DEPARTMENT OF
COMPUTER SCIENCE

EMA RODRIGUES VIEIRA
Bachelor in Computer Science

ECO SYNC TREE: A CAUSAL AND DYNAMIC BROADCAST TREE FOR EDGE-BASED REPLICATION

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

DEPARTMENT OF
COMPUTER SCIENCE

ECO SYNC TREE: A CAUSAL AND DYNAMIC BROADCAST TREE FOR EDGE-BASED REPLICATION

EMA RODRIGUES VIEIRA

Bachelor in Computer Science

Adviser: João Carlos Antunes Leitão
Assistant Professor, NOVA University Lisbon

Co-adviser: Nuno Manuel Ribeiro Preguiça
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

ECO SYNC Tree: a Causal and Dynamic Broadcast Tree for Edge-based Replication

This work was partially supported by FC&T through grants NOVA LINCS (UID/CEC/04516/2013) and project NG-STORAGE (PTDC/CC-INF/32038/2017).

To my parents, for their unconditional support.

Acknowledgements

Firstly, I want to express my sincere gratitude to the colleagues and friends that worked alongside me, but especially to Pedro Fouto and Pedro Ákos Costa, without whom the work presented in this thesis would not have been possible.

I want to thank all my friends for motivating me, but particularly Giovanna Lopez for being the ever present, incredible friend that she is.

Finally, I would like thank my mother and father for always encouraging me to *do* better, and *be* better.

Abstract

As current trends start to move storage and computation to the edge in order to provide support for latency constrained applications, new edge storage systems must emerge that optimise latency and reduce the cost of communication, so that they can provide users with the best possible experience. With this goal in mind, several new storage systems have surfaced that make the most of sophisticated replication techniques and weak consistency models, in particular the causal+ consistency model.

A way to build such a system is to use a causal broadcast algorithm to propagate write operations between replicas in an order that is compatible with the causal order. However, existing systems rely on mechanisms that have limitations: they either leverage on static tree topologies, not adapting to scenarios where replicas join or leave the system, or they use metadata that grows linearly with the number of replicas, being therefore incapable of scaling to the hundreds or thousands of (smaller) replicas and, as such, not being suitable for supporting the operation of edge data stores.

In this work, we propose a new edge-enabled replication scheme that ensures causal delivery of operations in all replicas and, when used together with CRDTs, guarantees causal+ consistency. Our solution is a decentralised causal broadcast algorithm, ECO SYNC Tree, that makes use of a dynamic tree topology, capable of quickly adapting to nodes joining and leaving the system, to offer causal delivery while using negligible metadata to encode causal dependencies.

We present an experimental evaluation of ECO SYNC Tree that shows that our solution captures “the best of both worlds” when it comes to the trade-off between broadcast latency and communication cost in stable environments, and in environments subject to events such as large groups of nodes joining or leaving the system. Moreover, when compared with state-of-the-art broadcast protocols, ECO SYNC Tree proved to be the one that is best suited for edge-based deployment, by exhibiting better performance and scalability in scenarios with high churn rates.

Keywords: Distributed Storage Systems, Geo-Replication, Causal+ Consistency, Edge Computing, Causal Broadcast, CRDTs.

Resumo

Conforme as tendências atuais começam a mover o armazenamento e a computação para a periferia da rede de a maneira suportar aplicações com restrições de latência, devem surgir novos sistemas de armazenamento para este paradigma que otimizem a latência e reduzam o custo de comunicação de forma a proporcionar aos utilizadores a melhor experiência possível. Com esse objetivo em mente, surgiram vários novos sistemas de armazenamento que utilizam técnicas sofisticadas de replicação e modelos de coerência fraca, em particular o modelo de coerência causal+.

Uma maneira de construir um destes sistemas é usar um algoritmo de difusão causal para propagar operações de escrita entre réplicas numa ordem que seja compatível com a ordem causal. No entanto, os sistemas existentes usam mecanismos que têm limitações: topologias em árvore estáticas que não se adaptam a cenários em que réplicas se juntam ou saem do sistema, ou metadados que crescem linearmente com o número de réplicas e impedem o sistema de escalar para centenas ou milhares de réplicas, não sendo adequados para suportar sistemas de armazenamento na periferia.

Neste trabalho, propomos um novo esquema de replicação adaptado à periferia que entrega operações por ordem causal a todas as réplicas e, quando usado em conjunto com CRDTs, garante coerência causal+. A nossa solução é um algoritmo de difusão causal descentralizado, ECO SYNC Tree, baseado numa topologia em árvore dinâmica capaz de se adaptar rapidamente à entrada e saída de nós do sistema, e que usa metadados de tamanho desprezível para codificar dependências causais.

Apresentamos uma avaliação experimental que mostra que a nossa solução captura "o melhor dos dois mundos" no *trade-off* entre latência de difusão e custo de comunicação, tanto em ambientes estáveis, como em ambientes sujeitos a eventos em que grandes grupos de nós entram ou saem do sistema. Além disso, quando comparado com protocolos do estado da arte, ECO SYNC Tree provou ser o mais adequado para ambientes na periferia, exibindo melhor desempenho e escalabilidade em cenários com elevadas taxas de *churn*.

Palavras-chave: Sistemas de Armazenamento Distribuídos, Geo-Replicação, Coerência Causal+, Computação na Periferia, Difusão Causal, CRDTs.

Contents

Annexes

List of Figures

List of Tables

List of Algorithms

1

Introduction

Distributed storage systems are essential for supporting large-scale on-line services. Today's web and mobile applications are becoming increasingly demanding when it comes to offering their users the best possible experience [11]. As such, they need scalable databases that provide high availability and low user perceived latency. Traditional ACID databases are not able to meet these requirements because they choose to offer strong consistency over availability [3]. Recently, numerous new storage systems have emerged that combine weak consistency models with replication techniques as a way to remain responsive and available [2, 5, 9, 12, 16, 27, 29, 30, 42].

Many large-scale on-line services have users spread around the globe. As such, they tend to use storage systems that are geo-replicated, i.e., systems that have replicas of the database in strategic geographical locations in order to be close to a majority of their users, thus improving the response times of applications while making the system fault tolerant. To take full advantage of data locality, some systems combine geo-replication with partial replication [9, 16, 27, 42]. This combination allows replicas to store only data objects that are accessed by the users in their region, thus requiring less resourceful devices to store and manage the data. Some of these systems opt for server-side approaches [9, 16] and store replicas in the cloud, while others bring the storage and computation to the client devices [27, 42].

Relying solely on cloud infrastructures tends to increase client perceived latency. This happens because the network may become a bottleneck when all client requests need to be forwarded to distant servers in the cloud. On the other end of the spectrum, moving storage and computation to the client-side is often too heavy for resource poor client devices. As of lately, attention has been brought to the edge computing paradigm, where storage and computation are moved to devices in the periphery of the network that are closer to clients (e.g., ISP servers and 5G towers), thus allowing applications to have lower response times and to take advantage of data locality.

Several recently proposed distributed storage systems have adopted weak consistency models, choosing to offer high availability over strong data consistency in order to meet the standards of responsive web applications. While some systems implement eventual consistency [11], many have been proposed to implement causal+ consistency [5, 12, 29, 30], as it provides availability while enforcing a set of guarantees that respect the cause and effect relationship (captured by Lamport's happens-before relationship [20]). This allows developers to more easily reason about the state of the system and provides sensible semantics for applications.

Most of the solutions that provide causal consistency while trying to minimise the amount of used metadata resort to simplistic conflict resolution policies that are not adequate for some applications (e.g., Last-Writer-Wins [5, 29, 30]). Conflict-free Replicated Data Types (CRDTs) [33] were created to solve this problem, allowing replicas to apply concurrent operations without any coordination, while still guaranteeing that their states will eventually converge. CRDTs offer sophisticated concurrency semantics for many data types, which can be used to configure conflict resolution policies adapted to the needs of different applications.

1.1 Motivation

Most of the existing systems that offer causal+ consistency are not able to scale with the number of replicas. In order to maintain the operation's causal dependencies, sizeable metadata is kept locally and transmitted alongside the operation payloads [16]. This metadata tends to grow linearly with the number of replicas, making it impossible for these systems to be used in environments with hundreds (or thousands) of replicas.

Some solutions greatly reduce the amount of metadata transmitted by leveraging tree shaped topologies [9]. However, because these topologies are computed off-line, they are not capable of adapting to changes in the configuration of the system (due to nodes joining and leaving the system, or alterations in the communication patterns between replicas).

Furthermore, while some recent systems have opted to use CRDTs for conflict resolution [2, 27, 42], they either fail to take advantage of partial replication, or they do so exclusively on the client-side, thus overwhelming resource-limited client devices by making them store large amounts of data and perform excessive amounts of computations.

By leveraging on existing edge nodes, replicas of the data can be stored closer to smaller groups of clients. Other than having the advantage of being replicated at a larger set of sites (when compared to the cloud), this decreases end-to-end latency, which is critical for responsive applications [36], and ensures that the durability of data is not compromised. However, developing dynamic and scalable replication schemes that are capable of extending towards the edge is a complex task [25], because in order to ensure convergence and availability, the protocol needs to adapt and remain correct when replicas fail, or when more replicas are added to the system.

Adding to the dynamic nature of the edge, come the challenges of implementing causally consistent systems. Choosing the right way to track causality is not an easy task due to the trade-off between data freshness and throughput [18]. By tracking causality with more fine grained approaches we incur in storage overhead by increasing the amount of metadata, which can degrade the throughput of the system. However, because the causality information is so detailed, the system is able to make remote updates visible more rapidly. On the other hand, when choosing to track causality with less granular approaches, the compression of metadata causes operations to have false dependencies, increasing the update visibility times [16].

Finally, in order to give support to demanding large-scale web applications, new edge-based storage systems need to be scalable enough to support hundreds or thousands of replicas. The challenge is to find ways to keep metadata constant and bounded as the number of operations and replicas of the data store increases.

1.2 Problem Statement

The primary goal of this work was to overcome the shortcomings mentioned above by exploring new edge-based causal+ geo-replication schemes. To this end, this work proposes a solution that retains efficiency and scalability by using minimal metadata to track causality, while enabling the use of sophisticated concurrency semantics by leveraging on CRDTs. Furthermore, our solution adapts to nodes joining and leaving the system without hindering the overall system performance.

1.3 Contributions

The main contributions of this thesis can be summarised as follows:

- A new decentralised causal broadcast algorithm, named SYNC Tree, that is scalable to hundreds of replicas by using minimal metadata to track causality, and efficiently adapts to membership changes by leveraging on a tree topology paired with a novel synchronisation mechanism.

- A configurable operation log garbage collection mechanism, and a state transfer mechanism, that reduce the memory requirements and the overall synchronisation time of our base solution, thus culminating in our final solution, ECO SYNC Tree.

- A small CRDT library that, allied to our causal broadcast protocol, can be used by distributed storage systems to offer causal+ consistency.

- An experimental evaluation of SYNC Tree and ECO SYNC Tree, using our CRDT library and a simple application, that shows the benefits of SYNC Tree when compared with other causal broadcast algorithms, as well as the improvements achieved with ECO SYNC Tree by adding the aforementioned mechanisms.

- A simple tree visualiser with an easy to use graphical user interface, created with the purpose of being a visual aid to understand the behaviour of our dynamic tree.

Research Context

The work conducted in the thesis is contextualised in the New Generation of Data Storage And Management Systems (NG-STORAGE) national funded project.

This project aims to tackle new emergent challenges in data storage and management systems related with three complementary aspects: i) flexible and large-scale partial replication; ii) self-management of replicas life-cycles, and iii) providing different consistency guarantees when replicating data objects.

This work brought relevant advances in the project's context by exploring a new edge-based replication scheme that ensures causal+ consistency, by leveraging on CRDTs, and adapts to membership changes with its dynamic tree topology.

Publications

The work presented here resulted in the following publication [40]:

- *Difusão Causal Flexível e Escalável para Replicação na Periferia.*

  Ema Vieira, Pedro Fouto, Nuno Preguiça and João Leitão.

  Proceedings of the 12th Simpósio de Informática (INForum 2021), Lisbon, Portugal, September 2021.

1.4 Document Structure

The rest of this document is organised as follows:

Chapter 2 discusses related work. It covers topics such as peer-to-peer networks and decentralised communication; different replication strategies; strong and weak consistency models, focusing primarily on causal (and causal+) consistency and causality tracking mechanisms; CRDTs as a conflict resolution technique that ensures replica convergence; and some relevant existing systems that provide causal+ consistency guarantees.

Chapter 3 presents the system model, architecture, and design of our new causal broadcast algorithm, SYNC Tree, as well as its improved final iteration, ECO SYNC Tree.

Chapter 4 details the implementation of the prototypes of both versions of our algorithm. It also discusses a simple visualisation tool created to help comprehend the behaviour of our protocol when faced with membership changes.

Chapter 5 presents an extensive experimental evaluation of both versions of our protocol. SYNC Tree is compared against other decentralised causal broadcast protocols. It is also compared with ECO SYNC Tree, to concretely demonstrate the benefits of the new mechanisms introduced.

Chapter 6 concludes this thesis and presents possible directions for future work.

2 Related Work

In this chapter, we present and discuss related work that is relevant for the context and objectives of this thesis. To help understand the contents of this document, we focus on the following topics:

In Section 2.1, we explore peer-to-peer networks, overlay networks, and decentralised communication strategies, due to their fulcral role in edge-based distributed storage systems.

In Section 2.2, we discuss replication, its benefits and downsides, and some design choices that may affect the performance of replicated storage systems.

In Section 2.3, we discuss several strong and weak consistency models, giving some emphasis to causal and causal+ consistency. Additionally, we cover some of the most common causality tracking mechanisms and conflict resolution techniques.

In Section 2.4, we cover the subject of CRDTs, discussing concurrency semantics and synchronisation methods. Furthermore, we present some examples of existing CRDT implementations.

In Section 2.5, we present an in depth description and analysis of some relevant existing replicated data storage systems, focusing mainly on those that use CRDTs to offer configurable conflict resolution policies.

2.1 Peer-to-Peer Networks

With the ever growing number of user devices, cloud infrastructures can suffer from network bottlenecks that increase client perceived latency and negatively impact user experience. Edge-based distributed infrastructures are able to solve this problem by relying on the hundreds (if not thousands) of existing devices on the periphery of the network, thus achieving lower response times [36]. In order for edge-based distributed storage systems to take full advantage of the data locality, the node's communication patterns must allow edge devices that are physically closer to each other to communicate and share information. Peer-to-peer (P2P) architectures are a viable and efficient way of doing this.

P2P networks are decentralised systems in which there is no central component. Each node stores data, performs computation, and is able to coordinate with other nodes that belong to the same network [32]. Nodes can join and leave the system without compromising its functionality, due to the network's automatic reconfiguration capabilities.

Because no central component is used, there is no single point of failure, making these types of networks extremely tolerant to faults and attacks. Furthermore, because data and computations are spread out between all the nodes of the network, the load on the physical devices in lessened, thus enabling P2P networks to scale to the hundreds/thousands of connected devices.

2.1.1 Overlay Networks

In large P2P systems where nodes can freely join and leave the network, ensuring that all nodes know all other nodes at all times (i.e., maintaining a globally known membership) can be very costly and unsustainable. As such, these systems make use of overlay networks.

An overlay network is a logical network built on top of another (logical or physical) network. In P2P systems, the overlay network creates logical point-to-point links between the nodes, that are physically materialised by one or more physical Internet links. Note that the nodes are not logically connected to all other nodes, but only to a subset of all nodes. Thus, each node effectively has a partial view of the whole system.

Overlays can be structured or unstructured. Each application should choose the appropriate type of overlay according to the expected usage of the network, as well as the patterns of nodes joining and leaving the system.

Structured overlays: In this type of overlay, some properties of the topology are known *a priori*. Most of the time, these properties are related with node identifiers which are then used to determine the node's position in the network (that usually has the structure of a ring) [31, 35, 39]. This is the case with Distributed Hash Tables (DHTs), where each node of the network is responsible for a given set of keys that depends on their unique identifier. Other structured overlays form topologies like trees [22], thus allowing for optimal propagation of messages with low overhead.

In general, structured overlays are efficient for particular types of applications, such as application level routing, exact match searches, as well as broadcasting messages. However, due to their rigid structure, they are less robust to failures (the choice of node replacement is limited), and they require algorithms with greater complexity.

Unstructured overlays: These overlays present random topologies, usually forming connected graphs [23, 41]. Because of this, they are highly resilient to failures, easily maintained, and great for cooperative message dissemination (which is useful for replication of data across large numbers of nodes). On the other hand, they are very inefficient in finding particular objects in a network because they have to query all the nodes in order to guarantee that they reach the desired one.

2.1.2 Overlay Management Protocols

Overlay management protocols are distributed algorithms used to maintain the partial views of the nodes of the overlay network. These protocols use two main strategies:

Cyclic strategy: Protocols that implement this strategy, such as Cyclon [41], periodically change the partial view by making a node exchange its information with its neighbours. As such, partial views change even if the system remains stable, i.e., if there are no changes to the membership of the system.

Reactive strategy: With this strategy, the partial view is changed due to external events of the system, like a node joining or leaving the network. So, if no changes happen, the views remain unaltered. HyParView [23] is an example of an overlay management protocol that uses a reactive strategy.

2.1.2.1 HyParView

HyParView [23] is an overlay management protocol that maintains two partial views of the system in each node: a small active view and a large passive view. Each node maintains symmetrical TCP connections to all the nodes in its active view, thus creating an overlay network. The passive view stores node identifiers so that the protocol can ensure connectivity when faced with large numbers of node failures, by keeping the active views filled.

The active view is managed with a reactive strategy, where each node uses TCP events to check if a node in its active view has crashed. When a node crashes, the protocol tries to connect to a random node from its passive view. If the temporary connection is successfully established, the node will send a neighbour request with his identifier and a priority to the chosen neighbour. When the neighbour replies, the node will either establish a permanent connection to that node and add it to its active view, or start the whole neighbour process again.

On the other hand, the passive view is maintained with a cyclic strategy. Each node will, periodically, send a shuffle message containing a set of node identifiers to a random node from its active view. The receiver node will reply with his own set of node identifiers. Finally, both nodes will try to incorporate the nodes that correspond to the received identifiers into their passive views.

2.1.3 Decentralised Communication

A decentralised distributed system is composed by a set of nodes that are interconnected through a network and seek to cooperatively perform tasks. In order to do this, the nodes must be able to communicate in a decentralised manner.

Gossip-based, or epidemic-based, algorithms were first introduced for their high reliability in disseminating messages over large-scale distributed systems built on top of unstructured overlay networks. Gossiping refers to the repeated probabilistic exchange of information between nodes.

As we mentioned in Section 2.1.1, for scalability and performance purposes, each node will only have a partial view of the system, because keeping an updated full membership of such large systems is not feasible. To counter the fact that each node only knows a subset of all the nodes in the system, these protocols introduce redundancy as a way to ensure reliability.

There are four main strategies to implement gossip protocols [24]:

Eager push strategy: Nodes send messages to a set of randomly selected neighbours when these are received for the first time. The number of neighbours the messages are sent to can vary, ranging from 1 neighbour (Random Walk) to all the neighbours of the partial view of the node (Flood).

Lazy push strategy: After receiving a message for the first time, the nodes send only the message identifiers to a set of randomly selected neighbours. If those neighbours have not received those messages, they can request that the messages be sent by the sender of the identifiers.

Pull strategy: Nodes periodically ask a set of randomly selected neighbours for information about their received messages. If there is a message that they have not yet received, they can request it.

Hybrid strategies: Use combinations of eager push and pull gossip, or eager push and lazy push gossip. Eager push is used to propagate the messages, while pull or lazy push gossip are used to fetch messages that were not received via eager push.

Other than these strategies, it is also possible for gossip protocols to embed secondary topologies on top of the overlay network. Next, we discuss one such protocol.

2.1.3.1 Plumtree

In eager push gossip protocols, after propagating the first message, the paths followed by that message form a spanning tree of the network. This fact lead to the emergence of tree-based gossip algorithms, such as Plumtree [22] (Epidemic Broadcast Trees).

This protocol uses a topology aware strategy that reduces the inherent redundancy of gossip protocols by creating a latency optimised multicast tree on top of the underlying unstructured overlay network. One of the key aspects of this algorithm is its ability to reconfigure the tree, in a decentralised manner, even when nodes are joining and leaving the system.

Plumtree makes use of two types of gossip strategies: eager push and lazy push gossip.

Eager push gossip is used to send messages to nodes via tree branches. The remaining links of the overlay are used to propagate message identifiers with lazy push gossip, so that nodes that have not received the corresponding messages can make pull requests. This lazy push mechanism provides high reliability and a way to heal the broadcast tree from disconnections.

Is is important to note that the overlay management protocol should guarantee that the nodes present symmetric partial views, thus allowing the tree to be shared by multiple nodes.

The Plumtree algorithm can be divided into two main mechanisms:

1. **Tree Creation:** The nodes keep two sets of peers: the eager push peers, and the lazy push peers. When a node first receives a message, it adds the sender to the eager push peer set. This creates a bidirectional link that belongs to the tree. When a duplicate message is received, the node moves the sender to the lazy push peer set and sends him a PRUNE message, so that he also moves the link to his lazy push peer set, effectively removing the tree branch. This mechanism creates a tree that minimises the latency after the first broadcast is finished.

2. **Announcements:** The lazy push strategy is implemented by periodically sending IHAVE messages (that contain the identifiers of the messages that the node has received since the last IHAVE message) to the lazy push peer set. These messages are used to pull missing messages and to reconstruct the tree when it becomes disconnected (e.g., in case of node failures). When a node receives an IHAVE message, it waits for a predetermined period of time. If, after that period time, it has not received the messages with the identifiers present in the IHAVE message through the tree branches, the node sends a GRAFT message to its sender. This message acts as a pull request of the message, but also adds the corresponding link to the tree.

2.1.4 Discussion

Our solution was specifically designed to be used by edge-based distributed data storage systems. As such, using the peer-to-peer approach seems appropriate. Unstructured overlay networks and gossip broadcast algorithms built on top of them are the most efficient at achieving replication when dealing with hundreds of replicas. For this reason, and because of its stability in the absence of nodes joining and leaving the system, our broadcast algorithm uses HyParView [23] as the underlying overlay management protocol. Furthermore, our solution is inspired by the Plumtree [22] protocol, due to its tree topology and dynamism.

2.2 Replication

Distributed storage systems that support large-scale on-line applications strive to provide their users with an optimal experience [11]. Nowadays, users are becoming more and more demanding, so these systems resort to replicating data across multiple replicas in order to provide some desirable properties:

Availability and fault-tolerance: A replica can become unavailable due to hardware failure, software fault, network partitions, or even due to external events like floods or power outages. By maintaining copies of the data in several replicas, we ensure that the system remains functional in the presence of a large number of replica failures, by guaranteeing that there is always a replica that is available to serve client requests.

High scalability: When several replicas maintain a copy of the data, the system can scale to large numbers of clients by using load balancing to distribute the requests across those replicas, particularly read operations (i.e., operations that do not modify the data).

Low latency: By spreading replicas of the system throughout distinct geographical locations (geo-replication), we can ensure that the users in their proximity experience lower user perceived latency, because the time to access the data decreases.

There is, however, a downside to using replication protocols: having multiple copies of the data may lead to consistency problems. When a copy of the data is modified at a given replica it becomes inconsistent with all other replicas. This creates the need for a protocol that propagates the modifications to all the remaining replicas so that they converge to a common state. Furthermore, systems may need to control which versions of the data objects can be observed by clients, so as to avoid exposing them to anomalies.

2.2.1 Geo-Replication

When replicating a storage system, we could choose to place all the replicas in close physical proximity to each other (e.g., in the same data centre, or in neighbouring data centres). However, this would not make the system fault-tolerant or available in the face of multiple replica failures caused, for example, by a natural disaster. If such an external event affected the geographical area of the data centre(s), it could cause all the replicas to fail simultaneously, rendering the system unusable [1].

To solve this problem we can geo-replicate the system by distributing the replicas across several distant geographical locations, making the system fault-tolerant by reducing the probability of all replicas becoming unavailable at the same time [2, 5, 12, 16, 30]. These replicas are usually placed in strategic locations, so as to be closer to the users of the system, reducing user perceived latency and giving them an overall improved experience.

Because replicas are far apart from each other, the time it takes for them to propagate updates to each other increases. However, these systems tend to prioritise user satisfaction, so they usually opt to give lower response times to users over making the replicas' synchronisation process faster.

2.2.2 Replication Schemes

Replication protocols often distribute data across replicas in a way that trades higher levels of redundancy for less communication and storage requirements.

Full replication: Refers to the approach where every replica of the system maintains a complete copy of the data [2, 9, 29, 30]. This type of replication is very costly when it comes to communication between replicas to propagate updates, because they must be sent to (and executed in) all sites. Additionally, as data grows so do the storage needs for the system. Oftentimes, it is not reasonable to assume that the replicas have enough resources to store the full dataset.

Partial replication: Refers to an approach in which replicas may not have full copies of the dataset, but only of a subset of it [9, 16, 27, 42]. This means that some replicas may store the same subset, while others store different parts of the full dataset. This scheme greatly reduces the cost of synchronising replicas, because the updates only need to be sent to the replicas that maintain a copy of the modified data item. However, this adds the additional complexity of keeping track of which replicas store each data item, so that the updates are only propagated to them. Furthermore, this approach reduces the (total) storage requirements of the replicas and, when combined with geo-replication, increases spacial locality, by storing data items only in the places where they are accessed and modified. These properties are essential in edge computing because edge nodes tend to have less resources, and thus less storage space, and because users in a given geographical area are more likely to access the same data items repeatedly.

2.2.3 Replication Models

Replication protocols can use different models when synchronising replicas (i.e., propagating operations from one replica to the others). There are two important models that choose between low user perceived latency and showing users consistent data.

Synchronous replication: When a system uses synchronous replication it ensures that the operation is propagated to (and executed in) all replicas before sending a response to the client. This type of synchronisation is normally employed when trying to provide strong consistency guarantees, because it forces all replicas to have consistent states. Due to the extra steps performed before sending a response to the client, this type of synchronisation makes for non-scalable systems (propagating updates takes longer if the number of replicas grows, and if they are geo-replicated), increases user perceived latency, but makes it easier to ensure durability.

Asynchronous replication: On the other hand, asynchronous replication sends a response to the client before propagating the operation to the other replicas [29]. This allows for lower user perceived latency and improves user experience. Because operations are propagated in the background after replying to the client, replicas can present divergent states for certain periods of time, thus allowing clients to have inconsistent views of the system. As such, this type of synchronisation is normally used when providing weaker consistency guarantees. These systems tend to scale better than systems with synchronous replication, because the synchronisation costs are mitigated by the fast responses to users. Note that, if a replica that has not propagated an update fails, the system might not be able to guarantee durability.

2.2.4 Horizontal Partitioning / Sharding

A distributed storage system usually has multiple machines in a single data centre. If the same copy of data is stored in all of them, each node will have to execute all updates to data items stored in that data centre, hindering the system's scalability. Sharding is a technique that consists in splitting the dataset into several fractions (often called partitions) and storing them on different nodes of the data centre [2, 5, 29, 30].

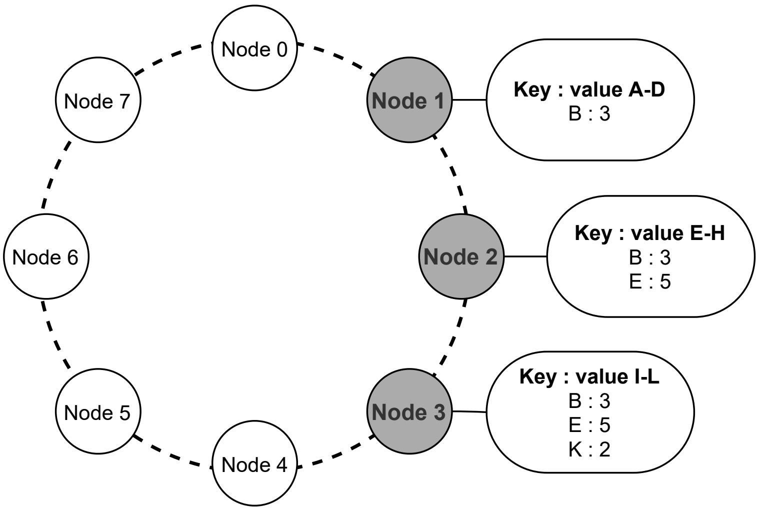


Figure 2.1: Example of sharding in data centre organised in a ring.

