Paulo Henrique Branco Dias

Degree in Computer Science and Engineering

Tree-based Decentralized and Robust Causal Dissemination

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Leitão, Assistant Professor,
NOVA University of Lisbon

Examination Committee

Chairperson:	Prof. Susana Nascimento, FCT/UNL
Raporteur:	Prof. Miguel Matos, IST/UL
Member:	Prof. João Leitão, FCT/UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

April, 2020

Tree-based Decentralized and Robust Causal Dissemination

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my family and friends.

Acknowledgements

The work conducted during this thesis had crucial assistance from several people and entities, to whom I am sincerely grateful.

Firstly, I would like to thank my adviser, Prof. João Leitão, who granted me this opportunity to research with him and continuously supported me with guidance and inspiration to achieve this outcome.

I want to proffer a special thanks to Pedro Fouto for all the suggestions and reviews that he made regarding this research and for accompanying me during the whole process. I also would like to thank Ricardo Loureiro that provided the initial prototype of HyParView and assisted me on the development of the final implementation.

I would like to mention the Department of Informatics and NOVA LINCS, where I found a great place to operate this research. This pleasant environment was enriched by my colleagues with whom I mutually exchanged many knowledge and advice.

This work was partially supported by the European Research Project H2020 LightKone (grant agreement ID 732505), and FCT/MCTES through the project NG-STORAGE (grant PTDC/CCI-INF/32038/2017) and NOVA LINCS (grant UID/CEC/04516/2013).

Experiments presented in this document were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Finally, I thank my family and friends for their unconditional support and endless motivation, which was vital to accomplish my goals.

Abstract

Managing a large-scale distributed system can be a really complex task that leads engineers to seek a high level of knowledge about the corresponding environment. New challenges and research are often emerging regarding this topic. One of the popular challenges that has gained significant attention recently is related to the pursuit of the maximum level of consistency that is possible to provide while remaining available in light of the CAP theorem. Causal consistency is one of the most common and interesting approaches that has been extensively studied in this context.

One way to build geo-replicated and highly distributed storage services providing causal (or causal+) consistency is to rely on a causal dissemination service that allows replicas to broadcast updates received locally to all replicas, ensuring that these are received in an order that respects causality. Building such dissemination service in a way that is scalable, robust and efficient is however a complex task. In this dissertation we will tackle this challenge by exploring how causal broadcast protocols certify the properties discussed above.

Our solution will have two main aspects: a robust and reliable dissemination protocol and an efficient causal consistency delivery enforcing mechanism. Moreover, we plan to implement our protocol in a realistic context, with large-scale systems that can validate the scalability of our proposed work.

Keywords: Distributed Systems, Causal Dissemination, Causal Consistency, Geo-Replication, Unstructured Overlay Networks

Resumo

Gerir um sistema distribuído de larga escala pode ser uma tarefa complexa, que leva engenheiros a estudar pormenorizadamente o contexto do sistema em questão. Novos desafios e pesquisas relativas a este tema estão a surgir cada vez com maior frequência. Um dos desafios mais populares relacionados com este tópico tem como objetivo alcançar o nível máximo de consistência possível, não comprometendo a disponibilidade, tendo em conta as premissas do teorema CAP. A consistência causal é uma das abordagens mais interessantes e comuns que tem sido estudada ultimamente.

Uma abordagem possível para desenvolver serviços de armazenamento de dados altamente distribuídos e geo-replicados que ofereçam consistência causal é confiar num serviço de disseminação causal que permita que as diversas réplicas possam propagar atualizações recebidas localmente para todas as outras réplicas, assegurando que estas são recebidas numa ordem que respeita a causalidade. Implementar um serviço de disseminação deste género e que seja escalável, robusto e eficiente é, contudo, uma tarefa difícil. Para contornar este desafio, nesta dissertação, o nosso principal objetivo é explorar a forma como os protocolos de disseminação causal certificam as propriedades referidas anteriormente.

A nossa solução terá duas características principais: um protocolo de disseminação robusto e confiável e um mecanismo para contemplar consistência causal de um modo eficiente. Além disso, queremos implementar o nosso protocolo em contexto real, recorrendo a sistemas de larga escala que possam validar a escalabilidade da nossa implementação.

Palavras-chave: Sistemas Distribuídos, Disseminação Causal, Consistência Causal, Geo-Replicação, Redes Sobrepostas Não Estruturadas

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Nowadays we can find distributed systems everywhere, with wide and heterogeneous structures supporting their operation. The sudden evolution of technology on a global scale highlighted the need of solutions to manage and spread information to different points of the world fast and reliably.

One of the main techniques that is used to overcome some performance issues is replication. This is motivated by the fact that most of the systems want to operate at a global scale, which lead to the emergence of the concept of geo-replication. Beyond the possibility of faster responses from the system, with geo-replication it is also possible to fulfill the need of fault-tolerance, since with this approach multiple copies of each data item are kept across different data centers.

Taking into account the existence of several user's operation entry points in this model, it is necessary to propagate the effect of the performed operations across all the replicas. This propagation takes some time to be performed and the replicas will receive the indication related to the operation at different moments, so it can be perceived that an operation performed by two distinct clients at the same instant, can result in two dissimilar behaviors, leading to an inconsistent state of the system.

Consistency is a crucial property in many contexts, but is not the only that have to be considered. The CAP theorem [3, 11] has famously established that it is only possible to assure two of the following three properties simultaneously: 1) strong consistency, 2) availability, and 3) partition tolerance. Since network errors are unpredictable and they will eventually occur, the partition tolerance property is mandatory. As a result of this assumption, one is left with two options: strong consistency or availability.

Despite the relevance of consistency in this kind of systems, availability represents a necessary requirement when the main focus of the service is usability, or in other words, when human users interact directly with the system.

There are many different consistency models studied in the context of distributed systems, although usually we classify them among two main categories: weak and strong consistency.

Strong consistency models ensure that every process of a system will return the same result to a certain request, at any given moment by ensuring that all replicas execute in a strongly synchronized way. However this guarantee has a high cost in performance, compromising availability, meaning that in the presence of faults or network partition, the system might be forced to stop its operation. Weak consistency models do not ensure this, enabling replicas to diverge and clients to observe outdated values, but allowing to considerably improve the performance of these operations.

In order to maintain availability, many weak consistency models were designed and implemented. One of the most popular models is causal consistency, that has been proven as one of the strongest weak consistency models that can be implemented .

The majority of the services that are provided currently focus heavily on user experience. Due to this fact, weak consistency models, namely causal consistency, become very attractive to implement and provide in these contexts.

To maintain a service that is required in many places of the planet, it is necessary to find a reliable way to exchange data between the different locations, considering that the machines that will support this service may be subjected to failures. In order to tackle this, we need to find a reliable approach to disseminate messages that ensure causal delivery, which is a relevant approach to build causally consistent systems, such as geo-replicated storage systems.

1.2 Problem statement

The main goal of the thesis is to develop a novel design for robust causal dissemination on dynamic large-scale distributed systems.

A service that relies on a geo-replication scheme is a perfect example of a dynamic large-scale distributed system. Due to that fact, our solution aims at being suitable to integrate in that class of distributed systems.

The main goal of our solution is to provide causal consistency, with high efficiency and robustness. It is also our intention to transform our replication structure into a tree topology, a standard type of graphs that can lead to the possibility of using very efficient algorithms, although significantly sensitive to failures.

1.3 Contributions

The resultant contributions from the work developed on this thesis can be summarized as follows:

- A novel broadcast protocol, named CPTCast, that constitutes a reliable solution of causal broadcast, relying on a tree-based dissemination strategy. The protocol provides high scalability, being capable to operate on large-scale distributed systems.
- A concrete implementation of this protocol over a convenient membership protocol, specifically HyParView [19], which is fully decentralized and highly robust to failures. Both protocols were developed on an equivalent codebase.
- An experimental evaluation of CPTCast conducted on a realistic large-scale testbed, that shows the benefits of adopting our solution in distinct scenarios. The experimental work required the development of several tools and led to the implementation of two broadcast protocols to compare with our solution, namely Plumtree [18] and PRC-broadcast [25].

1.4 Document structure

The present document is structured as follows:

- **Chapter 2: Related Work**

  This chapter presents the main concepts related to the work to be conducted in the thesis and discusses existing approaches that have been developed to address similar problems.
- **Chapter 3: CPTCast**

  Presents the novel causal broadcast protocol that we built, named CPTCast. We expose the rationale we adopted to design CPTCast and a complete specification of the solution.
- **Chapter 4: Evaluation**

  Covers all the aspects of the experimental work conducted to evaluate CPTCast, namely the methodology we followed to inspect our solution, the configuration details of the conducted experiments and the presentation and analysis of the results.
- **Chapter 5: Conclusion and Future Work**

  Concludes the document, summarizing the work performed on this thesis and providing possible directions for future work.

Chapter 2

Related Work

The purpose of this chapter is to expose the state-of-the-art on the topics related to the research conducted in the context of this thesis.

Firstly, we provide an overview of the most relevant consistency models and how causality fits in these models (Section 2.1). We proceed by describing some of the main overlay networks (Section 2.2) and the meaningful information about data dissemination through a network (Section 2.3), since this will help us in addressing some of the properties of the dissemination protocol to be developed.

Moreover, we detail data replication strategies (Section 2.4), such as geo-replication, and indicate some of the pertinent geo-replicated storage systems and describe how they can guide and inform our solution design (Section 2.5).

Finally, we present a few conclusions (Section 2.6) and a summary of this chapter (Section 2.7).

2.1 Consistency Models & Causality

There is a significant body of theoretical work (including many theorems and formal proofs) that describe some relevant restrictions on the domain of distributed systems, namely ACID, BASE and CAP [3].

The CAP theorem states that it is impossible for a distributed system to guarantee all of the following properties simultaneously:

- **Strong Consistency:** All the interactions performed by any client are in accordance with the most recent write;
- **Availability:** Every request must be able to complete and provide an answer to the client;

- **Partition tolerance:** The system works correctly (i.e. continuously provides all guarantees) even in the event of a network partition.

Consistency is a concept that can lead to misleading interpretations because it has a different meaning to the consistency concept in the context of ACID properties, which are defined considering traditional database systems. In that context, in order to accomplish consistency, it is required to continuously enforce all the rules and constraints associated with the data model of the database, between each transaction.

Due to the arbitrary behavior of the network that links every machine of a distributed system, it is infeasible to maintain a system that does not guarantee partition tolerance. So in practice we have to effectively select between strong consistency and availability when designing a distributed system. With this in mind, it is necessary to deeply understand the requirements of our environment, in order to establish the correct balance between consistency and availability.

Currently, most large-scale Internet services aim to offer the best usability to their users. Consequently, they have to attain high availability and hence, sacrifice strong consistency. However, to simplify the design and reasoning of the systems, it is important to provide some consistency guarantees within the spectrum of weak consistency models.

2.1.1 Strong Consistency

A system that enforces strong consistency, has to provide to the user the perception that the system is evolving through a single sequence of states. With this guarantee, whenever a pair of users perform the same read operation, at the same time, the result should be equal for both of them, independently of the replicas that they interact with.

To give this perception to the user, we can adopt some approaches, such as guarantying linearizability or serializability, which are relevant strong consistency models.

Linearizability [12] guarantees that for a single object, whenever a operation is performed, its effects become visible instantaneously. Therefore, once a read returns a certain value, all subsequent reads should return the same value or another that corresponds to a more recent write (considering the wall-clock time).

Serializability [26] is similar to linearizability, but it is usually applied on transactional systems, instead of single operations. A transaction is a group of operations, commonly, containing write and read operations. However, it does not impose any kind of strict order, but requires an execution that is equivalent to a serial execution of all transactions, even if this order does not respect the real time at which transactions were submitted to the system.

Strong consistency allows to easily reason about the evolution of the state of a system, which would be the ideal context to build applications. However, this is an unrealistic scenario in many cases, because it requires to contact a majority of the replicas for every operation, causing a huge overhead in the network. In scenarios where network partition can happen, it might be impractical to achieve strong consistency, since network partitions can make it impossible to contact a majority of the replicas.

2.1.2 Weak Consistency

Considering that strong consistency models are not much attractive for specific user-centric applications, since they are not able to ensure low latency and always available operations, the interest in weak consistency models has increased significantly in the recent past.

Weak (or available) consistency models provide high availability, although they forsake strong consistency. In certain circumstances, we need a considerable level of consistency in order to ensure some kind of arrangement between the operations performed by the clients. For instance, in a system with shared data items and access control based on roles, preserving some sort of temporal agreement between operations is crucial, to enforce the desired semantics of the system.

Regarding this previous example, consider a simple system with a certain data item *i* and two clients *c1*, *c2* that have privileges to access *i*. One example of an execution that can be problematic is: 1) revoke of the access privileges of *c1* on *i*, 2) write operation performed by *c2* on *i*, and 3) read operation performed by *c1* on *i*. If we consider an execution without consistency guarantees and since the two actions that were performed are not atomic, it is possible that *c1* obtains the current state of *i*, even after his privileges to *i* have been revoked, because it can read an outdated version of the access control data associated with *i*.

One of the more popular weak consistency models is eventual consistency, and it is an example of a model that is not able to avoid the issue illustrated above.

Eventual Consistency

Currently, several systems adopt eventual consistency in their architecture, since it provides them with high availability and very low latency. As we can deduce from the name of the model, the only guarantee that it provides is that eventually, when no more write or update operations are made, all replicas of the system will converge to the same state. Consequently, it is possible to observe some obsolete values in portions of the application state (sometimes referred as reading stale data).

Naturally, during the execution of a system providing eventual consistency, different replicas might observe different write operations, leading to their state to diverge. In these cases some mechanism to reconcile such divergent states must be employed. The most popular techniques are:

Last Writer Wins assures that for each set of concurrent operations, the most recent one is chosen to define the final state of the system. This technique can be difficult to implement using clocks, since different replicas can have their clocks desynchronized. Usually logical time is used for the purpose of defining a deterministic order among operations.

Merge Procedure entrusts on the application programmer the responsibility to solve the conflicts. In practice, this is verified by furnishing the replicas with some deterministic heuristic that leads replicas to authentically merge divergent states, using application-level semantics.

CRDTs [5] which stands for Conflict-free Replicated Data Types, are replicated data types that can be integrated into a distributed system. They work as an isolated module that guarantees eventual consistency by internally handling divergent states and reconciling them, discharging that responsibility from the rest of the system.

It has already been recognized that maintaining high availability allied with a satisfactory level of consistency is not a trivial task. With this in mind, the need for increasing the level of consistency in this type of models became increasingly relevant, leading to the appearance of stronger available consistency models.

2.1.3 Enforcing Causality

Recalling the system example that we referred previously, we were not able to guarantee that after the cancellation of the corresponding privileges, the solicitation of *c1* to access *i* would be denied when considering an operation that does not provide any consistency guarantees (offer more than eventual convergence). What we should aspire here is to define a logical relation between the two operations that would prevent such an execution to happen.

