood impose additional restrictions on message pipelining, that prevent nodes from using effectively the available network bandwidth.

Finally, it is important to emphasize that all algorithms use a number of optimizations in their implementation. Optimizations for LoCaMu have been briefly described in Section 3.9.3. CBCAST uses similar optimizations, where nodes only transmit the vector clock fields that have been updated since the last transmission. Causal Barriers has similar optimizations embedded in its main algorithm. For fairness, in these experiments, all 3 protocols use their optimized versions.

The following experiments use a base configuration consisting of:

- The system size was set to 511 nodes;
- The bandwidth of the nodes was set to 100 Mb/s;
- The payload size was set to 32 bytes (this value was chosen based on Facebook's production TAO

system [31], where 32 bytes represents the 90 percentile of data size that is stored);

- The number of tolerated faults in each neighbourhood was set to $f = 1$;
- The node degree was set to 3 (a binary tree).

In each experiment one of these variables is changed, keeping the remaining parameters unchanged. Information regarding the number of publishers and groups is declared for each experiment.

5.4 LoCaMu vs Delta-Neighbourhood

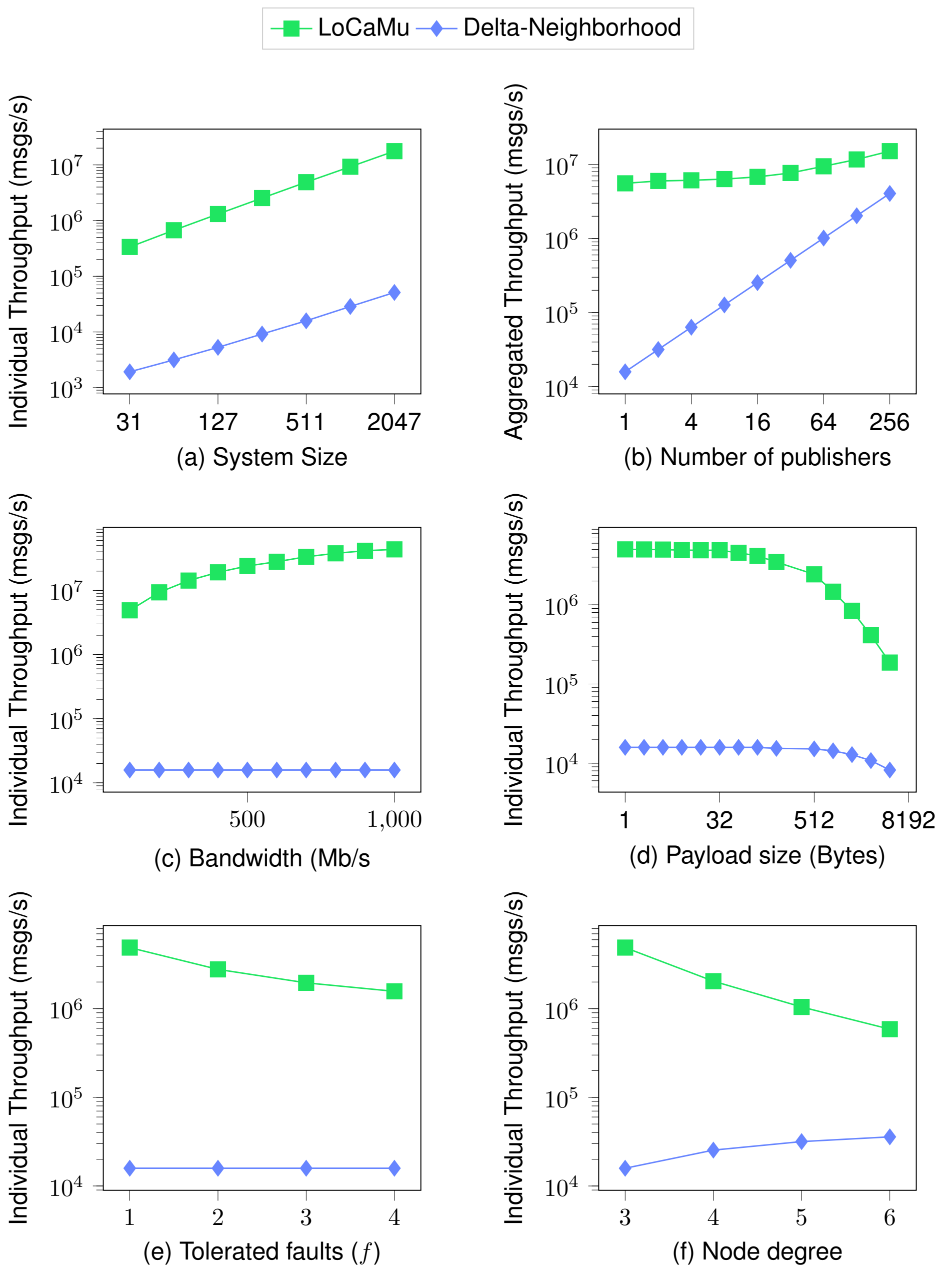


Figure 5.1: LoCaMu vs Delta-Neighbourhood under different settings

LoCaMu and Delta-Neighbourhood are similar in the sense that both use localized information, therefore it is important to compare them. Since the latter offers less guarantees (only supports FIFO), it should perform better than LoCaMu (that needs to keep track of causal dependencies among multiple publishers). As will be shown, Delta-Neighbourhood is, in fact, able to offer better maximum aggregated throughput if there are enough publishers and enough nodes in the system, but performs poorly when the maximum individual throughput is considered. This happens because in [5] a node is not allowed to send a new message before the previous one is received.

In these experiments, messages are sent to a single group that includes all nodes and by default there is only one publisher that publishes messages as fast as possible, saturating the bandwidth of the root node. Figure 5.1a shows the maximum individual throughput as the system size is changed. As this size increases, the diameter of the network grows and it takes longer for a message to reach its destinations. Since, Delta-Neighbourhood does not support