Consider the data centre illustrated in Figure 2.1 where the nodes are organised in a ring structure. This structure can be used to determine the nodes in which a partition is stored. In this example, *Node 1* stores all keys from A to D, *Node 2* stores all keys from E to H, and *Node 3* stores all keys from I to L. For increased tolerance to failures, each partition tends to be replicated in more than one node. In Figure 2.1, all keys that are stored in *Node 1* (i.e., key *B*) were also stored in *Node 2* and *3*, as these are the two closest clockwise neighbours of *Node 1*.

By using this mechanism, the data store is able to process several concurrent requests by distributing the load across the nodes of each data centre. Systems that use sharding tend to have good horizontal scalability (scaling out), because they can add more machines if the number of client requests requires it.

2.2.5 Discussion

As there are several edge points of presence spread out around the world, our solution must support geo-replication. In order to reduce client perceived latency, we opted for asynchronous replication. Although edge nodes would benefit from storing only a subset of all data objects, partial replication is not a feature our solution offers.

2.3 Consistency Models

The CAP theorem [17] states that it is impossible for a distributed system to, simultaneously, provide the following properties:

1. **Strong consistency:** The clients of the system always see the most recently written value for all data items;

2. **Availability:** All operations issued to non-faulty replicas eventually complete;

3. **Partition tolerance:** The system must remain functional even in the presence of network partitions that may affect the communication between replicas.

This impossibility result implies that a distributed system is only able to fully provide two of the three properties mentioned above. However, in large-scale distributed systems, network partitions are unavoidable and when they happen the system may stop functioning altogether (violating the availability property), or replicas' states can diverge (violating strong consistency guarantees) [10]. As such, most systems opt to be partition tolerant and, in the presence of partitions, they choose between providing availability (AP) or strong consistency (CP).

As mentioned before, nowadays most systems are concerned with providing users with the best experience possible and, consequently, tend to prioritise availability over strong consistency, adopting weaker consistency models.

Consistency model: Specifies how the replicas' states diverge due to write operations. It also refers to a set of safety restrictions that define what the clients can see, given the system's and the client's histories. Note that these restrictions only apply to read operations, so when a storage system does not receive read requests the consistency model is irrelevant.

There are two main categories of consistency models: strong and weak. In the following sections we define them and present some of their advantages and disadvantages.

2.3.1 Strong Consistency

When using strong consistency models, the system gives clients the illusion that a single replica exists despite the fact that it is distributed and that operations are executed in several replicas [17]. These models guarantee that clients always see the most recent values written in the system. In order to provide the illusion of a single replica, the system must guarantee the following key property:

Atomicity: Given a system trace, there must exist a serialisation point for each operation (between the point of its reception and the point in time when the reply is sent to the client) so that the story of the system as a whole, considering the relative order between operations, makes sense. Note that an operation that does not finish (and never returns a reply to the client) may or may not have a serialisation point defined. In other words, there must exist a total order for all operations, so that they are observed in the same order by all clients, maintaining the consistency of the data.

By ensuring this, it becomes easy to reason about the possible states of the system, and non-commutative operations can be executed without causing replicas to diverge.

Although it might appear simple to ensure that replicas execute all operations in the same order, it is actually a very complex problem to solve. This requires all replicas to agree on a sequence of operations to execute, effectively achieving consensus. Note that the consensus problem has been proved impossible to solve in the asynchronous model where replicas can fail [15]. Seen as this model is a good approximation of the real world, it is impossible to fully solve the consensus problem, but it is possible to implement algorithms that solve approximations of consensus with relaxed properties, such as Paxos [21].

The downside to these algorithms is that they require several high latency communications steps, for example: a replica must contact a majority of all replicas before confirming the operation to the client, so as to guarantee that the operation is executed in a quorum of replicas, enforcing data consistency. By itself, this hinders the availability of the system in the presence of multiple replica failures (i.e., the system stops if a majority of replicas becomes unavailable). Furthermore, because these systems tend to be geo-replicated, the latency's lower bound depends on the highest Round Trip Time (RTT) between two replicas.

Next, we discuss two of the most relevant strong consistency models:

Linearizability: A system that provides linearizability ensures that replicas execute all operations in the same order. Additionally, that total order must respect the real-time ordering of events, meaning that operations must be executed in the order by which clients issued them.

Serializability: On the other hand, serializability only ensures that there is a total order of operation executions across all replicas, even if that order does not respect the real-time ordering of events. This allows the rearranging of operations, as long as all replicas execute the same sequence of operations in the end.

2.3.2 Weak Consistency

In contrast to strong consistency models, when providing weak consistency the operations do not need to be executed in the same order across all replicas. As such, replicas may diverge and clients may read stale values, or even inconsistent ones.

These systems tend to have lower user perceived latency because replicas have no need to contact a majority quorum before confirming the operation to the client. This implies that the availability of these systems is not compromised even in the presence of multiple replica failures.

Next, we discuss three of the most relevant weak consistency models:

Eventual consistency: Systems that provide eventual consistency are highly available but have no safety guarantees [11]. Because of this, it is debatable if eventual consistency should be considered a consistency model. The only promise these systems make is that eventually, when no more operation requests are issued to the system, replicas will converge to the same state. This implies that, before this happens, the system might allow users to see stale or even inconsistent values. To ensure convergence, conflict resolution techniques (like the ones discussed in Section 2.3.4) must be employed.

Causal consistency: Causal consistency is one of the strongest forms of weak consistency a system can provide while still remaining available. This model guarantees that all replicas see states that respect the happens-before relationship between operations, thus respecting the notion of potential causality [20]. If we consider two operations o_1 and o_2, and a partial order $\prec$ that represents the happens-before relationship, if $o_1 \prec o_2$ then causal consistency guarantees that all replicas can only observe the effects of o_2 after observing the effects of o_1. More formally, we can say that o_1 happened before o_2, $o_1 \prec o_2$, iff one of the following rules applies:

- **Execution thread:** o_1 and o_2 are operations in a single thread of execution, and o_1 happens before o_2;
- **Reads-from:** o_1 is a write operation and o_2 is a read operation that reads the value written by o_1;
- **Transitivity:** Considering another operation o_3, if $o_1 \prec o_3$ and $o_3 \prec o_2$, then $o_1 \prec o_2$.

Note that the happens-before relationship is a partial order, meaning that not all operations are ordered by it. When two operation have no causal relation between them we say that they are concurrent. The effects of two concurrent operations can be observed in any order because they do not depend on each other.

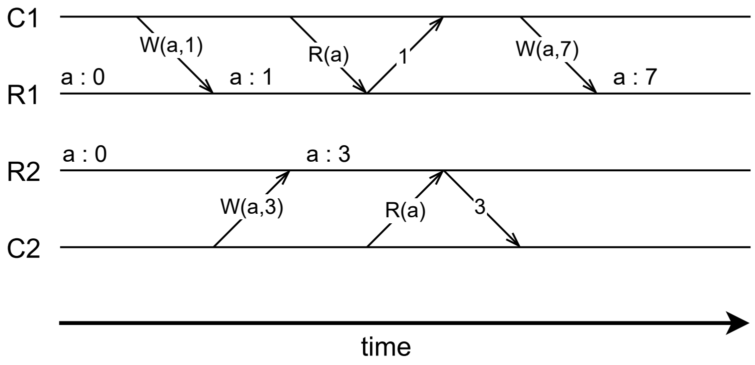


Figure 2.2: Execution with replica divergence in causal consistency.

One important aspect of causal consistency is that it does not guarantee replica convergence. Figure 2.2 presents an execution where the states of two replicas, *R1* and *R2*, diverge under causal consistency. *a* is the key of an object stored in both replicas. As can be seen in this execution, when two clients, *C1* and *C2*, write values for the same key in two different replicas without ever seeing the effects of the other ones' write, their operations do not have causal dependencies between each other (i.e., they are concurrent). As such, they will be executed and the replicas will diverge. If the clients always interact with their corresponding replica (i.e., if they have a sticky session), the replicas will never communicate and their states will diverge forever without ever violating causal consistency.

Causal+ consistency: Causal+ consistency, first coined in [29], combines the properties of causal consistency with those of eventual consistency, providing causal consistency with convergent conflict handling. Causal+ consistency does not order concurrent operations, with the exception of conflicting ones. Two unrelated operations are in conflict if they are writes for the same key happening at two different replicas. If we analyse the example from Figure 2.2 again, we can say that *W(a,1)*, in replica *R1*, and *W(a,3)*, in replica *R2*, are conflicting operations. Contrary to causal consistency, under causal+ consistency this conflict would have to be resolved in order for the replicas' states to converge.

2.3.3 Causality Tracking Mechanisms

There are several ways for a system to enforce causal consistency. As described in [28], some systems rely on the topology of the network to disseminate operations in causal fashion. Whether the nodes are arranged in a star or tree topology, if the channels between them are FIFO (First In, First Out) it is possible to guarantee that the dependencies of an operation have already been propagated at the time of its reception.

Other systems, such as C^3 [16] and Saturn [9], separate the metadata management from the data store itself, and deliver the metadata to each data centre while respecting causal order. The data centres have to wait for the reception of the metadata before applying the corresponding operation and making its effects visible to (local) clients.

In P2P networks, where all nodes can communicate among themselves, the system needs to keep track of causal dependencies between operations. The simplest way to track these dependencies is by using causal histories, explained in detail in [7].

Causal history: The causal history of an event can be represented by the set of all events that happened before it (i.e., all events it causally depends on).

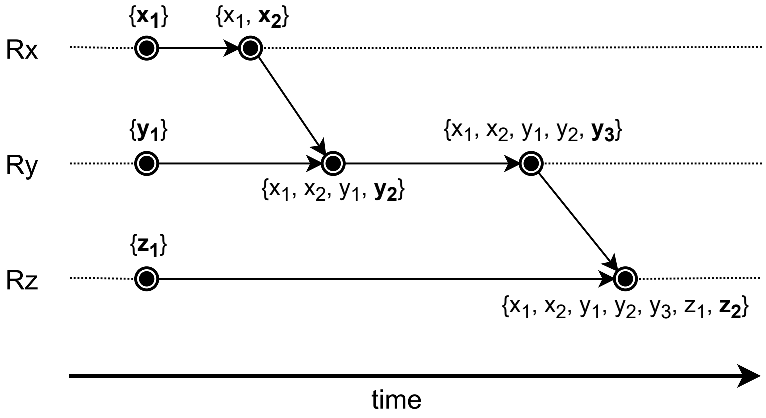


Figure 2.3: Example of causal histories.

Consider the system with three replicas, *Rx*, *Ry* and *Rz*, represented by Figure 2.3. In this system each event is assigned an unique identifier that consists of the replica's name and the value of a local monotonically increasing counter (logical clock). When a new event occurs, its causal history is the union of its unique identifier and the causal history of the previous event in that replica. For example, the causal history of the second event in replica *Rx* is $H_{x_2} = \{x_1, x_2\}$, because it is the union of its unique identifier, x_2, with the causal history of x_1. Furthermore, when a node sends a message to another one, the causal history of the send event is also transmitted along with the message so that the causal history of the remote node can be merged with the local history. This is exemplified in Figure 2.3 when *Rz* receives a message from *Ry* and creates a new event, z_2, with the corresponding merged causal history.

When using causal histories, to verify if an event is causally dependent on another we can use set inclusion:

- If the unique identifier of an event is contained in the causal history of another, then the latter is causally dependent on the former;
- If neither identifier is contained in the causal history of the other event, then the events are not causally related (i.e., they are concurrent).

Note that causal histories are an effective but inefficient way of tracking causality because the size of the metadata grows with the number of operations, becoming unsustainable. Next, we discuss some of the techniques proposed to overcome this problem.

Direct dependencies: One important property that holds in causal histories is transitivity. It can be used to compress causal histories into sets that only include the direct dependencies of an event, because all other dependencies are implicitly present. Consider the causal history of y_2 shown in Figure 2.3: it contains both x_1 and x_2 but, because the causal history of x_2 already contains x_1, it could be omitted without losing the causality information, still allowing us to build the full causal history transitively. This approach is used by systems like COPS [29] and Eiger [30], although the latter uses a superset of the nearest dependencies that includes all dependencies that have a path of length one to the current operation (one-hop dependencies).

Vector clocks: Another compact way of representing causal dependencies are vector clocks [14]. When analysing a given causal history, we can see that if an event is present then all the preceding events from that node are also present. This implies that it is enough to store the latest event from each node in order to know the full causal history. By taking advantage of this, we can represent causal dependencies in a very compact way by building a vector of Lamport clocks with one entry per node, like so: $H_{y_3} = \{Rx \mapsto 2, Ry \mapsto 3, Rz \mapsto 0\} = [2, 3, 0]$.

To know if there is a causal dependency between two operations we must check each entry in the respective vector clocks to see if one vector is strictly smaller than the other: $x \prec y$ iff $\forall i : V_x[i] \leq V_y[i] \wedge \exists j : V_x[j] < V_y[j]$. When a new event occurs in a node, we simply increase the corresponding entry in the vector clock instead of generating a new unique identifier for it. Finally, when two nodes communicate we need to merge the two causal histories. We can do this by maintaining the maximum value for each position in the vector clocks: $\forall i : V_z[i] = max(V_x[i], V_y[i])$. For example, if we merged [3,3,0] with [2,4,1] the result would be the vector [3,4,1].

Version vectors: When trying to enforce causal consistency in distributed storage systems, it is usually enough to register the events that effectively alter the replicas. Vector clocks that are only incremented when write operations occur are called version vectors. Systems such as C^3 [16] and ChainReaction [5] use versions vector to track causality relations between operations across different data centres.

2.3.4 Conflict Resolution Techniques

Systems that provide available forms of consistency, such as eventual or causal+ consistency, need to have conflict resolution mechanisms in place so that they can guarantee that, even when there are concurrent writes for the same key, the replicas of the system will eventually converge to a common state. There are three classical ways of implementing convergent conflict handling:

Last-Writer-Wins (LWW) policy: This approach consists of defining an order among any concurrent write operations and choosing to apply only the effects of the last one. Even though this is one of the most commonly used approaches [5, 29, 30], it is not adequate for all data types because the effects of some client operations are lost.

Consider a set $s = \{a, b, c\}$. When there are two concurrent add operations, $add(d)$ and $add(e)$, only one of the elements will be present in the final set, because the effects of one of the concurrent operations will be discarded. So, the final set will either be $s = \{a, b, c, d\}$ or $s = \{a, b, c, e\}$, and not the intended set, $s = \{a, b, c, d, e\}$.

Application-dependent policy: Some systems opt to expose the divergence to the application, delegating the task of solving the conflict. One such system is Amazon's Dynamo [11], that returns multiple values when executing a read operation that detected concurrent writes. It is the application's responsibility to merge the values and write the new value back to the system.

Merge procedure: The last approach is to give replicas access to a deterministic merge procedure that receives the two divergent states and computes a new merged state. This procedure is normally specific to each application. A particular case of this approach are CRDTs [33], where the logic for merging is encapsulated within data types. CRDTs are discussed in more detail in Section 2.4.

2.3.5 Discussion

To ensure that our solution remains available and tolerant to network partitions, we opted for a weak consistency model. As most applications need some guarantees regarding the order in which operations are received, our solution offers causal+ consistency, respecting the cause-effect relationship and ensuring replica convergence. As is the case with Saturn [9], we opted for a tree topology (in which the nodes are connected by FIFO channels) to ensure causal delivery while keeping the metadata size small. Because causal delivery, by itself, does not ensure replica convergence, we used CRDTs for conflict resolution.

2.4 CRDTs

CRDTs [33] are a family of abstract data types specifically designed to support replication and operation over replicas in large-scale distributed systems [2, 27, 42]. These data types can be concurrently updated with no need for coordination between replicas. CRDTs guarantee that any two replicas that have seen the same set of updates will, eventually and deterministically, converge to a common state. Interaction with CRDTs is made simple by their well defined interfaces that allow to update and query the state of the data structures.

To remain highly responsive and available in the presence of network failures and disconnections, all replicas accept updates at all times and propagate them in and asynchronous manner. Because global coordination is not used, replicas' states may diverge and reads may not reflect some modifications made in other replicas, allowing clients to see inconsistent states. Even though CRDTs intrinsically provide eventual consistency guarantees, when paired with a reliable causal broadcast middleware [8] they are able to provide per-object causal consistency.

Some CRDTs have been formally specified and verified by using the framework proposed in [43], which will not be detailed in this document. Instead, in the following sections we will discuss some key aspects of CRDTs, described in more detail in [33].

2.4.1 Concurrency Semantics

Because CRDTs are specifically designed to allow uncoordinated updates, it is important to define the behaviour of the data objects in the presence of concurrent updates.

CRDTs can be used to implement several different data types. For some of them, operations are inherently commutative and executing them in any given order will make the replicas converge to the same common state. However, this is not true for most data types and, in these cases, there are several possible concurrency semantics that can be used. Note that the expected behaviour in the presence of concurrency can be different depending on the purposes and requirements of an application. As such, CRDTs strive to give application developers the freedom to choose the most adequate semantics for their application.

We have previously mentioned two concepts that are important to keep in mind when defining concurrency semantics of CRDTs:

1. **Total order among operations:** It is possible to define a total order between all operations so that we can define the **last-writer-wins semantics**, where the state of the replicas will present the effects of the last operation according to that total order;

2. **Happens-before relationship:** When using CRDTs, we can say that $o_1 \prec o_2$ iff the effects of o_1 had been applied in the replica where o_2 was first executed. This partial relationship can be used to define several concurrency semantics.

Going back to the example of the set data type introduced in Section 2.3.4, it is clear that add and remove operations for the same element are not commutative. With this in mind, we can define two concurrency semantics based on the happens-before relationship.

Add-wins semantics: In the presence of a concurrent add and remove for the the same element, the add always wins over the remove and the element in question will be added to the set. An example can be seen in Figure 2.4, where replica *Rx* adds *b* to

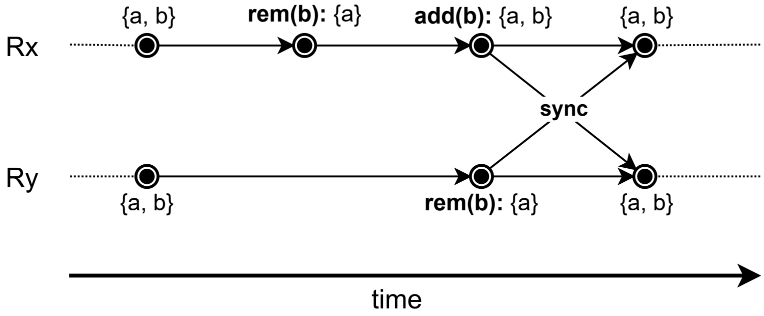


Figure 2.4: Execution with add-wins set.

the set and, concurrently, replica *Ry* removes it from the set. After the synchronisation, *b* belongs to the set (in all replicas).

Remove-wins semantics: In contrast, in the presence of a concurrent add and remove for the the same element, the remove always wins over the add and the element in question will be removed from the set. An example can be seen in Figure 2.5, where replica *Rx* adds *b* to the set and, concurrently, replica *Ry* removes it from the set. After the synchronisation, *b* no longer belongs to the set, having been removed in both replicas.

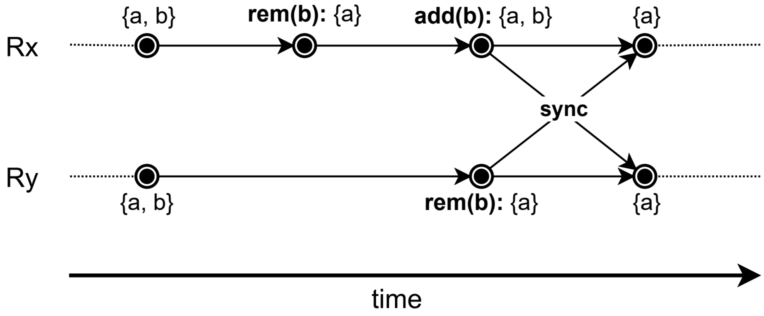


Figure 2.5: Execution with remove-wins set.

2.4.2 Synchronization Methods

When using CRDTs, the system needs to have a synchronisation mechanism that allows the replicas to propagate updates to all other replicas so that they can execute them and reach a common state. There are two main types of CRDTs that differ in the synchronisation method:

State-based CRDTs: These CRDTs use state-based replication to propagate the effects of update operations. Replicas synchronise by periodically sending a payload with their full state to a neighbouring replica. When that replica receives the state, it merges the received state with its local state by using a merge function. If the update propagation graph is connected, then every update will, eventually, reach every replica.

It has been proven, in [38], that all replicas of a CRDT will converge if three conditions are met:

1. The state, s, of the CRDT is partially ordered by $\leq$, forming a join semilattice. A **join semilattice** is a partially ordered set that has a **least upper bound** (LUB), $\sqcup$, for all pairs of elements. $m = x \sqcup y$ is a LUB of $\{x, y\}$ according to $\leq$ iff $\forall m' : x \leq m' \land y \leq m' \Rightarrow x \leq m \land y \leq m \land m \leq m'$. Note that $\sqcup$ is commutative, associative, and idempotent;
2. The state of a replica is monotonically non-decreasing across updates, i.e., an update operation always modifies the state of a replica by inflation, producing a new state that is larger or equal to the previous state according to $\leq$. Given an operation m, $s \leq m(s)$;
3. Merging the local state, s, with a remote state, s', using the merge function produces a final state that is the LUB of the two states: $s \sqcup s'$.

The main downside to state-based CRDTs is that, if the size of the data objects grows to be significantly large, they tend to become inefficient because they use more bandwidth to propagate their full state during synchronisation steps.

Operation-based CRDTs: These CRDTs use operation-based replication. When a replica receives an update that mutates its state, it applies the update locally and then propagates it to all other replicas. For each operation, the CRDTs must define two functions:

- **Generator function:** This is a function with no side-effects that executes in the original replica. It looks at the operation and the current state of the replica and generates an effector (a message) that encodes the side-effects of the operation. In Section 2.4.3 we refer to this function as *atSource*.
- **Effector function:** This function reliably delivers the effector operations to all replicas so that they can be executed, effectively updating their states. In Section 2.4.3 we refer to this function as *downstream*.

This type of CRDT uses less bandwidth than state-based CRDTs, because the payloads of the effectors are much smaller than the payload of the full state. However, it requires that all effector operations be delivered to every replica in a reliable manner and, usually, according to an order that respects the happens-before relationship (causal order). Because concurrent updates can be delivered in any order, to guarantee convergence, they must commute. Otherwise, if the underlying broadcast primitive does not ensure causal delivery, then all operations must commute.

In addition to the two main types of CRDTs, several alternative models that try to improve some of their downsides have been proposed.

Pure Operation-based CRDTs [6]**:** The definition of standard operation-based CRDTs is very relaxed. In order to deal with non-commutative operations, these CRDTs include additional causality information in the state, using it in the generate phase and sending it in messages to be used in the effect phase. This makes them able to send the full state between replicas, blurring the distinction between operation-based and state-based CRDTs.

Pure operation-based CRDTs draw a clear line between the two previous types of CRDTs by only sending operations (and their arguments) to other replicas. Data types with non-commutative operations are implemented by making use of the tagged reliable causal broadcast (TRCB) messaging API, that provides causality information when it delivers operations. It does so by assigning a vector clock to each message so that it can be used to make sure that an operation is only delivered at a replica after all its preceding operations also have been.

To represent the state of CRDTs, this implementation makes use of a partially ordered log (PO-log) of operations. Additionally, the authors introduced a PO-log compaction framework that uses causality and causal stability information in order to reduce the storage overhead. This framework removes obsolete operations from the log, keeping only the minimum number of operation needed (e.g., in the set data type, an add operation makes any previous add or remove operations for the same element obsolete). Finally, the framework uses causal stability information to determine when the causal information regarding an operation can be stripped from the log.

In [8], the authors describe a join model for dynamic CRDT environments, and propose a technique to remove metadata of causally stable operations quicker by relying on acknowledgements.

Just like operation-based CRDTs, pure operation-based CRDTs send smaller messages between replicas, using less bandwidth. However, they always depend on the existence of a reliable causal broadcast middleware. One other issue is that the state of the CRDTs is larger because the PO-log stores operations and their causal metadata (even though some operations and metadata are removed under specific conditions).

(Small) δ-CRDTs [4]**:** These CRDTs try to combine the advantages of both operation-based CRDTs, by sending only small incremental states between replicas, and state-based CRDTs, by disseminating the messages over unreliable communication channels. When using these CRDTs, operations are encoded as δ-mutators that change the state of the CRDT by generating a delta, d, that represents the effects of the most recent update operation on the state. These delta-states are sent to other replicas, instead of shipping the full CRDT state, so that they can be merged with the local replica states. Delta-states can also be combined to form delta-groups before being shipped.

If the causal order of operations is not important, then the delta-groups can be shipped using an unreliable dissemination layer. However, an anti-entropy algorithm that enforces causal delta-merging was defined, so as to provide causal consistency guarantees.

The downside to δ-CRDTs is that they assume continuous and static synchronisation patterns between replicas. This is not ideal for scenarios where clients have unreliable communication channels that manifest in highly dynamic communication patters [26].

Furthermore, delta-propagation algorithms tend to disseminate redundant information between replicas. As such, in [13] the authors find the sources of redundancy (i.e., back-propagation of δ-groups and redundant state in received δ-groups) and introduce a way of computing optimal deltas, as well as an improved synchronisation algorithm that reduces the amount of state transmission, memory consumption and processing time for synchronisation.

(Big) Δ-CRDTs [26]: These CRDTs were specifically designed to overcome the problem of having highly dynamic communication patterns between replicas. They are an extension of δ-CRDTs that remove the assumption that pairs of replicas continuously communicate, and reduce the storage overhead by not maintaining pairwise communication buffers. To do this, they use internal metadata to compute the minimal delta that needs to be propagated to the other replicas.

In order to compute the delta from a given causal context, Δ-CRDTs need to store metadata regarding deleted elements in the CRDT state (tombstones). However, these CRDTs provide a mechanism to periodically garbage collect these tombstones so that the space overhead is kept low.

The main downside to using Δ-CRDTs is the increase in latency for replicas to receive operations, incurred from the additional communication step needed for the receiving replica to send their version vector before being sent the corresponding minimal delta.

2.4.3 Examples of CRDTs

In this section, we present a number of CRDT designs for both simple data types, such as counters and registers, and more advanced data types, such as sets and maps. The designs presented below were based on those proposed in [37].

2.4.3.1 Counter

A counter is a replicated integer that supports *increment* and *decrement* operations to update it, and a *value* operation to query it. The integer returned by the *value* operation should be the total number of increments minus the total number of decrements.

Algorithm 2.1: Operation-based Counter CRDT

```
payload integer $c$
    initial 0
query $value()$: integer $r$
    let $r = c$
update $increment()$
    atSource()                      // No initial processing done at source
    downstream()
        $c := c + 1$
update $decrement()$
    atSource()                      // No initial processing done at source
    downstream()
        $c := c - 1$
```

An operation-based counter is presented in Algorithm 2.1. The payload is an integer with initial value of zero. Assuming that there are no overflows or underflows, *increment* and *decrement* operations are intrinsically commutative. The *downstream* phases (where the operations are propagated to all replicas) consist in simply incrementing or decrementing the counter by 1.

Note that it is very simple to extend this counter with support for operations that increment or decrement the counter by a specific amount, by simply passing it as an argument in, for example, *incrementValue(integer v)* and *decrementValue(integer v)* operations.