Leslie Lamport devoted some of his research tackling the challenges inherent to this particular topic and introduced key concepts [16] that will guide our path towards the model we are pursuing.

Happened-Before Relation ($\rightarrow$)

Given two events *e1* and *e2*, we can define that *e1* happened before *e2* (symbolized by $e1 \rightarrow e2$), if we verify at least one of the following conditions: i) *e1* and *e2* are events in the same process and *e1* preceded *e2*; ii) a certain message *m* was sent by a process *p1* to other process *p2* then *e1* corresponds to the send event on *p1* and *e2* to the receive event on *p2*; iii) given some additional event *e3*, such that $e1 \rightarrow e3$ and $e3 \rightarrow e2$. Therefore, by transitivity $e1 \rightarrow e2$, which indicates that *e1* causally affects *e2*.

Causal order

A system is said to ensure causal order, if all the happened-before relations are preserved across the execution of operations across all replicas of the system.

In our context, an execution of a broadcast protocol that guarantees causal order requires that for each pair of messages *m1* and *m2* respectively sent by *p1* and *p2* such that $broadcast_{p1}(m1) \rightarrow broadcast_{p2}(m2)$, the relation $deliver_p(m1) \rightarrow deliver_p(m2)$ is preserved, for every process p.

Causal consistency

Causal consistency is known as one of the strongest weak consistency models and can be implemented in an available way. Intuitively, this consistency model enforces that clients observe the evolution of the system in a way that respects the causal order among all write operations. It can be also described as a replicated system that enforces the four session guarantees [4]:

1. **Read your Writes:** A client must always be able to observe the effects of all previous writes issued (and for which it got a reply from the system) by itself;

2. **Monotonic Reads:** Subsequent reads issued by the same client should observe either the same state or an inflation of the system state;

3. **Monotonic Writes:** The effects of multiple write operations issued by a given client, must be observed respecting that order by every other client.

4. **Writes follows Reads:** If a client observes the effects of a write operation in its session, then any subsequent write operation issued by that client must be ordered after the previously observed write (which means that no client should be able to observe the effects of that write without observing the effects of all writes that precede it, even in different objects).

Causal+ consistency

Causal+ consistency [23] is obtained from the combination of causal consistency with eventual consistency, or in other words, it is causal consistency with guarantees of convergence of the state across the whole network, at some point in time after no more update operations are issued to the system.

2.1.4 Discussion

As we have been discussing throughout this section, large-scale Internet services pursue to have the best possible usability, which implies both continuous availability and low latency. Consequently, it is necessary to adopt weak consistency models in order to provide high availability.

Our research will target this type of systems and hence, we will focus on solutions that provide causal+ consistency, which is the strongest approach that is employed currently when considering weak consistency models. In practice, we just need to focus on causal consistency dissemination primitives, since we are aiming at designing broadcast solutions and these already guarantee that every node will converge to the same state, if the broadcast process is reliable (i.e., if every correct process receives and delivers all messages).

2.2 Overlay Networks

An overlay network can be defined as a logical network, that is built on top of another network. Each process has a set of neighbors and each neighboring relationship is represented by a logical link. In order for an overlay network to be considered as correct, it has to accomplish the following requirements: 1) for each pair of two correct processes, there must be a sequence of links that results on a path connecting them, and 2) eventually, when a process fails, every other process that had a link to that failed process, will remove it.

The concept of overlay network has an ambiguous meaning, since any logical network implemented on top of another network fits this definition. For instance, peer-to-peer networks are built on top of the Internet, the most common and general network that exists.

In the context of this research, our purpose is to focus on overlay networks that are leveraged to provide some kind of service. In particular, we plan to build our dissemination scheme on top of overlay networks. Hence, we can look at overlay networks as an additional level of abstraction that is implemented over a network, with the intention of increasing the performance and usability of the underlying network.

Furthermore, overlay networks can be categorized by the way they structure their processes and manage their logical links.

2.2.1 Unstructured Overlays

In unstructured overlay networks, as the name suggests, there is no evident structure or meaningful topology that can be identified. Moreover, the neighbors that are linked to a certain node are usually selected by an arbitrary procedure. The neighborhood of each process is usually captured through their local partial view. There are many implementations of this type of overlay and is precisely in this previous point that they mainly differ, that is in the way they maintain and update their partial views.

Scamp [10] is distinguished by its partial view managing mechanism. The partial views are updated when a node joins or leaves the network. Periodically, every node in the network sends an heartbeat message to all the nodes of its partial view. If a long period of time has elapsed since the last heartbeat message arrival in a certain node, then this node rejoins the network, as the lack of heartbeats is most likely an indicator that the node has become isolated in the overlay.

Cyclon [31] has a very different mechanism from the one that Scamp provides. This mechanism relies on a cyclic behavior of exchanging neighbors between each node and its oldest neighbor. In order to perform this exchange, both nodes select a few neighbors from its partial view to send to the other node. Symmetrically, each node collects the selected node information and uses it to update the contents of its own partial view.

HyParView [19] is a sort of a hybrid solution of the two above, extracting the best features of both and presenting better performance and resilience to failures than the previous two. In HyParView, each node has two partial views: 1) a small active view, that supports the main purpose of the network, namely the dissemination of messages, and 2) a wider passive view, whose objective is to provide substitute nodes to the active view in case of failures.

The active view management follows a reactive behavior, similar to Scamp while ensuring that neighbouring relations are symmetric. On the other hand, the passive view is maintained in a way that is similar to the one employed by Cyclon, through a cyclic (or periodic) procedure.

2.2.2 Structured Overlays

In this kind of overlays, a structure that shapes the network can be perceived. Generally, each node is enhanced with an identifier that determines the relative position of the node in the network, in order to enable hashing techniques that will lead to efficient ways of locating a specific resource that is stored in a deterministically selected node. Some concrete techniques are further detailed in the following algorithm descriptions:

Chord [29] uses *consistent hashing* [14] in order to assign each data object, and each node defines its identifier to be in the range of the hash function output (e.g., by hashing its IP and port). This procedure leads Chord to provide the functionality of a Distributed Hash Table (DHT). The network topology can be defined as a ring, as each node is linked to the successor, which is the node with the following (i.e., closest) identifier. Hence, the last node's successor turns out to be the first node, consummating the ring.

Besides the successor's identifier, each node also keeps other useful information, namely a *finger table*. As the name suggests, this table presents some *fingers*, that are a sort of shortcuts to convenient zones of the ring, that make the resource location process and application-level routing more efficient and faster.

Pastry [27] is identical to Chord, since it also follows a DHT philosophy and its network is structured as a ring. Pastry differs from Chord in the identifier generation process, which selects a random hexadecimal regarding a predetermined upper bound, and in the data that each node keeps, as it also stores a *neighborhood set* that maintains information about nodes that are nearby, in terms of network proximity (i.e. low latency nodes).

2.2.3 Partially Structured

The genesis of this classification arises from the lack of efficiency of unstructured overlays on providing adequate support for specific use cases and properties, such as exact match resource location and topology adaptability. Therefore, partially structured overlays implement certain optimization techniques on unstructured overlays. Consequently, this withdraws the total randomness that was verified in the original unstructured overlays. The fundamental pursuit of these overlays is to effectively support those particular use cases, maintaining the benefits that unstructured overlays provide, such as low management overhead and robustness.

T-MAN [13] is known for its ability to adapt any overlay topology to another topology that the user requests. The technique that accomplishes this ensures that every node periodically exchanges their partial views with its neighbors. A merging function is applied, enabling each node to find it target position, in order to achieve the predefined topology. However, this ability to modify the topology can compromise some properties of the primordial overlay, such as the in-degree distribution of nodes, and hence, a few problems can emerge, such as unbalanced load distribution and overall connectivity loss.

X-BOT [20, 22] enables unstructured overlays to bias their topologies concerning a target efficiency metric. It introduces the *oracles*, which are components that are responsible to determine the cost of linking two given nodes, according to the predetermined efficiency criteria. In contrast to T-MAN, X-BOT aims to preserve as much as possible the underlying network's (i.e. HyParView) properties, such as low diameter and clustering, and connectivity.

Plumtree [18] combines tree-based broadcast and gossip primitives. It propagates messages through the dissemination tree, while promoting the use of the remaining links of the underlying overlay in order to recover the tree from failures in a decentralized fashion. This technique allows the protocol to increase fault-tolerance, in contrast to the pure tree-based broadcast protocols, which are not highly resilient, due to the fact that each failure triggers the *tree repair*, which is a slow process. Despite all these positive characteristics, Plumtree does not provide any causal guarantees.

Thicket [6] also uses tree-based broadcast and gossip approaches, like Plumtree does. However, it introduces the technique of combining multiple spanning trees on top of the underlying overlay. This method promotes a balanced load, since the messages being disseminated are split among the set of trees. Furthermore, Thicket presents the possibility of building trees with limited weight, that originates the decrease of the overhead imposed on the trees.

2.2.4 Discussion

Regarding overlay networks, we can conclude that structured overlays are not the best suitable for our goals, since they have lower robustness to failures as a result of the limitations that have to be imposed on the neighborhood to enforce the target topology.

Unstructured and partially structured overlays are easier to implement and maintain, leveraging our goal of supplying a highly dynamic and large network.

Taking into consideration our interest on the integration of tree topology into our protocol, Plumtree and Thicket arise as interesting starting points for our research. Both operate on top of HyParView, which can also be leveraged to design our solution.

2.3 Data Dissemination

In the context of distributed systems, a fundamental concern is how to transmit a given message from a sender process to a receiver process. This procedure must be as fast and safe as possible, therefore many approaches have been discussed. One of the most relevant solutions is the Point-to-Point protocol, that is the responsible layer for data link between the two corresponding nodes. Since most of these transmissions are over the Internet, it became crucial to develop a protocol that deals with the transport layer and was with that purpose that TCP appeared.

Concerning the previous solution for the transmission of a message between two points, the next step is to understand the applications that it might fulfill. We now present two of the most general data dissemination methodologies.

Broadcast

Broadcast is the most general one-to-many communication method. A given message is spread to every device on the network. Despite the simple definition of broadcast, there are several approaches that are employed in order to accomplish this apparently basic task. There are other versions of this technique that only addresses a subset of the whole network, namely multicast, unicast, and anycast. However, we will focus on broadcast, since our goal will focus on this type of primitives.

Publish-Subscribe

This scheme commonly has two types of users: publishers and subscribers. The publisher is responsible to inject new messages into the network, that are categorized into classes.

Each subscriber has a list of classes that it is interested and thereby it only receives the messages that are linked to the classes he has explicitly subscribed to. The filtering process of deciding in which class should a message be inserted can have different behaviors and is precisely on this point that the implementations of this method contrasts among different alternatives. This sort of primitive can be implemented in a centralized way, however, our interest lies on decentralized approaches, such as Scribe, that is a well known solution.

Scribe [28] is one of the most popular implementations of the primitive described above. It lays on top of Pastry, a structured overlay network which was previously discussed. In this solution, every node can subscribe, unsubscribe, create a topic, or publish a new message into the network. Each topic has an identifier, which is defined during the topic's creation, and every message is related to a topic. Scribe totally relies on Pastry to route the messages through the network, leveraging some of Pastry's properties, such as scalability and fault-resilience.

2.3.1 Properties

Depending on the type of data dissemination that is required, there are some explicit properties that can be provided by a broadcast solution. We proceed with the description of some relevant properties and how they are used in practice.

Best Effort

Services that achieves this property do not provide any guarantee about the delivery of the data. As the name suggests, in a best effort solution the dissemination procedure aims to spend the least amount of resources as possible. For instance, an implementation of the best effort broadcast can be simply described as a single transmission from the sender to each of the receivers, without performing any type of validation afterwards or recovery mechanisms for lost messages. Consequently, there are no guarantees of total coverage of the network.

Reliable

In contrast to best effort, this property is able to provide delivery guarantees regarding the transmitted messages. It can be built on top of best effort solutions, so it has to introduce those guarantees by its own. Techniques like infinite retransmissions, *acks* and reception windows are used to surpass several technical challenges that result from implementing this property.

An example in a realistic context is the layering of TCP over IP, a combination that is known as TCP/IP. On this example, TCP focus on ensuring the reliable property and IP the best effort property, since TCP develops a reliable delivery protocol on top of IP, which is unreliable.

Total Order

This property is verified only if every process strictly receives the same sequence of messages in the exact same order. We can easily deduce that this is not a trivial task to accomplish, for instance, it is only possible to achieve in synchronous systems, as it has been proved by FLP[1] [7]. Despite that fact, it is possible to circumvent this issue, developing solutions that only evolve when the system behaves in a synchronous way and halts otherwise, such as Paxos [17].

Causal

As we have previously discussed, causal guarantees have several benefits regarding the levels of consistency on many systems. To reach this property, a network has to assure that all the causal dependencies between events are respected in every node of the network. In this context, the main events are the sending and receiving of the messages and, for instance, every receiving event must appear after the corresponding sending event, in order to secure this specific causal relation.

PRC-broadcast [25] is a causal broadcast implementation that aims to improve the space complexity of the data that forbids multiple delivery. It introduces a new technique that permits to identify the necessary data to maintain, without keeping obsolete data. This technique lays on the determination of the active messages that can be delivered at a given moment and for those messages that were not chosen, it simply discards their data. To determine these obsolete control information, it uses *link memories*. *Link memory* can be defined as a way to guarantee exactly-once delivery while safely removing obsolete data. These *link memories* are the logical relations that are established between the nodes of the network in order to propagate messages among them. Assuming reliable FIFO links to ensure causal order, in a link (a,b), process b remembers among its delivered messages those that it may receive from this specific link and forgets those that it will never receive from it. To accomplish this, in the initialization procedure is necessary to exchange a few control messages with the set of delivered messages of each process. This implementation provides a relevant improvement in the complexity of the data kept in each point of the system.

2.3.2 Techniques

Many specific techniques have been built in order to address the different primitives of data dissemination that are required. Depending on the environment of the network and on the properties that we want to employ, we can develop a distinct technique concerning

[1]FLP proved that consensus problem and all equivalent problems are only possible to solve in synchronous systems. Since total order broadcast is equivalent to the consensus problem, we can conclude that total order broadcast is also impractical in asynchronous systems.

those characteristics. We now discuss two of the main techniques that we identified as the most relevant in the literature.

Gossip

Gossip, or epidemic, protocols have a generic behavior in order to flood a network: When a node receives a certain message, it verifies if it has already delivered this specific message and if he did not, he send it to *t* other nodes. With this approach we can, with a huge probability, make sure that the message will eventually be delivered by every process of the network.

Despite the fact that gossip is not able to give total guarantees that the message will be delivered in every process, in practice, it will eventually happen. Therefore, we can even configure this probability, adjusting the parameter *t*, that is also named *fanout*.

Other adaptable aspect of gossip is the communication mode. The common communication modes are:

- **Eager push:** The sender transmits the entire message in the first interaction with the receiver;
- **Pull:** Every node periodically asks other nodes for new messages, if a node that receives this request has a new message, then it sends the message to the corresponding node;
- **Lazy push:** The sender transmits a message identifier to receiver and if it still does not have that specific message, it asks the sender for that message. Finally, the sender transmits the entire message to the node that asked for the message.