pipelining, the throughput of this protocol scales poorly with the system size; LoCaMu does not have such drawback. In Figure 5.1b, the number of publishers is changed in order to measure the maximum aggregated throughput. Since LoCaMu exploits pipelining, it is able to approximate the maximum network capacity even with a single publisher. Delta-Neighbourhood has less individual throughput but, as it uses much less metadata, it supports a much higher aggregated throughput, growing linearly with the number of publishers and eventually overcoming LoCaMu's. In Figure 5.1c, the bandwidth available to the nodes is changed. Again, given that the maximum individual throughput of Delta-Neighbourhood is constrained by the end-to-end latency, it cannot benefit from the extra available bandwidth, while in LoCaMu a single publish can fully exploit the available bandwidth. In Figure 5.1d, the payload size is varied; as the size of the payload increases it becomes the main source of bandwidth usage in the network and both algorithms start behaving similarly. In Figure 5.1e, the f value is changed; because the size of the metadata in LoCaMu is a function of the size of the safe neighbourhood, and this size grows exponentially with f, the performance of LoCaMu drops notably for large values of f. Finally, in Figure 5.1f, the node degree (i.e., the breadth of the tree) is changed; again, LoCaMu is affected due the size of the safe neighbourhood grows exponentially while Delta-Neighbourhood is able to improve because the diameter of the network is reduced.

5.5 LoCaMu vs Causal Multicast Algorithms

Causal order is only relevant when there are multiple publishers, hence it makes no sense to measure the maximum individual throughput of an isolated node. Therefore, this section concentrates on assessing the maximum aggregated throughput achieved by the different algorithms. For these experiments the same base configuration is used as before. However, to make sure that all nodes send messages and causal dependencies are established among these messages, the throughput of each publisher is limited to 100 msg/s. Note that the size of the metadata maintained by LoCaMu and Causal Barriers is similar, regardless of the number of groups in the system. CBCAST uses a vector clock for each group, thus the metadata changes with the number of groups. The experiments were ran with 1 group and with 255 groups. Two different bandwidths were also used, in order to see the impact of the metadata increase on the throughput.