Algorithm 2.2: State-based PN-Counter CRDT

```
payload integer[$n$] $P$, integer[$n$] $N$        // $n$: the number of replicas
    initial [0,0,...,0], [0,0,...,0]
update $increment()$
    let $g = myID()$                        // $g$: index of the source replica
    $P[g] := P[g] + 1$
update $decrement()$
    let $g = myID()$
    $N[g] := N[g] + 1$
query $value()$: integer $r$
    let $r = \sum_{i=0}^{n-1} P[i]$ - $\sum_{i=0}^{n-1} N[i]$
compare(X,Y): boolean $b$
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \wedge \forall i \in [0, n-1] : X.N[i] \leq Y.N[i])$
merge(X,Y): payload $Z$
    let $\forall i \in [0, n-1] : Z.P[i] = max(X.P[i], Y.P[i])$
    let $\forall i \in [0, n-1] : Z.N[i] = max(X.N[i], Y.N[i])$
```

Because the operations of a counter are commutative, it is inherently an operation-based CRDT. However, a state-based CRDT version of a counter is not so trivial to specify. In Algorithm 2.2, we present a specification of a PN-Counter (Positive-Negative Counter) CRDT that supports the *decrement* operation by combining two Grow-only Counters (G-Counters), i.e., counters that only accept increments.

The payload of the counter consists of two vectors with one entry per replica of the CRDT. Vector P registers the increments, and vector N registers the decrements. The value of the counter is the difference between the sums of all elements in P and N. The *merge* operation chooses, for each position of P and N, the maximum value of the two counters it is merging, X and Y.

Note that, like the operation-based CRDT counter, the PN-Counter also assumes there are no overflows or underflows, and that the set of replicas is well-known (and static).

2.4.3.2 Register

A register is used to store an object of any type. It supports an *assign* operation to update its value, and *value* operation to query it (i.e., obtain the value currently being stored). Two concurrent *assign* operations are non-commutative. In Algorithm 2.3, we present the LWW-Register CRDT, that uses the last-writer-wins policy to decide which operation has precedence over the other. There is also another specification, the Multi-Value Register (MV-Register) CRDT, that keeps both values in the presence of concurrent *assign* operations.

To apply the LWW policy, this CRDT creates a total order of *assign* operations. This is done by associating a unique timestamp that is consistent with causal order to each update. The *assign* operation generates the new timestamp at the source replica and then the operation is sent to all replicas. In the *downstream* phase, the update only modifies the state if the new timestamp is greater than the local timestamp of the replica.

Algorithm 2.3: Operation-based LWW-Register CRDT

```
payload X $x$, timestamp $t$                              // X can be any type
    initial $\bot$, 0
query value(): X $r$
    let $r = x$
update assign(X $x'$)
    atSource()
        let $t' = now()$                                  // Timestamp
    downstream($x'$, $t'$)
        if $t < t'$ then $x, t := x', t'$
```

2.4.3.3 Set

A set is a collection of distinct elements, i.e., a collection with no duplicates. This collection supports an *add* operation, to add an element to the set, and a *remove* operation, to remove an element from the set. Additionally, it usually has support for a *lookup* operation to check if an element belongs to the set, and an *elements* operation that returns all the elements of the set.

Algorithm 2.4: Operation-based OR-Set CRDT

```
payload set $S$                                   // S: set of pairs (element, unique-tag)
    initial $\varnothing$
query lookup(element $e$): boolean $b$
    let $b = (\exists u : (e,u) \in S)$
update add(element $e$)
    atSource($e$)
        let $\alpha = unique()$                                    // $\alpha$: unique-tag
    downstream($e$, $\alpha$)
        $S := S \cup \{(e,\alpha)\}$
update remove(element $e$)
    atSource($e$)
        pre lookup($e$)
        let $R = \{(e,u) | \exists u : (e,u) \in S\}$
    downstream($R$)
        pre $\forall (e,u) \in R : add\,(e,u)$ has been delivered   // Causal order is enough
        $S := S \setminus R$                           // Remove pairs observed at source
```

Because *add* and *remove* operations are non-commutative, the main challenge when specifying a set CRDT is to define how to deal with concurrent adds and removes. There are several ways to do this but, in this document, we will only present the specification of the OR-Set (Observed-Remove Set) CRDT, that gives precedence to *add* operations over *remove* operations, effectively providing add-wins semantics.

The OR-Set specification can be seen in Algorithm 2.4. The payload is a set S that includes pairs (element, unique-tag). When an *add* operation occurs, an unique tag is generated in the *atSource* phase (where some pre-processing is done in the source replica). Then, the element and its tag are propagated to the other replicas in the *downstream* phase, so that the replicas can add the element and its unique tag to their respective sets. The *lookup* operation implicitly hides duplicates, so this CRDT supports concurrent adds for the same element. When a *remove* operation occurs, in the *atSource* phase, all pairs to be removed are collected in set R. This set is then propagated to all replicas in the *downstream* phase, so that they can remove the pairs from their local set S.

For this specification to be correct, it is assumed that the underlying dissemination protocol delivers pairs in an order that is consistent with the happens-before relationship, so that a *remove(e)* operation is able to remove all pairs (e, u) added before it.

2.4.3.4 Map

A map is a collection of pairs (key, value), where each key is unique and maps to a value. These values can be of any type. Maps normally have *add*, *remove*, *get*, *contains*, *keys* and *elements* operations. The *add* operation is used to map a value to a key. The *remove* operation removes a key a its corresponding value from the map. The *get* operation returns the value that is mapped to a given key. The *contains* operation returns true if a specific key has a mapping. Finally, the *keys* and *elements* operations return, respectively, the set of all keys or all values in the map.

Map CRDTs can be built upon any set CRDT specification, by making the payload a mapping of keys to sets of values. Map specifications make the *add* operation remove all previous values associated with the key in question. However, if there are two concurrent adds, both values can be kept.

The specification presented in Algorithm 2.5 was adapted from [34]. It is an OR-Map based on a OR-Set CRDT and, as such, when there are concurrent *add* and *remove* operations, precedence is given to the *add*.

Algorithm 2.5: Operation-based OR-Map CRDT

```
payload map $M$                      // M: map of keys ↦ set of pairs (element, unique-tag)
    initial $\varnothing$
query contains(key $k$): boolean $b$
    let $b = (k \in M)$                      // If M[k] is an empty set, then k is not in M
query get(key $k$): set $E$
    pre contains($k$)
    let $E = \{e | \exists\, (e,u) : (e,u) \in M[k]\}$
query keys(): set $K$
    let $K = \{k | \exists k : k \in M\}$
query elements(): set $E$
    let $E = \{e | \exists\, (k,e,u) : (e,u) \in M[k]\}$
update add(key $k$, element $e$)
    atSource($k$, $e$)
        let $u = unique()$                                   // u: unique-tag
        let $R = M[k]$                               // Pairs to remove from key
    downstream($k$, $e$, $u$, $R$)
        pre $\forall\, (e,u) \in R : add(k,e,u)$ has been delivered     // Causal order is enough
        $M[k] := M[k] \cup \{(e,u)\} \setminus R$
update remove(key $k$, element $e$)
    atSource($e$)
        pre contains($k$)
        let $R = M[k]$
    downstream($k$, $R$)
        pre $\forall\, (e,u) \in R : add\,(k,e,u)$ has been delivered     // Causal order is enough
        $M[k] := M[k] \setminus R$
```

When an *add* operation occurs, a unique tag and the set of values, R, associated to the key are computed, in the *atSource* phase. Then, the pair (key, value), the tag and R are propagated to every replica in the *downstream* phase, in which the replicas map the new pair to the given key and remove all previous mappings. When a *remove* operation occurs, the set of mappings for the key, R, is computed in the *atSource* phase. That set is then passed to every replica, so that they can remove the pairs from the map in the *downstream* phase. The *contains* operation checks if there exists a mapping for the key k in the map. Note that if the mapping is an empty set, the *contains* operation should return false.

The *get* operation returns the set of elements that are mapped to the key k, if there is mapping different from an empty set. The *keys* operation returns all the keys that have mappings, and the *elements* operation returns all the elements that exist in the map.

2.4.4 Discussion

To guarantee replica convergence in our solution we opted to create a small library of CRDTs. As such, we implemented the operation-based CRDTs presented in Algorithm 2.1, 2.3, 2.4 and 2.5. In addition to accepting individual operations, our CRDTs accept full state installation because our garbage collection mechanism discards older operations after a period of time, thus requiring our protocol to perform state transfers when new nodes join the system.

2.5 Case Studies

In this section, we present descriptions of some existing causally consistent data storage systems.

The following systems are discussed in a less detailed manner because they are not as close to our proposed approach, mostly because they do not use CRDTs to guarantee that all replicas converge to the same state despite concurrent updates. All these systems support geo-replication because they are built to serve as base for large-scale services with users all over the world.

COPS [29] (Clusters of Order-Preserving Servers) is the distributed key-value store that first defined the concept of causal+ consistency. It keeps a full copy of the data at each data centre (i.e., COPS cluster). However, within the data centres, the key space is partitioned and the keys are stored in different nodes by using consistent hashing. Chain replication is used to replicate each key across a small number of replicas, thus providing linearizability within the cluster. On the other hand, replication between COPS clusters happens asynchronously and provides causal+ consistency. This system uses client-side metadata to track the nearest dependencies on versions of keys and explicitly checks that they have been satisfied locally before making the effects of an operation visible. There is another variant of this system, COPS-GT, that maintains multiple older versions of the values associated with the keys so that it can offer multiple key read-only transactions.

Eiger [30] is a low latency geo-replicated data store for the column-oriented data model. Similarly to COPS, Eiger keeps a full copy of the data in each data centre, partitions the data across several machines and uses explicit dependency checking in order to guarantee that an operation can be applied. However, contrary to COPS, Eiger keeps track of one-hop dependencies on operations and not of nearest dependencies on versions of keys. This system supports improved read-only transactions when compared to those of COPS, and write-only transactions that work for keys spread across multiple servers of a data centre.

GentleRain [12] is a geo-replicated key-value store. The system is split into several partitions that are replicated across multiple replicas, with consistent hashing of the keys being used to assign data items to a specific partition. Additionally, this data store maintains older versions of the items that are periodically garbage collected. This system tracks causality with a less granular approach by summarising dependencies in a single scalar (timestamp), thus reducing the storage and communication overhead. Unlike COPS and Eiger, GentleRain uses a periodic aggregation protocol to determine if an update can be made visible, resulting in improved throughput but also higher update visibility latency.

ChainReaction [5] is a geo-distributed key-value store that uses a new variant of chain replication that offers causal+ consistency. Inside each data centre, the replicas are organised in a DHT and the data items are replicated only in a chain of nodes of a specified size by using consistent hashing. This allows to distribute the load of read operations across multiple replicas within the same data centre. This system deals with causal metadata in an efficient manner by implementing a stabilisation procedure that preserves causal guarantees while introducing low metadata overhead, for both the client and the data store.

Saturn [9] is a metadata management service that can be used in combination with existing geo-replicated storage systems. The underlying system can use a full replication scheme, but genuine partial replication is also supported, guaranteeing that each data centre only receives and stores information regarding data items that it replicates, and ensuring the scalability of the system. Saturn can be used to provide causal consistency to storage systems by delivering metadata to the data centres in causal order, as long as they apply an update only after receiving the corresponding metadata. To ensure causal delivery, Saturn internally organises the data centres in a tree connected by FIFO channels, where the data centres are the leaves. By decoupling the metadata management from data dissemination and using clever metadata propagation mechanisms, it ensures that the visibility latency of updates approximates that of weak-consistent systems that do not store metadata.

C^3 [16] is a solution that can be used to extend existing distributed storage systems. It introduces a novel replication scheme that supports partial geo-replication in order to reduce the storage and communication overhead of full replication schemes. The system also works if the underlying storage system partitions data items across multiple nodes inside each data centre, even though it is not required that it does. C^3 uses an approach that separates the data store layer from the causality tracking layer. It works in a P2P fashion, where each data centre has a causality layer instance associated, and the instances communicate with each other by exchanging labels (i.e., small pieces of data that contain a vector clock and an unique identifier). When executing an operation, its label is propagated to the causality layer of the interested data centres to be added to their log of pending operations until all the operations it depends upon have been completed. This solution allows concurrency in the execution of remote operations, thus lowering their visibility times and enabling the data store to scale through sharding.

The following systems are discussed in a more detailed manner because they are the ones that most closely approximate our proposed approach, namely by using CRDTs to ensure replica convergence.

Cure [2] is a replication protocol for highly available geo-replicated key-value stores that supports causal+ consistency. It uses operation-based CRDTs to guarantee convergence, and provides an interface that offers interactive transactions that combine read and write operations. Cure assumes that, in each data centre, the system stores the full set of objects but creates non-overlapping partitions of the key-space that are the same across data centres.

Cure uses multi-versioning to allow clients to read from causally consistent snapshots of the database. To do this, each version is stored with its causal dependencies in the form of a vector clock. Old versions are garbage collected by making partitions periodically exchange the oldest snapshot vector clock of its active transactions. The minimum of these vectors clocks is computed so that partitions can remove all objects with versions prior to it.

To replicate updates, each partition periodically synchronises with the partitions that store the same key set in other data centres. If there are no new updates, the partitions send heartbeats. To ensure causal consistency guarantees, an update received from a remote replica is only made visible after the updates it depends on have been received and applied at the local data centre. As such, this system relies on a background stabilisation protocol that builds a tree over all servers in a data centre and makes them exchange their vector clocks in order to compute their the global stable snapshots.

By using vector clocks instead of a single scalar to track causality, Cure's update visibility latency depends on the latency to the data centre where the update originated, and not on the latency to the furthest data centre, as in GentleRain. However, this improvement in the visibility latency comes at the cost of higher computation and storage overhead related with metadata management, which negatively affects the throughput of Cure.

SwiftCloud [42] is a distributed object database that gives client-side applications local access to a causally consistent cache that contains a partial replica of the database. This system provides immediate responses for reads and writes on local objects, while using the cloud as a back-up for cache misses, thus offering throughput that is equal or better to that of systems that use server-side geo-replication. Because causal consistency does not guarantee the convergence of replicas, this system relies on CRDTs to do so. SwiftCloud is scalable to thousands of clients, providing consistent versioning while using small metadata. It is also tolerant to data centre failures, connecting clients to a different data centre while maintaining causal consistency guarantees, at the cost of a slight increase in staleness.

The cloud infrastructure of the system connects a small set of geo-replicated data centres to a large set of clients. Each data centre maintains a full copy of the database, while clients only keep the objects that interest them (partial replicas). A client usually connects to a single data centre but, in case of failure, it can connect to a new one. Because client replicas are partial, the system can only guarantee conditional availability, by which an operation only returns without remote communication if the requested object is in the local cache. If the object is not in the cache, it must be retrieved from a data centre. If, for some reason, the data centres are not able to satisfy the request, the operation may block or return an error. Each data centre also maintains a best-effort notification session to each of its clients, over FIFO channels, so that it can issue notifications to them regarding updates to objects in their interest sets.

SwiftCloud uses the approach of "reading in the past" to ensure that data centres only expose causally consistent views of the data, even though some of that data may already have newer versions. In order to do this, the system keeps metadata in the form of a version vector. When a data centre receives an update operation, it must check if all its dependencies are satisfied before applying it. Otherwise, it must buffer the operation until all dependencies have been satisfied, so as not to cause clients to see inconsistent views of the data.

As is, this solution is not tolerant to data centre failures. Because replication between data centres is done asynchronously, when a client switches to a new data centre the updates that were delivered in that data centre may be stale when compared to the ones of the previous data centre. This could cause a causal gap in the client and, as such, the new data centre must reject the client. SwiftCloud provides a solution to this problem by maintaining, in each data centre, a K-stable version that contains the updates for which the data centre has received acknowledgements for at least $K-1$ distinct data centres. The base version (stored in the cache) of all clients must be K-stable, so that clients depend only on external updates that are likely to be found in any data centre, or internal ones that the client can transfer to the new data centre.

Even though SwiftCloud provides causal consistency under a partial replication scheme, remains fault tolerant, and uses CRDTs to ensure replica convergence, it does so for client-side applications. As such, it does not entirely approximate our proposed approach because it brings the storage and computation to the client devices.

Legion [27] is a framework that allows web applications to use client-side replication of data. Each client maintains a local data store with a partial replica of the shared application objects that can be modified without coordination with any other replicas. Legion offers causal consistency guarantees by propagating updates asynchronously to other replicas.

Contrary to SwiftCloud, that caches data at the client and synchronises with the servers, Legion allows for both synchronisation with servers and among the clients themselves, using P2P interactions. As such, overlay networks are created between clients that share objects so that they can propagate updates to each other. Some of the clients also act as a bridge to the servers (and to clients that are outside the Legion logical networks), allowing the remote updates to eventually reach all clients. This reduces the update visibility latency of clients that are close to each other.

To ensure that all replicas' states converge, Legion uses Δ-CRDTs [26] to resolve conflicts. These CRDTs are efficient when used with decentralised dissemination protocols, such as epidemic protocols, allowing replicas to synchronise by using deltas with the effects of one or more operations (or the full state, if need be). The objects of the data store are encoded as CRDTs and grouped within containers of related objects.

To propagate deltas in causal order, Legion uses a multicast primitive implemented using a push-gossip protocol similar to the one described in [23]. Each client maintains a causally ordered list of received deltas for each container, and propagates them (using FIFO channels) to every client it connects to. Upon receiving a delta, a client either discards it (if when checking the version vector of the container it detects that it already received it), or it integrates the delta with its state and adds it to the list of deltas to be propagated to other clients.

Note that, to reduce storage overhead, Legion only stores a suffix of the list of deltas received. Because clients exchange their vector clocks at the start of every synchronisation step, this allows them to generate deltas that contain only operations that the other peer has not yet seen. However, when connecting for the first time, it might be impossible, or highly inefficient, to compute the appropriate delta between two clients. As such, sometimes clients may have to do a synchronisation step where they exchange their full state, which may cause a wave of synchronisations that is propagated throughout the network leading to high communication cost.

Legion improves the latency for update propagation when compared with server-based systems. Furthermore, it reduces the load to the centralised component by using P2P interactions between clients, allowing disconnected operation. However, because client-side devices are often resource poor, this system imposes a large overhead on them by making clients communicate with each other, store data objects, and perform advanced computation. On the other hand, our approach aims to utilise edge nodes for storage and computation, thus reducing client perceived latency without overloading clients, while still allowing us to take advantage of data locality.

2.5.1 Discussion

In Table 2.1, we present a comparison between the several studied systems and our solution, to demonstrate that our solution solves problems that the current state of the art systems do not. Note that, of all the case studies, only Cure, SwiftCloud, and Legion offer configurable conflict resolution policies by leveraging on CRDTs.

COPS, Eiger, GentleRain, ChainReaction, C^3, Cure, and SwiftCloud were designed to be deployed with few data centres, using non scalable techniques to encode causal dependencies for inter data centre (remote) operations. These systems also require each data centre to communicate with all others to ensure causality, making them efficient only when working with reduced numbers of data centres.

Although Saturn is scalable and decentralised, it uses a static tree topology making it incapable of being used in edge scenarios where nodes constantly join and leave the network. Just like Saturn, all the studied systems except for Legion do not easily adapt to new replicas being added to the system. These systems could be changed in order support a dynamic membership, but they would still not be able to scale adequately to hundreds of data centres.

	Scalable Causality Tracking	Decentralised	Dynamic Membership	Edge Enabled	CRDTs for Conflict Resolution
COPS	✗	✓	✗	✗	✗
Eiger	✗	✓	✗	✗	✗
GentleRain	✗	✓	✗	✗	✗
ChainReaction	✗	✓	✗	✗	✗
Saturn	✓	✓	✗	✗	✗
C^3	✗	✓	✗	✗	✗
Cure	✗	✓	✗	✗	✓
SwiftCloud	✗	✓	✗	✗	✓
Legion	✓	✗	✓	✗	✓
SYNC Tree	✓	✓	✓	✓	✓

Table 2.1: Comparison between the studied systems and our solution.

Legion (which is the closest to being suitable for the edge) is not completely decentralised, requiring client replicas to access the system via a central data centre, and some of those replicas to remain connected to that data centre to guarantee essential functionality, making those replicas of more importance than others.

On the other hand, our solution is adequate for edge-based distributed storage systems. It relies on truly decentralised P2P interactions, uses scalable causality tracking (a single scalar), and is based on a dynamic tree topology that allows it to react efficiently to membership changes. Lastly, our solution also resolves conflicts by using CRDTs.

Summary

In this chapter, we presented related work that is relevant for the context and objectives of this thesis. We discussed overlay networks and gossip-based communication, which are at the foundation of our edge-based P2P system. We detailed different consistency models, causality tracking mechanisms, and conflict resolution policies, with emphasis on causal+ consistency, vector clocks, and CRDTs, as these are the ones used in our solution.

In the following chapter, we present our novel causal broadcast protocol.

3

Designing a New Causal Broadcast Algorithm

In this chapter, we present our novel causal broadcast algorithm, suitable for supporting replication in edge-based storage systems. The presentation of our solution is structured as follows:

In Section 3.1, we discuss some assumptions made regarding the expected behaviour of the system in order for the presented algorithm to work as expected.

In Section 3.2, we talk about the architecture of our solution.

In Section 3.3, we present SYNC Tree, the first iteration of our causal broadcast algorithm, and go into detail about choices made during its development.

In Section 3.4, we discuss the changes made in order to devise ECO SYNC Tree, the final iteration of our algorithm.

In Section 3.5, we present the correctness arguments for our algorithm.

3.1 System Model

In this work, we assume a system composed of several replicas that can be located in data centres, or in edge access points. All interactions between replicas are performed through the exchange of messages via the Internet, which are guaranteed to be delivered in the order that they were sent by using TCP channels. As we are working in the edge computing paradigm, we assume that the total number of replicas can vary over time, being in the order of hundreds or thousands. We consider a system with full replication, where all replicas maintain the same set of data objects encoded in the form of CRDTs.

In our model, clients interact with the application at the closest replica. We assume that they always interact with the same replica (replica migration has been explored elsewhere, e.g., in [9]). Both read and write operations are executed immediately at the local replica. Additionally, write operations are associated with control metadata and asynchronously disseminated by our causal broadcast algorithm to all other replicas. By pairing causal delivery with the use of CRDTs, our solution as a whole offers causal+ consistency.

3.2 Architecture

Figure 3.1 depicts the architecture of our system, where each of the nodes is composed by a protocol stack with four layers: i) the membership layer is responsible for managing the overlay network; ii) the broadcast layer ensures messages are delivered to every node in causal order; iii) the replication layer provides applications with an interface to use replicated CRDTs; and iv) the application layer consists of any user application that maintains its data in CRDTs.

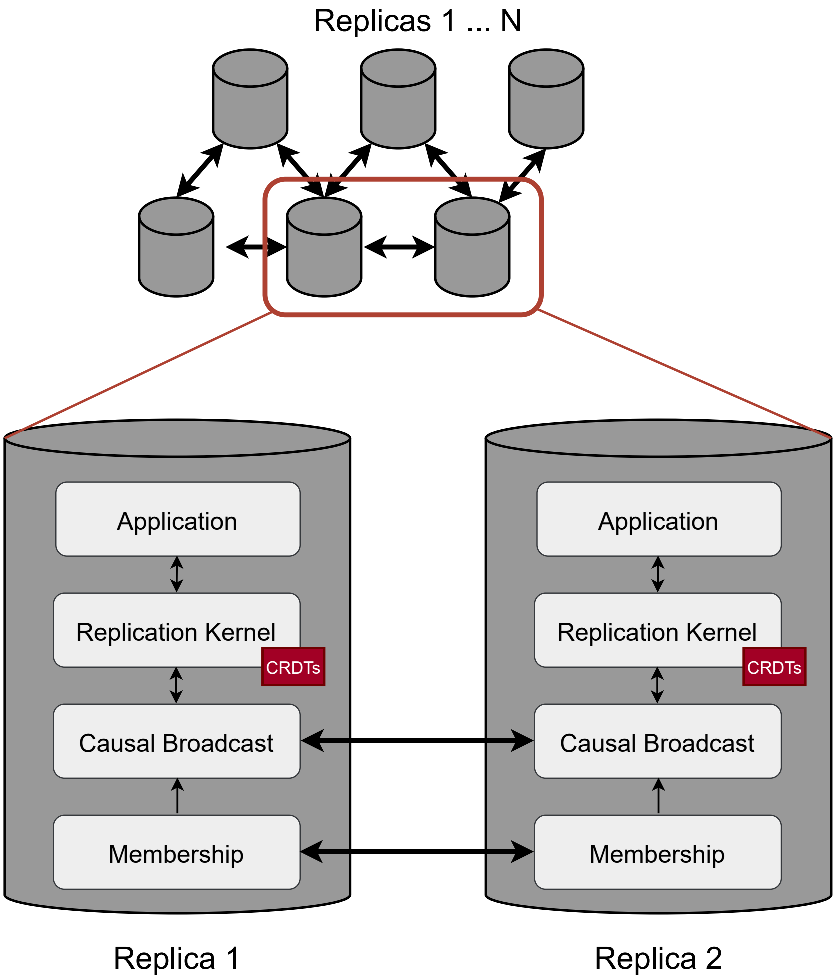


Figure 3.1: System architecture of our solution.

Membership Layer: This layer is composed by an overlay management protocol that builds an overlay network over the nodes of the system. While any overlay management protocol could be used, our causal broadcast algorithm works best when this layer offers symmetric partial views, and is relatively stable when no external events occur (i.e., when no nodes crash, join, or leave the system). In our prototype we used HyParView [23], as it meets all these requirements.

The membership layer does not receive any outside information. It does, however, provide information to the broadcast layer in the form of a set of neighbours the node knows. The broadcast layer is also notified when one of those neighbours is removed from the system, or when a new one is added.

Causal Broadcast Layer: The causal broadcast layer is responsible for disseminating messages (operations) through the overlay in causal order. As mentioned before, this layer is independent from the choice of overlay management protocol, but requires that partial views be symmetric and that the underlying overlay remain stable.

When this layer receives an operation from the replication layer, it propagates the operation while respecting causal order and, when it receives a remote operation, it delivers the operation to the local replication layer.

Replication Layer: The replication layer, also referred to as replication kernel, provides an interface for client applications that want to access and manipulate replicated data objects in the form of CRDTs. For this, it includes a small library of CRDTs which are used to represent the application's data objects, and can even be shared between applications. This layer acts as an additional layer of abstraction for clients, hiding any replication logic from them.

The replication kernel receives read and write operations from client applications. It communicates with the causal broadcast layer in order to disseminate write operations and, when it receives one from the broadcast layer, it executes the operation locally.

Application Layer: The application layer may be any client application that utilises the interface offered by the replication layer to interact with replicated CRDTs. For this interaction, this layer sends read and write requests to the replication layer.

The main contribution of this work is the causal broadcast layer, which we discuss in more detail in Sections 3.3 and 3.4. Note, however, that the replication layer, as well as the small library of CRDTs it contains, are secondary contributions that allowed our system, as a whole, to offer causal+ consistency.

3.3 SYNC Tree

In this section, we present SYNC Tree (cau**S**al d**YN**ami**C** **Tree**), our causal broadcast algorithm. We start by giving a brief overview of the protocol, followed by detailed explanations of its core mechanisms.

The pseudo code for our protocol is presented in Algorithms 3.1, 3.2, 3.3, and 3.4, which will be referenced throughout this section.

3.3.1 Overview

The purpose of this work is to create a new and efficient causal broadcast algorithm that, in combination with a small library of CRDTs, can be used to offer causal+ consistency guarantees to edge-based replicated storage systems. In order to do so, we must minimise the amount of metadata used to encode the causal dependencies of operations, as well as the number of messages transmitted by the algorithm to ensure delivery to all nodes.

SYNC Tree, inspired by Plumtree [22], builds a decentralised dynamic tree shaped overlay (broadcast tree), on top of the overlay created by the membership protocol, that is used to disseminate messages through the network. As long as the links between nodes adhere to a FIFO discipline and each node propagates messages in the order it received them, the tree topology inherently offers causal delivery. This means that nodes can receive messages in different orders, but those orders will always respect causality.