Due to these adjustable features of gossip, it is possible to define different flavors of this method, in order to fulfill the needs of the network environment.

Group Communication

Group communication techniques are employed when a set of processes collaborate to achieve a common goal. Usually, this type of cooperation aims to provide certain services, implement specific algorithms or ensure a given set of relevant properties. We now describe a suitable implementation of this methodology.

Group Communication Service [30] aims to support local workstations, providing an adequate environment to distributed activities that require a group of participants cooperating, such as managing a shared document or interacting with a replicated database. This service focus on three main components: 1) manage the groups memberships, allowing the dynamic creation and reconfiguration of groups, 2) provide efficient support for exchange of information between group members, and 3) supply an execution environment that execute specific algorithms in order to achieve some desirable properties.

2.3.3 Discussion

In this section we examined the state-of-the-art in data dissemination, presenting the relevant properties and techniques related to this topic.

Group communication techniques are not applicable to our target protocol, since this kind of approaches rely on abstractions, such as view synchrony, whose implementations have limited scalability.

Nevertheless, gossip protocols have interesting properties and can be integrated in our solution, namely methods that can ensure an efficient and reliable broadcast implementation.

2.4 Data Replication

Another crucial topic that is essential for the operation of many distributed system is replication. Actually, it is imperative to replicate data among the different nodes of the network, since it addresses some important properties of the system, such as fault-tolerance and load balance.

Nowadays, most distributed systems employ a replication strategy, but there are many different flavors that can be attained. In order to capture some of these, we proceed by presenting some generic categories, in which they are commonly classified.

2.4.1 Full Replication

A system uses full replication if every replica keeps the same entire copy of the system state. When an operation is performed on a certain replica, the effects are propagated to every other replica. Since all the replicas have the same state, read operations might be treated locally, in any of those replicas.

2.4.2 Partial Replication

In opposition to full replication, partial replication solutions have different replicas, containing just a part of the total amount of the system state. Therefore, a strategy has to be outlined in order to conceive replies to the requests that the clients issued, since with this approach, replicas may not have a local copy of a requested data item.

One concrete example of partial replication is caching, since a cache acts like a partial replica, maintaining only data that is often required, but supporting only read operations.

2.4.3 Geo-Replication

Geo-replication is a particular case of replication. It stands for replication schemes that allocate the replicas in different places of the globe. Usually, those locations are chosen wisely, aiming popular areas where clients interacts more frequently with the system.

This kind of strategy is commonly related with dynamic large-scale systems, that are dispersed all over the world. Due to that fact, a large number of replicas might be needed, requiring coordination schemes among the replicas that can overcome the high latency that can be verified on the communication between distant replicas.

2.4.4 Edge Replication & Dynamic Replication

Edge computing [21] is emerging as a very interesting alternative to cloud computing. This tendency brings computing power and memory geographically closer to users, reducing the need of communicating with remote data centers, which widely increases latency. Edge replication arises in this scope, assigning replicas into near infrastructures, such as regional data centers, routers or mobile devices.

In dynamic replication schemes, the location and the number of the replicas can change dynamically according to the operation data requirements and system conditions. The environment associated with edge computing perfectly fits this definition, since edge devices are really dynamic and are only able to supply resources in a heterogeneous way.

2.4.5 Discussion

In the work to be conducted in the thesis we aim at a causal dissemination solution for dynamic large-scale systems and hence, we highlight two of the categories that we described: full replication and geo-replication.

These two classifications can be seen as complementary, since we want to propagate messages across every node of a geo-replicated network. Therefore, full replication schemes are better suited in this context than partial replication approaches.

As we have also stated, edge replication is a recent topic that is emerging and has potential to start to be employed in current systems. The systems will have to adopt themselves to operate with a larger number of replicas, which also motivates our work in this context.

2.5 Relevant Geo-Replicated Storage Systems

This section's goal is to introduce the most convenient geo-replicated storage systems, which can be useful to compare results or to extract some of their features in order to integrate them into our protocol.

2.5.1 COPS & Eiger

COPS [23] was the pioneer system providing causal+ consistency guarantees. It is a distributed key-value storage system that is only capable to run across a small amount of data centers. Each data center is represented as a *COPS cluster* that has a complete copy of the stored data. COPS requires that each *cluster* ensures linearizability. However, in order to guarantee causal+ consistency through the whole system, COPS tracks causal dependencies, maintaining for each operation a list of the dependencies that it has to respect. In order to respect these dependencies, an operation, after its local commit, has to be asynchronously replicated to the other clusters. Once this operation arrives at other cluster, it has to wait until all its dependencies are locally executed before being able to execute the received operation.

Eiger [24] is recognized as an evolution of COPS. It strictly follows the COPS structure, but providing some extra features, such as supporting column data models, implementing a dependency tracking mechanism based on operations and supplying write-only and read-only transactions.

2.5.2 ChainReaction

ChainReaction [1] is a geo-distributed key-value data store that provides causal+ consistency adopting a slightly different version of chain replication. It has an implementation for systems with a single data center and other for systems with multiple data centers. In the first variant, it uses the same metadata as COPS for tracking causal dependencies, although this metadata is only maintained for write operations whose dependencies are not yet stable, softly decreasing the amount of metadata. In the second, it is used all the techniques of the first variant and a version vector with one entry per data center, in order to maintain causality across the different data centers.

2.5.3 Saturn

Saturn [2] is a metadata service that provides causal consistency across different geo-replicated data services. The main focus of Saturn is to solve two problems: 1) eliminate the trade-off between throughput and data freshness that other solutions can't address, and 2) fully benefit from partial geo-replication, requiring datacenters to manage only metadata concerning items replicated locally. Saturn pursues an efficient management of metadata, keeping it with small and constant size. To address this goal, Saturn implements a technique to propagate the operations, based in a global dissemination tree interconnecting all data centers.

2.5.4 C^3

C^3 [8] is a replication scheme that offers causal+ consistency in partial geo-replicated scenarios. C^3 explicitly divides itself in two layers: the causality layer and the data store layer. The first layer propagates causality tracking information across the replicas. This information is named as *labels* and exists a *label* for each write operation, containing an unique identifier and the corresponding dependencies. The data store layer executes the operations in the local data center and propagates them to the others data centers. The two layers have to agree when an operation can be executed in order to respect all the causal dependencies.

2.5.5 Cassandra

Apache Cassandra [15] is a popular distributed storage system that ensures high scalability and fault-tolerance. It was developed and used by Facebook, resulting in a highly available service. As it can be deduced from the huge amount of data that Facebook processes, its platform has strict operational requirements in terms of performance and efficiency. Cassandra was designed based on the design of Dynamo to offer similar features. However, the price paid to provide this in Cassandra is the lack of consistency guarantees that the system offers, since Cassandra only ensures eventual consistency.

2.5.6 Discussion

Saturn and C^3 can be perceived as plugins to storage systems, contrasting with COPS, Eiger, and ChainReaction that are authentic storage systems, in addition to provide causal guarantees.

Our protocol can be used as a building block for systems that operate at the same level of Saturn and C^3, since we aim to implement a protocol that can be attached to systems that do not have any mechanisms to track causality. With this in mind, we will deeply investigate each of the approaches, with special attention to Saturn and C^3.

2.6 Discussion

Throughout this chapter, we have been able to study many essential topics and, for each of them, we identified the most important concepts and designs to further explore in the next phase of this project.

In first section, we verified the usefulness of causality in consistency models, concluding that we will pursue causal consistency. Regarding overlay networks, we discarded structured approaches and targeted more interesting and scalable unstructured overlays. We proceeded by investigating significant properties and techniques of data dissemination and the main outcome was the identification of gossip techniques as a necessary ingredient to our solution.

Finally, we assimilated that full geo-replication has to be our approach and defined that Saturn and C^3 will be our main influences, in understanding how a causal dissemination primitive can be employed.

2.7 Summary

In this chapter we explored several key topics regarding our research, leading to the selection of the most suitable approaches for the desired protocol.

Modern distributed systems are increasing in number and size at a breathtaking pace. With this abrupt growth, their performance is depreciating and urgent solutions are required in order to maintain consistency guarantees combined with highly available services. With this in mind, we chose to focus on causal consistency models, since they are able to ally both demands. Therefore, our first concern was to present all the important concepts and definitions related with causality.

Besides causality, it is also crucial to deeply understand how to efficiently disseminate a message across an entire network. Consequently, we presented the generic methodologies regarding this wide topic, such as overlay networks and dissemination primitives.

We continued by describing some pertinent categories of replication in order to perceive what kind of approaches we should adopt, considering this theme. One of the categories we introduced is geo-replication, which is going to be our main target type of distributed systems. Hence, in our last section we presented a few geo-replicated storage systems that we might use to attach our solution and compare performances.

Chapter 3

CPTCast: Causal Plumtree Broadcast

In this chapter, we propose a novel protocol, named CPTCast, that manages causal dissemination on large-scale distributed systems that are able to mutate into a tree topology.

We begin by presenting the expected properties of the systems where CPTCast will be capable to operate (Section 3.1) and the requirements for our solution that were initially outlined (Section 3.2). We proceed by specifying the general architecture of the solution (Section 3.3), followed by all the implementation details (Section 3.4), such as the techniques that are implemented in order to manipulate the tree and the approach that is used to handle the dynamic membership, preserving the causality of the system. Lastly, we provide a brief summary (Section 3.5) of the present chapter.

3.1 System Model

In order to CPTCast to behave properly, we consider that the system on which it will be operating must fulfill some specific properties. In particular, we consider a system model, where processes communicate through the exchange of messages. Each process of the system can be divided into three primary components: i) **application**, ii) **broadcast**, and iii) **membership**. In this context, CPTCast is classified as a broadcast protocol and it cooperates with the other components to allow the system to perform its operation.

We assume that the connections between each pair of processes provides **FIFO ordering** and these links are established and managed by an appropriate membership protocol. This membership protocol must ensure some crucial properties, in particular: i) preserve the **connectivity** of the overlay network, despite any failure that might occur ii) support a

high scalability of the network, since we are aiming to integrate large distributed applications iii) promote **reactive membership**[1], that will allow the overlay to converge to a stable state, preserving the links that are used in the broadcast protocol. This is essential to approach tree topologies, because when a link is created or destroyed, that leads to a tree repair step, which is a slow process.

The machines where each process will be operating must grant an **accurate timestamp** or any feature that provides a reliable counter for their own broadcasts. This requirement is related with the need to generate distinct message identifiers.

3.2 Requirements

From the study of the state-of-the-art, we determined a proper set of requirements that should be accomplished, in order to validate our protocol:

- **Causal Dissemination:** The solution has to perform dissemination in a reliable and efficient way, enforcing causal delivery across the entire system;
- **Support for Highly Dynamic Networks:** Many nodes will join and leave the network frequently and, consequently, it is essential to build an agile protocol that supports these frequent changes on the system's membership;
- **Low Metadata Overhead:** A large amount of metadata has to be kept in order to implement causal dissemination primitives, although our solution will seek to minimize this amount of metadata as much as possible;
- **Low Network Overhead:** The solution should strive to lower as much as possible the overheads, both in terms of network usage and CPU consumption, as to minimize the impact on applications operations on top of this protocol;
- **High Scalability:** The network has to be prepared to grow and keep the performance adequate with the number of nodes.

3.3 Architecture

In Section 3.1, we mentioned the inner structure of a process, introducing the three main components that form the stack of protocols in each process. Notice that all processes are similar regarding this stack. Figure 3.1 presents a simple diagram that summarizes how these components relate to each other.

[1]Reactive membership preserves the stability of the partial view of each node, since it only promote changes in the views, when a node joins or leaves the overlay.

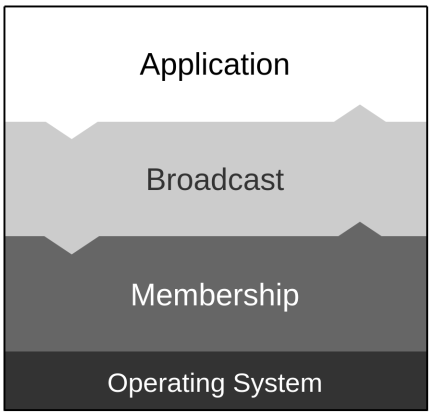


Figure 3.1: Simple architecture of a single process

The **application** layer can have any kind of purpose, concerning what the user demands for the system itself. However, it must require to perform broadcasts at some point. To execute that, it may request the underlying layer, the broadcast layer, to perform a broadcast of a certain message.

Therefore, the **broadcast** layer is intended to deliver to the application layer, those messages that arrive following a broadcast. The essential goal of this component is to disseminate messages according to the broadcast protocol that it attends.

Finally, the **membership** layer is responsible to manage the overlay network, providing the links that might be used by the broadcast protocol to disseminate messages. Likewise the upper layer, this component should follow a certain protocol, that will specify how each node is supposed to behave.

Despite the straightforward cooperation between the first two layers, the broadcast and the membership layers exchange a more complex set of interactions. Hence, we summarize this relationship by describing the events that each protocol triggers to the other protocol handle.

Broadcast → Membership

- `SEND(msg, node);`

  The broadcast layer requests the membership layer to send `msg` to `node`.

- `SWITCH_NEIGHBOUR(node);`

  The broadcast layer requests the membership layer to remove `node` from the view and replace it with another unspecified node.

Membership → Broadcast

- `RECEIVE(msg, node);`

  The membership layer requests the broadcast layer to receive, and consequently deliver, `msg` from `node`.

- `NEIGHBOUR_UP(node);`

  The membership layer notifies the broadcast layer that `node` joined its view.

- `NEIGHBOUR_DOWN(node);`

  The membership layer notifies the broadcast layer that `node` left its view.

3.4 Implementation Details

CPTCast stands for Causal Plumtree Broadcast, due to the fact that our protocol is inspired by the tree construction mechanisms that Plumtree [18] has introduced. Allied to the ability to build a tree topology, we leveraged techniques that ensure causal consistency across the entire system.

As we follow a gossip strategy, which has natural redundancy in the dissemination process, it is necessary to ensure that each message is delivered just once, even though each node expects to receive multiple copies of every message. Consequently, it is required to implement a technique that forbids multiple delivery of a single message. The most common approach to deal with this issue is to save control information that remembers every messages that were already delivered. However, this method leads to a large amount of control information that might be discarded.

If we guarantee that every link only carries each message once, a certain node can destroy the control information regarding a specific message, once that node received that message from every neighbour. To guarantee that every link only carries each message once, we have to ensure that every node only disseminates each message exactly one time. Our approach to tackle this issue is by certifying that every node forwards each message on the moment it arrives for the first time. It is also on this instant that each process delivers the message to the application. The broadcaster node behaves equivalently to the remaining nodes: once a broadcast is requested, it delivers the message and disseminates it to its neighbours.