5.5.1 Single Group

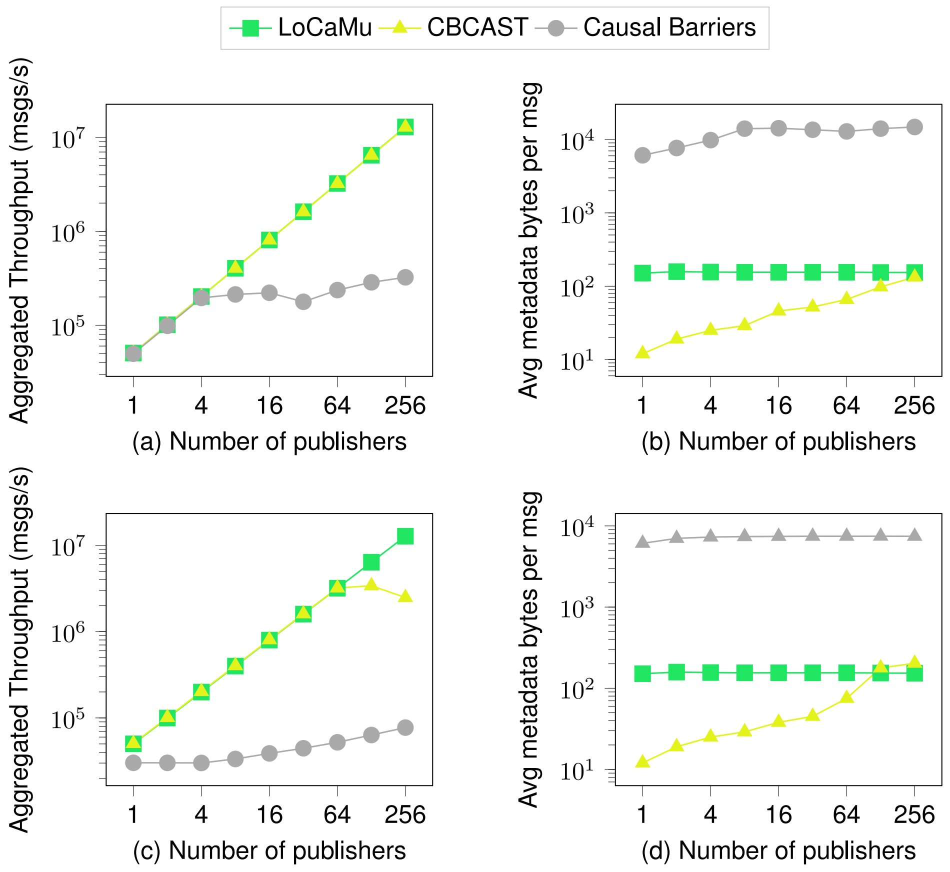


Figure 5.2: Throughput comparison (left) in a system with 511 nodes with 1 group between the causal algorithms and the corresponding average metadata size per message (right). Bandwidth is 100 mbps (top) vs 10 mbps (bottom)

Figure 5.2a and Figure 5.2c compare the maximum aggregated throughput achieved by the different algorithms when all nodes send messages to a single group that includes all members. As before, as more publishers are added to the system, the aggregate throughput continuously increases up to a point where the maximum flow of the network is reached. Figure 5.2b and Figure 5.2d show the corresponding average metadata present in each message when the number of concurrent publishers is increased. By analysing these two last figures, Causal Barriers is shown to use a very high amount of metadata and while CBCAST initially uses less metadata than the other algorithms due to its optimizations compressing the clocks, as the number of parallel publishers increases, the effectiveness of the optimizations decreases, until it eventually starts requiring more metadata than LoCaMu. This in turn results in its aggregate throughput decreasing as seen in Figure 5.2c, where, due to the network having a smaller bandwidth, the network is saturated.