If no nodes leave or join the network, the broadcast tree topology remains unaltered and there will be no duplication of messages or causality guarantees broken. This, of course, assumes that we are using an underlying overlay management protocol that is stable when no external events occur (e.g., HyParView [23]). When nodes join or leave the network, or when the underlying overlay management protocol introduces changes to the partial views of nodes, cycles and partitions can be temporarily introduced to the tree. One of the main challenges of this work is to ensure that the topology created by our protocol adapts to these changes and quickly converges into a new broadcast tree. In Section 3.3.2, we explain how our algorithm is able to create and maintain the broadcast tree.

Another main challenge, possibly the hardest one to overcome, is to ensure that, in these moments of instability, the delivery order of messages will not break causality guarantees. To do this, we introduce a novel synchronisation mechanism, detailed in Section 3.3.3, based on the exchange of vector clocks and re-transmission of causally-ordered operations. This mechanism allows nodes to synchronise with each other when joining the network, or when the tree suffers reconfigurations, while still guaranteeing causal delivery. Through the exchange of vector clocks, which summarise the operations executed by a node, it is possible for two nodes to synchronise by sending each other the missing operations, thus becoming neighbours in the tree.

For our algorithm to work, each node of the system must maintain information regarding its neighbours, the identifiers of messages it has (or has not) received, and timers to perform certain tasks (Alg. 3.1 - line 3-15). Furthermore, for the purpose of the synchronisation mechanism and to detect duplicate messages, each node maintains a local vector clock to capture their local state (Alg. 3.1 - line 16). However, it is important to note that this vector clock does not need to be disseminated through the tree with each operation, as this would incur an excessive communication cost.

In our solution, each node maintains a local sequence number, initialised to zero (Alg. 3.1 - line 17, 34), which is incremented and assigned (atomically) to each local

Algorithm 3.1: SYNC Tree: Initialisation & Gossip

```
Local State:
  myself // Local host
  announcementTimeout // Timeout to send announcements
  treeMsgTimeout // Timeout to send tree message
  checkTreeMsgsTimeout // Timeout to check if tree messages have been received recently
  sendTreeMsgTimer // Timer to send tree messages
  nReceivedTreeMsgs // Number of tree messages received from relevant peers
  // since the last timer expired
  eager // Map of eager push peers to the vector clock received when synchronising
  lazy // Set of lazy push peers
  incomingSync // Peer of current incoming synchronisation
  pendingIncomingSyncs // Queue of pending incoming synchronisations
  outgoingSyncs // Set of peers we have outgoing synchronisations with
  receivedTreeIDs // Set of IDs of received tree messages
  missingTreeIDs // Map of missing tree message IDs to queue of peers that have them
  ongoingTimers // Timers for the reception of tree messages
  vectorClock // Locally stored vector clock
  seqNumber // Sequence number of the last operation sent by the local host

Upon Init(myself, s, t1, t2, t3, t4) do:
  myself ⟵ myself;
  announcementTimeout ⟵ t2;
  treeMsgTimeout ⟵ t3;
  checkTreeMsgsTimeout ⟵ t4;
  sendTreeMsgTimer ⟵ ⊥;
  nReceivedTreeMsgs ⟵ 0;
  eager ⟵ {};
  lazy ⟵ {};
  incomingSync ⟵ ⊥;
  pendingIncomingSyncs ⟵ {};
  outgoingSyncs ⟵ {};
  receivedTreeIDs ⟵ {};
  missingTreeIDs ⟵ {};
  ongoingTimers ⟵ {};
  vectorClock ⟵ ⊥;
  seqNumber ⟵ 0;
  setup periodic timer CheckReceivedTreeMsgsTimeout() with checkTreeMsgsTimeout;

Upon BroadcastRequest(originalSender, content) do:
  trigger DeliverNotification(content);
  seqNumber ⟵ seqNumber + 1;
  call handleGossipMessage(originalSender, seqNumber, content, myself);

Upon Receive GossipMessage(originalSender, msgSeqNumber, content) from from do:
  if vectorClock[originalSender] = msgSeqNumber - 1 then:
    trigger DeliverNotification(content);
    call handleGossipMessage(originalSender, msgSeqNumber, content, from);

Procedure handleGossipMessage(originalSender, msgSeqNumber, content, from) :
  vectorClock[originalSender] ⟵ vectorClock[originalSender] + 1;
  writeOperationToFile(originalSender, msgSeqNumber, content, vectorClock);
  call forwardGossipMessage(originalSender, msgSeqNumber, content, from);

Procedure forwardGossipMessage(originalSender, msgSeqNumber, content, from) :
  forall peer, peerVC ∈ eager : peer ≠ from do:
    if peerVC[originalSender] < msgSeqNumber then:
      trigger send GossipMessage(originalSender, msgSeqNumber, content) to peer;
```

write operation (Alg. 3.1 - line 36-39). The operations sent through the tree contain the identifier of the original sender node, as well as its sequence number (Alg. 3.1 - line 51), allowing each node to update its own vector clock locally (Alg. 3.1 - line 45). Lastly, for the synchronisation mechanism to work, each node of the system must maintain a causally-ordered list of all operations. To keep the main memory requirements of our protocol constant, this list of operations is written to, and read from, disk (details on how this was implemented are given in Section 4.1.4).

3.3.2 Tree Creation and Maintenance

Each node of our protocol maintains two different sets of peers that are initially empty: i) the *eager* peers; and ii) the *lazy* peers (Alg. 3.1 - line 25-26). Eager push gossip is used to disseminate *Gossip* messages to the *eager* peers, whose connections form the branches of the broadcast tree. On the other hand, a lazy push gossip strategy is used with the *lazy* peers, whose connections can be used, when needed, to optimise or repair the tree, but are not used to disseminate *Gossip* messages.

When new neighbours are added to the partial view of a given node, the node will add them to the *lazy* peers set (Alg. 3.3 - line 94-95). As such, initially, no nodes belong to the broadcast tree and no *Gossip* messages are disseminated through the branches (because there are none). Note that this is a departure from the original Plumtree protocol, where links are initially created as part of the *eager* peers set. Before *Gossip* messages can flow in the system (conveying operations over data objects), the tree needs to be created and, to do that, we introduced a new special kind of message that we call *Tree* message.

Periodically, one of the nodes of the system will disseminate a *Tree* message through the tree branches (if there are any), as well as send an *Announcement* message with the identifier of that *Tree* message to all the *lazy* peers (Alg. 3.2 - line 86-88; and Alg. 3.3 - line 115-118). When nodes receive an *Announcement* message, they start a timer to wait for the reception of the corresponding *Tree* message through the tree branches and memorise the node that announced the message (Alg. 3.2 - line 67-70). Because there are no tree branches initially, those timers will expire. When they do, the nodes will synchronise with the sender of the *Announcement* message (Alg. 3.2 - line 89-93), effectively creating new tree branches. To expedite the process of creating the tree, when a node that is completely isolated from the tree (i.e., a node that has no neighbours in the *eager* peers set and is not synchronising with any other node) receives an *Announcement* message, it will start synchronising with the sender of the message immediately, instead of setting a timer and waiting for it to expire (Alg. 3.2 - line 64-65). The synchronisation mechanism will be explained in detail in Section 3.3.3.

As branches are added to the tree, the *Tree* messages will start to be propagated through those branches. When that happens, some nodes will receive duplicate *Tree* messages, due to the cycles that emerge when nodes create tree branches simultaneously. The next step is to remove those cycles by pruning the redundant tree branches. As such, when a node receives a duplicate *Tree* message, it will remove the sender from the *eager* peers and add him to the *lazy* peers, as well as send him a *Prune* message (Alg. 3.2 - line 58-61). When the sender receives that *Prune* message it will also move his connection to the node to the *lazy* peers, thus removing the redundant tree branch (Alg. 3.2 - line 71-74).

Algorithm 3.2: SYNC Tree: Tree Creation & Management - Part 1

```
52 Upon Receive TreeMessage($mid$, $originalSender$) from $from$ do:
53   if $mid \notin$ receivedTreeIDs then:
54     call handleTreeMessage($mid$, $originalSender$, $from$);
55   else:
56     if $from \in$ outgoingSyncs then:
57       outgoingSyncs $\longleftarrow$ outgoingSyncs $\setminus \{from\}$;
58     if eager[$from$] $\neq \perp$ then:
59       removeKey(eager, $from$);
60       lazy $\longleftarrow$ lazy $\cup \{from\}$;
61     trigger send PruneMessage() to $from$;

62 Upon Receive AnnouncementMessage($mid$) from $from$ do:
63   if $mid \notin$ receivedTreeIDs then:
64     if isEmpty(eager) $\wedge$ isEmpty(outgoingSyncs) then:
65       call startOutgoingSync($from$, $true$);
66     else:
67       if ongoingTimers[$mid$] $= \perp$ then:
68         tid $\longleftarrow$ setup timer AnnouncementTimeout($mid$) with announcementTimeout;
69         ongoingTimers[$mid$] $\longleftarrow$ tid;
70       missingTreeIDs[$mid$] $\longleftarrow$ missingTreeIDs[$mid$] $\cup \{from\}$;

71 Upon Receive PruneMessage() from $from$ do:
72   if eager[$from$] $\neq \perp$ then:
73     removeKey(eager, $from$);
74     lazy $\longleftarrow$ lazy $\cup \{from\}$;
75   if $from \in$ pendingIncomingSyncs then:
76     pendingIncomingSyncs $\longleftarrow$ pendingIncomingSyncs $\setminus \{from\}$;
77   elif $from =$ incomingSync then:
78     call tryNextIncomingSync();

79 Upon Receive GraftMessage() from $from$ do:
80   call startOutgoingSync($from$, $false$);

81 Upon CheckReceivedTreeMsgsTimeout() do:
82   if nReceivedTreeMsgs = 0 then:
83     sendTreeMsgTimer $\longleftarrow$ setup periodic timer SendTreeMessageTimeout() with treeMsgTimeout;
84   else:
85     nReceivedTreeMsgs $\longleftarrow$ 0;

86 Upon SendTreeMessageTimeout() do:
87   mid $\longleftarrow$ randomID();
88   call handleTreeMessage(mid, myself, myself);

89 Upon AnnouncementTimeout($mid$) do:
90   if ongoingTimers[$mid$] $\neq \perp \wedge mid \notin$ receivedTreeIDs then:
91     sender $\longleftarrow$ poll(missingTreeIDs[$mid$]);
92     if sender $\neq \perp$ then:
93       call startOutgoingSync($sender$, $true$);
```

The repetition of this process will, eventually, connect all nodes in a (acyclic) broadcast tree that is latency optimised (for the sender of the *Tree* message).

For the tree to stabilise, we need to make sure that only one node is sending *Tree*

Algorithm 3.3: SYNC Tree: Tree Creation & Management - Part 2

```
 94 Upon NeighbourUpNotification($neighbour$) do:
 95   lazy ⟵ lazy ∪ {$neighbour$};

 96 Upon NeighbourDownNotification($neighbour$) do:
 97   eager ⟵ eager \ {$neighbour$};
 98   lazy ⟵ lazy \ {$neighbour$};
 99   pendingIncomingSyncs ⟵ pendingIncomingSyncs \ {$neighbour$};
100   outgoingSyncs ⟵ outgoingSyncs \ {$neighbour$};
101   forall queue ∈ values(missingTreeIDs) do:
102     queue ⟵ queue \ {$neighbour$};
103   if $neighbour$ = incomingSync then:
104     call tryNextIncomingSync();

105 Procedure handleTreeMessage($mid$, $originalSender$, $from$) :
106   if $originalSender$ = myself ∨ isSenderSmaller($originalSender$, myself) then:
107     nReceivedTreeMsgs ⟵ nReceivedTreeMsgs + 1;
108     if $originalSender$ ≠ myself then:
109       cancel timer sendTreeMsgTimer;
110   receivedTreeIDs ⟵ receivedTreeIDs ∪ {$mid$};
111   tid ⟵ removeKey(ongoingTimers, $mid$);
112   if tid ≠ ⊥ then:
113     cancel timer $tid$;
114     removeKey(missingTreeIDs, $mid$);
115   forall peer ∈ keys(eager) : peer ≠ $from$ do:
116     trigger send TreeMessage($mid$, $originalSender$) to peer;
117   forall peer ∈ lazy : peer ≠ $from$ do:
118     trigger send AnnouncementMessage($mid$) to peer;
```

messages. The reasoning behind this comes from the fact that, if there are multiple nodes sending *Tree* messages and their corresponding *Announcement* messages, the topology may become trapped in an endless cycle of nodes creating and removing branches, due to the detection of duplicates at numerous nodes at the same time. This makes it impossible for the topology to stabilise in a tree structure.

To guarantee that only one node is sending *Tree* messages, the protocol must be able to compare nodes and define a total order among them. In our implementation we compare nodes by their IP address and port, but any comparison method could be used as long as it allows a given node to determine if it is "equal" to, "bigger" than, or "smaller" than another node.

In order to know if it should be sending *Tree* messages, each node keeps a counter of *Tree* messages received from nodes "equal or smaller" than itself (Alg. 3.3 - line 105-107). Nodes will periodically check their counter and, if they have not received any messages, they start sending the *Tree* messages themselves (Alg. 3.2 - line 81-85). If, meanwhile, they receive a *Tree* message from a node with the specified restrictions, they will stop sending *Tree* messages (Alg. 3.3 - line 108-109). This mechanism ensures that, eventually, there will only be one node in the network sending *Tree* messages. If that node dies, another will take its place. If there are network partitions, each partition will have one sender and, when the partitions are healed, this process will again ensure that only one node remains sending *Tree* messages.

3.3.3 Synchronisation Mechanism

As we explained in the previous section, the topology of our protocol is not always stable, with branches being created and removed in periods of instability. When we want to create a new branch, the nodes on either side of the branch cannot simply start sending *Gossip* messages to each other. If that happened, those messages might not be delivered in causal order because the nodes might not be in compatible states (i.e., they may not have received the same set of *Gossip* messages). Because of this, we now introduce our new synchronisation mechanism that allows the nodes of the branch extremities to reach compatible states.

Because we assume an asynchronous system, the nodes may appear in each other's partial views at different points in time. As such, the synchronisation must be done separately in each direction of the branch. This means that a branch may, temporarily, be used in only one direction, but the protocol guarantees that the topology will eventually converge and, either the branch will work both ways, or it will be removed.

When a node A wants to create a branch to a neighbour B, A starts the synchronisation process from his side and sends B a *Graft* message (Alg. 3.4 - line 143), requesting him to start the synchronisation in the opposite direction (Alg. 3.2 - line 79-80).

The synchronisation (in either direction) occurs as follows:

- When node A wants to synchronise with node B, it will ask node B for its vector clock by sending him a *SendVectorClock* message (Alg. 3.4 - line 145);

- When node B receives the *SendVectorClock* message, it will reply with a *VectorClock* message that contains its vector clock (Alg. 3.4 - line 134);

- Node A will then compare the received vector clock to its own, and send node B a *Synchronisation* message containing a causally-ordered list of operations unknown to B. Simultaneously, node A adds node B to his *eager* peers, thus propagating future *Gossip* messages to him (Alg. 3.4 - line 119-130). Note that, node B (and each of the other nodes in *eager* peers) is tagged with the vector clock received when synchronising so that node A can avoid sending him duplicate *Gossip* messages by checking if they are included in his vector clock (Alg. 3.1 - line 48-51);

- Upon receiving the *Synchronisation* message, node B will execute each of the operations within and forward them through his other tree branches in the order they were received, ignoring any duplicate operations that he may have received meanwhile (Alg. 3.4 - line 153-160).

The synchronisation process happens concurrently in both directions. When the synchronisation finishes in one of those directions, the "one-way" branch can be used to propagate *Gossip* messages without ever breaking causality guarantees. Eventually, the synchronisation finalises in both directions and *Gossip* messages can be propagated both ways.

Algorithm 3.4: SYNC Tree: Synchronisation

```
Upon Receive VectorClockMessage($msgVC$) from $from$ do:
  if $from \in$ outgoingSyncs then:
    syncOps $\longleftarrow$ call readSyncOpsFromFile($msgVC$, vectorClock);
    trigger send SynchronisationMessage(syncOps) to $from$;
    eager[$from$] $\longleftarrow$ $msgVC$;
    outgoingSyncs $\longleftarrow$ outgoingSyncs \ {$from$};
    lazy $\longleftarrow$ lazy \ {$from$};
    forall (mid, queue) $\in$ missingTreeIDs do:
      if $from \in$ queue then:
        tid $\longleftarrow$ removeKey(ongoingTimers, mid);
        cancel timer tid;
        removeKey(missingTreeIDs, mid);

Upon Receive SendVectorClockMessage() from $from$ do:
  if incomingSync = $\perp$ then:
    incomingSync $\longleftarrow$ $from$;
    trigger send VectorClockMessage(vectorClock) to $from$;
  else:
    pendingIncomingSyncs $\longleftarrow$ pendingIncomingSyncs $\cup$ {$from$};

Upon Receive SynchronisationMessage($syncOps$) from $from$ do:
  call executeSyncOperations($syncOps$, $from$);
  call tryNextIncomingSync();

Procedure startOutgoingSync($neighbour$, $sendGraft$) :
  if $neighbour \in$ lazy $\wedge$ $neighbour \notin$ outgoingSyncs) then:
    if $sendGraft = true$ then:
      trigger send GraftMessage() to $neighbour$;
    outgoingSyncs $\longleftarrow$ outgoingSyncs $\cup$ {$neighbour$};
    trigger send SendVectorClockMessage() to $neighbour$;

Procedure tryNextIncomingSync() :
  nextIncomingSync $\longleftarrow$ poll(pendingIncomingSyncs);
  if nextIncomingSync $\neq \perp$ then:
    incomingSync $\longleftarrow$ nextIncomingSync;
    trigger send VectorClockMessage(vectorClock) to incomingSync;
  else:
    incomingSync $\longleftarrow$ $\perp$;

Procedure executeSyncOperations($syncOps$, $from$) :
  forall op $\in$ $syncOps$ do:
    originalSender $\longleftarrow$ getOriginalSender(op);
    msgSeqNumber $\longleftarrow$ getSeqNumber(op);
    content $\longleftarrow$ getContent(op);
    if vectorClock[$originalSender$] = msgSeqNumber - 1 then:
      trigger DeliverNotification(content);
      call handleGossipMessage(originalSender, msgSeqNumber, content, $from$);
```

Note that, the missing operation's list sent during synchronisation steps is computed by comparing node A's vector clock with node B's vector clock, and iterating over the causally-ordered list of all operations until the first operation that is in node A's vector clock but not in node B's clock is found. After this point, each operation's sequence number is analysed in order to see if it is already contained in node B's vector clock, only being sent if it is not. This ensures that no operation will be sent before all its dependencies, and reduces the number of duplicate messages by filtering out the ones that are contained in node B's vector clock. However, it is possible that node B receives operations through other tree branches after sending its vector clock to node A. When that happens, node A may send node B duplicate operations in the missing operation's list. This is not a problem because our protocol also filters duplicates at the reception of the missing operations (Alg. 3.4 - line 158).

We opted for allowing only one incoming synchronisation to be active at a time in each node (Alg. 3.4 - line 131-136). This was done to further reduce the number of duplicate messages that our solution sends, because it ensures that a node does not send his vector clock to, and receive missing operations from, two (or more) neighbours at the same time. By doing this, a node will never receive two *Synchronisation* messages that include the same operation.

On the other hand, a node can have multiple outgoing synchronisations happening at the same time, effectively sending lists of missing operations to multiple neighbours (Alg. 3.4 - line 144, 120).

3.3.4 Discussion

In Section 3.3, we presented SYNC Tree, our new causal broadcast algorithm based on a tree topology. Our broadcast tree is able to dynamically reconfigure itself when nodes join or leave the system and when there are changes in the patterns of communication of nodes. This is achieved by using an efficient synchronisation mechanism that ensures that in these moments of instability, even though our topology becomes a cyclic graph, our protocol still delivers messages in causal order.

One shortcoming of this solution is that the causally-ordered list of operations written to disk will grow indefinitely as the system remains on-line, effectively increasing the disk space requirements of the machines used to run our protocol. When new nodes join the system, this also causes the synchronisations to be slower because lengthier lists of missing operations must be transmitted. In the following section, we present a version of our protocol that garbage collects old operations in order to address these problems.

3.4 ECO SYNC Tree: Adding Garbage Collection

In this section, we present ECO SYNC Tree (garbag**E** **CO**llecting cau**S**al d**YN**ami**C** **Tree**), the evolution of SYNC Tree that supports state transfer and introduces garbage collection of the operations written to disk in order to reduce the disk usage requirements and the synchronisation times of our solution. We give a brief overview of the final protocol and explain the changes introduced for it to work.

The pseudo code that features the changes made to the original protocol is presented in Algorithm 3.5, which will be referenced throughout this section. Any event handler or procedure that is not redefined in Algorithm 3.5 remains the same as in the original protocol from Algorithms 3.1, 3.2, 3.3, and 3.4.

3.4.1 Overview

ECO SYNC Tree was created to solve the problem of the causally-ordered list of operations written to disk growing indefinitely. This negatively affected the behaviour of SYNC Tree when new nodes joined the system, because it meant that those nodes had to receive long lists of operations in the synchronisation steps. In this case, it would be much more efficient to send the new nodes a copy of the local state along with a smaller list with the necessary operations, thus using significantly less bandwidth and reducing the time it takes to finalise a synchronisation.

With this said, ECO SYNC Tree enriches our protocol with a state transfer mechanism, as well as a garbage collection mechanism. Given that we can only garbage collect operations if we are able to perform state transfer, we will start by detailing the state transfer mechanism, and only then will we discuss the garbage collection of old operations.

3.4.2 State Transfer

To allow operations from disk to be garbage collected, we first need to make sure that they will not be needed when synchronising with other nodes. This means that each node of our protocol must be able to produce snapshots of its state and transfer them to new nodes that may join the network. Only then can we erase the operations reflected in those snapshots from disk without violating causality guarantees.

Our solution periodically sends a request to the replication kernel asking it for a (serialised and opaque) snapshot of its state (Alg. 3.5 - line 34-35). This is necessary because the broadcast layer is agnostic to everything it disseminates, and any new node that joins the system needs to become up-to-date with the data stored only at the replication layer. Note that the computed state is tagged with a vector clock that summarises the operations it reflects (Alg. 3.5 - line 6, 15-16), and that it is also possible to configure the frequency of the state computation (Alg. 3.5 - line 10).

Suppose a new node, say B, joins the system and node A starts to synchronise with him. Instead of sending him all the operations that have passed through the network, node A will send node B its most recently computed state (and corresponding vector clock), as well as a smaller list of operations containing all the operations that have not been garbage collected (more details on this are given in Section 3.4.3). If no local state has been computed, this means that the list of missing operations has not yet been garbage collected for the first time, and still contains all operations of the system. In this case,

Algorithm 3.5: ECO SYNC Tree: State Transfer & Garbage Collection

```
Local State:
  ...
  garbageCollectionTimeout // Timeout to garbage collect old operations
  saveStateTimeout // Timeout to update current state
  garbageCollectionTTL // Time to live of operations written to disk
  stateAndVC // Current serialised state and corresponding vector clock

Upon Init(myself, s, t1, t2, t3, t4, t5, t6, ttl) do:
  ...
  garbageCollectionTimeout ⟵ t5;
  saveStateTimeout ⟵ t6;
  garbageCollectionTTL ⟵ ttl;
  stateAndVC ⟵ (⊥, ⊥);
  setup periodic timer GarbageCollectionTimeout() with garbageCollectionTimeout;
  setup periodic timer SaveStateTimeout() with saveStateTimeout;

Upon UpdateStateRequest(newState, newVC) do:
  stateAndVC ⟵ (newState, newVC);

Upon Receive VectorClockMessage(msgVC) from from do:
  if from ∈ outgoingSyncs then:
    syncOps ⟵ call readSyncOpsFromFile(msgVC, vectorClock);
    currState, _ ⟵ stateAndVC;
    if currState ≠ ⊥ ∧ isEmptyExceptFor(msgVC, from) then:
      trigger send SynchronisationMessage(stateAndVC, syncOps) to from;
    else:
      trigger send SynchronisationMessage((⊥, ⊥), syncOps) to from;
    ...

Upon Receive SynchronisationMessage(msgStateAndVC, syncOps) from from do:
  if msgStateAndVC ≠ (⊥, ⊥) then:
    call installStateAndExecuteSyncOperations(syncOps, from, msgStateAndVC);
  else:
    call executeSyncOperations(syncOps, from);
  call tryNextIncomingSync();

Upon GarbageCollectionTimeout() do:
  garbageCollectOperations(garbageCollectionTTL);

Upon SaveStateTimeout() do:
  trigger SendStateNotification(vectorClock);

Procedure installStateAndExecuteSyncOperations(syncOps, from, msgStateAndVC) :
  msgState, _ ⟵ msgStateAndVC;
  trigger InstallStateNotification(msgState);
  stateAndVC ⟵ msgStateAndVC;
  call reexecuteMyOperations();
  forall op ∈ syncOps do:
    originalSender ⟵ getOriginalSender(op);
    msgSeqNumber ⟵ getSeqNumber(op);
    content ⟵ getContent(op);
    if vectorClock[originalSender] = msgSeqNumber - 1 then:
      trigger DeliverNotification(content);
      call handleGossipMessage(originalSender, msgSeqNumber, content, from);
    elif vectorClock[originalSender] ≥ msgSeqNumber then:
      writeOperationToFile(originalSender, msgSeqNumber, content, vectorClock);
      call forwardGossipMessage(originalSender, msgSeqNumber, content, from);

Procedure reexecuteMyOperations() :
  myLateSeqNumber ⟵ vectorClock[myself];
  myLateOps ⟵ getMyLateOperationsFromFile(myself, seqNumber, myLateSeqNumber);
  forall op ∈ myLateOps do:
    vectorClock[myself] ⟵ vectorClock[myself] + 1;
    mid ⟵ getId(op);
    content ⟵ getContent(op);
    trigger DeliverNotification(mid, content);
```

the *Synchronisation* message will be sent with the list of missing operations and without a state (Alg. 3.5 - line 17-25).

Upon receiving a *Synchronisation* message that contains a state, node B stores the state locally and requests that the replication kernel install it (Alg. 3.5 - line 36-39). Then, as in the previous iteration of our protocol, node B executes the missing operations in the order they were received, filtering out any duplicates (Alg. 3.5 - line 41-47).

However, node B must write all the operations received to disk and forward them to his neighbours, even if the received state already reflected them and B did not re-execute them (Alg. 3.5 - line 48-50). This is done to ensure that: i) node B writes any operations that he may need to send to any future neighbours in causal order to disk; and ii) the current neighbours of node B receive all operations he has received in this synchronisation step, before node B starts propagating new *Gossip* messages to them, thus enforcing causality guarantees. Additionally, node B must re-execute any locally generated operations that may have been executed after B sent its vector clock to node A and, as such, have been overwritten by the received state (Alg. 3.5 - line 40, 51-58).

Note that, as was previously mentioned in Section 3.3.3, when forwarding the received operations to his neighbours, node B filters any possible duplicates by comparing their sequence number with the vector clock received when synchronising with the neighbour (Alg. 3.1 - line 48-51). This verification ensures that the number of duplicates sent when forwarding *Gossip* messages received via synchronisation is relatively low.

3.4.3 Garbage Collection

Having implemented the state transfer mechanism described in the previous section, there is no need to store all the operations that have passed through the system forever. As such, we introduced a garbage collection mechanism to our solution that, periodically, erases operations from disk after $garbageCollectionTTL$ seconds have passed since they were received (Alg. 3.5 - line 32-33). The frequency at which our protocol executes the garbage collection mechanism, as well as the time to live (TTL) of operations, are parameterisable (Alg. 3.5 - line 9, 11).

Note that, the TTL should be configured to a value larger than the frequency at which the protocol executes state updates to ensure that the protocol does not garbage collect operations that are not yet reflected in the current state. Furthermore, due to the fact that state-based synchronisation is only performed when nodes are new to the system, the TTL should ensure that no operation is erased before all nodes that are already in the system receive it. This means that, when parameterising $garbageCollectionTTL$, the maximum broadcast latency between the nodes of the system should be taken into account, as well as possible network partitions.