Once a node n receives a repeated message m from a neighbour n', it acknowledges that m has been transmitted by n'. As soon as all the neighbours become checked, n can finally discard the data kept related to m. By exploiting this strategy, we are able to dismiss multiple delivery, employing a data structure that does not monotonically increase.

Algorithm 1 depicts a simple implementation of causal broadcast for static systems[2] following this strategy that improves the space complexity of the solution. Assuming that the network provides reliable FIFO connections, causal consistency is ensured in this context of a static system [9].

Algorithm 1: Naive causal broadcast for static systems

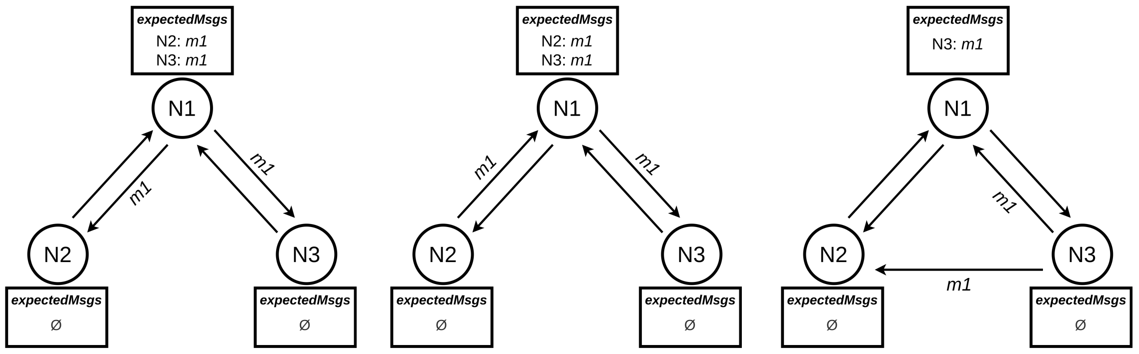
```
1  inView //set of incoming peers
2  outView //set of outgoing peers
3  expectedMsgs //map that keeps for each peer p, a set of the messages expected to arrive from p
4  upon init
5      inView ← ∅
6      outView ← ∅
7      expectedMsgs ← ∅
8  upon broadcast(msg)
9      mID ← concat(self, SYSTEM_TIME)
10     call receiveMsg(⟨GOSSIP, mID, msg⟩, ⊥)
11 procedure receiveMsg(message, sender)
12     ⟨_, mID, msg⟩ ← message
13     //if message was not received yet
14     if ∄n ∈ inView : mID ∈ expectedMsgs[n] then
15         trigger deliver(msg)
16         foreach n ∈ inView do
17             if n ≠ sender then
18                 expectedMsgs[n] ← expectedMsgs[n] ∪ {mID}
19         call forward(message, sender)
20     else
21         expectedMsgs[sender] ← expectedMsgs[sender] \ {mID}
22 procedure forward(msg, sender)
23     foreach n ∈ outView do
24         call sendMsg(msg, n)
25 procedure sendMsg(msg, node)
26     trigger send(msg, node)
```

To univocally identify each message, the broadcaster node constructs an identifier by concatenating its address and current timestamp (Alg.1, line 9). When a broadcast is requested, the node acts like it had received that message and proceeds to forward the message. However, there is a meaningful difference in this specific scenario, which is the absence of a *sender* (Alg.1, line 10).

[2]Static systems are defined by managing a constant set of nodes during the whole execution time. Additionally, each node holds the same group of neighbours.

As previously mentioned, when a node *n* receives a message *m*, *n* verifies if this was the first time it received *m* (Alg.1, line 14). If it was the first time, *n* progresses by delivering *m*, marking *m* as expected from the remaining incoming links and, finally, forwarding *m* through all its outgoing links (Alg.1, lines 15-19). If it was not, *n* simply acknowledges that *m* has already been sent by the corresponding neighbour (Alg.1, line 21).

This approach seems to implement a causal broadcast solution in a very straightforward way. However, it will only work on static systems, which are not the kind of systems we are pursuing. We aim to be able to employ our protocol in large-scale networks, where nodes are joining or leaving constantly. Figure 3.2 depicts an execution of this solution on a dynamic system, endorsing its inability to prevent multiple delivery on this kind of environment.



(a) N1 broadcasts *m1* and expects to receive it from N2 and N3

(b) N2 receives and forwards *m1*. N2 does not expect to receive *m1* again

(c) N3 adds a link to N2 and then N3 receives and forwards *m1*

Figure 3.2: Naive causal broadcast (Alg.1) on dynamic systems

Figure 3.2 is the aggregate of three consecutive events that lead to the delivery of a repeated message, or in other words, multiple delivery. Firstly, N1 is requested to broadcast *m1* and, consequently, forward it to all its neighbours (N2 and N3) and expect a copy from them (Fig.3.2a). N2 immediately receives and forwards *m1* to the only neighbour it has, which is N1 (Fig.3.2b). N2 does not expect to receive *m1* again since the sender (N1) is its only neighbour. Due to an unpredictable delay on the network communication between N1 and N3, the receipt of *m1* on N3 occurs after the creation of a new link that connects N3 to N2 (Fig.3.2c). Therefore, once N3 receives *m1*, forwards it to its neighbours and, hence, causing N2 to mislead *m1* as a new message since it was not expecting to receive *m1* from N3.

A single occurrence of multiple delivery is not the only problem by itself, because it will trigger a chain of effects throughout the whole network, such as generating extra retransmissions that lead to additional redundant deliveries, violating causal consistency and therefore, compromising the validity of the solution.

3.4.1 Dynamic Membership Mechanisms

In dynamic systems, nodes are constantly joining and leaving the network. Therefore, the network is often reconfiguring itself, which leads to the establishment of new links. As a result, it is necessary to safely create those new links to avoid issues, as highlighted in Figure 3.2.

When a new link is announced by the membership layer, it is immediately disabled and recognized as unsafe. By suspending a recently created link, the corresponding nodes can progressively integrate this link, without compromising the consistency of their actual state. This integration employs some techniques, namely message buffering and round trips of control messages. In our solution, the procedure of setting up a safe link is based on the proposal employed in PRC-broadcast [25].

Discovery mechanism

To initialize a link *N1*→*N2*, as soon as *N1* is notified by the membership protocol of its new link to *N2*, it prompts the *discovery* mechanism as specified by Algorithm 2. The goal of this mechanism is to find a safe path, which is a sequence of safe links, that connects *N1* to *N2*. Once the path is constituted, the two nodes can start to exchange the necessary control messages in order to safely synchronize their states.

Algorithm 2: Handling changes on the membership - NEIGHBOUR_UP (1/2)

```
eagerPushPeers //set of outgoing peers on eager push mode
lazyPushPeers //set of outgoing peers on lazy push mode
unsafeIn //map that keeps for each unsafe incoming peer u, the last control message sent to u
unsafeOut //map that keeps for each unsafe outgoing peer u, the last control message sent to u
traversals //set of the DISCOVERY and REDISCOVERY messages that traversed this node
upon init
    $eagerPushPeers \leftarrow \emptyset$
    $lazyPushPeers \leftarrow \emptyset$
    $unsafeIn \leftarrow \emptyset$
    $unsafeOut \leftarrow \emptyset$
    $traversals \leftarrow \emptyset$
upon neighbourUp(node, isJoin)
    //if the new neighbour is joining the system by contacting this node
    if isJoin then
        $eagerPushPeers \leftarrow eagerPushPeers \cup \{node\}$
        $inView \leftarrow inView \cup \{node\}$
    else
        setup DiscoveryTimer(node, 0, DISCOVERY_TIMEOUT)
        $mID \leftarrow$ concat(self, SYSTEM_TIME)
        $unsafeOut[node] \leftarrow mID$
        $traversals \leftarrow traversals \cup mID$
        call flood($\langle DISCOVERY, mID, self, node, mID, \emptyset, \emptyset \rangle, \perp$)
```

In Algorithm 1, we defined a structure *outView* that keeps all the outgoing links to the neighbours of a certain node. However, in CPTCast we employed a strategy to adapt the network to a tree topology. For that reason, we split the *outView* into two different sets: *eagerPushPeers* and *lazyPushPeers*. We further detail the purpose of each data structure in Section 3.4.2.

If the new link *N1*→*N2* is introduced following the join of *N1* or *N2* to the system, the *discovery* mechanism is not applied. In this scenario, this new node is unable to violate any protocol properties, since it is not aware of any other node besides its contact node[3] and it had not any contact with the network before. Consequently, the link is instantly declared as safe (Alg.2, lines 14-16).

However, if the new link does not follow a join of a node to the network, the *discovery* mechanism is prompted. The initial step of this mechanism leads process *N1* to flood[4] the network with a *discovery* message towards *N2*. The content required in this *discovery* message (Alg.2, line 22) is the following: i) message identifier, ii) *N1*'s address, iii) *N2*'s address, iv) preceding message identifier, v) *N1*→*N2* path, and vi) *N2*→*N1* path. The *discovery* message traverses the network until it reaches *N2*, saving each safe link that it crossed into *N1*→*N2* path. Once it arrives at *N2*, the path is marked as completed. This path only can be trusted to travel on *N1*→*N2* direction, since we are not able to guarantee that every link is bidirectional. For that reason, *N2* triggers another *discovery* message in the opposite direction, although on this travel, it is the *N2*→*N1* path that needs to be completed. Finally, when this last *discovery* message reaches *N1*, the procedure had constituted safe paths for both directions and, hence, the *discovery* mechanism had successfully concluded.

Algorithm 3: Handling changes on the membership - NEIGHBOUR_UP (2/2)

```
1  upon DiscoveryTimer(node, version)
2      if $version < MAX\_DISCOVERIES$ then
3          setup DiscoveryTimer(node, version+1, DISCOVERY_TIMEOUT)
4          $mID \leftarrow$ concat($self$, SYSTEM_TIME)
5          $unsafeOut[node] \leftarrow mID$
6          $traversals \leftarrow traversals \cup mID$
7          call flood($\langle DISCOVERY, mID, self, node, mID, \emptyset, \emptyset\rangle, \perp$)
8      else
9          trigger switchNeighbour(node)
10 procedure flood(msg, sender)
11     foreach $n \in eagerPushPeers \cup lazyPushPeers : n \neq sender$ do
12         trigger send(msg, n)
```

[3] A contact node is assigned to each node that is joining the system to introduce this new node into the network.

[4] Flooding is a message dissemination approach that leads every incoming message to be sent through every outgoing link except the one it arrived on.

Additionally, the *discovery* mechanism is supplied with some techniques that improve its performance and reliability. When a *discovery* is sent, a timer is defined (Alg.2, line 18) to trigger the resend of the *discovery* message, once the time limit is exceeded. With this technique, it is possible to attempt to find each safe path multiple times. Other technique is employed with the *traversals* set, which memorizes every *discovery* message that has traversed the corresponding process. Consequently, the process only forwards a *discovery* if it had not been already inserted into *traversals* (Alg.4, line 5). Lastly, the *unsafeOut* and *unsafeIn* structures save the last control message sent to each unsafe neighbour. Accordingly, the first is related to the outgoing links and the last to the incoming. This technique intends to forbid any delayed control message to interrupt the current control message exchange. Therefore, every control message should display the expected preceding message identifier. Otherwise, the message will be ignored (Alg.4, line 19).

Algorithm 4: *Discovery* mechanism

```
1  safePaths //map that keeps for each unsafe peer u, a path consisted of safe links to u
2  upon init
3      safePaths ← $\emptyset$
4  upon receive(⟨DISCOVERY, mID, from, to, prevMID, path, wayBack⟩, sender)
5      if mID $\notin$ traversals then
6          traversals ← traversals ∪ {mID}
7          path ← path ∪ {self}
8          if self = to then
9              mID2 ← concat(self, SYSTEM_TIME)
10             //if DISCOVERY is going on from->to direction
11             if wayBack = $\emptyset$ then
12                 //if this DISCOVERY results from a preceding REDISCOVERY
13                 if mID ≠ prevMID then
14                     unsafeOut ← unsafeOut \ {from}
15                     safePaths ← safePaths \ {from}
16                 unsafeIn[from] ← mID2
17                 call flood(⟨DISCOVERY, mID2, self, from, mID, $\emptyset$, path⟩, ⊥)
18             //if DISCOVERY is going on to->from direction
19             else if unsafeOut[from] = prevMID then
20                 unsafeOut[from] ← mID2
21                 safePaths[from] ← wayBack
22                 call sendCtrlMsg(⟨$\alpha$, mID2, self, from, mID, wayBack, path⟩)
23         else
24             call flood(⟨DISCOVERY, mID, from, to, prevMID, path, wayBack⟩, sender)
```

Now that we have presented how safe paths are built, we proceed by explaining their purpose. As we previously mentioned, our approach to coordinate the state of the nodes that have recently established a new link relies on exchanging a sequence of control messages through safe paths.

Control messages exchange

By exploiting the safe paths that were constructed by the *discovery* mechanism, it becomes possible to exchange control messages respecting the causal guarantees of the system, since the safe links enable a causal order of the messages that are being disseminated. Algorithm 5 describes how processes behave to conduct the control messages through all the safe links that constitute each safe path.

Algorithm 5: Control messages sending and *rediscovery* mechanism

```
1  procedure sendCtrlMsg($msg$)
2      $\langle \_, \_, from, to, \_, path, \_ \rangle \leftarrow msg$
3      $next \leftarrow poll(path)$ //retrieves and removes the head of the queue
4      //if path defined by DISCOVERY mechanism is not adulterated
5      if $next \in eagerPushPeers \cup lazyPushPeers$ then
6          call sendMsg(msg, next)
7      else
8          $mID2 \leftarrow$ concat($self$, SYSTEM_TIME)
9          $traversals \leftarrow traversals \cup \{mID2\}$
10         call flood($\langle REDISCOVERY, mID2, from, to \rangle, \perp$)
11 upon receive($\langle REDISCOVERY, mID, from, to \rangle, sender$)
12     if $mID \notin traversals$ then
13         $traversals \leftarrow traversals \cup \{mID\}$
14         if $self = from \land to \in unsafeOut$ then
15             $safePaths \leftarrow safePaths \setminus to$
16             $mID2 \leftarrow$ concat($self$, SYSTEM_TIME)
17             $unsafeOut[to] \leftarrow mID2$
18             $traversals \leftarrow traversals \cup \{mID2\}$
19             setup DiscoveryTimer(to, 0, DISCOVERY_TIMEOUT)
20             call flood($\langle DISCOVERY, mID2, self, to, mID, \emptyset, \emptyset \rangle, \perp$)
21         else call flood($\langle REDISCOVERY, mID, from, to \rangle$, sender)
```

A safe path can be defined as a sequence of safe links and, concerning that, in order to transport a control message, each node of the path directly forwards the control message to the next node in the path (Alg.5, lines 5-6). However, if the membership protocol discards a certain link of the safe path, it becomes obsolete. Therefore, a *rediscovery* message is transmitted towards the origin of the link that is being integrated (Alg.5, line 10). This *rediscovery* mechanism is very simple, it traverses the networks similarly to the *discovery* mechanism and once it arrives at the origin of the corresponding link, it restarts the *discovery* mechanism (Alg.5, lines 11-21).