5.5.2 Various Groups

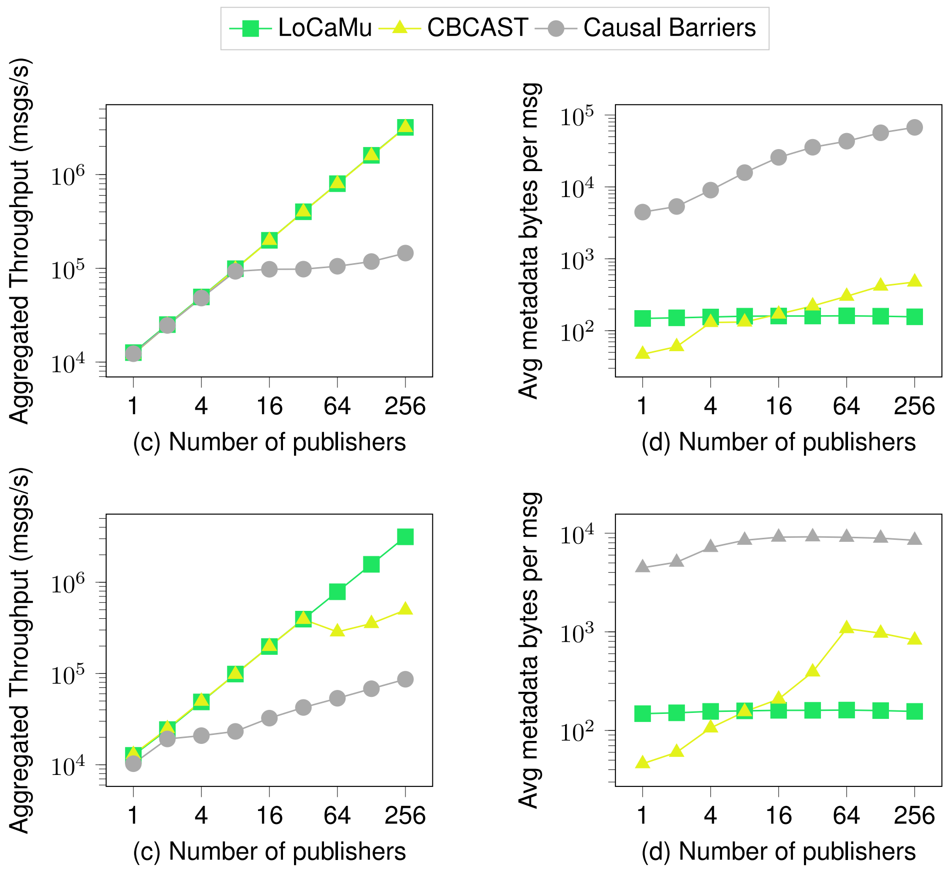


Figure 5.3: Throughput comparison (left) in a system with 511 nodes with 255 groups between the causal algorithms and the corresponding average metadata size per message (right). Bandwidth is 100 mbps (top) vs 10 mbps (bottom)

Figure 5.3a and Figure 5.3c compare the maximum aggregated throughput achieved by the different algorithms when all nodes send messages to the existing 255 groups of different size (as explained in 5.3). Figure 5.3b and Figure 5.3d represent the corresponding average message metadata. Causal Barriers' and LoCaMu's average metadata is similar when there is one group or several groups, as their metadata is independent of the number of groups, so these algorithms' performance is similar to the previous section. CBCAST, however, scales much worse, because messages are required to carry several vector clocks (in the worst case, as many as the number of groups that exist in the system).

5.6 Metadata Size Comparison

From the results of the previous experiments, it is clear that the size of the metadata exchanged in the header field of messages has a significant impact on the performance of the different algorithms.