The garbage collection mechanism can be implemented in several different ways, as long as it adheres to the restrictions mentioned above. The details on how we implemented garbage collection in our prototype are presented in Section 4.2.1.

3.4.4 Discussion

In Section 3.4, we presented ECO SYNC Tree, the final iteration of our causal broadcast algorithm based on a tree topology. Our solution supports state transfer to new nodes that join the system and introduces a garbage collection mechanism capable of erasing old operations written to disk. This evolution allowed us to greatly reduce the disk usage requirements for machines to run our protocol.

As we mentioned, to ensure that we have not garbage collected operations that have not been received by all nodes of the system, there is some overlap between the operations reflected in the transferred state and the operations present in the additional list of missing operations. Despite this, the final iteration of our protocol lowers the amount of time spent synchronising and allows nodes to enter the system efficiently, even if the system has been on-line for some time.

With this said, our solution has some limitations, namely when it comes to being able to withstand long network partitions. As explained in Section 3.4.3, due to the fact that state-based synchronisation is only performed when nodes are new to the system, the TTL of operations written to disk should ensure that no operation is erased before all nodes that are already in the system receive it. Accordingly, we must parameterise $garbageCollectionTTL$ to a value that accounts for the maximum broadcast latency between the nodes of the system. However, in distributed systems, where network partitions are unavoidable, it is impossible to determine the maximum period of time a network partition will take to heal.

Our solution assumes that any distributed storage system that uses our causal broadcast algorithm will configure the TTL to a value greater than the maximum time they expect any network partition to exist. If any network partition exceeded that duration, nodes would not be able to synchronise when the partition healed because they would detect that operations were not being delivered in causal order (due to the garbage collection mechanism having erased operations that had not yet been propagated to the nodes on the other side of the partition).

To overcome this limitation, the system could implement some kind of mergeable state transfer, in which two nodes (even if they were previously partitioned) exchange states and are able to compute a common final state through a (deterministic) merge function. Another possible solution for this problem is to reset all the nodes on one side of the partition and subsequently make them install the state of one of the nodes of the opposite side. However, the nodes would have to disconnect themselves from all their neighbours and re-synchronise, which could cause a domino effect of resets and synchronisations. In sum, this a complex problem to overcome, and our solution offers neither of these possibilities.

3.5 Correctness Arguments

In this section, we present some informal arguments for the correctness of our algorithm.

Tree-based topologies, when used to disseminate operations via FIFO channels, inherently ensure causal delivery [9, 28]. When a node receives an operation, it executes it and atomically sends it to its neighbours. The same happens when a node generates a local operation and atomically disseminates it. This ensures that, when an operation reaches a given node, all of the operations on which it depends have already been delivered and executed. Furthermore, this approach does not require any metadata to encode causal dependencies, because operations will always be received in causal order. This serves to show that, when the tree topology in stable, our solution ensures causal delivery.

Nevertheless, when the tree suffers reconfigurations, we had to make sure that the mechanisms implemented never violated causal consistency guarantees, because our topology can no longer be considered a tree: it temporarily becomes a cyclic graph. We introduced additional techniques to impose causality in those instability moments, such as maintaining a causally-ordered list of operations to serve the purposes of our synchronisation mechanism, and filtering out any duplicate operations upon their reception by comparing their sequence number with the locally kept vector clock.

The new synchronisation mechanism, detailed in Section 3.3.3, has the purpose of allowing nodes in different states to establish a connection by becoming up-to-date with each other. This mechanism never violates causal consistency guarantees because it uses the causally-ordered list of operations to send nodes all the messages they are missing in causal order, thus making it impossible for operations to be executed before all their dependencies. The intuition for this is that when a node, A, is synchronising with another node, B, by sending him the causally-ordered list of missing operations, for each operation o with a set of dependencies D, there are only two possibilities: i) either B already has all operations of D, meaning that all dependencies of o have already been satisfied; or ii) B does not have some (or all) operations of D and, in that case, all those missing dependencies were known by A and will be included in the received list before o.

To show that our synchronisation mechanism works as described, we now give some examples of situations when causality could have been violated but, in the case of our algorithm, was not.

Example 1

Suppose a new node, B, wants to join the system while we have a fully formed (stable) tree topology. Node B will receive *Announcement* messages, via its *lazy* peers, that will eventually trigger him to send a *SendVectorClock* message, as well as a *Graft* message, to the sender of said announcement, say node A. When that happens, the first message will start the synchronisation process from node B to A, and the second message will start the synchronisation from node A to B. The nodes will then exchange a series of messages that will culminate in each node receiving a list of causally-ordered operations that they do not have (and possibly a state). In this specific case, node B would receive the state node A most recently computed, as well as the subsequent list of operations, while node A would receive an empty list of operations (assuming that node B did not execute any prior local operations). During this process, in no moment is it possible for our mechanism to break causality guarantees.

Example 2

Now let us consider the opposite case, where a node that belongs to the tree leaves the system, causing a tree branch to become separated from the rest of the tree. Eventually, this event would trigger all the same mechanism as in the previous example, except that both nodes would, most likely, receive non-empty lists of operations upon completing the synchronisation process. This reinforces our premise that causal delivery is never compromised.

Example 3

Lastly, let us examine what happens when a given node, C, tries to synchronise with another node, D, while a third node, E, is synchronising with node C:

1. Suppose node C wants to initiate the synchronisation process to node D. Node C will send him a *SendVectorClock* message, as well as a *Graft* message, but before receiving node D's vector clock, it receives a *SendVectorClock* and a *Graft* message from node E. Even though node C has an outgoing synchronisation to node D, he has no incoming synchronisations. As such, he will respond to node E with his vector clock, and will also send him a *SendVectorClock* message that will trigger the synchronisation process in the opposite direction.

2. Assume now that, node D finally sends C his vector clock, immediately followed by a *SendVectorClock* message. Node C will proceed to send him a *Synchronisation* message with his missing operations and, then, he will notice that he already has an incoming synchronisation from node E. Node C will place node D on hold until the synchronisation from node E finishes.

3. Once node C receives a *Synchronisation* message from node E, he will, at last, send his vector clock to node D, who will later send him a *Synchronisation* message, thus concluding the pending incoming synchronisation.

This example shows that, even with complex interleavings of messages from different nodes, our protocol is able to complete synchronisation in a causal manner.

With the reasons presented above, and the specific examples provided, we argue that our algorithm correctly imposes the guarantees of causal consistency in all possible scenarios.

Summary

In this chapter, we presented our base solution, SYNC Tree, as well as its final iteration, ECO SYNC Tree. Our work gave way to a new causal broadcast algorithm that offers causal consistency by making use of a tree topology paired with a new synchronisation mechanism. Our solution is able to dynamically and efficiently adapt to membership changes in a scalable and decentralised manner, thus being edge-enabled. Furthermore, we discussed our state transfer and garbage collection mechanisms that make our solution need less disk space to work, and become faster when new nodes need to synchronise with others to join the system. Finally, we presented some informal arguments to prove the correctness of our work.

In the following chapter, we briefly discuss some prototype implementation details, as well as a visualisation tool developed in the course of this work.

4 Prototype Implementation

In this chapter, we present some details regarding the implementation of the prototypes of both versions of our algorithm, and discuss a simple visualisation tool created to help in the implementation process. The rest of this chapter is organised as follows:

In Section 4.1, we discuss the implementation details of the SYNC Tree prototype.

In Section 4.2, we present additional details that apply only to the prototype of the final iteration of our algorithm, ECO SYNC Tree.

In Section 4.3, we briefly explore our tree visualiser, implemented to help understand the intricate behaviour of the nodes in our solution.

4.1 SYNC Tree Prototype

This section discusses some details regarding the implementation of the prototype of SYNC Tree. These details do not change the inner workings of the presented algorithm, they only give specific insights into how our Java-based implementation operates.

4.1.1 The Babel Framework

To abstract the communication between the different nodes of the network, as well as between the different layers of the system within each node, we used a framework developed at NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) called Babel[1].

Babel executes each of the layers of the system (membership, broadcast, replication and application) as a thread with an event queue. These layers run in parallel, independently from each other. Communication between layer A and B is achieved by thread A enqueueing an event into the event queue of thread B.

Furthermore, Babel can establish independent TCP connections between the same layers of different nodes, allowing them to communicate with each other. To do this, messages received from a TCP connection are also placed in the event queue of the receiving layer (thread). In our prototype, both the membership layer and the broadcast

[1] https://github.com/pfouto/babel-core

layer maintain their own TCP connections. This means that, even if the broadcast layer channels are saturated, the membership messages are still being exchanged and are not stuck in a shared event queue that is full with broadcast related messages.

4.1.2 Fault Detection

In our solution, fault detection is done only at the membership layer, meaning that the broadcast layer does not try to infer any information regarding node faults or network failures, it simply reacts blindly to the information received from the membership layer. For example, if an attempt at a TCP connection fails at the broadcast layer, the broadcast protocol will keep trying to open that connection until the membership layer notifies it that the node has failed.

4.1.3 Layer Independence and Modularity

The code developed for our prototype was written with modularity in mind, i.e., the several components (layers) are independent from each other and can be altered without compromising the functionality of the remaining ones. Some details that ensure modularity are: i) the membership layer only needs to generate *NeighbourUp* and *NeighbourDown* events that the broadcast layer will react to (Alg. 3.3 - line 94-104), and ii) the broadcast layer disseminates gossip messages without knowing anything regarding their content, i.e., it receives requests to propagate opaque arrays of bytes from the replication layer and guarantees that they are delivered to the replication layer of every node. This ensures that any type of information can be disseminated by the broadcast layer.

4.1.4 Writing to, and Reading from Disk

In order to make the process of reading missing operations from disk more efficient and consequently shortening the time it takes for two nodes to synchronise, we implemented index-based search over the file in which the causally-ordered list of operations is written. This enhanced search prevents nodes from reading all the content of the file (which becomes bigger over time), when it is likely that all operations required are near the end of the file. Figure 4.1 shows a representation of the indexes used to read missing operations from file.

Each node maintains an index for each of the nodes of its local vector clock. Each index contains entries that represent the sequence number of an operation generated by that node and point to the location of the file where that operation is stored. The spacing between each entry on our indexes is configurable, meaning that we can use denser or sparser indexes according to our needs. The spacing used in Figure 4.1 was 10, so there is one index entry for each 10 operations of a given node.

We will now use the example presented in Figure 4.1 to detail how we use the indexes to efficiently search for the missing operations, upon receiving a remote vector clock:

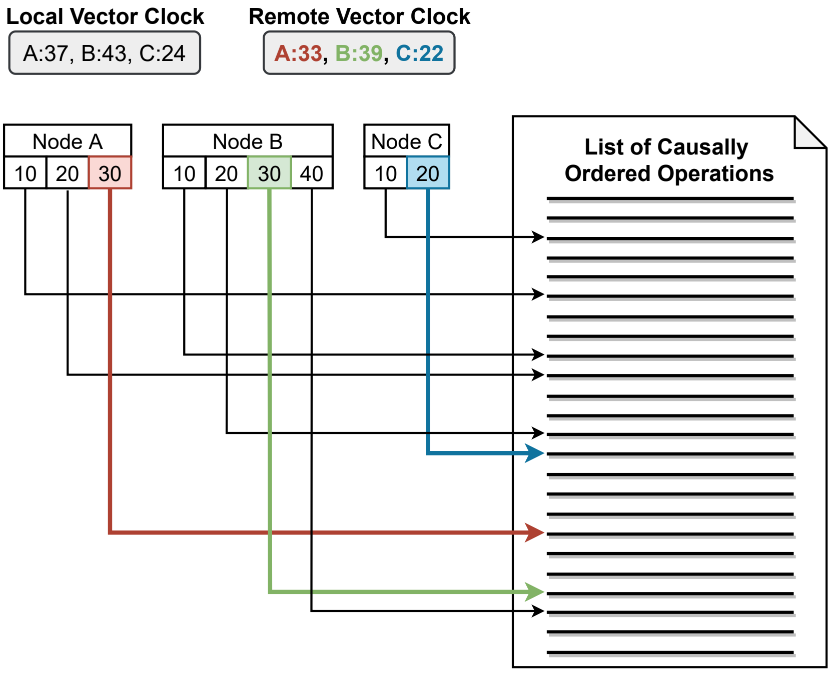


Figure 4.1: Indexes created to read operations from file.

- First, we analyse the index for node A and find the greatest entry whose sequence number is smaller or equal than 33. This corresponds to the entry of sequence number 30, which points to the RED location of the file;

- Then, we do the same for node B finding entry 30, that is the greatest one smaller or equal to 39. That entry points to the GREEN location of the file;

- Finally, we find entry 20 of node C's index, which is the greatest entry smaller or equal than 22 and points to the BLUE location of the file;

- After finding a location for each of the indexes, we then choose the minimum between all of them, i.e., we choose the one that is closest to the beginning of the file, in this case, the BLUE location. This is done to guarantee that we do not skip operations that the remote node does not yet have;

- Finally, we read the file by skipping to the BLUE location and comparing each operation individually against the remote vector clock to filter out duplicates, as detailed in Section 3.3.3.

If there are any entries in the local vector clock that are not in the remote one, the file is read from begging to end because the remote node does not have any operations from the node of the corresponding entry. If it is the other way around and there is an entry in the remote vector clock that is not in the local one, that entry is ignored because it shows that the remote node has operations from a node that we have no information about and have not received operations from.

4.2 ECO SYNC Tree Prototype

This section discusses some details regarding the implementation of the prototype of ECO SYNC Tree. These details are incremental to the ones presented in the previous section and do not change the inner workings of the presented algorithm, only giving specific insights into how our Java-based implementation operates.

4.2.1 File Organisation and Erasure

In order to facilitate the garbage collection of old operations from disk, we decided to partition the list of causally-ordered operations and distribute it across several files, instead of writing it in its entirety in a single file. We did this to allow the deletion of an entire file when the garbage collection mechanism was executed, instead of requiring us to delete some bytes at the beginning of the file and then shift all the remaining ones. Because we changed the file structure, we had to adapt the way in which missing operations are read from the files in order to keep it efficient. Figure 4.2 shows a representation of the file organisation, as well as the metadata associated with each file.

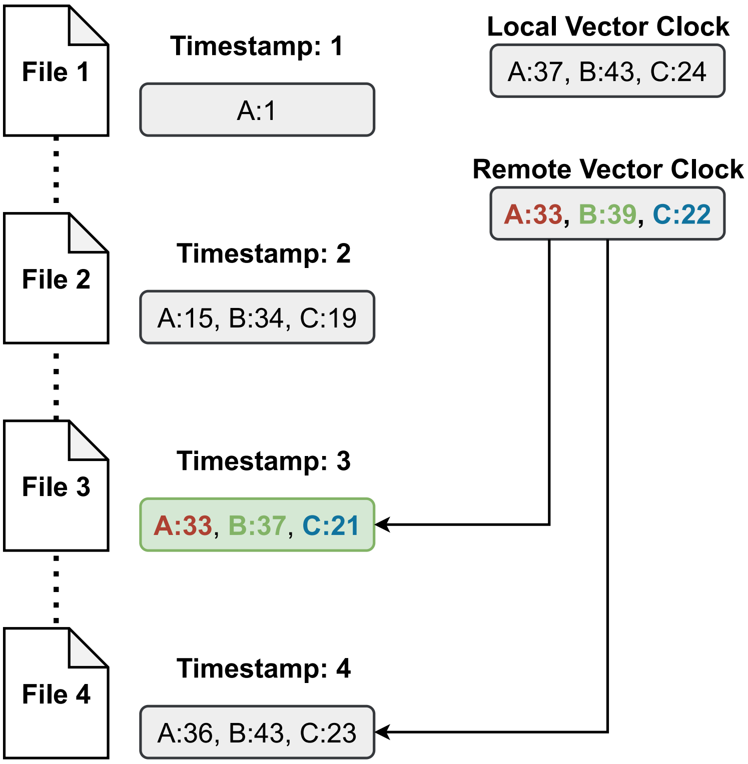


Figure 4.2: Organisation in multiple files with associated timestamps and vector clocks.

Similarly to systems like ZooKeeper [19], to know when to create or delete a file we use a basic time unit (in seconds), based on the *garbageCollectionTimeout*, that we will call *tickTime*. Each node of the system creates a new file every *tickTime* seconds, tags it with a timestamp that encodes the time of its creation (in *tickTime* units) and with the vector clock of the first operation written to it, and adds it to the tail of a list of files ordered by their timestamps.

We will now use the example presented in Figure 4.2 to detail how the vector clock of each file is used to search for the missing operations, upon receiving a remote vector clock. Note that, we analyse the files starting from the tail of the list and ending at its head. This is done because there is a high probability that the operations needed are written in the files that were most recently created, i.e., the files at the end of the list.

- First, we analyse the vector clock of *File 4* against the remote vector clock. Because the remote vector clock is "smaller" than that of *File 4*, we know that the remote node needs operations from earlier files;
- Then, we do the same for *File 3*, finding that the remote vector clock is "larger" than the vector clock of the file. This means that the operations the remote node needs start somewhere within *File 3*. This stops the iteration over the files, because we have found the first file we need to read;
- Finally, the search proceeds by reading the missing operations from *File 3* and *File 4*, with the help of the file indexes detailed in Section 4.1.

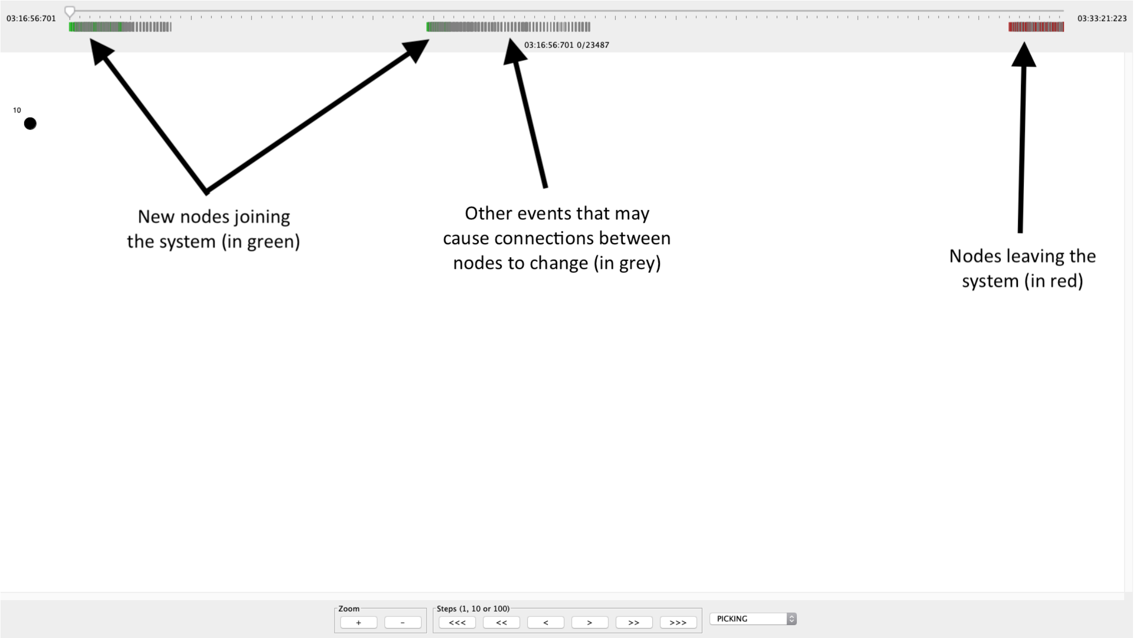
To garbage collect operations that are no longer needed, we simply iterate over the list of files, from head to tail, and remove from disk, and from the list itself, the files whose timestamp is older than *garbageCollectionTTL* (in *tickTime* units).

4.3 Visualisation Tool

During the development of our prototypes we faced several difficulties. Due to the complex interactions between the nodes of the system, it became impossible for us to understand the root cause of most problems simply by analysing application logs. Some of these problems triggered causality guarantees to be violated, others made it impossible for the broadcast tree to stabilise after new nodes joined the system, or caused nodes to become isolated.

These problems impaired core features of our protocol, so we had to resort to implementing a visualisation tool that parsed our application logs and allowed us to visually analyse the connections and interactions between nodes, to better understand the causes of our problems. This visualiser focused on showing: i) changes made to the *eager* peers of nodes, ii) changes made to the *lazy* peers of nodes, iii) active synchronisations, and iv) "one-way" tree branches.

Figure 4.3 shows the graphical user interface (GUI) of the visualiser. At the top, we can see a slider that allows us to move back and forth in the time of the experience. Below the time slider, we opted to present the different changes that can occur to the nodes of the system:

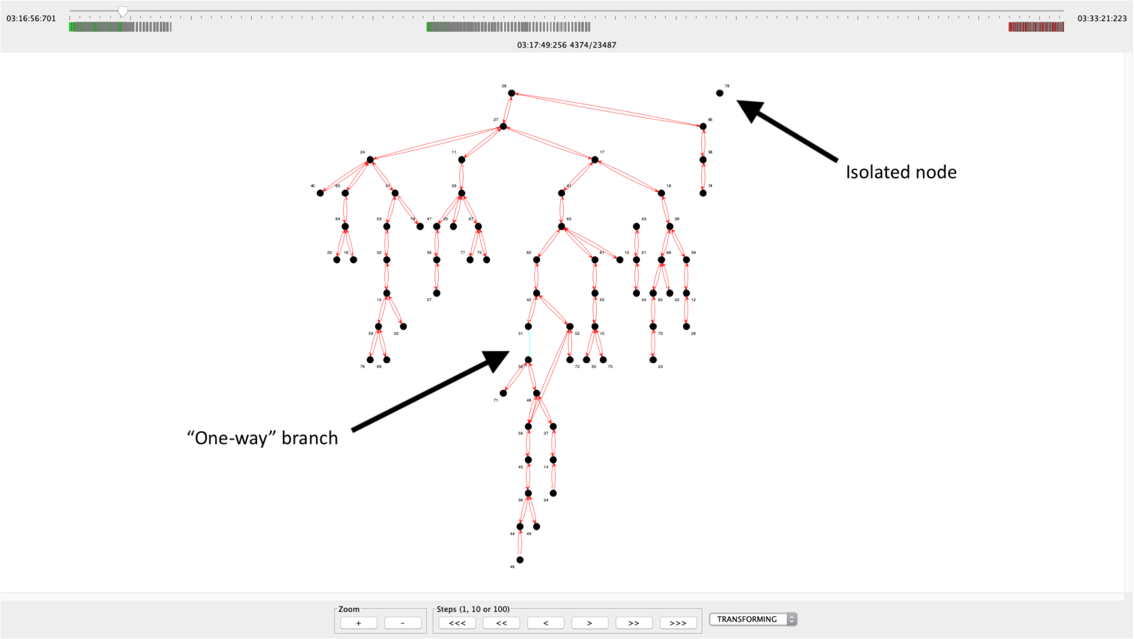


New nodes joining the system (in green)

Other events that may cause connections between nodes to change (in grey)

Nodes leaving the system (in red)

Figure 4.3: Visualisation tool: components of the GUI.



Isolated node

"One-way" branch

Figure 4.4: Visualisation tool: isolated node and one-way branch.

- In green, we can see that new nodes entered the system;
- In red, we marked the times at which nodes leave the system;
- Finally, in grey, we represent all other events that changed the partial views of the nodes.

At the bottom, we have several other buttons that allow us to zoom in and out, move

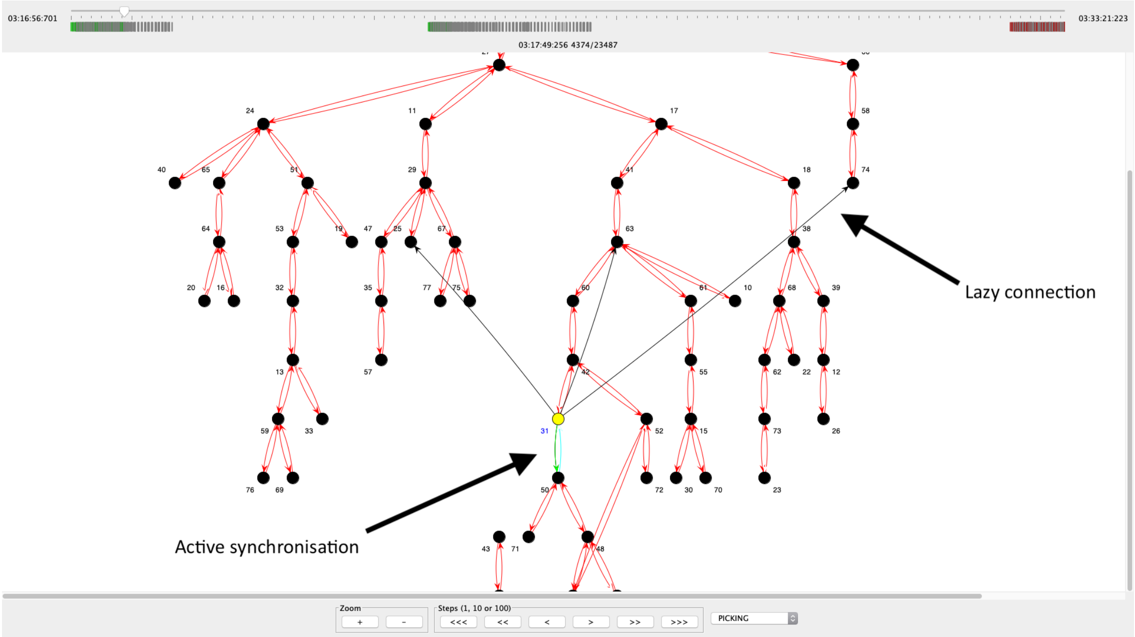


Figure 4.5: Visualisation tool: active synchronisation and lazy connection.

forward or backward a predefined number of events, and select nodes.

Figure 4.4 visually demonstrates how we were able to detect when nodes became isolated or when there was a "one-way" tree branch. When this happened, we had to understand if these events were temporary or permanent, by moving along the time variable of the experiment.

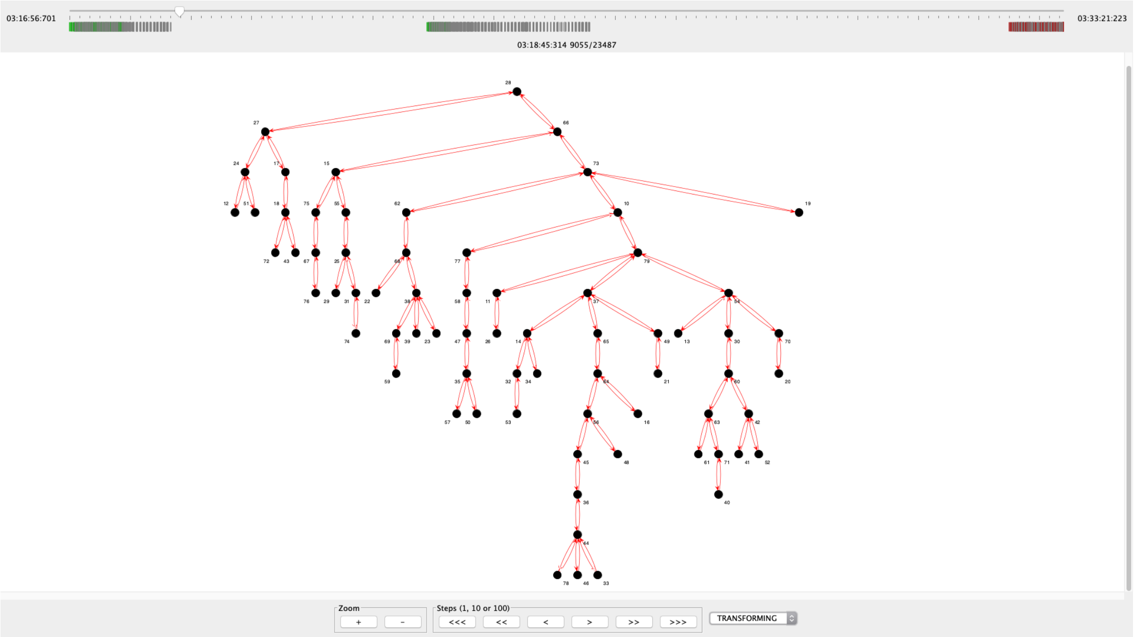


Figure 4.6: Visualisation tool: causal broadcast tree after stabilisation.