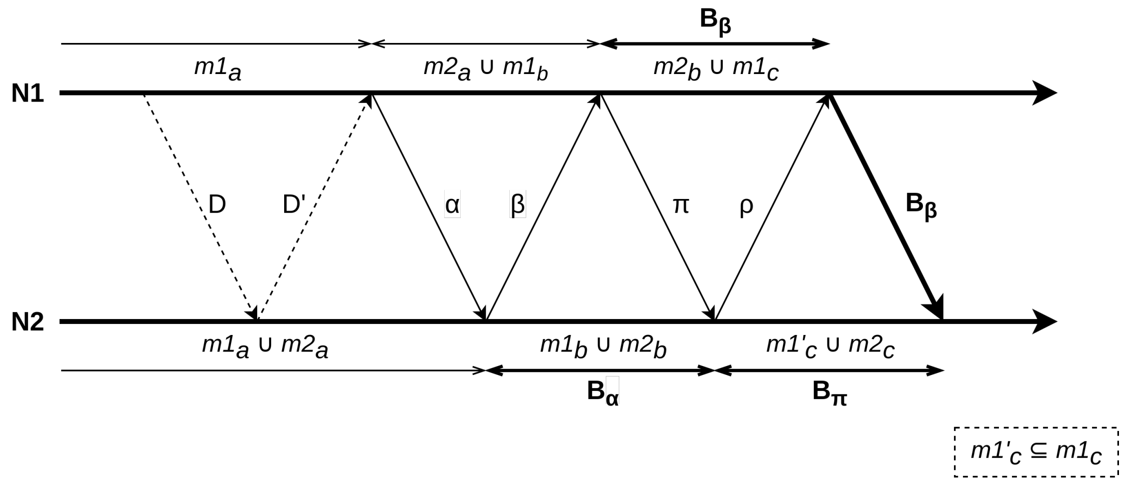


Figure 3.3: Control messages exchange regarding new link *N1*→*N2*

Figure 3.3 illustrates the complete flow of a control messages exchange, which emerges from the initialization of a new link *N1*→*N2*. As the diagram depicts, after a round trip of *discoveries*, *N1* prompts the control messages exchange by sending a notification α to *N2* (Alg.4, line 22). By transmitting α through safe links, we ensure that α follows a causal order. Therefore, when *N2* receives α, it is implicitly guaranteeing that all messages delivered by *N1* ($m1_a$) had previously been delivered by *N2*. Since *N2* had already received α, it can start buffering into B_α (Alg.6, line 7) all the broadcast messages that arrive from now on. This message buffering method triggers the identification of concurrent messages and hence, the possibility to decide the purpose for each message: i) **deliver** it to the application component, ii) **expect** to receive it from neighbours, or iii) **ignore** it. To proceed with the exchange, *N2* acknowledges the receipt of α by sending a notification β to *N1* (Alg.6, line 11).

Algorithm 6: Control messages exchange - α

```
1  rBuffers //map that keeps for each unsafe peer u, a buffer with messages to receive from u
2  upon init
3      rBuffers ← ∅
4  upon receive(⟨α, mID, from, to, prevMID, path, wayBack⟩, _)
5      if self = to then
6          if unsafeIn[from] = prevMID then
7              rBuffers[from] ← ⟨∅, ⊥, false⟩
8              safePaths[from] ← wayBack
9              mID2 ← concat(self, SYSTEM_TIME)
10             unsafeIn[from] ← mID2
11             call sendCtrlMsg(⟨β, mID2, from, to, mID, safePaths[from]⟩)
12     else
13         call sendCtrlMsg(⟨α, mID, from, to, prevMID, path, wayBack⟩)
```

At the arrival of β, *N1* implicitly ensures that it had already received every message that *N2* delivered before the sending of β ($m1_a \cup m2_a$). With these changes, we can prevent the issue that Figure 3.2 illustrates since $m1$ would match $m1_a$ or $m2_a$. Nevertheless, the initialization of the new link is not finished yet. Equivalently to the handling of α, *N1* starts gathering every message following β's arrival (Alg.7, line 7). However, the messages are inserted into another buffer, named B_β. *N1* also replies with another notification, entitled as π (Alg.7, line 10).

Algorithm 7: Control messages exchange - β

```
1  sBuffers //map that keeps for each unsafe peer u, a buffer with messages to send to u
2  upon init
3      sBuffers ← ∅
4  upon receive(⟨β, mID, from, to, prevMID, path⟩, _)
5      if self = from then
6          if unsafeOut[to] = prevMID then
7              sBuffers[to] ← ∅
8              mID2 ← concat(self, SYSTEM_TIME)
9              unsafeOut[to] ← mID2
10             call sendCtrlMsg(⟨π, mID2, from, to, mID, safePaths[to]⟩)
11     else
12         call sendCtrlMsg(⟨β, mID, from, to, prevMID, path⟩)
```

The exchange progresses as *N2* collects π. With this notification, *N2* closes its first buffer B_α and starts storing the subsequent delivered messages into a new buffer B_π (Alg.8, lines 4-5). This buffer may have some of the messages that *N2* should expect from *N1*, but *N2* is not aware of which at that moment. For that reason, *N2* advances with a final notification ρ to *N1* (Alg.8, line 8).

Algorithm 8: Control messages exchange - π

```
1  upon receive(⟨π, mID, from, to, prevMID, path⟩, _)
2      if self = to then
3          if unsafeIn[from] = prevMID then
4              ⟨B_α, _, _⟩ ← rBuffers[from]
5              rBuffers[from] ← ⟨B_α, ∅, true⟩
6              mID2 ← concat(self, SYSTEM_TIME)
7              unsafeIn[from] ← mID2
8              call sendCtrlMsg(⟨ρ, mID2, from, to, mID, safePaths[from]⟩)
9      else
10         call sendCtrlMsg(π, mID, from, to, prevMID, path)
```

After receiving ρ, *N1* can finally close its buffer, send it to *N2* through the new link and discard all the control information regarding this initialization (Alg.9, lines 4-10). The new link is suitable to transmit B_β since we know that it will arrive before any subsequent broadcast message and B_β carries the remaining information that is required to securely initialize this link on *N2*. Henceforth this link can be safely employed for dissemination.

Algorithm 9: Control messages exchange - ρ

```
1  upon receive(⟨$\rho$, mID, from, to, prevMID, path⟩, _)
2      if self = from then
3          if unsafeOut[to] = prevMID then
4              msg ← ⟨$B_\beta$, mID, sBuffers[to]⟩
5              trigger send(msg, to)
6              eagerPushPeers ← eagerPushPeers ∪ {to}
7              sBuffers ← sBuffers \ {to}
8              unsafeOut ← unsafeOut \ {to}
9              safePaths ← safePaths \ {to}
10             cancel DiscoveryTimer(to)
11     else
12         call sendCtrlMsg(⟨$\rho$, mID, from, to, prevMID, path⟩)
```

Upon the receipt of B_β, *N2* stops buffering into B_π and starts processing the buffers. With the three buffers (B_α, B_β and B_π), *N2* can diagnose the purpose for each message in the buffers B_β and B_π, concerning that from the sending of ρ, every message in B_α has already been implicitly certified to respect causal properties. The messages that *N2* should deliver are in $B_\beta \setminus B_\alpha \setminus B_\pi$ (Alg.10, lines 4-5) and it must expect to receive from *N1* the content of $B_\pi \setminus B_\beta$ (Alg.10, line 6). All the remaining messages are ignored, which are contained in $B_\beta \cap (B_\alpha \cup B_\pi)$.

Algorithm 10: Control messages exchange - B_β

```
1  upon receive(⟨$B_\beta$, prevMID, $buffer_\beta$⟩, sender)
2      if unsafeIn[sender] = prevMID then
3          ⟨$B_\alpha$, $B_\pi$, _⟩ ← rBuffers[sender]
4          foreach msg ∈ $buffer_\beta \setminus B_\alpha \setminus B_\pi$ do
5              call receiveMsg(msg, sender)
6          expectedMsgs[sender] ← $B_\pi \setminus buffer_\beta$
7          inView ← inView ∪ {sender}
8          rBuffers ← rBuffers \ {sender}
9          unsafeIn ← unsafeIn \ {sender}
10         safePaths ← safePaths \ {sender}
11         cancel DiscoveryTimer(sender)
```

The control message exchange is completed and thereby the new link *N1→N2* is fully integrated in the network. Now that we have described how we handle changes on the membership preserving causal dependencies, we proceed by explaining the implementation of the mechanisms that shape the overlay into a tree and how the message dissemination works on this scenario.

3.4.2 Broadcast Tree Management & Dissemination Primitives

As we have been mentioning throughout this document, our broadcast model establishes a tree among all peers, because this allows to decrease substantially the message complexity and support the execution of relevant algorithms and methodologies. However, they can not offer high fault-tolerance since any change on the overlay triggers the reconfiguration of the whole tree. To tackle this issue, we focused on the mechanisms that Plumtree [18] has originally proposed to reconfigure its tree in the presence of faults, which achieves an efficient trade-off between epidemic and tree-based dissemination primitives.

Those mechanisms introduce the division of each partial view in two disjoint subsets: i) *eagerPushPeers*, and ii) *lazyPushPeers*. As the names suggest, these structures will separate the neighbours of each node by the communication mode that it will employ when disseminating messages to them. The broadcast tree will be constituted by the links where eager push gossip is used and the remaining links, the lazy push links, are leveraged to achieve dissemination reliability by promoting those links to become eager push in the presence of faults. When a node is added to a partial view, it is inserted in *eagerPushPeers* by default (Alg.2, line 15; Alg.9, line 6). Therefore, in the beginning, the protocol will behave as a pure gossip protocol. Nevertheless, the tree will acquire shape, as the neighbours that transmit redundant messages are moved to the *lazyPushPeers* set. The exchanges between the two sets are performed according to Algorithm 11, which describes the procedure of tree construction.

When a broadcast message arrives, each node verifies if the message was expected and operates as the following rules specify: i) if a node receives a message for the first time, the sender is inserted or maintained in *eagerPushPeers* (Alg.11, line 6), or ii) if a node receives a repeated message, the sender enters or remains in *lazyPushPeers* and a *prune* message[5] is sent to the sender (Alg.11, line 10). Consequently, after the first broadcast, the eager push links will converge to a tree topology. The tree will keep its shape, as long as the overlay remains stable and the broadcasts are performed by the same node.

As any regular gossip protocol, once a node receives a message for the first time, it delivers and forwards the message. However, on this method, the messages are not eagerly disseminated towards every neighbour, since the lazy push links require the use of a slightly different behaviour.

[5]When a node receives a *prune* message, it moves the specified link to the *lazyPushPeers* (Alg.11, lines 21-23).

Algorithm 11: Broadcast tree construction

```
grafted //map that keeps for each message m, a set of the peers to which this node grafted m
upon init
    $grafted \leftarrow \emptyset$
upon receive($\langle GOSSIP, mID, msg \rangle$, sender)
    if $\nexists n \in inView : mID \in expectedMsgs[n]$ then
        call elongate(sender)
        trigger deliver(msg)
        call forward(mID, msg, sender)
    else
        call prune(sender)
        $expectedMsgs[sender] \leftarrow expectedMsgs[sender] \setminus \{mID\}$
        $grafted[mID] \leftarrow grafted[mID] \setminus \{sender\}$
        call checkWaiting(mID, sender)
procedure elongate(node)
    $eagerPushPeers \leftarrow eagerPushPeers \cup \{node\}$
    $lazyPushPeers \leftarrow lazyPushPeers \setminus \{node\}$
procedure prune(node)
    $eagerPushPeers \leftarrow eagerPushPeers \setminus \{node\}$
    $lazyPushPeers \leftarrow lazyPushPeers \cup \{node\}$
    trigger send($\langle$PRUNE$\rangle$, node)
upon receive($\langle PRUNE \rangle$, sender)
    $eagerPushPeers \leftarrow eagerPushPeers \setminus \{sender\}$
    $lazyPushPeers \leftarrow lazyPushPeers \cup \{sender\}$
```

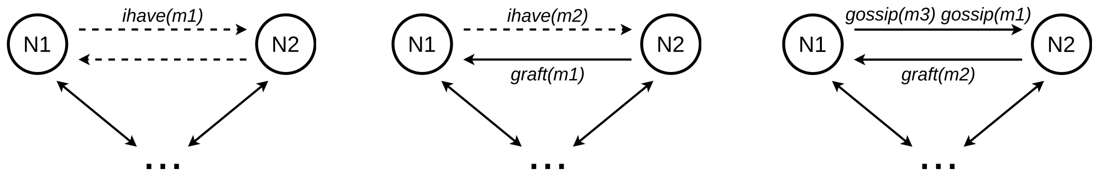
To perform a lazy push, the node sends an *ihave* message to its *lazyPushPeers* to warn them that it has the message that matches the identifier enclosed to the *ihave*. When an *ihave* arrives at a certain node and it refers an unexpected (i.e., new) message, then the node sets a timer and registers that the corresponding message is missing (Alg.12, lines 5-8). Once the broadcast message is received, the node cancels the timer and marks that the message is not missing anymore. The timeout value should be configured at deployment time and must be defined concerning the expected size of the system and the network latency.

Due to the unpredictable behaviour of the network, many events might delay a scheduled transmission, such as a different node performing the broadcast or a link removal. Therefore, if a timer expires at a specific node before it receives the corresponding message, the protocol needs to proceed by replacing the obsolete tree branch. For that purpose, the node selects the first neighbour that announced to have the missing message (Alg.12, line 14). Afterwards, it appends the corresponding link to the tree to replace the obsolete tree branch (Alg.12, line 15). Finally, it sends a *graft* message to the selected neighbour (Alg.12, lines 16-17) to trigger the corresponding message transmission. Furthermore, another timer is set (Alg.12, line 13) to guarantee that an extra *graft* will be issued to an alternative neighbour if the previous *graft* does not succeed in the meantime.