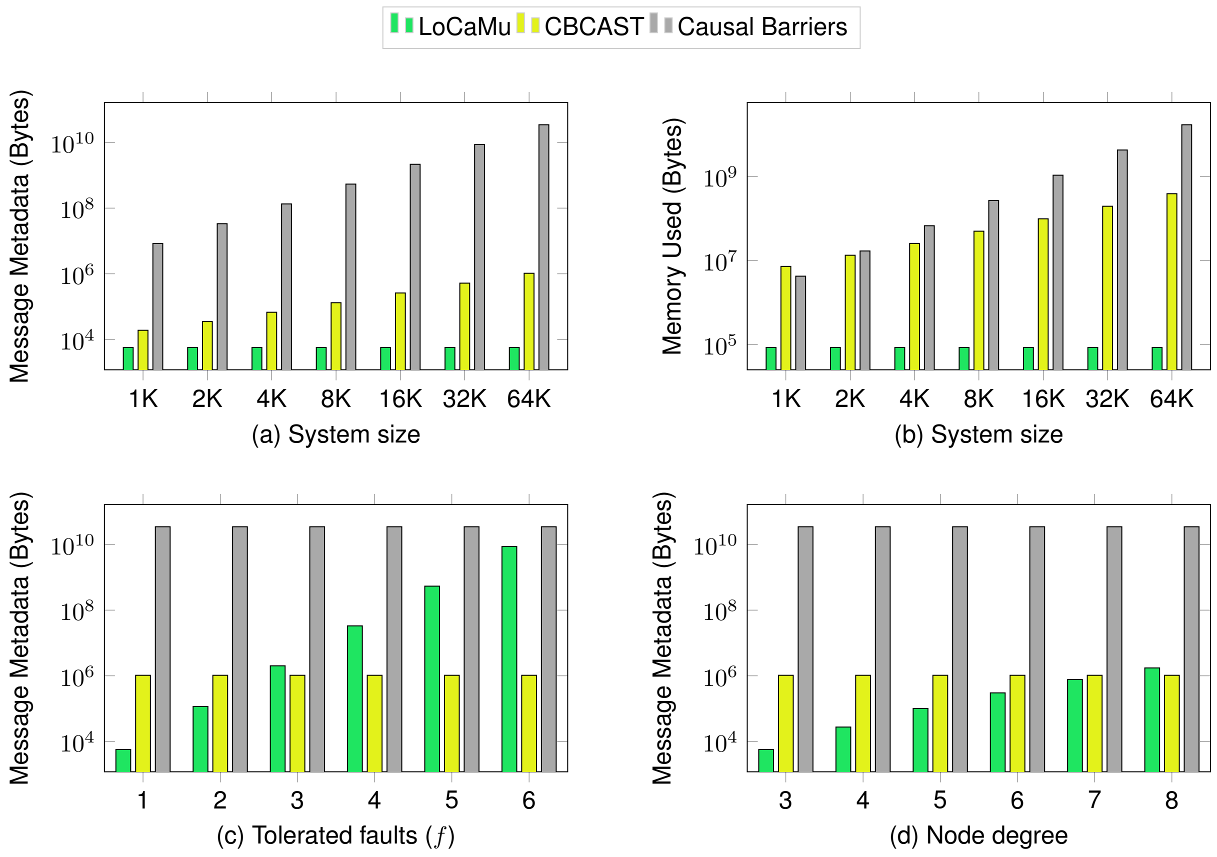


Figure 5.4: Metadata worst case and memory required (causal algorithms). Default settings: $f = 1$, node degree = 2, nodes = $65,536$

Detailed data is now provided for the amount of metadata required by each algorithm.

A key aspect that the reader should retain is that Causal Barriers and CBCAST are inherently non scalable: the size of the metadata required by the algorithms quickly becomes unfeasible to manage in practice as the size of the system grows. Recall that in Causal Barriers the size of metadata may be roughly quadratic with the system size, for CBCAST is linear with the system size and linear with the number of groups, and with LoCaMu is exponential with the size of the safe neighbourhood. The size of the safe neighbourhood is a function of f and of the degree of the nodes in the overlay, but does not depend on the entire system size. For instance, for a system with $1,024$ nodes, assuming that every node has sent a message to every other node, Causal Barriers would be required to send a message with over 2^{20} entries. Assuming 8 bytes per entry (an entry is a tuple that includes an identifier and a sequence number), Causal Barriers uses 8 Megabyte (MB). In the same setting, CBCAST would require a vector clock with size 4 Kilobyte (KB) for a single broadcast group. In turn, LoCaMu requires 540 bytes assuming a node degree of 3 and $f = 1$, *no matter the system size*. Figure 5.4 illustrates these facts with more detail. Here, consider the existence of $1,500$ groups and network size of $65,536$ nodes. A Zipf-like distribution is used for assigning the size of the group membership such that *Members*$(g) =$

$[Ng^{-1.25} + 0.5]$, where g is the unique number of the group (from 1 to $1,500$ in this particular case) and N is the amount of nodes in the system. This results in the sum of the amount of members of every group being $237,895$. This distribution was used for the membership of the groups, because it typically used in others works that consider large-scale publish-subscribe settings (such as [11]).