Furthermore, in Figure 4.5 we can see that the visualiser also allows us to see active synchronisations. In the presented example, we can clearly detect that the "one-way" branch between node 31 and node 50 is temporary, because there is a synchronisation still happening in the opposite direction. In cases where the problems persisted, we were able to pinpoint the time at which the error occurred and easily comprehend the problem by looking at the application logs at that time. Additionally, Figure 4.5 also shows that it is possible to select nodes and, when that happens, its lazy connections become visible. Finally, Figure 4.6 shows the broadcast tree fully formed, after the protocol stabilises.

This visualisation tool allowed us to better understand the dynamics of our algorithm, helped us perfect the implemented prototypes, and be more confident that they worked as we intended them to.

Summary

In this chapter, we discussed some implementation details regarding the prototypes of both SYNC Tree, and ECO SYNC Tree. Furthermore, we presented a visualisation tool created to aid in the development of our prototypes. Both prototypes of our protocol can be found in https://github.com/Sunnout/ECO-SYNC-Tree, and the visualisation tool implementation can be found in https://github.com/Sunnout/TreeVisualiser.

In the following chapter, we present a thorough experimental evaluation of both SYNC Tree and ECO SYNC Tree.

5 Evaluation

In this chapter, we present an extensive experimental evaluation of the prototypes of both iterations of our algorithm, so as to demonstrate their applicability in edge-based scenarios. The rest of this chapter is organised as follows:

In Section 5.1, we discuss the main objectives of the performed experimental evaluation.

In Section 5.2, we start by describing the environment in which our experiments were conducted and then proceed to explain the several parameters and scenarios used.

In Section 5.3, we detail the setup of the experimental evaluation of SYNC Tree against other broadcast protocols, as well as present and analyse the results obtained.

In Section 5.4, we show the benefits of ECO SYNC Tree when compared with SYNC Tree.

5.1 Objectives

In the first part of this experimental evaluation, we try to measure the performance of SYNC Tree against other broadcast protocols and answer the following questions:

- How does our solution perform in stable environments, in terms of latency and communication cost? Does it perform better than other alternative protocols?
- Is SYNC Tree able to deal with increasing number of nodes, payload size, or load of the system?
- Is our new causal broadcast protocol able to outperform other broadcast protocols in edge-based scenarios?
- Is SYNC Tree capable of handling events, such as joins or failures of larger groups of nodes, while still remaining efficient?

The second part of the experimental evaluation tries to analyse the impact of the state transfer and garbage collection mechanisms. As such, we compare ECO SYNC Tree against SYNC Tree and answer the following questions:

- Is our state transfer mechanism capable of decreasing synchronisation times, communication cost, and the overall broadcast latency?
- Does our garbage collection mechanism solve the problem of the ever increasing disk usage?
- Does ECO SYNC Tree make the tree topology stabilise faster than SYNC Tree?

Before we address these questions, we detail our experimental methodology.

5.2 Experimental Methodology

The experiments discussed in this chapter were carried out on the Grid'5000[1] testbed. All the machines used have an 18-core CPU (Intel Xeon Gold 5220), 96GB RAM, 480GB of SSD storage, and are connected by a 25 Gbps network.

With the aim of evaluating our solution as realistically as possible, we used Docker's swarm mode to emulate a distributed network. Each of the Grid'5000 machines that we used was host to 50 Docker containers, thus ensuring that the load was evenly distributed, and that the environment remained unaltered for all experiments. Within each Docker container, we run a single node of the broadcast protocol.

To more accurately emulate a real world environment, we introduced latency between each pair of nodes with the help of Linux Traffic Control (Linux TC). We varied the latencies between 10 and 100 milliseconds in order to approximate the latencies of real-world geo-distributed systems. To compute the latencies between each pair of nodes, we distributed the nodes in a two-dimensional grid with `N` positions (where `N` is the number of nodes), e.g., if an experiment had 200 nodes, we considered a 20x10 grid. Finally, we computed the euclidean distance between each pair of nodes and then scaled the resulting values so that they were all in the interval $[10, 100]$. We chose to dispose the nodes in this grid and compute the latencies in the previously described way to ensure that the latencies between nodes adhered to the Triangle Inequality Theorem.

In our experiments, we took several parameters into account. Firstly, the number of nodes of the broadcast protocol was varied (50, 100, and 200). In order to emulate lighter workloads, we also varied the probability of each node sending an operation ($p = 0.3$, $p = 0.6$, and $p = 1$). Lastly, we varied the size of the payload of the operations (128KB, 1024KB, and 4096KB).

Our experiments consisted of launching the initial nodes, one at a time, giving a small interval between each launch so that HyParView [23] did not cause nodes to become isolated. Each of the nodes waited a predefined amount of time, which we call `warmup` time, to ensure that the overlay created by the membership management protocol had stabilised.

[1] All experiments presented in this work 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 organisations (see https://www.grid5000.fr).

Then, during the next 10 minutes, which we call the run time, each node, with probability p, generated a new operation every second for a LWW Register CRDT (implemented as in Alg. 2.3) and disseminated it by using the broadcast protocol. After the run time passed, each node stopped generating operations.

Bear in mind that, since not all nodes joined the network at the same time, they stop generating operations and leave the network at different times. With this said, each node waited an extra 3-6 minutes before leaving the system, which we call cooldown time, to ensure that all nodes had finished sending operations and that those operations had been received by every other node.

Additionally, during the runtime of our experiments, we used four different experimental scenarios where we, sometimes, introduced events in which nodes joined or left the system:

1. **Stable scenario:** This scenario is meant to show the performance of our system when the network remains stable. This is the base scenario, already described above, in which all nodes join the system before operations start being propagated, and then simply leave the system after operations stop being propagated;

2. **Churn scenario:** This scenario is the one that tries to approximate an edge-based deployment, i.e., a deployment that has higher churn rates due to the high probability of nodes constantly joining and leaving the system. In this scenario, a group of nodes enters the system before operations start being propagated and, after a short period of time, small groups of nodes (with size of 4% of the initial nodes) begin to leave and join the network, periodically (every 30 seconds), until a given time at which this process stops. All nodes that remain in the system will leave after operations stop being propagated;

3. **Massive join scenario:** This scenario is meant to showcase the performance of our protocol when there is a sudden spike in nodes joining the system. In this scenario, a group of nodes enters the system before operations start being propagated. Then, mid way through the experiment, a block of nodes (with size of 30% of the final nodes) joins the system simultaneously. All nodes that remain in the system will leave after operations stop being propagated;

4. **Catastrophic failure scenario:** This scenario is the opposite of scenario 3, where there is a sudden spike in nodes leaving the system. Similarly, a group of nodes enters the system before operations start being propagated and, mid way through the experiment, a block of nodes (with size of 30% of the initial nodes) leaves the system simultaneously. All nodes that remain in the system will leave after operations stop being propagated.

Each of the experiments was executed 5 times, and we extracted the results presented in the following sections by averaging the results of the individual runs.

5.3 SYNC Tree Evaluation

In this section, we report our experimental evaluation of SYNC Tree against other broadcast protocols, which we proceed to describe below. Then, we discuss how those protocols were parameterised for our experiments and analyse the obtained results.

5.3.1 Additional Broadcast Protocols

To measure the performance of SYNC Tree, we compared it to two other alternative broadcast protocols: flood and periodic pull. As such, we had to implement their causal variants and, to do so in a fair way, they maintain and propagate the same information as our solution (i.e., each node keeps a local vector clock, and each message is propagated along with a sequence number). The implemented variants work as follows:

Causal Flood: a protocol that employs an eager push gossip strategy (flood), in which each node propagates operations to all the neighbours of his partial view, when they are first received. To achieve causal delivery in the Causal Flood protocol, each node uses our synchronisation mechanism, described in Section 3.3.3, to synchronise with each of his neighbours before starting to propagate messages to them. This protocol tends to deliver messages to all nodes very quickly, but incurs in additional communication cost due to the large quantity of duplicate messages sent.

Periodic Synchronisation: a protocol that uses a periodic pull gossip strategy, in which each node periodically synchronises with a randomly chosen neighbour, i.e., each node asks another to send him all the needed missing operations. The Periodic Synchronisation protocol used a simplified version of our synchronisation mechanism, where a given node sends a *VectorClock* message and, as a response, receives a *Synchronisation* message with his missing operations. This protocol sits on the other end of the trade-off between broadcast latency and communication cost, delaying message delivery but avoiding duplicate messages altogether.

To ensure that the way in which we store operations does not affect performance, both of these protocols use the approach detailed in Section 4.1.4 to write to, and read from, disk in the same manner as our solution.

5.3.2 Protocol Parameterisation

SYNC Tree was compared with the Causal Flood protocol, as well as two versions of the Periodic Synchronisation Protocol: one with a larger period of 1000 milliseconds, i.e., each node of the protocol waits 1000 milliseconds between each synchronisation step, and one with a smaller period of 200 milliseconds.

The parameters chosen for SYNC Tree were: a $treeMsgTimeout$ of 100 milliseconds, an $announcementTimeout$ of 3 seconds, and a $checkTreeMsgsTimeout$ of 5 seconds. This meant that *Tree* messages were sent every 100 milliseconds, the timers for their announcements expired after 3 seconds, and at every 5 seconds each node verified if it had received any *Tree* messages from relevant neighbours.

When writing operations to disk, we used an index spacing of 50 for all protocols. Finally, all experiments were run using HyParView as the membership management protocol, with the partial view of each node corresponding to the active view of HyParView, which was parameterised with a size of 5.

5.3.3 Results and Analysis

In this section, we present the results obtained by comparing SYNC Tree with the three other broadcast protocols described above: Causal Flood, Periodic Synchronisation (with period of 1000 ms), and Periodic Synchronisation (with period of 200 ms).

To assess our protocol's performance, we measured the following metrics:

- **Average Broadcast Latency:** The average time (in seconds) it takes for a *Gossip* message to be delivered to all nodes of the system.
- **Number of Duplicate Messages:** The sum of the number of times each node of the system received a duplicate *Gossip* message.
- **Communication Cost:** The total bandwidth usage (in GB) of the broadcast protocols, i.e., the sum of the number of GBs that passed through the broadcast layer links of every node in the system.

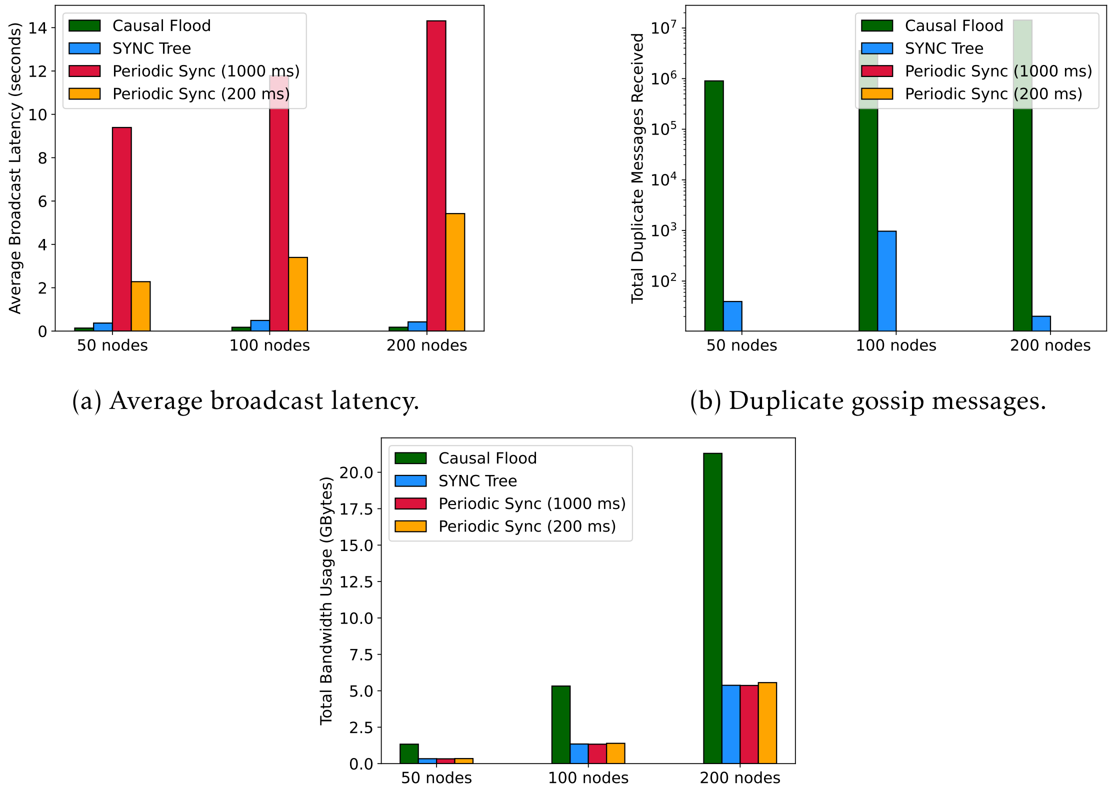
Additionally, we measured how these metrics vary over time for the duration of the experience.

In each of the following sections, we show only the most relevant results for the scenario being studied. In Annex I, we present some of the additional results extracted during our experimental work.

Stable Scenario

In a stable scenario, we wanted to infer whether or not our protocol was able to remain efficient with increasing numbers of nodes, increasing payload size, and increasing p (which represents the probability of a node sending a *Gossip* message each second). Furthermore, we wanted to compare its performance against the other competing protocols.

Figures 5.1a, 5.1b, and 5.1c show, respectively, the average broadcast latency (in seconds), the number of duplicate *Gossip* messages, and the communication cost (in GB) of all four broadcast protocols when we fix p to 1, the payload size to 1024KB, and vary the number of nodes in a stable scenario. Note that, the y axis of Figure 5.1b is in logarithmic scale and, as such, the disparities presented are much bigger than they might appear at first sight.



(a) Average broadcast latency.

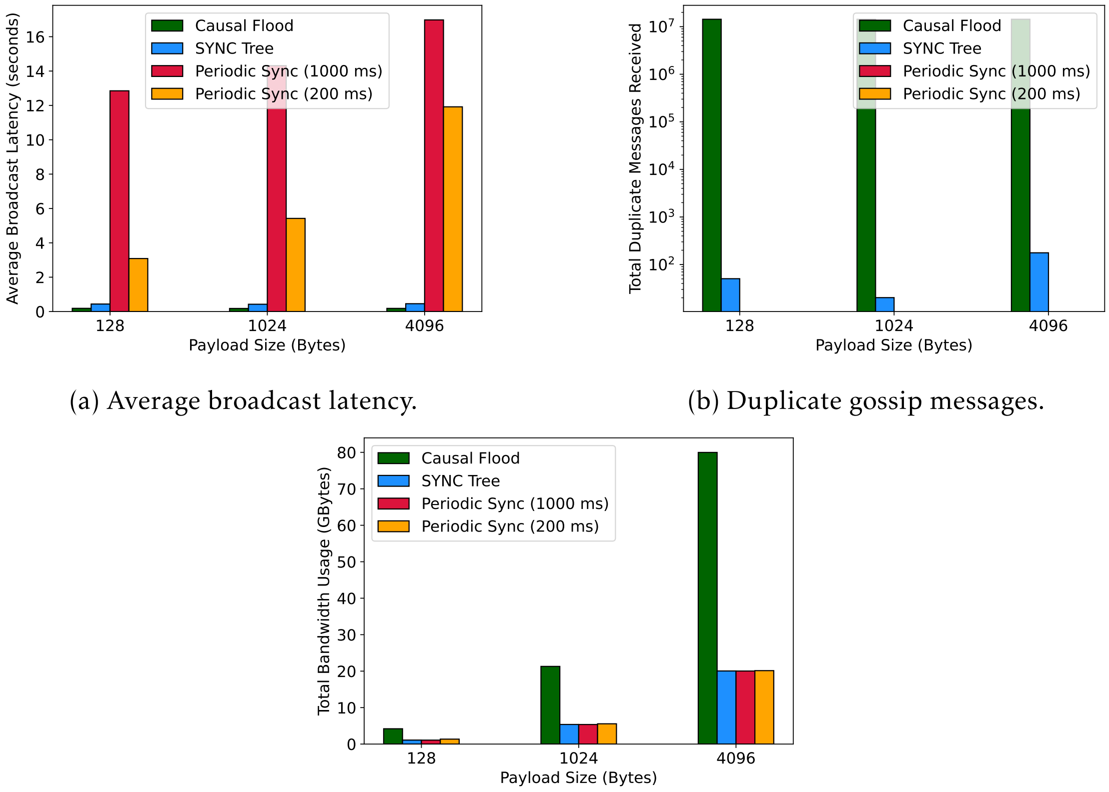
(b) Duplicate gossip messages.

(c) Communication cost.

Figure 5.1: Metrics in stable scenario with $p = 1$, payload of 1024KB, and varying number of nodes.

In Figure 5.1a, we can observe that SYNC Tree has extremely low average broadcast latency when compared to both configurations of the Periodic Synchronisation protocol. This is to be expected, since our protocol receives and propagates the operations as they are generated, whereas the Periodic Synchronisation protocol waits for the next synchronisation step to propagate the accumulated set of operations. Note that, even with smaller synchronisation periods of 200 milliseconds, SYNC Tree still shows lower latency. On the other hand, our solution has higher latency than the Causal Flood protocol, which is justified by the fact that, when flooding, each node sends messages directly to all its neighbours, significantly reducing the number of hops that a message needs until it is delivered to all nodes. However, the latency of SYNC Tree when compared to that of the Causal Flood is only marginally higher and remains relatively constant in all tested scenarios.

As expected, in Figure 5.1b, we can see that since both configurations of the Periodic Synchronisation protocol perform one synchronisation at a time, they do not incur in duplicate messages. SYNC Tree, by removing tree branches through which duplicates are received, shows significant improvements when compared to the Causal Flood protocol which, due to its underlying cyclic graph topology, presents high levels of redundancy



(a) Average broadcast latency.

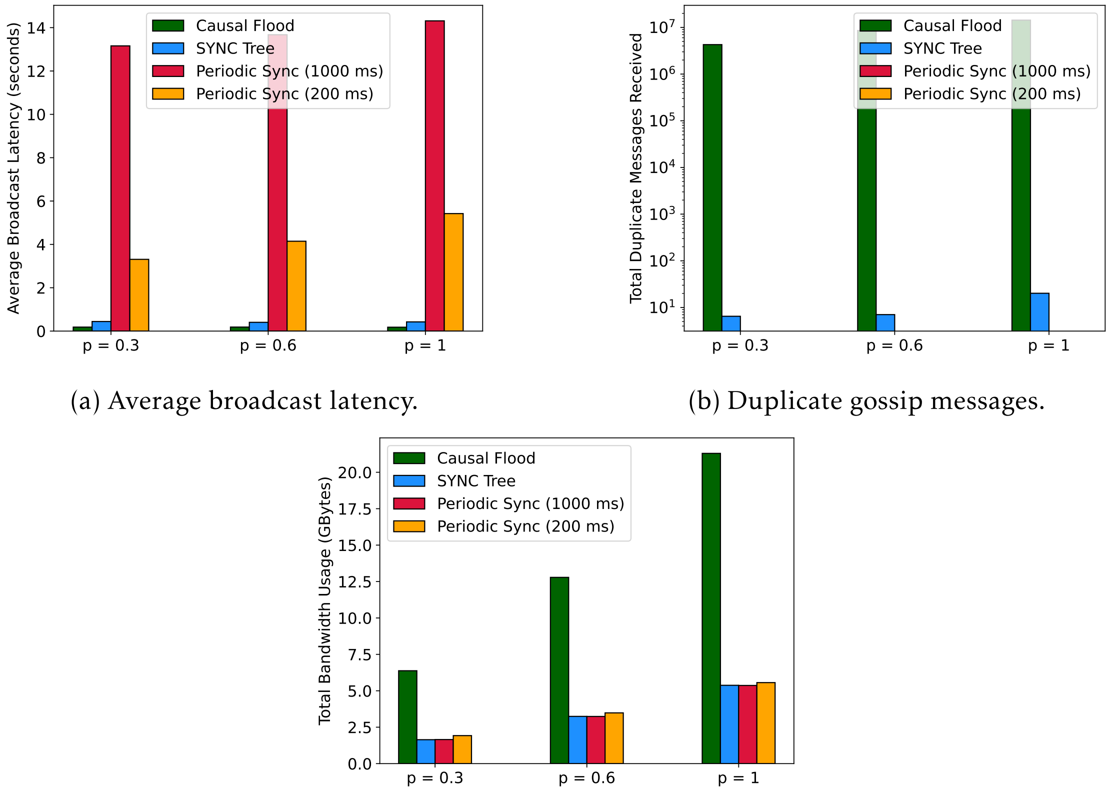
(b) Duplicate gossip messages.

(c) Communication cost.

Figure 5.2: Metrics in stable scenario with 200 nodes, $p = 1$, and varying payload size.

and, consequently, an elevated number of duplicate messages. For reference, the total number of non-duplicate messages received in each experiment was 1.5×10^6, 6×10^6, and 2.4×10^7, for 50, 100, and 200 nodes, respectively. Note that, in SYNC Tree the time it takes for the tree to stabilise is not constant and, as such, the number of duplicates may vary regardless of the number of nodes, as shown in the figure. However, that number is a very small percentage of the total number of non-duplicate *Gossip* messages received and is always several orders of magnitude lower than the number of duplicates of the Causal Flood protocol.

Furthermore, in Figure 5.1c we show that SYNC Tree has a communication cost that is significantly lower than that of the Causal Flood protocol. This is expected, as each node of the Causal Flood protocol sends messages to all the neighbours in its partial view, thus increasing the number of bytes transmitted when compared to SYNC Tree, where messages are only sent via tree branches. Our solution incurs in additional communication cost by sending *Tree* messages and their respective *Announcement* messages. However, we can see that the overall cost in only marginally higher than the cost of the Periodic Synchronisation protocol with period of 1000 milliseconds, and lower than the cost of the Periodic Synchronisation protocol with period of 200 milliseconds, since this version of the protocol propagates large *VectorClock* messages very frequently. This serves to



(a) Average broadcast latency.

(b) Duplicate gossip messages.

(c) Communication cost.

Figure 5.3: Metrics in stable scenario with 200 nodes, payload of 1024KB, and varying *p*.

show that the cost of introducing *Tree* messages to stabilise the tree does not greatly affect our protocol's performance, and proves that by increasing the frequency of synchronisations of the Periodic Synchronisation protocol, it will incur in higher communication cost without ever achieving a lower broadcast latency than SYNC Tree (as shown by Fig. 5.1a).

Finally, by analysing Figure 5.1 as a whole, we can conclude that SYNC Tree is able to remaining efficient and scale to a large number of nodes, since the broadcast latency grows logarithmically. The communication cost is minimal, being negligibly affected by the low number of duplicates messages (as seen by comparing the communication cost of our solution with that of the Periodic Synchronisation protocol), and growing linearly with the total number of messages generated, which is unavoidable since every node needs to receive every message.

Figures 5.2a, 5.2b, and 5.2c show the same metrics of Figure 5.1, when we fix *p* to 1, the number of nodes to 200, and vary the payload size in a stable scenario.

In these figures, we can see that the latency of the Periodic Synchronisation protocols tends to increase with the size of the payload. This happens because synchronisation steps take longer to process operations and complete. Furthermore, the figure shows that SYNC Tree's performance withstands increases in the size of the payload of operations. As the payload size grows, all the measured metrics remain relatively stable, except for the bandwidth used by the protocol. This rise in communication cost is expected (and unavoidable) because we increased the number of bytes of each operation payload, effectively propagating more bytes through the network.

Figures 5.3a, 5.3b, and 5.3c show the same metrics of Figure 5.1 when we fix the payload size to 1024KB, the number of nodes to 200, and vary p in a stable scenario.

Again, these figures show that SYNC Tree tolerates heavier workloads, i.e., our protocol still performs well when more operations are sent by each node. As with the results presented in Figure 5.2, the latency of the Periodic Synchronisation protocols is negatively affected by the increase in p, because synchronisations carry more messages and take longer to complete. Obviously, the greater the probability p of a node sending an operation, the greater the communication cost will be for all protocols.

All the results presented above show that, in a stable scenario, SYNC Tree is able to withstand increases in numbers of nodes, payload size, and p (workload intensity), while still remaining efficient. Furthermore, we can also conclude that SYNC Tree shows a good balance in the trade-off between average broadcast latency and communication cost, being a considerably better choice than the competing broadcast alternatives.

Unless explicitly stated otherwise, all results presented henceforth represent runs with 200 nodes, $p = 1$, and payload of size 1024KB, which we adopted as our base test case.

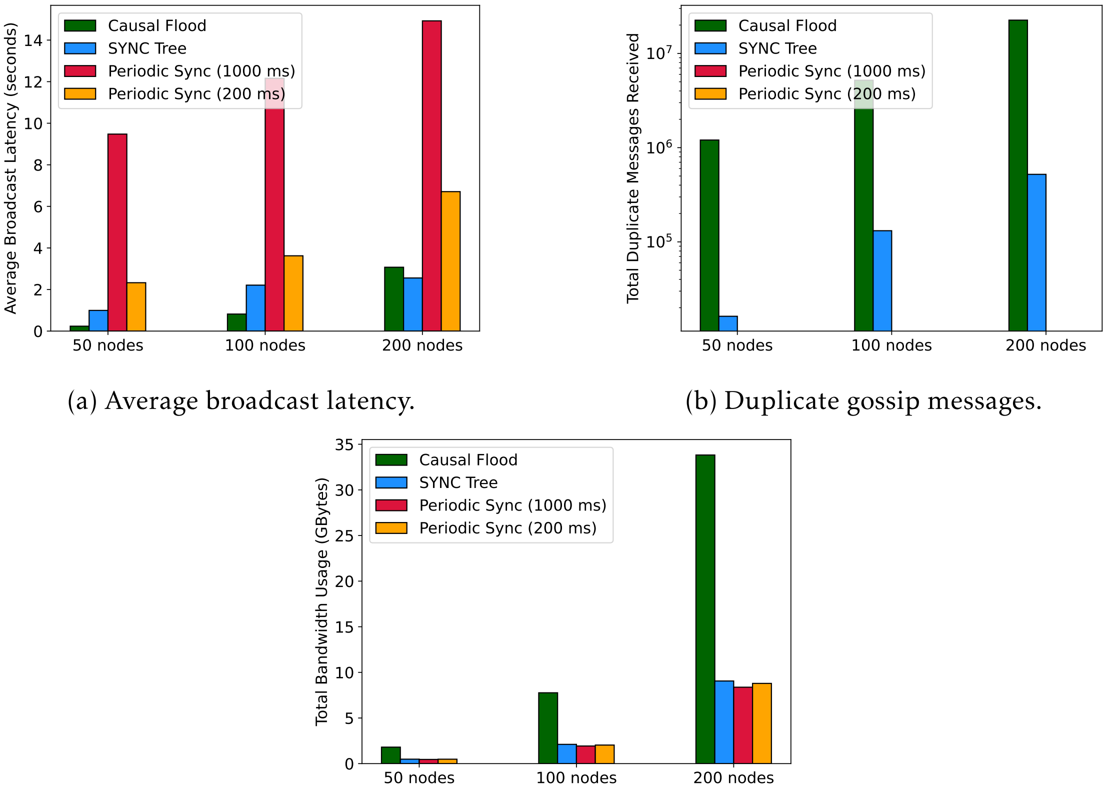
Churn Scenario

With churn scenarios, we wanted to approximate edge conditions, where nodes are constantly joining and leaving the system, to evaluate how our protocol reacts and to study how it compares to the other broadcast protocols.

Figures 5.4a, 5.4b, and 5.4c show the same metrics of Figure 5.1 when we fix p to 1, the payload size to 1024KB, and vary the number of nodes in a churn scenario.