Algorithm 12: Broadcast tree repair

```
1  missing //map that keeps for each message m, a queue of the peers that announced to have m
2  upon init
3      $missing \leftarrow \emptyset$
4  upon receive(⟨IHAVE, mID⟩, sender)
5      if $\nexists n \in inView : mID \in expectedMsgs[n]$ then
6          if $\nexists AnnouncementTimer(id) : id = mID$ then
7              setup AnnouncementTimer(mID, TIMEOUT1)
8          $missing[mID] \leftarrow missing[mID] \cup \{sender\}$
9      else
10         $expectedMsgs[sender] \leftarrow expectedMsgs[sender] \setminus \{mID\}$
11         call checkWaiting(mID, sender)
12 upon AnnouncementTimer(mID)
13     setup AnnouncementTimer(mID, TIMEOUT2)
14     $node \leftarrow poll(missing[mID])$
15     call elongate(node)
16     $grafted[mID] \leftarrow grafted[mID] \cup \{node\}$
17     trigger send(⟨GRAFT, mID⟩, node)
```

Upon receiving a *graft*, the node instantly adds the sender to the *eagerPushPeers* (Alg.13, line 5) to consummate the symmetric link for the tree. After that, the process should effectuate the expected transmission of the message payload. This content was kept in *waiting*, a blocking data structure that saves, for each neighbour, the sequence of all the messages that were requested to be forwarded after a given *ihave*. Without this technique, it would not be possible to ensure the FIFO order on every link, which is required to preserve causal guarantees. The following execution elucidates how the absence of this method may endanger causality:



(a) *N1* is requested to broadcast *m1*, so it sends an *ihave(m1)* to its lazyPushPeer *N2*

(b) *N1* is requested to broadcast *m2*. *N2* receives the *ihave(m1)* and replies with a *graft(m1)*

(c) *N1* forwards *m1* and is requested to broadcast *m3*. *N2* receives the *ihave(m2)* and replies with a *graft(m2)*

Figure 3.4: Sample execution that expose how the lack of *waiting* mechanism leads to causal dependencies violations

The execution begins with *N1* being requested to broadcast *m1* and hence, *N1* forwards it to its neighbours. One of its neighbours is *N2*, which is connected through a lazy push link and thereby *N1* sends an *ihave(m1)* (Fig.3.4a).

After the arrival of *ihave(m1)*, *N2* sets a timer and marks *m1* as missing, since it has not received *m1* yet. Meanwhile, *N1* gets another request to broadcast, this time a message *m2*. Due to the absence of the *waiting* mechanism, an *ihave(m2)* is promptly sent to *N2*, despite the lack of acknowledgement for the previous *ihave*. Once the timer related to *m1* expires, *N2* moves *N1* to *eagerPushPeers* and transmits a *graft(m1)* (Fig.3.4b).

Upon receiving *graft(m1)*, *N1* proceeds with the dissemination of *m1* and switches *N2* to an eager push link. In the meantime, *N2* gets *ihave(m2)* from *N1* and consequently, starts another timer. A final broadcast is requested to *N1*, which leads the message *m3* to be instantly propagated to *N2* since the corresponding link was recently changed to eager push mode. Once the timer regarding *m2* terminates, *N2* triggers a *graft(m2)* towards *N1* (Fig.3.4c). Lastly, *N1* obtains the *graft(m2)* and spreads *m2* to *N2*, causing *m2* to be the last delivery of *N2*.

Algorithm 13: Receiving *graft* message & checkWaiting

```
waiting //map that keeps for each peer p, a set of the messages waiting to be sent to p
upon init
    waiting ← $\emptyset$
upon receive(⟨GRAFT, mID⟩, sender)
    call elongate(sender)
    msg ← poll(waiting[sender])
    while msg $\neq \perp$ do
        trigger send(msg, sender)
        msg ← poll(waiting[sender])
procedure checkWaiting(mID, sender)
    msg ← peek(waiting[sender]) //retrieves without removing the head of the queue
    if msg $\neq \perp$ then ⟨_, mID2, _, _, _, _, _⟩ ← msg
    if mID = mID2 then
        waiting[sender] ← waiting[sender] \ {msg}
        m ← peek(waiting[sender])
        if m $\neq \perp$ then ⟨type, id, _, _, _, _, _⟩ ← m
        while m $\neq \perp \wedge$ type $\neq$ GOSSIP do
            waiting[sender] ← waiting[sender] \ {m}
            trigger send(m, sender)
            m ← peek(waiting[sender])
            if m $\neq \perp$ then ⟨type, id, _, _, _, _, _⟩ ← m
        if type = GOSSIP then trigger send(⟨IHAVE, id⟩, sender)
    else if mID ∈ waiting[sender] then
        waiting[sender] ← waiting[sender] \ {msg}
        trigger send(⟨IHAVE, mID⟩, sender)
```

Process *N1* had broadcast three distinct messages in the following order: i) *broadcast(m1)*, ii) *broadcast(m2)*, and iii) *broadcast(m3)*. From the definition of causal order (Section 2.1.3), we can extract two main causal dependencies that must be respected: $delivery(m1) \rightarrow delivery(m2)$ and $delivery(m2) \rightarrow delivery(m3)$. Therefore, *N2* violates the last dependency because it delivered *m2* after *m3*.

The *waiting* mechanism prevents this type of situations by blocking all disseminations employed by *N1* following a certain *ihave*. As a consequence, if the corresponding *graft* arises, *N1* replies with the requested message and all the other messages that were waiting to be transmitted through the link to *N2*, preserving the FIFO order (Alg.13, lines 6-9). However, *N2* may get the message from another link before its timer expires, avoiding the *graft* transmission. In that case, *N2* will forward the message to its neighbours and hence, it will send an *ihave* or the whole content to *N1*, according to the communication mode of the corresponding link. Even though every link may converge to be symmetric and to have a common communication mode, there might be some temporary mismatch between the two nodes that form a link, due to the asynchronous behaviour of the system. Therefore, despite the link *N1*→*N2* being on lazy push mode, the symmetric link might be temporarily on eager push mode.

When *N1* receives the expected message from *N2*, it checks if this message is blocking *waiting* (Alg.11, line 13; Alg.12, line 11). Therefore, *N1* verifies if the first message of *waiting[N2]* corresponds to the message that had recently arrived from *N2* (Alg.13, line 13) and if it does, the message is removed from *waiting* (Alg.13, line 14). After this removal, to fully unblock *waiting*, it is necessary to check if there are any subsequent obstructed messages that need to be sent. To operate that, *N1* inspects the type of the following message in *waiting[N2]*. As long as the extracted type applies to a control message, the process keeps successively removing and sending messages from *waiting[N2]* (Alg.13, lines 17-21) since the control messages are invariably exchanged in eager push mode. Once a broadcast message is selected, an *ihave* is sent (Alg.13, line 22) and the *waiting* set holds this message as its first element until it receives any feedback from the corresponding neighbour. This transmission is issued in lazy push mode because the *waiting* structure is only checked following the arrival of an already expected message, which always activates that mode.

However, if there is another message on the first position of *waiting[N2]*, then *N1* searches for the expected message in the remaining structure. If the message is found, it is consequently removed from the set and an *ihave* is sent towards *N2* (Alg.13, lines 23-25), with the purpose of informing that *N1* already possesses the message.

By employing this *waiting* technique and all the distinct mechanisms that we have been describing throughout this chapter, arises the requirement to update some procedures from the naive approach to causal broadcast, namely `forward` and `sendMsg`. These novel implementations will rearrange the dissemination primitives of our protocol, in order to merge all those mechanisms.

Algorithm 14: `forward` & `sendMsg` updated

```
1  procedure forward(message, sender)
2      $\langle \_, mID, msg \rangle \leftarrow message$
3      foreach $n \in sBuffers[n]$ do
4          $sBuffers[n] \leftarrow sBuffers[n] \cup \{\langle mID, msg \rangle\}$
5      foreach $\langle B_\alpha, B_\pi, received_\pi \rangle \in rBuffers$ do
6          if $received_\pi$ then $B_\pi \leftarrow B_\pi \cup \{\langle mID, msg \rangle\}$
7          else $B_\alpha \leftarrow B_\alpha \cup \{\langle mID, msg \rangle\}$
8      $notToSend \leftarrow \emptyset$
9      //if this node is not the message broadcaster
10     if $sender \neq \perp$ then
11         if $sender \in eagerPushPeers \cup lazyPushPeers$ then trigger send($\langle IHAVE, mID \rangle$, sender)
12         $notToSend \leftarrow notToSend \cup \{sender\}$
13         cancel AnnouncementTimer(mID)
14         foreach $n \in missing[mID]$ do
15             $expectedMsgs[n] \leftarrow expectedMsgs[n] \setminus \{mID\}$
16             trigger send($\langle IHAVE, mID \rangle$, n)
17         $notToSend \leftarrow notToSend \cup missing[mID]$
18         $missing \leftarrow missing \setminus \{mID\}$
19         $grafted[mID] \leftarrow grafted[mID] \setminus \{sender\}$
20         foreach $n \in grafted[mID]$ do trigger send($\langle IHAVE, mID \rangle$, n)
21         $notToSend \leftarrow notToSend \cup grafted[mID]$
22         $grafted \leftarrow grafted \setminus \{mID\}$
23     foreach $n \in eagerPushPeers \setminus notToSend$ do
24         call sendMsg($\langle GOSSIP, mID, msg \rangle$, n)
25     foreach $n \in lazyPushPeers \setminus notToSend$ do
26         if $waiting[n] = \emptyset$ then trigger send($\langle IHAVE, mID \rangle$, n)
27         $waiting[n] \leftarrow waiting[n] \cup \{\langle GOSSIP, mID, msg \rangle\}$
28 procedure sendMsg(msg, node)
29     if $waiting[node] = \emptyset$ then trigger send(msg, node)
30     else $waiting[node] \leftarrow waiting[node] \cup \{msg\}$
```

When a new message arrives, besides the inherent demand to propagate it to the neighbours, the receiver node has to incorporate it into the control information that support all the described mechanisms. One of those techniques is the message buffering that leverages the control message exchange. In that sense, the process registers this recent message on its active buffers (Alg.14, lines 3-7).

There are some neighbours that only need an acknowledgement regarding this new message, dismissing the transmission of the entire content. Naturally, one of these neighbours is the message sender, considering that it is not the original broadcaster and that exists a corresponding outgoing link (Alg.14, lines 11-12). The remaining neighbours might result from preceding *ihaves* that were eventually received regarding this message. In that scenario, the process promptly propagates acknowledgements to all the neighbours that announced to have this message and discards all the control information regarding this supposedly missing message (Alg.14, lines 13-18). Nevertheless, if the node had previously *grafted*, it is impossible to prevent the transmission of the message and thereby, the process should expect to receive it. However, the related control information must get dropped and an acknowledgement should be sent (Alg.14, lines 19-22).

To conclude the procedure, the message is forwarded through all the outgoing links according to their communication mode. If the link is on eager push mode (Alg.14, line 23), the process verifies if the *waiting* structure regarding this link is empty and if it is not, the message is attached to *waiting* (Alg.14, line 29). Otherwise, the whole message is instantly transmitted (Alg.14, line 30). If it is on lazy push mode (Alg.14, line 25) and assuming that the *waiting* for this link is empty, then the node sends an *ihave* (Alg.14, line 26). Despite the fact that an *ihave* was sent or not, the message is kept on the first position of the *waiting* for this link (Alg.14, line 27).

Now that we presented all the mechanisms of CPTCast and the consequent data structures that were employed, we are able to disclose the expected behaviour of a process, when a given neighbour is dropped. Summarily, the process removes all the control information related to the withdrawn node, as it can be depicted in Algorithm 15.

Algorithm 15: Handling changes on the membership - NEIGHBOUR_DOWN

```
1  upon neighbourDown($node$)
2      $eagerPushPeers \leftarrow eagerPushPeers \setminus \{node\}$
3      $lazyPushPeers \leftarrow lazyPushPeers \setminus \{node\}$
4      $inView \leftarrow inView \setminus \{node\}$
5      foreach $mID \in missing$ do
6          $missing[mID] \leftarrow missing[mID] \setminus \{node\}$
7      $expectedMsgs \leftarrow expectedMsgs \setminus \{node\}$
8      $sBuffers \leftarrow sBuffers \setminus \{node\}$
9      $rBuffers \leftarrow rBuffers \setminus \{node\}$
10     foreach $mID \in grafted$ do
11         $grafted[mID] \leftarrow grafted[mID] \setminus \{node\}$
12     $waiting \leftarrow waiting \setminus \{node\}$
13     $unsafeIn \leftarrow unsafeIn \setminus \{node\}$
14     $unsafeOut \leftarrow unsafeOut \setminus \{node\}$
15     $safePaths \leftarrow safePaths \setminus \{node\}$
16     cancel DiscoveryTimer(node)
```

3.5 Summary

In this chapter we presented our protocol, named CPTCast, that provides a scalable and reliable solution of causal broadcast, adopting a tree-based approach.

We started the chapter by delineating the system model on which CPTCast can be employed and the necessary requirements that our implementation must fulfill. To give an insight about the structure of each process that constitutes the system, we depicted their architecture and how their components relate to each other. Finally, we detailed our implementation, specifying all the mechanisms that were exploited.

As we had previously mentioned, our solution is based on some mechanisms from both Plumtree and PRC-broadcast. However, their semantics are not fully compatible and therefore, those mechanisms had to be adjusted and we had to introduce several other techniques with the purpose of merging the two dissimilar strategies.

The following chapter describes all the experimental work that we conducted in order to evaluate CPTCast.

Chapter 4

Evaluation

In this chapter, we describe the experimental work conducted to evaluate CPTCast. We begin by introducing the experimental methodology that we adopted, including the metrics that we selected to perceive the performance of our solution when compared with relevant baselines (Section 4.1). We proceed by specifying the configuration details of all the experiments that we conducted (Section 4.2), followed by the presentation and consequent analysis of the experimental results (Section 4.3). Lastly, we present a summary of this chapter (Section 4.4).

4.1 Methodology

To evaluate a broadcast protocol, such as CPTCast, it is fundamental to submit a proper implementation of the protocol to a considerable set of tests with diverse configurations. With this approach, it is possible to validate if the solution accomplishes all the predefined requirements and to scrutinize its performance across several contexts with distinct settings. Moreover, to obtain a meaningful perception of how CPTCast performs, it is essential to compare it with other broadcast protocols. For that purpose, we implemented three different protocols: i) CPTCast, ii) Plumtree, and iii) PRC-broadcast.

To employ tree-based broadcast primitives, CPTCast adopts some techniques based on Plumtree [18]. However, it also applies procedures that ensure causal consistency, in contrast to Plumtree, which do not provide any ordering guarantees. Hence, it is relevant to examine their performances to perceive the overhead associated with the causality mechanisms that CPTCast utilizes.

These mechanisms are inspired by PRC-broadcast [25], which does not pursue any specific topology, managing pure eager gossip dissemination. Therefore, the comparison between the performances of PRC-broadcast and CPTCast enables the identification of the different configurations where each protocol is more suitable to operate. For instance, in a system where all disseminated messages are small and every node plays often the broadcaster role, it would be more appropriate to adopt a pure eager broadcast approach. However, if large messages are travelling across the network and all the broadcasts are issued by the same process or a small number of processes, it would be convenient to follow a tree-based approach, such as CPTCast or Plumtree, where a tree rooted on the broadcaster node is inscribed in the network and used to guide the flow of large messages. This tree covers every node of the system, although is constituted by a subset of all the links that shape the network and consequently, the remaining links are spared from performing the costly and redundant propagation of those messages, simply transmitting small control information.

To achieve an unbiased evaluation of the referred protocols, every implementation must rely on an equivalent codebase. Therefore, all the solutions are written in Java and rely on the same membership protocol, which is HyParView [19]. We further detail how HyParView is implemented and configured in Section 4.2.1.

Once we implemented all the protocols, we should start to test their implementations, by applying the same instructions to each protocol under identical configurations. After executing the requested instructions, each protocol must present all the gathered statistical information, which will produce the metrics for an insightful analysis of their performances.