Figure 5.4a and Figure 5.4b show the size of the header fields and the amount of local memory used by each algorithm, as the size of the system increases. These plots consider a node degree of 3 and $f = 1$. In Figure 5.4c and Figure 5.4d the system size is kept equal to $65,536$ and the size of the header fields is compared as a function of f (keeping node degree equal to 3) and also as a function of the node degree (keeping $f = 1$). As expected, only the metadata of LoCaMu is affected by these parameters. An interesting observation is that, even for large values of f and large values of the node degree, LoCaMu still requires much less metadata than Causal Barriers.

5.7 Discussion

There are two main points to discuss regarding the evaluation. The first one is regarding the trade-offs in both locality algorithms. While Delta-Neighbourhood uses less metadata per message, it has a much smaller throughput per publisher (due to requiring each message to be acknowledged by every target in the system). LoCaMu loses when comparing the throughput of multiple publishers, however, this downside can be neglected if the bandwidth between the nodes is sufficiently big. Delta-Neighbourhood's downside (latency) currently does not have a solution, as there is a maximum speed at which data can travel.

Regarding the causality algorithms, there is one difference between CBCAST and both Causal Barriers and LoCaMu, namely CBCAST tracks causality using metadata specifically per ground while Causal Barriers and LoCaMu don't consider the concept of groups in their metadata. The trade-off between CBCAST and Causal Barriers, however, is that while in CBCAST a node may take a long time to join a group because it needs to synchronize with the other nodes in the system, in Causal Barriers there is no synchronization required. This allows CBCAST to optimize the metadata much more efficiently than Causal Barriers. However, this optimization eventually stops being efficient when either one or more of the following is true:

- The system is very large;
- There are several nodes publishing;
- There are several groups of communication.

LoCaMu's performance does not suffer from any of these conditions. The system size has no impact on any individual node, each node is only affected by the publishing or forwarding nodes inside its neighbourhood and groups have no impact on the metadata. This advantage of LoCaMu is notable on Figure 5.3c, as there are several nodes publishing and several groups of communication, therefore CBCAST's throughput is very affected but LoCaMu's able to continuously increase.

Summary

This chapter presented the evaluation of LoCaMu. By using a simulator, two sets of comparisons were made. The first set compares the performance of the locality algorithms, LoCaMu and Delta-Neighbourhood. The conclusion is that while Delta-Neighbourhood only offers FIFO, LoCaMu is better in every scenario except when there is a sufficiently large number of concurrent publishing nodes. The second set compares the algorithms that offer causality: LoCaMu, CBCAST and Causal Barriers. The general conclusion is that Causal Barriers severely under performs in large systems, while CBCAST is able to have a good performance if there are not a lot of concurrent publishing nodes and there is a small number of groups. Meanwhile, LoCaMu has good performance in every scenario because of the concept of locality. Finally, the worst case of how much metadata a message may carry of these causality algorithms is displayed and both CBCAST's and Causal Barriers' worst case scales significantly with the system size, while LoCaMu's remains constant.

6 Conclusion

Contents

6.1 Conclusions 64
6.2 System Limitations and Future Work 64

6.1 Conclusions

This dissertation has presented LoCaMu, an algorithm that guarantees message delivery with causal order in a publish-subscribe system built on top of a broker overlay. LoCaMu is the first algorithm to offer causal order and fault tolerance while using localized information, i.e. each node maintains state regarding only the nodes on its neighbourhood and not every other node on the system. Thus, LoCaMu can be used in large scale systems, while previous work requires so much metadata that any practical implementation becomes infeasible. For systems with hundreds of nodes, where the previous work can still be applied, LoCaMu shows clear advantages, given that it makes a much better use of the available bandwidth.

6.2 System Limitations and Future Work

As this work considers a very large system, it was unfeasible to deploy LoCaMu in a real system with the same amount of considered nodes. Therefore, the evaluation was entirely produced based on simulations, which may produce results that are different from those obtained with a real implementation. There are some details that were not explored, namely having the system overlay be dynamic, meaning neighbourhoods could change by having new nodes and having other nodes leave the system. Group membership changes could also be explored, which would require subscription messages that would guarantee causality between groups.

Bibliography

[1] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *TOCS*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

[2] J. Lin and S. Paul, "RMTP: a reliable multicast transport protocol," in *INFOCOM*, San Francisco (CA), USA, Mar. 1996, pp. 1414–1424 vol.3.