Figures 5.5, 5.6, and 5.7 show, respectively, how the average broadcast latency, the number of duplicates, and the communication cost varied over time. We extracted the presented metrics with a granularity of 1 second and, to obtain the communication cost and the number of duplicate *Gossip* messages, we measure the total number of bytes sent, and the total number of duplicate *Gossip* messages received by every node, in each second of the experiment. For the average broadcast latency, each measure represents the time it took for the messages sent in that second to reach all nodes. Note that these figures omit some time at the beginning and end of the experiment, to focus only on the portion of the experience where the churn steps occurred.

One of the first things to note in Figure 5.4a is that, as the number of nodes increases in churn scenarios, the average broadcast latency of the Causal Flood protocol surpasses that of SYNC Tree. As made clearer in Figure 5.5, this happens because, with the Causal Flood protocol, the latency spikes get bigger in each consecutive churn step, as opposed to SYNC Tree, whose spikes grow in a more controlled manner.



(a) Average broadcast latency.

(b) Duplicate gossip messages.

(c) Communication cost.

Figure 5.4: Metrics in churn scenario with $p = 1$, payload of 1024KB, and varying number of nodes.

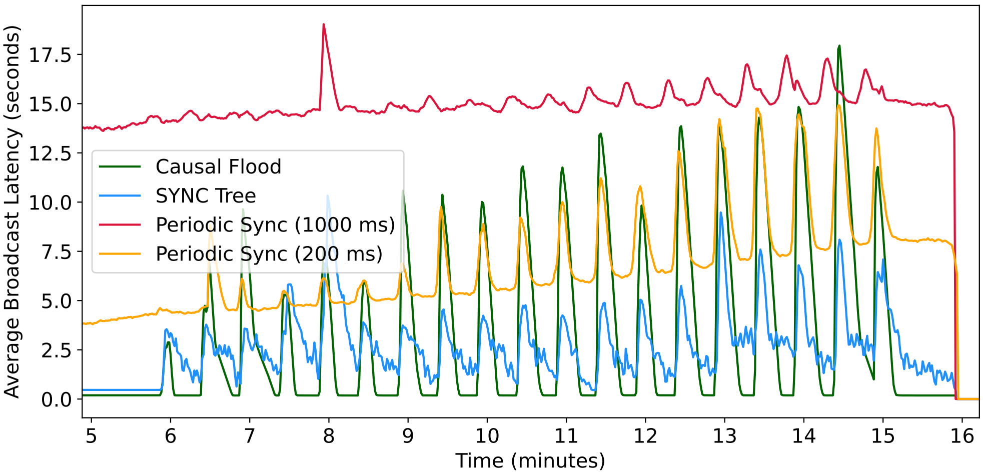


Figure 5.5: Churn: Average broadcast latency of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

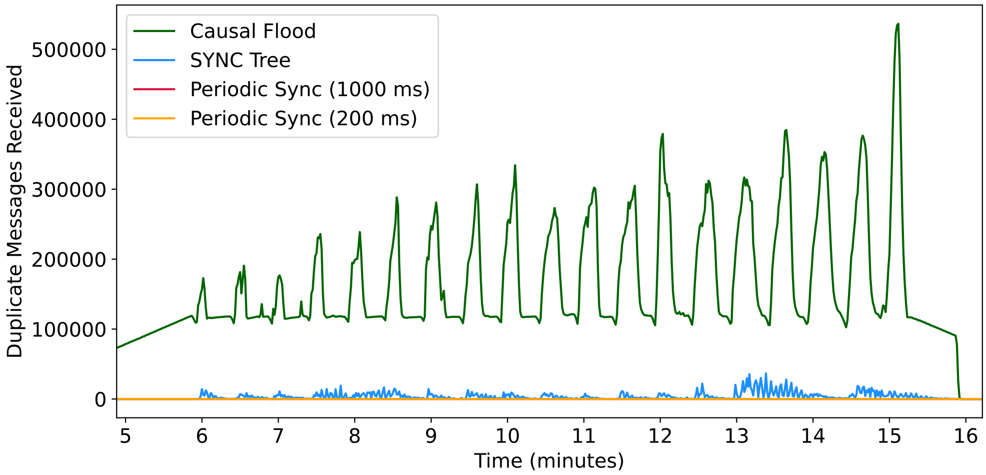


Figure 5.6: Churn: Number of duplicate messages of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

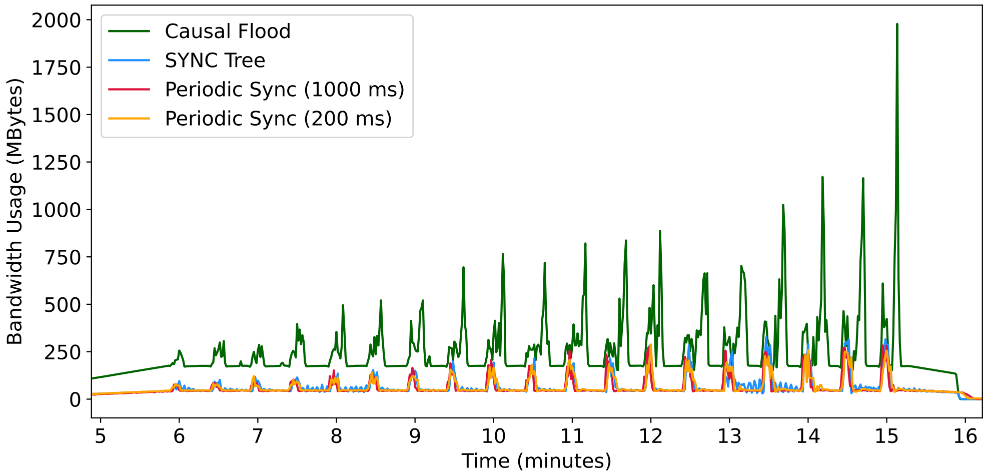


Figure 5.7: Churn: Communication cost of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

As evidenced by Figure 5.6 and 5.7, these latency spikes occur due to the significant increase in the propagation of duplicate messages in synchronisation steps and, consequently, the increase in bytes passed through the network. The performance of SYNC Tree, in this aspect, is better due to the fact that less synchronisations need to be performed, because each node is only required to synchronise to form tree branches, contrary to what happens in the Causal Flood protocol where each node needs to synchronise with all its neighbours.

Similarly, the latency of both Periodic Synchronisation protocols increases over time, not because the protocols present duplicates, but because of the increase in the number of messages that need to be sent to new nodes that join the system. Each individual synchronisation will take longer to complete, and a new node may need multiple synchronisations before receiving every missing operation. In the case of the Periodic Synchronisation protocol with period of 1000 milliseconds, this increase is only noticeable towards the end of the experiment because, as it presents a higher latency to begin with, the latency will take a longer time to start rising. SYNC Tree is able to deal with the churn events more efficiently, effectively presenting a gentler growth in latency and, therefore, being more suitable for edge-based scenarios.

Massive Join Scenario

In the experiences with massive join scenarios, we wanted to examine if our protocol behaves as expected when a large group of nodes joins the system simultaneously, and how its performance in those conditions compares to the performance of the other broadcast protocols.

Both before and after the massive join event, all protocols show the same behaviour as in the stable scenario (presented in Figure 5.1): SYNC Tree presents a good balance between average broadcast latency and communication cost, when compared to the other broadcast protocols. Our solution is still able to, simultaneously, offer a lower latency and communication cost than the Periodic Synchronisation protocol with period of 200 milliseconds, while the Causal Flood protocol presents lower latency, but higher communication cost and number of duplicates.

Figures 5.8, 5.9, and 5.10 show the same metrics presented in Figures 5.5, 5.6, and 5.7, for a massive join scenario. These figures also omit some time at the beginning and end of the experiment, to focus only on the portion of the experience where the massive join event occurred. The grey vertical dashed line in the figures represents the time at which the join event started.

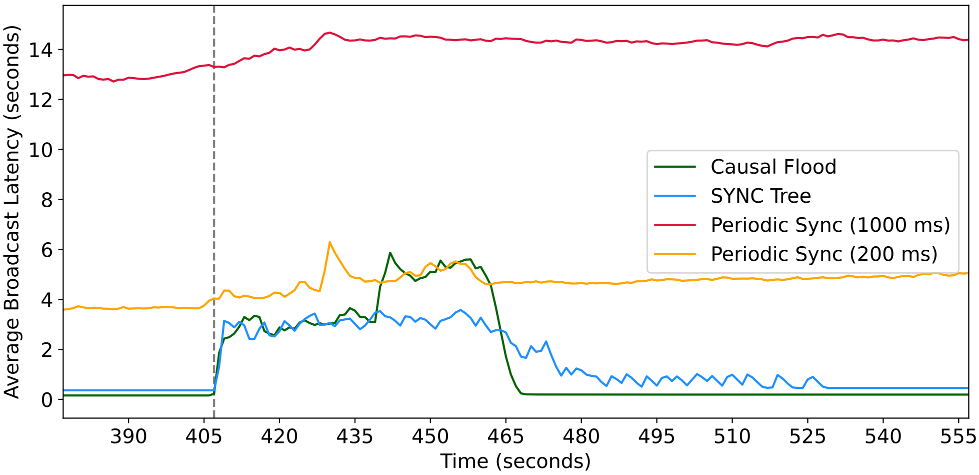


Figure 5.8: Massive Join: Average broadcast latency of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

In Figure 5.8, we can see that all protocols present increased latency when the massive join event occurs, but then settle on a lower latency, that is slightly higher than before the event because of the increase in the number of nodes in the system. The latency increase presented by the Periodic Synchronisation protocols is less noticeable, as these protocols already show high latency and are less affected by such an event. On the other hand, the Causal Flood and SYNC Tree protocols present steeper peaks, which we can attribute to the synchronisations with the new nodes, that require them to execute all old operations before executing the new ones. Our solution's peak is lower than that of the Causal Flood protocol, but takes longer to disappear because our tree topology needs time to stabilise.

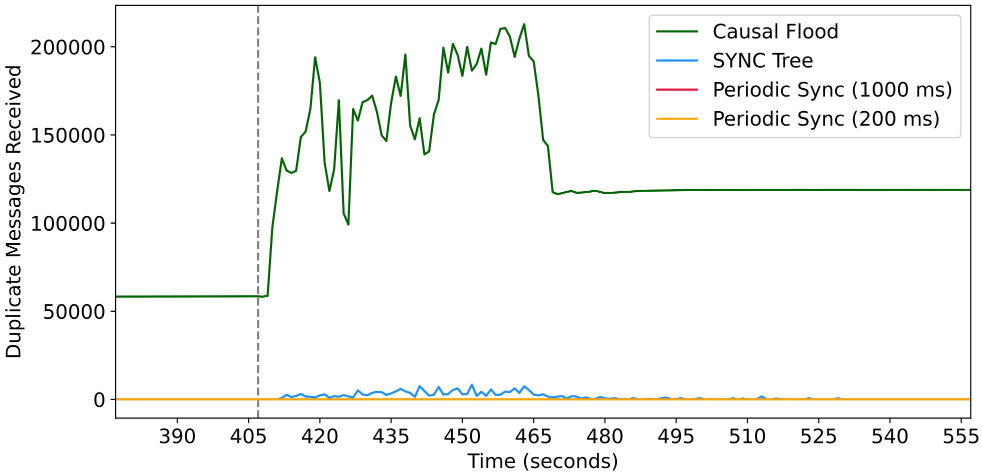


Figure 5.9: Massive Join: Number of duplicate messages of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

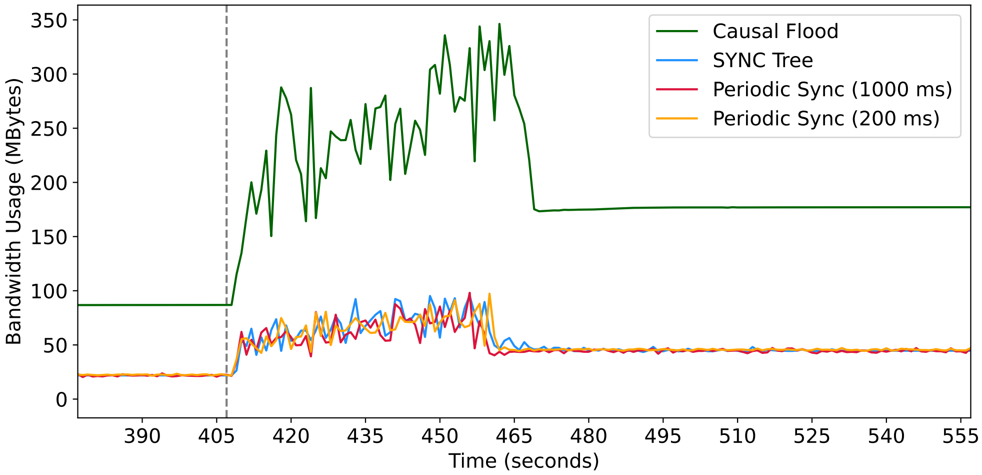


Figure 5.10: Massive Join: Communication cost of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

Furthermore, this instability causes the number of duplicate *Gossip* messages received to rise, as seen in Figure 5.9. This number is considerably higher in the Causal Flood protocol than in SYNC Tree, which is to be expected because, in the Causal Flood protocol nodes connect with each other, forming a cyclic graph, and send operations (including duplicate ones) to each other, while in SYNC Tree nodes join the tree topology and, if any duplicate messages are detected, the branch that forms a cycle is removed from the tree, thus ensuring lower numbers of duplicates.

In Figure 5.10, we can see that due to the additional *Synchronisation* messages (that carry all old operations of the system) exchanged with the new nodes joining the system (and the rise in duplicates for the Causal Flood and SYNC Tree protocol), all protocols present an increased bandwidth usage at the time of the massive join event, which then subsides after the repercussions of the event. However, the peak in bandwidth usage in much lower in SYNC Tree and the Periodic Synchronisation protocols, because they present few or no duplicate messages, whereas the Causal Flood protocol shows a sharp increase in bandwidth usage, due to the high number of duplicate messages received.

With these results, we can conclude that during massive join scenarios SYNC Tree adapts better than the alternative broadcast protocols. It presents an increase in average broadcast latency similar to that of the Causal Flood, but avoids its large amount of duplicates, thus maintaining a bandwidth usage similar to that of the Periodic Synchronisation protocols.

Catastrophic Failure Scenario

In the experiences with catastrophic failure scenarios, we wanted to examine if our protocol behaves as expected when a large group of nodes leaves the system simultaneously, and how its performance in those conditions compares to the performance of the other broadcast protocols.

Both before and after the catastrophic failure event, all protocols show the same behaviour as in the stable scenario (shown in Figure 5.1): SYNC Tree presents a balance between average broadcast latency and communication cost, when compared to the other broadcast protocols. Our solution is still able to, simultaneously, offer a lower latency and communication cost than the Periodic Synchronisation protocol with period of 200 milliseconds, while the Causal Flood protocol presents lower latency, but higher communication cost and number of duplicates.

Figures 5.11, 5.12, and 5.13 show the same metrics presented in Figures 5.5, 5.6, and 5.7, for a catastrophic failure scenario. Again, these figures omit some time at the beginning and end of the experiment, to focus only on the portion of the experience where the catastrophic failure event occurred. The grey vertical dashed line in the figures represents the time at which the failure event started.

By analysing these figures, it is interesting to see that SYNC Tree is the only one that, at the time of the catastrophic failure event, presents an increase in latency, number of duplicates, and communication cost, while the metrics for all the other protocols either remain stable or decrease.

The rise of the measured metrics is justified by the fact that our protocol is the only one that suffers reconfigurations to its topology when nodes leave the system. The catastrophic failure event causes one or more branches (or single nodes) of the tree to become isolated from the tree itself, causing the average broadcast latency to increase. New branches are then created that make the affected nodes recover from this isolation but, to do this, nodes must synchronise with their new tree neighbours, thus leading to spikes in the communication cost incurred by the protocol. Furthermore, when new branches are added to the tree, cycles may be introduced to the topology causing the number of duplicate *Gossip* messages received to rise. Eventually, these cycles are removed and the tree stabilises, which is when SYNC Tree stops presenting duplicates, as seen in Figure 5.12.

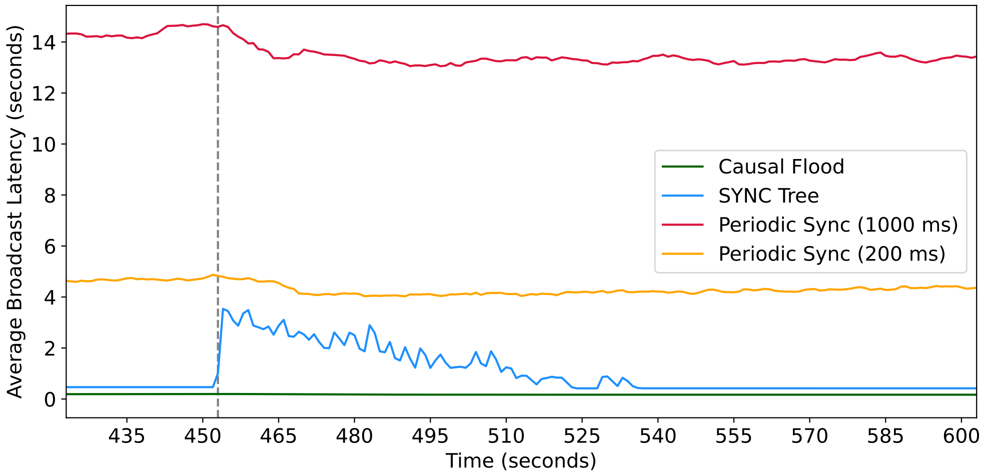


Figure 5.11: Catastrophic Failure: Average broadcast latency of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

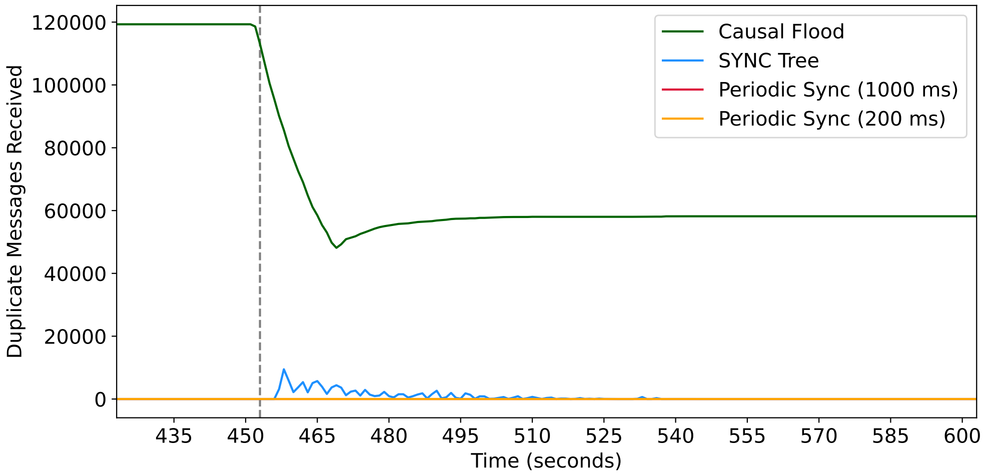


Figure 5.12: Catastrophic Failure: Number of duplicate messages of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

On the other hand, the Causal Flood and Periodic Synchronisation protocols remain unaffected by nodes leaving the system, because of the way in which the underlying overlay management protocol forms the network. As explained in Section 2.1.1, HyParView makes each node maintain two sets of views, a smaller active view that is the partial view for the broadcast protocols, and a larger passive view. HyParView uses the passive view to fill the active view, stopping it from ever becoming empty, thus ensuring that no node is ever isolated from the rest of the network, even in the case of a catastrophic failure.

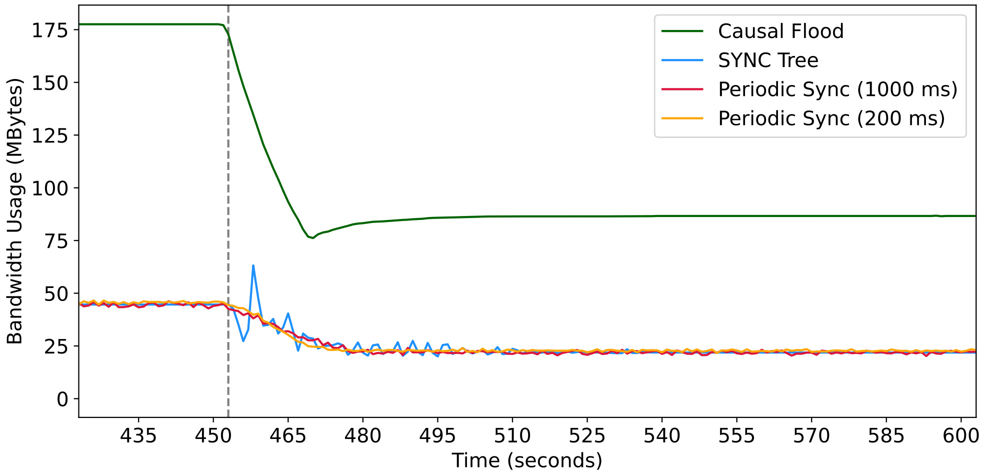


Figure 5.13: Catastrophic Failure: Communication cost of the 4 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

As can be seen in Figures 5.11 and 5.13, when the catastrophic failure occurs, the Periodic Synchronisation protocols are not negatively affected because each node always has neighbours. As such, all nodes are able continue performing synchronisations and their latency and communication cost simply lower in response to the fewer nodes that remain in the system.

In the Causal Flood protocol, because the active views of nodes may temporarily become less than full (but never empty), we can see a dip in the number of duplicates and in the communication cost, because each node sends messages to fewer neighbours. However, when the active view is filled, those metrics rise and stabilise at a lower value than the initial one, because there are fewer nodes in the system. Note that, when new neighbours are added to the active view of a given node in the Causal Flood protocol, that node still has to synchronise with them. But, because none of those neighbours was ever isolated from the network and the messages kept being delivered to all of them, those synchronisations will be very light, i.e., they will have very few operations and will not make the number of duplicates and the communication cost spike.

The figures also confirm that, despite presenting these temporary increases in latency, duplicates, and communication cost, SYNC Tree always presents lower latency than the Periodic Synchronisation protocol, even when configured with period of 200 milliseconds, and never uses more bandwidth or propagates more duplicate messages than the Causal Flood protocol.

5.3.4 Discussion

With this extensive experimental evaluation of the performance of SYNC Tree against other broadcast protocols in various scenarios, we were able to conclude that our solution seems to capture "the best of both worlds" when it comes to the trade-off between broadcast latency and communication cost in stable environments, and even if there are unexpected events that may cause large numbers of nodes to join or leave the system in a short period of time. SYNC Tree presents latency almost as low as that of the Causal Flood protocol, and communication cost similar to the Periodic Synchronisation protocol.

Furthermore, SYNC Tree is the broadcast protocol that performs the best in churn scenarios, being able to surpass the lower bound of latency that the Causal Flood algorithm imposes in other scenarios. Bearing in mind that edge networks have higher tendency for nodes to constantly join or leave the system, i.e., present higher churn rates, we believe our new causal broadcast protocol to be the most adapted to use at the edge.

5.4 ECO SYNC Tree Evaluation

In this section, we analyse the benefits brought to our solution by the state transfer and garbage collection mechanisms. In order to do this, we performed an experimental evaluation of ECO SYNC Tree against SYNC Tree, so we begin by explaining how both protocols were parameterised for our experiments and then analyse the results obtained.

5.4.1 Protocol Parameterisation

When it comes to the parameterisation of SYNC Tree and ECO SYNC Tree regarding the $treeMsgTimeout$, $announcementTimeout$, and $checkTreeMsgsTimeout$, the values remained the same as the ones presented in Section 5.3.2. The same applies for the index spacing and the membership management protocol.

However, ECO SYNC Tree required additional parameters for the garbage collection and state transfer mechanisms. We used a $garbageCollectionTTL$ of 60 seconds, a $garbageCollectionTimeout$ of 15 seconds, and a $saveStateTimeout$ of 30 seconds. This meant that, at every 30 seconds a new state was computed and stored, and at every 15 seconds the operations that had been delivered more that 60 seconds ago were erased from disk.

5.4.2 Results and Analysis

In this section, we present the results obtained by comparing ECO SYNC Tree, presented in Section 3.4, with SYNC Tree. To assess the benefits of the new mechanisms introduced, we measured not only the metrics described in Section 5.3.3, but also the following ones:

- **Time Spent Synchronising:** The sum of the time spent synchronising by all nodes of the system, e.g., assuming that a node A is synchronising with another node B, we measured the time between node A sending a *SendVectorClock* message to B, and the time at which node B finishes executing the received missing operations.
- **Average Synchronisation Duration Over Time:** The average time (in seconds) it takes for a single synchronisation to finish, during the time of the experiences.
- **Disk Usage Over Time:** The sum of the disk space occupied by the causally-ordered list of operations of every node, during the time of the experiences.
- **Tree Stabilisation Time:** The time it took for the tree topology to stabilise after a massive join event, or a catastrophic failure event.

In the following sections, we only present the most relevant results and, in Annex II, we present some of the additional results extracted during our experimental work.

Benefits of the State Transfer Mechanism

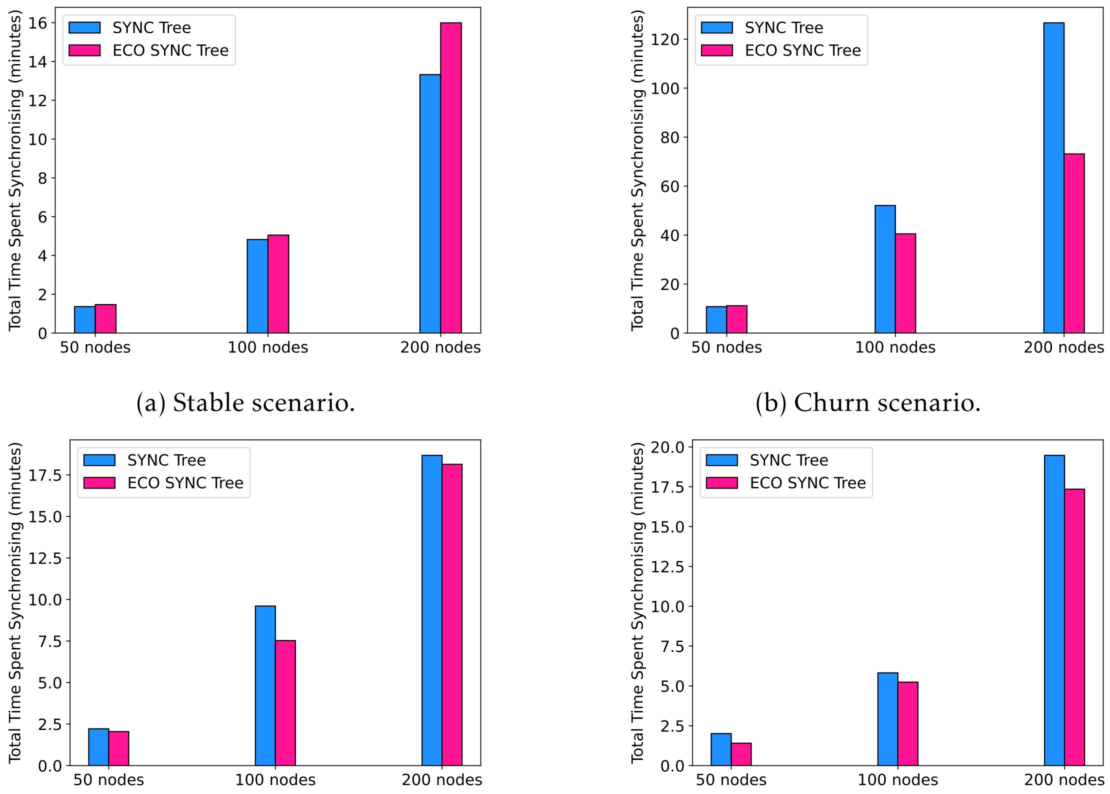
To assess the benefits of the state transfer mechanism, we mostly focused on churn scenarios, where new nodes join the network very frequently, thus requiring the protocols to perform several synchronisations that employ said mechanism.