4.1.1 Metrics

To administer an accurate evaluation of the previously indicated broadcast protocols, we selected the following metrics:

- **Reliability** is the percentage of active nodes that delivered a specific broadcast.
- **Latency** of a broadcast b towards a process p is the period of time elapsed between the initial dissemination of b and the moment when p receives it for the first time.
- **Relative Message Redundancy (RMR)** is a metric derived from the original experimental work of Plumtree [18] and it measures the messages overhead in a broadcast protocol. The metric is defined by the following formula:

  $$\frac{m}{n-1} - 1$$

  where m is the total of broadcast messages exchanged during a certain broadcast and n is the number of nodes that received it.

Now that we had described the methodology that conducted the experimental work in order to validate our solution, we proceed by detailing the configuration process of those experiments.

4.2 Configuration Details

Each process of the system has three main components that interact with each other, as we presented in Section 3.3. Our goal in this experimental phase is to compare CPTCast to the mentioned broadcast solutions and hence, the broadcast layer must alternate each protocol throughout the different experiments. However, the other layers are kept unchanged to provide equivalent test settings. Therefore, the application layer is in charge of carrying out the experiments by demanding appropriate sets of broadcast operations to examine how each protocol behaves when handling the same sequence of instructions. In addition, the membership layer invariably follows one specific protocol, which is HyParView.

4.2.1 HyParView

We selected HyParView to handle the membership layer, due to the fact that it provides all the properties stated in Section 3.1. It follows a hybrid approach of maintaining two distinct views at each node: a small *active view*, of size $log_{10}(n) + c$, and a larger *passive view*, of size $k(log_{10}(n) + c)$. Regarding the original experimental work conducted for HyParView: $c = 1$, $k = 6$ and n is the number of nodes in the network. All our experiments were employed in a network of 10,000 nodes and consequently, we used *active views* of size 5 and *passive views* of size 30. The *active view*, as the name suggests, is the list of neighbours that is actually shared with the broadcast protocol to perform its inherent activity and the *passive view* is used to replace any obsolete link of the *active view*.

Its implementation were also written in Java and integrates Netty, an asynchronous event-driven network framework for development of high performance protocols.

4.2.2 Grid'5000

As we have been stated throughout this document, CPTCast must be suitable to operate on large-scale distributed systems. Therefore, the tests that are conducted on this experimental phase should be executed on several machines simultaneously. Another property that we pursued is the ability to perform efficiently, even if those machines are on distant points of the planet, as a geo-replicated distributed system. With that in mind, we searched for a testbed that would match our requirements and Grid'5000 emerged as a proper solution.

Grid'5000 is a large-scale testbed for experiment-driven research, that offers access to a large amount of computational resources spread in distinct regions of France. As it can be depicted in Figure 4.1, there are eight main sites and each one keeps multiple clusters of compute nodes. In a nutshell, Grid'5000 gathers more than 15000 cores, distributed by approximately 800 nodes.

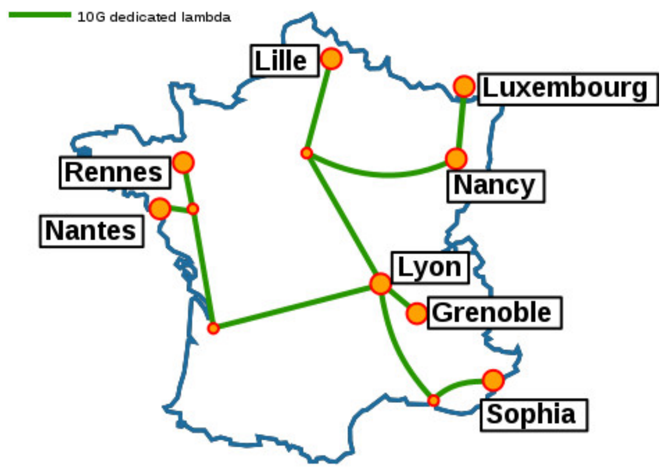


Figure 4.1: Geographic distribution of all sites on Grid’5000

The resources provided by Grid’5000 are shared among many users and hence, it relies on a resource manager, named OAR. This tool operates with a simple language that expresses operations to be performed in the platform, such as create, delete or edit a reservation. To reserve a certain set of resources, the user has to access the corresponding site via SSH and then specify the job using OAR commands.

For instance, the following command indicates a reservation of 100 nodes in either *gros* or *grisou*, which are clusters located in *Nancy*, for 10 hours, starting on 20 February 2020, at 20:00:

```
$ oarsub -p "cluster='gros'␣or␣cluster='grisou'" -l nodes=100,walltime=10 \
> -r "2020-02-20␣20:00:00"
```

All experiments were conducted on a network with 5000 nodes in *Nancy* and another 5000 processes in *Rennes*. We reserved 100 machines on each site and launched 50 processes on every machine. Table 4.1 shows the hardware specifications of the devices on which we operate.

Site	Cluster	Nodes	CPU	Cores	Memory	Storage	Network
Nancy	gros	100	Intel Xeon Gold 5220	18 cores/CPU	96 GiB	480 GB SSD + 960 GB SSD	2 x 25 Gbps
Rennes	paranoia	8	2 x Intel Xeon E5-2660 v2	10 cores/CPU	128 GiB	1 x 600 GB HDD + 4 x 600 GB HDD	1 Gbps + 2 x 10 Gbps
Rennes	parasilo	22	2 x Intel Xeon E5-2630 v3	8 cores/CPU	128 GiB	1 x 600 GB HDD + 4 x 600 GB HDD + 200 GB SSD	2 x 10 Gbps
Rennes	paravance	70	2 x Intel Xeon E5-2630 v3	8 cores/CPU	128 GiB	1 x 600 GB HDD + 1 x 600 GB HDD	2 x 10 Gbps

Table 4.1: Hardware specifications of the employed machines

4.2.3 Experiments Workflow

To accomplish a complete evaluation of CPTCast, we conducted a large number of experiments. Each experiment can be translated into a sequence of steps, beginning at the network setup and ending at the causality inspection.

Network Setup

The first step is to launch the network by prompting a determined number of nodes onto the system. Since we are aiming large-scale systems and a single computer has a limited amount of computational resources, the load has to be split by multiple devices. Therefore, we initialize an appropriate number of processes on each machine in order to reach the total number of nodes defined for the experiment. To differentiate the processes, each one concatenates a unique port to the network address of the machine, constituting its identifier. To boot a process, we need to execute a JAR file, specifying which protocol will be employed in that experiment. Since we want to launch a significantly large amount of nodes, it is not feasible to manually initialize each process. To automate this procedure, we developed a shell script that carries out the whole bootstrap, with only three parameters: i) the site where the job was reserved on Grid'5000, ii) the number of processes to boot on each machine, and iii) the number of nodes on broadcaster mode.

Experiments Execution & Metrics Extraction

After the network had fully initialized and converged to a stable state, it is ready to accommodate the experiments. An experiment can be interpreted as a sequence of broadcasts with certain specifications and hence, to trigger a test, it is necessary to indicate the following information: i) the number of broadcasts, ii) the message payload size, and iii) the interval between each broadcast. These parameters must be deployed to a contact node, that will notify the whole network of the ongoing test. From then on, the broadcasters propagate the requested messages until the defined number of broadcasts is achieved. After receiving the final dissemination, each node prints a log file with all the statistical data collected during the test. To inspect the results of each experiment, we implemented a Python script that digests all the logs and convert them into a smooth report with all the relevant metrics.

Causal Guarantees Verification

CPTCast and PRC-broadcast commit to provide causal consistency, which implies that the causal order among all disseminated messages must be preserved.

We can inspect the full execution of the system during a certain experiment because whenever a message m is delivered by a particular node p ($delivery_p(m)$), it instantly records the message identifier in its deliveries log. Consequently, we can determine the order that the delivery operations were issued on every node. Furthermore, when the broadcast of the message m is requested at a certain process p ($broadcast_p(m)$), then the corresponding $delivery_p(m)$ is instantly issued by p. For that reason, the moment when m was recorded in p's log indicates when p operated $broadcast_p(m)$. In a nutshell, when we are inspecting the logs and we find a message identifier containing the same address of the current log's owner, it means that the corresponding message was broadcast by such owner at that instant.

Due to our context of large-scale distributed systems, it is impossible to scrutinize an absolute order for the multiple events that are occurring concurrently. However, if a certain process executes a sequence of two broadcasts, a happened-before relationship between them must be preserved. Therefore, we can conclude that all the broadcasts displayed before a broadcast b in the log of a certain process, have a causal relationship with b. Besides the broadcast operations, we also know that every delivery presented in a particular log before a specific broadcast b was introduced by an initial dissemination and hence, that corresponding broadcast operation should preserve a causal dependency with b. In conclusion, the causal history of b is constituted by all the broadcast operations of the messages that are displayed before b in the log of b's broadcaster.

An execution respects causal order if, for all the broadcast operations that happened before the broadcast of a certain message, the corresponding delivery operations also precede the delivery of that specific message. With that in mind, we developed a tool that checks if a system execution respects this previous rule. Algorithm 16 presents the pseudo-code of this tool.

Algorithm 16: Algorithm to check causal order

```
Data: logs, a map that for each process p, keeps the sequence of message IDs delivered by p
Result: true, if the causal order was not violated; false, otherwise
1  foreach $p \in logs$ do
2      $l \leftarrow \emptyset$
3      foreach $m \in logs[p]$ do
4          $\langle addr, \_ \rangle \leftarrow m$
5          //if p had broadcast m
6          if $p = addr$ then
7              $causalHistory[m] \leftarrow l$
8              $l \leftarrow \emptyset$
9          $l \leftarrow l \cup \{m\}$
10 foreach $m1 \in causalHistory$ do
11     $\langle addr, \_ \rangle \leftarrow m1$
12     foreach $m2 \in causalHistory[m1]$ do
13         foreach $p \in logs$ do
14             if $p \neq addr$ then
15                 foreach $m3 \in logs[p]$ do
16                     if $m2 = m3$ then break
17                     if $m1 = m3$ then return false
18 return true
```

Firstly, the tool extracts the causal history of every broadcast from the logs. For that purpose, it traverses each file and whenever a broadcast is identified, it saves the list of all the message identifiers that were traversed previously (Alg.16, lines 1-9). For instance, a certain process operates two subsequent broadcasts *b1* and *b2*. Consequently, *b1*'s causal history will be contained in *b2*'s history. For that reason, if *b1* already validates its causal history, then it is redundant to *b2* check those dependencies again. With that in mind, the tool only produces causal histories with the message identifiers that are presented since the previous broadcast. After all the causal histories are completed, the tool starts to validate them.

For every message *m* contained in the causal history of a message *b*, the tool needs to validate that $broadcast(m) \rightarrow broadcast(b)$ is preserved. To accomplish that, it has to certify that the corresponding *m*'s delivery was issued before *b*'s delivery by every node. In practice, the tool checks that *m*'s identifier is displayed earlier than *b*'s identifier in every log (Alg.16, lines 10-17).

The tool was written in Java and since it reveals a considerable complexity, the logic of the pseudo-code was slightly adjusted to operate using a parallel programming model. In opposition to Plumtree, CPTCast and PRC-broadcast offer causal guarantees and hence, only experiments over these two protocols include this experimental step.

4.3 Experimental Results

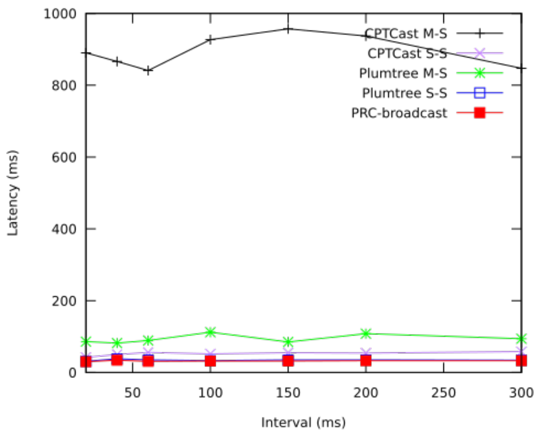
Now that we had described the configuration process of our experimental work, we proceed by presenting and discussing the results obtained. As we discussed previously, we implemented three broadcast protocols to compare their performances. For CPTCast and Plumtree, we used two different modes: i) single-sender, and ii) multi-sender. In single-sender, the broadcasts are issued exclusively by one specific node and in multi-sender, any process is allowed to disseminate messages. PRC-broadcast behaves similarly in both modes since it determines that every node invariably forwards each incoming message in eager push mode to all its neighbours, in contrast to the other protocols, which use a tree-based dissemination strategy. In multi-sender mode, the dissemination tree will constantly change its shape, trying to adapt to each broadcaster. Consequently, its performance will decrease compared to the single-sender mode.

We tried several inputs for the experiments in order to identify an appropriate range of values for each parameter. For the message payload size, we outlined four different values: i) 1 kilobyte, ii) 100 kilobytes, iii) 500 kilobytes, and iv) 1 megabyte. Regarding the interval between each broadcast, we ranged between 20 and 300 milliseconds. This parameter controls the broadcast throughput since decreasing this interval prompts more messages to be produced in a shorter period and increasing causes the opposite effect. The number of messages disseminated on each experiment is not so relevant to fluctuate because every metric produces an average value. For that reason, it is pointless to perform experiments with a large number of messages. To obtain a more accurate output, we operated a considerable amount of tests with 100 broadcasts each and calculated the average value for each metric.

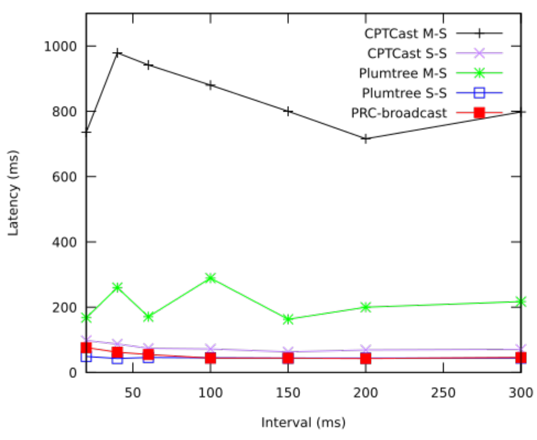
All the experiments exhibited a reliability of 100%, corroborating their classification as reliable broadcast protocols. However, the remaining metrics had not disclosed such a straightforward output. Therefore, we will discuss the results for each one.

4.3.1 Latency

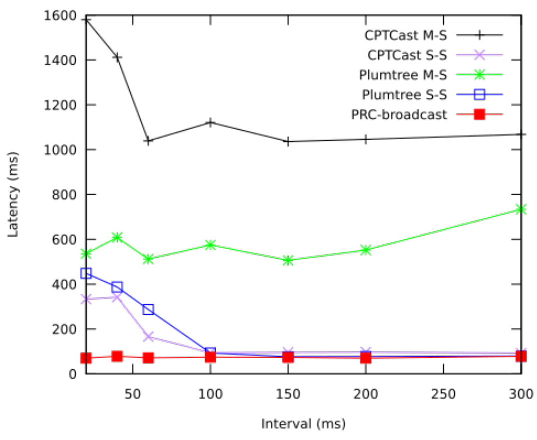
Figure 4.2 reports, for each implementation, the average latency extracted from all the possible combinations of the predetermined values for the parameters. It includes four separate charts, each one regarding a different payload size.