[3] I. Gupta, R. van Renesse, and K. Birman, "Scalable fault-tolerant aggregation in large process groups," in *DSN*, Göteborg, Sweden, Jul. 2001, pp. 433–442.

[4] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *DSN*, Bethesda (MD), USA, Jun. 2002, pp. 7–16.

[5] R. Kazemzadeh and H. Jacobsen, "Partition-tolerant distributed publish/subscribe systems," in *SRDS*, Madrid, Spain, Oct. 2011, pp. 101–110.

[6] R. Prakash, M. Raynal, and M. Singhal, "An efficient causal ordering algorithm for mobile computing environments," in *ICDCS*, Hong Kong, May 1996, pp. 744–751.

[7] M. Bravo, L. Rodrigues, and P. van Roy, "Saturn: a distributed metadata service for causal consistency," in *EuroSys*, Belgrade, Serbia, Apr. 2017, pp. 111–126.

[8] R. V. Renesse and et. al, "Horus: A flexible group communications system," Cornell University, Ithaca (NY), USA, Tech. Rep., Mar. 1995.

[9] JGroups, http://www.jgroups.org/index.html, accessed: 2019-06-29.

[10] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *SOCC*, New York (NY), USA, Nov. 2014, pp. 1–13.

[11] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *NGC*, London, UK, Jul. 2001, pp. 30–43.

[12] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *P2P*, Seattle (WA), USA, Sep. 2009, pp. 99–100.

[13] V. Santos and L. Rodrigues, “Difusão em grupo tolerante a faltas com ordem causal usando informação localizada,” in *Inforum*, Guimarães, Portugal, Sep. 2019, pp. 217–228.

[14] ——, “Localized reliable causal multicast,” in *NCA*, Cambridge, MA, USA, Sep. 2019.

[15] L. Lamport, “Time, clocks and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[16] J. Leitão, J. Pereira, and L. Rodrigues, “Hyparview: A membership protocol for reliable gossip-based broadcast,” in *DSN*, Edinburgh, UK, Jun. 2007, pp. 419–429.

[17] A. I. T. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *Middleware*, Berlin, Heidelberg, Jan. 2001, pp. 329–350.

[18] J. Leitão, J. Pereira, and L. Rodrigues, “Epidemic broadcast trees,” in *SRDS*, Beijing, China, Oct. 2007, pp. 301–310.

[19] M. Ferreira, J. Leitão, and L. Rodrigues, “Thicket: A protocol for building and maintaining multiple trees in a p2p overlay,” in *SRDS*, New Delhi, Punjab, India, Oct. 2010, pp. 293–302.

[20] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *CSUR*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[21] Y. Zhao, D. C. Sturman, and S. Bhola, “Subscription propagation in highly-available publish/subscribe middleware,” in *Middleware*, Berlin, Heidelberg, Oct. 2004, pp. 274–293.

[22] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, “An efficient multicast protocol for content-based publish-subscribe systems,” in *ICDCS*, Austin (TX), USA, Jun. 1999, pp. 262–272.

[23] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, “Matching events in a content-based subscription system,” in *PODC*, New York, NY, USA, May 1999, pp. 53–61.

[24] J. P. d. Araujo, L. Arantes, E. P. Duarte, L. A. Rodrigues, and P. Sens, “A publish/subscribe system using causal broadcast over dynamically built spanning trees,” in *SBAC-PAD*, Campinas, Brazil, oct 2017, pp. 161–168.

[25] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer, 2006.

[26] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *JACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[27] M. Raynal, A. Schiper, and S. Toueg, “The causal ordering abstraction and a simple way to implement it,” *Inf. Process. Lett.*, vol. 39, no. 6, pp. 343–350, Oct. 1991.

[28] J. 11, https://openjdk.java.net/projects/jdk/11/, accessed: 2019-08-10.

[29] Maven, https://maven.apache.org/, accessed: 2019-08-10.

[30] JUnit, https://maven.apache.org/, accessed: 2019-08-10.

[31] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, “Tao: Facebook’s distributed data store for the social graph,” in *ATC*, San Jose (CA), USA, Jun. 2013, pp. 49–60.