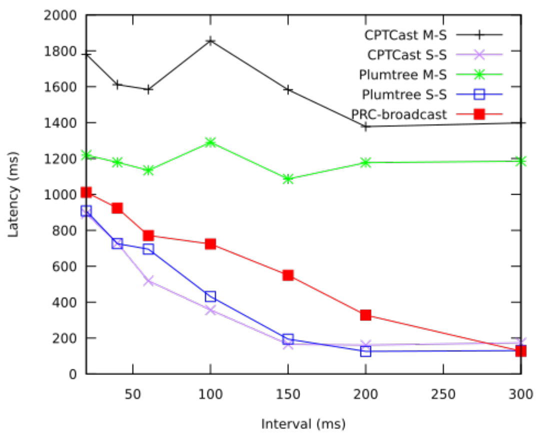
(a) 1 kilobyte



(b) 100 kilobytes



(c) 500 kilobytes



(d) 1 megabyte

Figure 4.2: Average latency for different payload sizes

One of the first observations that we can perceive from Figure 4.2 is that CPTCast with multi-sender (CPTCast M-S) presents significantly higher latency than the other configurations on every chart. As we mentioned earlier, it is caused by the continuous changes in the broadcast tree, that is seeking to adjust itself to the source of each broadcast. The other multi-sender configuration, Plumtree M-S also displays higher latency than the remaining configurations, although exhibiting a lower difference, when compared to CPTCast M-S.

An interesting aspect of both implementations in multi-sender is their low sensibility to the interval fluctuation, comparing to the other configurations. When the interval between broadcasts is decreased, the throughput inflates and hence, the network reacts to that stimulation by queuing the constantly incoming messages. This phenomenon increases the latency and this shift gets even more accentuated with heavier messages. If a message gets blocked enough time to trigger the lazy push mode, then it even induces mutations on the tree. Since these changes are endless in multi-sender configurations, the resulting latency value is already maximized. Consequently, the interval changes do not significantly influence the average latency on this mode. However, single-sender solutions maintain a stable state on their dissemination trees, which are only disturbed on heavy throughput scenarios. For that reason, these configurations display higher responsiveness to the interval changes.

Both single-sender implementations exhibit optimal latency across all the plots, demonstrating that the causal mechanisms introduced by CPTCast do not produce a significant overhead on this mode. However, CPTCast displays considerably higher values than Plumtree on multi-sender mode.

PRC-broadcast also displays optimal latency, although it performs slightly worse than single-sender configurations for 1 megabyte of payload size. A reasonable explanation comes from the fact that it produces more messages in general and consequently, more situations of message queuing. Even considering that it does not have a tree that would be constantly affected, the extra amount of large messages getting blocked induces this overhead.

4.3.2 Relative message redundancy

To report the relative message redundancy of our experiments, we plotted a bar chart, aggregating all the values extracted for each payload size since the interval variations did not significantly influence the output of the experiments. Figure 4.3 depicts, for every configuration, the average relative message redundancy obtained from each predetermined value for the payload size.

As expected, PRC-broadcast presents the worst results for this metric, with a nearly constant RMR of 4, which is closely related to the number of neighbours of each node, that we defined as 5. The rationale behind this pattern is the eager gossip behaviour that this protocol follows by forwarding every incoming message through all its outgoing links. Consequently, every node will receive five copies of each message, although only the first transmission is strictly necessary. In conclusion, when a certain process forwards a message to its five neighbours, four of these disseminations tend to be unnecessary.

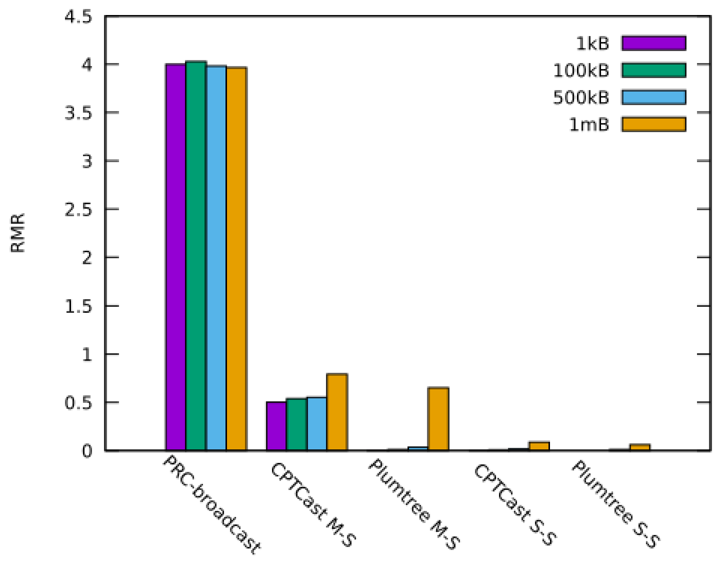


Figure 4.3: Average relative message redundancy for different payload sizes

In contrast to PRC-broadcast, the single-sender implementations exhibit RMR values tending to 0, which indicates that they are promoting non-redundant disseminations. Nevertheless, CPTCast M-S registers some notable redundancy and even Plumtree M-S for 1 megabyte of payload size. As we explained previously, the multi-sender mode causes continuous changes on the dissemination tree, which explains the extra redundancy that is displayed. However, there is still a large disparity between them and PRC-broadcast results since their RMR does not even reach the value of 1.

4.3.3 Results Analysis

After studying all the results that we extracted from the experimental work, we acquired an insightful perception of how CPTCast performs and thereby, we will present the main lessons that we learned.

PRC-broadcast leads each process to utilise all its links to propagate messages, triggering their dissemination through all the existent paths on the network and consequently, the shortest path will also be employed. Therefore, if the network is under regular conditions of broadcast throughput, PRC-broadcast will display optimal latency. On single-sender mode, CPTCast presented similar values of latency to PRC-broadcast, which indicates that it manages to maintain a concise tree connecting every node. With 1 megabyte of payload size, CPTCast S-S even outperforms PRC-broadcast, corroborating the inadequate network usage that PRC-broadcast conducts, which leads its processes to accumulate many heavy messages. The RMR results reinforce its inefficient network usage and support the accurate dissemination strategy that CPTCast introduces.

Regarding the comparison with Plumtree, they revealed similar performances, except for some particular scenarios where CPTCast performed worse, for instance, in the multi-sender mode. Therefore, we can deduce that the mechanisms that ensure causal consistency on our solution are not producing a significant overhead.

Considering all the premises that we identified, we conclude that CPTCast will fit perfectly on applications with broadcast primitives that run on top of large-scale distributed systems and require causal guarantees. In particular, CPTCast is mostly adequate in scenarios where messages have large size, as it can achieve causal delivery with reduced usage of the network.

Finally, due to lack of time, we could not compare the performance of the solutions in scenarios where node failures can happen. We expect CPTCast to be able to cope well with failures, as it inherits some of the fault-tolerance properties of Plumtree. However, fault recovery might lead to additional network usage, and potentially, result in a slight increase in latency. This experimental direction will be explored as future work.

4.4 Summary

In this chapter, we presented all the experimental work that we operated in order to understand how CPTCast performs. Firstly, we revealed the methodology that we followed to evaluate our solution, which led to the implementation of two referential broadcast protocols.

After specifying the methodology, we proceeded by describing all the details of the configuration process that the experiments had required. Those experiments generated some data that was digested to extract interesting analytical results, which were further analyzed and discussed in order to conclude the verdict regarding the applicability of CPTCast.

The following chapter concludes this document, presenting our conclusions regarding all the conducted work and outlining some future work possibilities.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The causal consistency model has emerged as a relevant answer for those systems that require to present an adequate performance and provide consistency guarantees, namely ensuring causal order on the system. Causal consistency reveals a great trade-off, producing solutions that can fulfil both requirements. With that in mind, the design of our protocol pursued this consistency paradigm.

From the study of the state-of-the-art, we identified several approaches that seek causal consistency and hence, we decided to exploit a novel technique introduced by PRC-broadcast [25]. It enables the safe removal of obsolete control information about broadcast messages, promoting causal dissemination with an optimal space complexity, which is a desirable feature to address on our solution. We also aimed to employ a tree-based dissemination strategy, which grants the application of distinguished algorithms and supports an efficient usage of the network, reducing the inherent message traffic. Therefore, we explored the hybrid dissemination method that Plumtree [18] originally proposed, which incorporates gossip and tree-based broadcast primitives. From these two baselines, we composited our solution, which unveiled to be capable of combining these distinct approaches.

Besides the design of our solution, we also provided an implementation of CPTCast, coupling with an appropriate membership protocol, such as HyParView. To validate that it comprises all the predefined requirements and obtain a perceptive evaluation of its performance, we operated an extensive experimental work, adopting the implementations of the two correlated protocols as reference. The experiments were conducted on a proper and realistic context since we administered them on the wide testbed that Grid'5000 offers.

The results from the experiments demonstrated that CPTCast proposes a proper and innovative solution for causal dissemination, which will ideally adapt to widely distributed systems that need to provide causal guarantees, such as geo-replicated storage services with causal order among their operations. The causal broadcast primitives introduced by CPTCast will lead the actions performed locally on a certain node to be securely spread throughout the whole network.

5.2 Future Work

During the evolution of the thesis, we have outlined some potential directions for future work in order to improve our solution. These prospects are mainly focused on validating the applicability of CPTCast on alternative scenarios.

The first relevant possibility that we identified was to attach CPTCast to a concrete database service without causal guarantees. A proper example of this class of systems is Apache Cassandra [15], which ensures high availability and fault-tolerance on large-scale networks, although only ensuring eventual consistency. If the causal broadcast mechanisms that CPTCast propose were able to integrate into Cassandra's logic, it would be a perfect setting to measure the overhead that those techniques produce, since the comparison between this version of Cassandra and regular Cassandra, would fully disclose the impact of adding those mechanisms.

We also would like to perform an extensive evaluation of CPTCast's performance under an environment with massive failures occurring. Although the recognition that Plumtree provides high fault-tolerance be an encouraging aspect to presume that CPTCast would also guarantee such property, we think that it would be important to verify that, by conducting several experiments on that scenario.

Finally, it is also our intention to couple CPTCast to an alternative membership protocol with different characteristics. Even though the alliance with HyParView [19] revealed to be flawless, we consider that testing our solution with other membership services would be beneficial to approve its performance, regardless of the underlying service. For instance, a protocol that comprises routing primitives would be interesting to adopt, since it would allow to contrast with the *discovery* mechanism that we designed to route the control messages through safe links. Therefore, we would have a baseline to validate the efficiency of our mechanism.

Bibliography

[1] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 85–98.

[2] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A distributed metadata service for causal consistency." In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, pp. 111–126.

[3] E. A. Brewer. "Towards robust distributed systems." In: *PODC*. Vol. 7. 2000.

[4] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. "The Bayou architecture: Support for data sharing among mobile users." In: *1994 First Workshop on Mobile Computing Systems and Applications*. IEEE. 1994, pp. 2–7.

[5] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão. "Efficient Synchronization of State-based CRDTs." In: *arXiv preprint arXiv:1803.02750* (2018).

[6] M. Ferreira, J. Leitao, and L. Rodrigues. "Thicket: A protocol for building and maintaining multiple trees in a p2p overlay." In: *2010 29th IEEE Symposium on Reliable Distributed Systems*. IEEE. 2010, pp. 293–302.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. *Impossibility of distributed consensus with one faulty process.* Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1982.

[8] P. Fouto, J. Leitão, and N. Preguiça. "Practical and Fast Causal Consistent Partial Geo-Replication." In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2018, pp. 1–10.

[9] R. Friedman and S. Manor. *Causal ordering in deterministic overlay networks*. Tech. rep. Computer Science Department, Technion, 2004.

[10] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. "Scamp: Peer-to-peer lightweight membership service for large-scale group communication." In: *International Workshop on Networked Group Communication*. Springer. 2001, pp. 44–55.

[11] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[12] M. P. Herlihy and J. M. Wing. “Linearizability: A correctness condition for concurrent objects.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[13] M. Jelasity and O. Babaoglu. “T-Man: Gossip-based overlay topology management.” In: *International Workshop on Engineering Self-Organising Applications*. Springer. 2005, pp. 1–15.

[14] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.” In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. STOC ’97*. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: http://doi.acm.org/10.1145/258533.258660.

[15] A. Lakshman and P. Malik. “Cassandra: a decentralized structured storage system.” In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[16] L. Lamport. “Time, clocks, and the ordering of events in a distributed system.” In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[17] L. Lamport et al. “Paxos made simple.” In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[18] J. Leitao, J. Pereira, and L. Rodrigues. “Epidemic broadcast trees.” In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE. 2007, pp. 301–310.

[19] J. Leitao, J. Pereira, and L. Rodrigues. “HyParView: A membership protocol for reliable gossip-based broadcast.” In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*. IEEE. 2007, pp. 419–429.

[20] J. Leitao, J. P. Marques, J. Pereira, and L. Rodrigues. “X-bot: A protocol for resilient optimization of unstructured overlay networks.” In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (2012), pp. 2175–2188.

[21] J. Leitão, P. Á. Costa, M. C. Gomes, and N. Preguiça. “Towards Enabling Novel Edge-Enabled Applications.” In: *arXiv preprint arXiv:1805.06989* (2018).

[22] J. C. A. Leitao, J. P.d.S. F. Moura, J. O.R. N. Pereira, L. E. T. Rodrigues, et al. “X-bot: A protocol for resilient optimization of unstructured overlays.” In: *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE. 2009, pp. 236–245.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. “Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS.” In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 401–416.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. “Stronger semantics for low-latency geo-replicated storage.” In: *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, pp. 313–328.

[25] B. Nédelec, P. Molli, and A. Mostefaoui. “Causal Broadcast: How to Forget?” In: *The 22nd International Conference on Principles of Distributed Systems (OPODIS)*. 2018.

[26] C. H. Papadimitriou. “The serializability of concurrent database updates.” In: *Journal of the ACM (JACM)* 26.4 (1979), pp. 631–653.

[27] A. Rowstron and P. Druschel. “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems.” In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.

[28] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. “SCRIBE: The design of a large-scale event notification infrastructure.” In: *International workshop on networked group communication*. Springer. 2001, pp. 30–43.

[29] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. “Chord: a scalable peer-to-peer lookup protocol for internet applications.” In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32.

[30] W. Vogels, L. Rodrigues, and P. Veríssimo. “Fast group communication for standard workstations.” In: (1992).

[31] S. Voulgaris, D. Gavidia, and M. Van Steen. “Cyclon: Inexpensive membership management for unstructured p2p overlays.” In: *Journal of Network and systems Management* 13.2 (2005), pp. 197–217.