Figures 5.14a, 5.14b, 5.14c, and 5.14d show the total time the two protocols spent synchronising when we fix p to 1, the payload size to 1024KB, and vary the number of nodes in, respectively, the stable, churn, massive join, and catastrophic failure scenarios.

In Figure 5.14, we can see that the time both protocols spent synchronising increases with the number of nodes in the system. This is not surprising, since the more nodes there are, the more synchronisations need to be performed.

Furthermore, this figure shows that ECO SYNC Tree is not able to reduce the time spent synchronising in the stable scenario. This happens because all synchronisations are performed before nodes start sending operations and, in this specific case, the state transfer mechanism is not able to improve the synchronisation process as the synchronisations themselves transfer no state and no operations. In fact, when no nodes have performed operations, the state transfer mechanism slightly increases the time it takes for a synchronisation to finish because it introduces additional logic to the algorithm.

In all other scenarios, our state transfer mechanism significantly decreases the time nodes spend synchronising. We expected these results, because if new nodes join the network when operations are already being disseminated, sending them a state plus a subset of all past operations of the system will always be lighter than sending the whole set of past operations. As we can see, the churn scenario is the one that benefits the most from the state transfer mechanism because it is the one where the most synchronisations occur. With this said, the next results focus on that scenario.



(a) Stable scenario.

(b) Churn scenario.

(c) Massive join scenario.

(d) Catastrophic failure scenario.

Figure 5.14: Total time spent synchronising with $p = 1$, payload of 1024KB, and varying number of nodes.

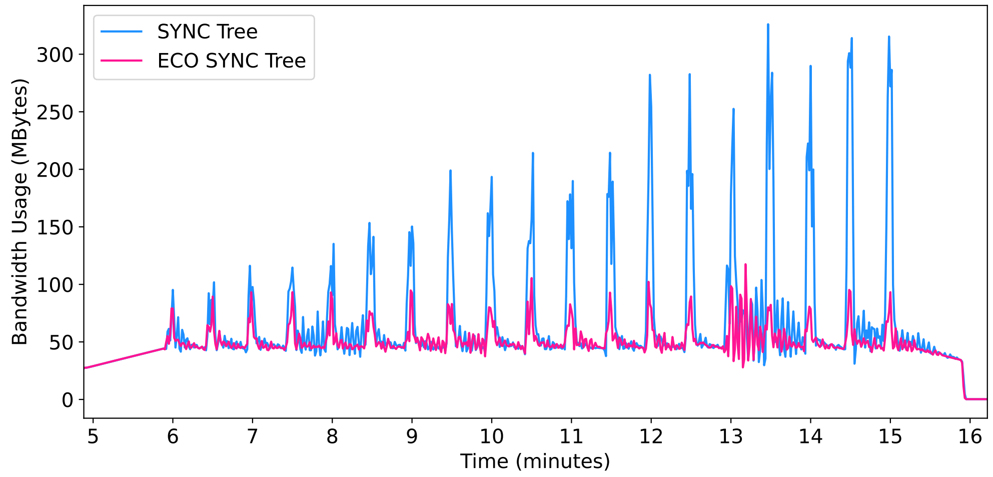


Figure 5.15: Churn: Communication cost of the 2 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

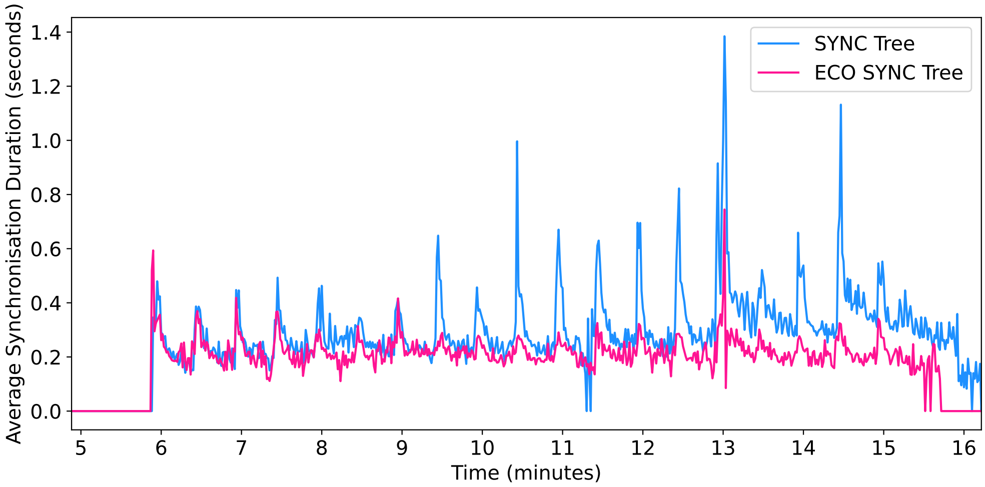


Figure 5.16: Churn: Average synchronisation duration of the 2 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

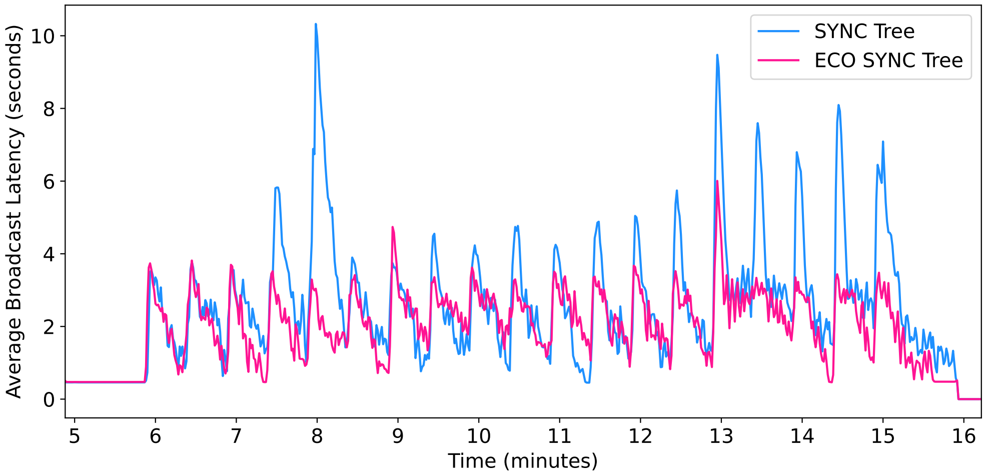


Figure 5.17: Churn: Average broadcast latency of the 2 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

Figures 5.15, 5.16, and 5.17 show, respectively, how the communication cost, the average synchronisation duration, and the average broadcast latency varied over time, in the churn scenario. As in Section 5.3.3, we took measures with a granularity of 1 second. To compute the average synchronisation duration, each measure represents the average time it took for the synchronisations that started in that second to finish. Again, these figures omit some time at the beginning and end of the experiment, to focus only on the portion of the experience where the churn events occurred.

In Figure 5.15, we can see that the bandwidth usage spikes at each churn step for both protocols. This is expected, because large *Synchronisation* messages that contain past system information are being propagated through the network. However, it is very clear that ECO SYNC Tree greatly reduces the bandwidth usage when compared to SYNC

Tree (which tends to show peaks that grow over time). As there are significantly less bytes being propagated between nodes, synchronisations also take a much shorter time to finish, thus decreasing the average synchronisation duration in churn steps, as seen in Figure 5.16.

Note as well that, as time moves forward in the experience and more operations are disseminated to the nodes of the system, the discrepancy in synchronisation duration between the protocols becomes more evident. This happens because SYNC Tree must propagate ever increasing lists of missing operations, while in ECO SYNC Tree the size of the transferred information remains relatively stable, as it consists of the last computed state plus the operations received in the last $garbageCollectionTTL$ seconds.

Furthermore, because synchronisations finish faster, new nodes that join the system are able to more quickly execute new operations that are being propagated and, because of this, the average broadcast latency of ECO SYNC Tree is lower than that of SYNC Tree, as evidenced by Figure 5.17.

The presented results prove that the introduced state transfer mechanism does effectively improve the performance of our solution. By decreasing the amount of information propagated in *Synchronisation* messages, synchronisation processes are able to complete much faster, thus decreasing both the average broadcast latency and bandwidth usage of ECO SYNC Tree when compared to SYNC Tree.

Benefits of the Garbage Collection Mechanism

To measure the benefits of the garbage collection mechanism, we present only the results of the stable scenario, because the results in all other scenarios were the expected ones.

Figure 5.18 shows how the disk usage varied over time, in the stable scenario.

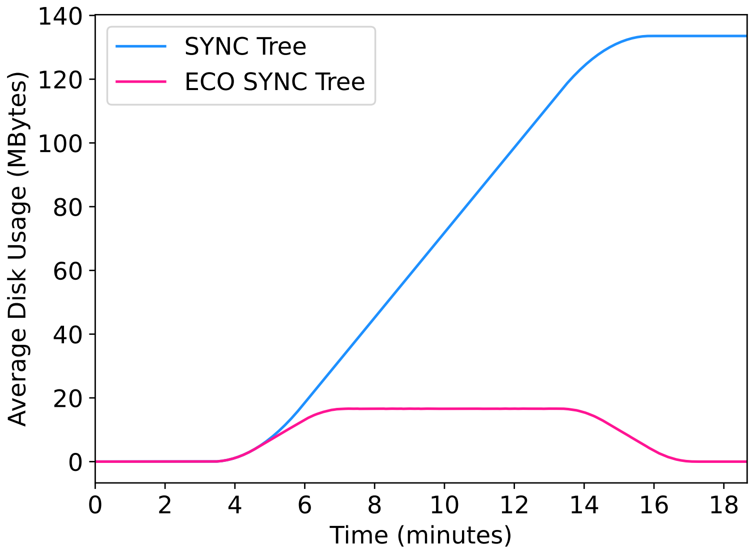


Figure 5.18: Stable: Disk usage of the 2 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

As we anticipated, Figure 5.18 proves that our garbage collection mechanism is able to drastically reduce the amount of disk space required by our solution. As we can see, while the disk usage of SYNC Tree slowly rises over time, ECO SYNC Tree is able to maintain its disk usage within a certain range. Of course, this range will vary according to the parameterisation of the protocol, but this mechanism ensures that, as our system remains on-line, the disk space needed for it to work will not tend to infinity.

Tree Stabilisation Time

Lastly, we evaluated how ECO SYNC Tree improves the time it takes for our tree topology to stabilise after events such as massive joins or catastrophic failures of large groups of nodes, when compared to SYNC Tree.

Table 5.1 shows the tree stabilisation time for massive joins of 15, 30, and 60 nodes, in experiments where the number of final nodes in the system is, respectively, 50, 100, and 200 nodes.

Table 5.2 shows the tree stabilisation time for catastrophic failures of 15, 30, and 60 nodes, in experiments where the number of initial nodes in the system is, respectively, 50, 100, and 200 nodes.

The results reported on Table 5.1 show that, in all cases, ECO SYNC Tree is able to reduce the time it takes for the tree to stabilise after a massive join event. It is also interesting to observe that, as the number of final nodes in the system increases, so does the difference in stabilisation time between both protocols. This happens because, as we have previously shown in Figure 5.16, ECO SYNC Tree is able to decrease the time it takes for synchronisations to finish and, because several synchronisations need to be executed in order for the tree to stabilise, is able to more efficiently make the topology converge into a tree.

	15 nodes	**30 nodes**	**60 nodes**
SYNC Tree	42	96	125
ECO SYNC Tree	41	67	88

Table 5.1: Massive Join: Tree stabilisation time in seconds.

	15 nodes	**30 nodes**	**60 nodes**
SYNC Tree	27	46	88
ECO SYNC Tree	13	40	64

Table 5.2: Catastrophic Failure: Tree stabilisation time in seconds.

In the results reported on Table 5.2 we can see that ECO SYNC Tree also presented lower tree stabilisation times than SYNC Tree for scenarios with catastrophic failures. However, the difference is not as noticeable as in the massive join scenario. This happens because when nodes leave the system they may not always cause the tree to reconfigure, e.g., when a leaf node dies, the tree topology will remain unaltered. When catastrophic failures occur, less synchronisations tend to be needed when compared to when massive joins occur and, as such, the improvement in the tree stabilisation time is not as significant as when massive joins occur.

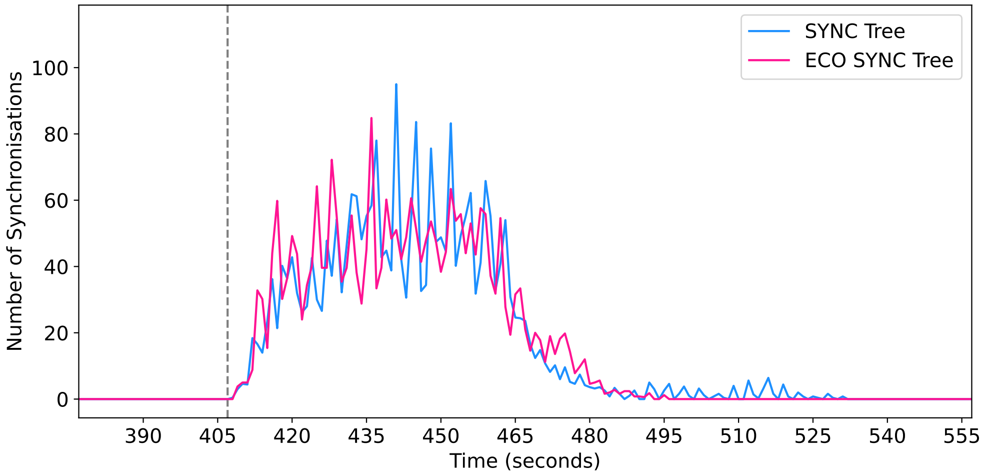


Figure 5.19: Massive Join: Number of synchronisations of the 2 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.

In Figure 5.19 we can see a more detailed visual representation of the tree stabilisation time. This figure shows how the number of synchronisations varied over time, in the massive join scenario with 200 final nodes, which corresponds to the 60 nodes column of Table 5.1. In the figure, we can see that in ECO SYNC Tree the tree topology converges (i.e., the number of synchronisations becomes zero) quicker than in SYNC Tree. This is due to the faster synchronisations of ECO SYNC Tree , which allow the protocol to form branches and detect and remove any introduced cycles faster.

5.4.3 Discussion

The evaluation reported in this section shows that the new state transfer and garbage collection mechanisms added to ECO SYNC Tree have greatly improved its performance, and that our solution is able to quickly and efficiently perform lighter synchronisations whose communication cost remains stable as time passes. This enables the tree topology to reconfigure itself quicker in the face of changes to the membership of the system. Lastly, ECO SYNC Tree reduces the disk space needed to store the causally-ordered list of operations, being more suited for use in resource constrained edge nodes.

Summary

In this chapter, we presented the experimental evaluation of the causal broadcast algorithm developed in the course of this thesis, SYNC Tree, and of its evolution that incorporates a state transfer and a garbage collection mechanism, ECO SYNC Tree.

In the next chapter, we conclude this thesis and present some features that could be added to our solution in the future.

6

Conclusion and Future Work

Conclusion

As current trends start to move storage and computation to the edge in order to provide support for latency constrained applications, new edge storage systems must emerge that optimise latency and reduce the cost of communication. The causal consistency model allows these edge-based storage systems to combine the availability and low latency of weak consistency models, while still providing some consistency guarantees that allow developers to more easily reason about the state of the system, and offering sensible semantics for applications.

With this in mind, in this thesis we studied the challenges faced when attempting to create a protocol capable of offering causal consistency guarantees in resource constrained geo-replicated edge settings. By exploiting tree topologies, which have previously been used to offer causal consistency with small metadata overhead, and combining them with automatic reconfiguration features, based on those that the Plumtree broadcast protocol uses to deal with membership changes, we envisioned our solution. As a result, we presented a new causal broadcast algorithm suited for use at the edge. ECO SYNC Tree makes use of a dynamic tree topology, capable of quickly adapting to nodes joining and leaving the system, to offer causal delivery while using negligible metadata to encode causal dependencies.

In addition to the design of ECO SYNC Tree, this work also presents some details regarding the implementation of its prototype. The experimental evaluation conducted over this prototype allowed us to conclude that our solution captures "the best of both worlds" when it comes to the trade-off between broadcast latency and communication cost in stable environments, and in environments subject to events such as large groups of nodes joining or leaving the system. In these scenarios, our protocol has average broadcast latency almost as low as the Causal Flood protocol, and communication cost similar to that of the Periodic Synchronisation protocol. Furthermore, out of all the broadcast protocols evaluated, ECO SYNC Tree has proven to be the one most suitable for edge-based deployment by presenting the lowest average broadcast latency and communication cost in scenarios with high churn rates, thus enabling it to scale to hundreds of nodes. Finally, the state transfer and garbage collection mechanisms our solution uses greatly improve its performance, allowing the tree topology to reconfigure itself quicker in the face of changes to the membership of the system, and reducing the disk space needed for successful long-term deployment.

Lastly, by combining ECO SYNC Tree with a library of CRDTs, this work offers support for any edge-based storage systems that may want to offer causal+ consistency guarantees with configurable conflict resolution policies adapted to the needs of different applications.

Future Work

Here, we present some possible future work directions that can be pursued to improve on the solution presented in this thesis.

Partial replication: Edge-based geo-replicated storage systems often aim at exploiting data locality by storing data items only in the places where they are accessed and modified. This requires them to support partial replication, i.e., making replicas store only a subset of all data items, in this case, only the data items that are relevant to the applications that directly access that edge node. This is an essential feature in edge computing because edge nodes tend to have less resources, and thus less storage space, and because users in a given geographical area are more likely to access the same data items repeatedly. As such, adding support for partial replication in our edge-enabled causal broadcast algorithm seems the only logical next step. To this end, our protocol could create several broadcast trees, one for each data partition. Then, an additional mechanism to solve dependencies between trees would have to be devised.

Partition-tolerant state transfer: As we mentioned Section 3.4.4, our solution is not able to withstand network partitions whose duration exceeds the maximum broadcast latency of the system. Because it is impossible to determine the maximum amount of time a network partition will take to heal, our solution assumes that any distributed storage system that uses our causal broadcast algorithm will configure the TTL of old operations to a value greater than the maximum time they expect any network partition to exist. A possible way to overcome this limitation is to extend our protocol to support mergeable state transfers, in which any two nodes (even if they were previously partitioned) exchange states and are able to compute a common final state through a merge function, thus ensuring that our solution is tolerant to all partitions, regardless of the time they take to heal.

Bibliography

[1] *7 Data Center Disasters You'll Never See Coming*. https://www.informationweek.com/cloud/7-data-center-disasters-youll-never-see-coming/d/d-id/1320702. Accessed: Feb. 2021.

[2] D. D. Akkoorath et al. "Cure: Strong Semantics Meets High Availability and Low Latency". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 2016, pp. 405–414. DOI: 10.1109/ICDCS.2016.98.

[3] D. D. Akkoorath and A. Bieniusa. "Antidote: the highly-available geo-replicated database with strongest guarantees". In: *SyncFree Technology White Paper* (2016).

[4] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient State-based CRDTs by Delta-Mutation". In: *CoRR* abs/1410.2803 (2014). arXiv: 1410.2803. URL: http://arxiv.org/abs/1410.2803.

[5] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction". In: (2013), p. 85. DOI: 10.1145/2465351.2465361.

[6] C. Baquero, P. Almeida, and A. Shoker. "Making Operation-Based CRDTs Operation-Based". In: *Proceedings of the 1st Workshop on the Principles and Practice of Eventual Consistency, PaPEC 2014* (Apr. 2014). DOI: 10.1145/2596631.2596632.

[7] C. Baquero and N. Preguiça. "Why Logical Clocks Are Easy". In: *Communications of the ACM* 59.4 (Apr. 2016), pp. 43–47. ISSN: 0001-0782. DOI: 10.1145/2890782.

[8] J. Bauwens and E. Gonzalez Boix. "Memory Efficient CRDTs in Dynamic Environments". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 48–57. ISBN: 9781450369879. DOI: 10.1145/3358504.3361231. URL: https://doi.org/10.1145/3358504.3361231.

[9] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A distributed metadata service for causal consistency". In: *Proceedings of the 12th European Conference on Computer Systems, EuroSys 2017* (2017), pp. 111–126. DOI: 10.1145/3064176.3064210.

[10] E. Brewer. "CAP twelve years later: How the "rules"have changed". In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[11] G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: https://doi.org/10.1145/1323293.1294281.

[12] J. Du et al. "GentleRain: Cheap and scalable causal consistency with physical clocks". In: *Proceedings of the 5th ACM Symposium on Cloud Computing, SOCC 2014* (2014). DOI: 10.1145/2670979.2670983.

[13] V. Enes et al. "Efficient synchronization of state-based CRDTs". English. In: *Proceedings - 2019 IEEE 35th International Conference on Data Engineering, ICDE 2019*. Proceedings - International Conference on Data Engineering. IEEE Computer Society, Apr. 2019, pp. 148–159. DOI: 10.1109/ICDE.2019.00022.

[14] C. J. Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering". In: *Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88)*. 1988, pp. 56–66.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121.

[16] 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)*. 2018, pp. 1–10. DOI: 10.1109/NCA.2018.8548067.

[17] 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. ISSN: 0163-5700. DOI: 10.1145/564585.564601.

[18] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. "Trade-offs in replicated systems". In: *IEEE Data Engineering Bulletin* 39.ARTICLE (2016), pp. 14–26.

[19] P. Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *USENIX Annual Technical Conference*. 2010.

[20] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[21] L. Lamport. "The Part-Time Parliament". In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. ISSN: 07342071. DOI: 10.1145/279227.279229.

[22] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic Broadcast Trees". In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. 2007, pp. 301–310. DOI: 10.1109/SRDS.2007.27.

[23] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: July 2007, pp. 419–429. ISBN: 0-7695-2855-4. DOI: 10.1109/DSN.2007.56.

[24] J. Leitão, J. Pereira, and L. Rodrigues. "Gossip-Based Broadcast". In: Oct. 2010, pp. 831–860. DOI: 10.1007/978-0-387-09751-0_29.

[25] J. Leitão et al. "Towards Enabling Novel Edge-Enabled Applications". In: *CoRR* abs/1805.06989 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989.

[26] A. Linde, J. Leitão, and N. Preguiça. "Δ-CRDTs: making δ-CRDTs delta-based". In: Apr. 2016, pp. 1–4. DOI: 10.1145/2911151.2911163.

[27] A. van der Linde et al. "Legion: Enriching Internet Services with Peer-to-Peer Interactions". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 283–292. ISBN: 9781450349130. DOI: 10.1145/3038912.3052673. URL: https://doi.org/10.1145/3038912.3052673.

[28] A. van der Linde et al. "The intrinsic cost of causal consistency". In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2020* (2020). DOI: 10.1145/3380787.3393674.

[29] W. Lloyd et al. "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*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 401–416. ISBN: 9781450309776. DOI: 10.1145/2043556.2043593. URL: https://doi.org/10.1145/2043556.2043593.

[30] W. Lloyd et al. "Stronger Semantics for Low-Latency Geo-Replicated Storage". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 313–328. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd.

[31] P. Maymounkov and D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems*. Ed. by P. Druschel, F. Kaashoek, and A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-45748-0.

[32] D. Milojicic et al. "Peer-to-Peer Computing". In: (Apr. 2002).

[33] N. M. Preguiça, C. Baquero, and M. Shapiro. "Conflict-free Replicated Data Types (CRDTs)". In: *CoRR* abs/1805.06358 (2018). arXiv: 1805.06358. URL: http://arxiv.org/abs/1805.06358.

[34] A. Rijo. "Building Tunable CRDTs". MA thesis. https://run.unl.pt/handle/10362/55171: FCT NOVA, Nov. 2018.

[35] A. Rowstron and P. Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Middleware 2001*. Ed. by R. Guerraoui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 329–350. ISBN: 978-3-540-45518-9.

[36] M. Satyanarayanan. “The Emergence of Edge Computing”. In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[37] M. Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[38] M. Shapiro et al. *Conflict-free Replicated Data Types*. Research Report RR-7687. INRIA, July 2011, p. 18. URL: https://hal.inria.fr/inria-00609399.

[39] I. Stoica et al. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’01. San Diego, California, USA: Association for Computing Machinery, 2001, pp. 149–160. ISBN: 1581134118. DOI: 10.1145/383059.383071. URL: https://doi.org/10.1145/383059.383071.

[40] E. Vieira et al. “Difusão Causal Flexível e Escalável para Replicação na Periferia”. In: *Proceedings of the 12th Simpósio de Informática (INForum 2021), Lisbon, Portugal, September 2021*.

[41] S. Voulgaris, D. Gavidia, and M. van Steen. “CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays”. In: *J. Network Syst. Manage.* 13 (June 2005), pp. 197–217. DOI: 10.1007/s10922-005-4441-x.

[42] M. Zawirski et al. “Write fast, read in the past: Causal consistency for client-side applications”. In: *Middleware 2015 - Proceedings of the 16th Annual Middleware Conference* Dc (2015), pp. 75–87. DOI: 10.1145/2814576.2814733.

[43] P. Zeller, A. Bieniusa, and A. Poetzsch-Heffter. “Formal Specification and Verification of CRDTs”. In: June 2014, pp. 33–48. ISBN: 978-3-662-43612-7. DOI: 10.1007/978-3-662-43613-4_3.

I

SYNC Tree: Additional Results

Figures I.1, I.2, I.3, I.4, I.5, and I.6 complement the results presented in Figures 5.5, 5.6, and 5.7 from Section 5.3.3, when varying the number of nodes in the churn scenario.

Figure I.7 shows the average broadcast latency (in seconds), the number of duplicate *Gossip* messages, and the communication cost (in GB) of all four broadcast protocols when we fix p to 1, the payload size to 1024KB, and vary the number of nodes in a massive join scenario. Furthermore, Figures I.8, I.9, I.10, I.11, I.12, and I.13 complement the results presented in Figures 5.8, 5.9, and 5.10 from Section 5.3.3, when varying the number of nodes in the massive join scenario.

Figure I.14 shows the average broadcast latency (in seconds), the number of duplicate *Gossip* messages, and the communication cost (in GB) of all four broadcast protocols when we fix p to 1, the payload size to 1024KB, and vary the number of nodes in a catastrophic failure scenario. Furthermore, Figures I.15, I.16, I.17, I.18, I.19, and I.20 complement the results presented in Figures 5.11, 5.12, and 5.13 from Section 5.3.3, when varying the number of nodes in the catastrophic failure scenario.

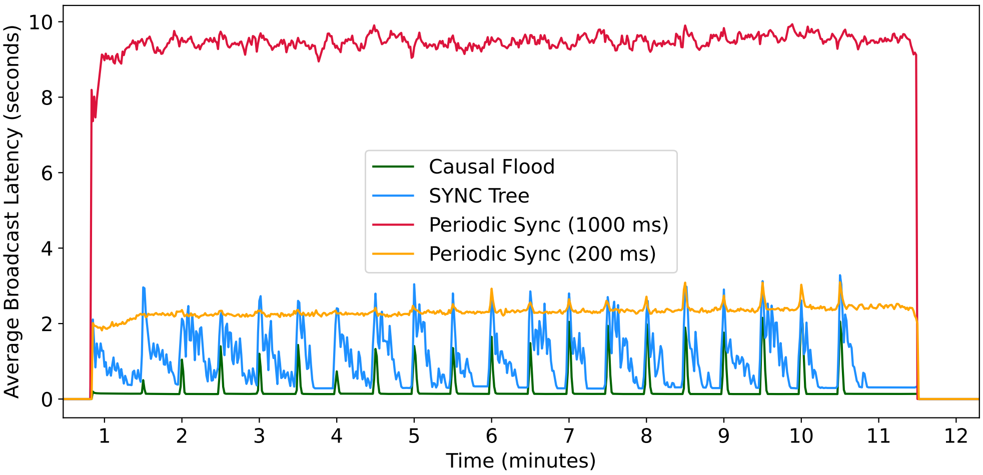


Figure I.1: Churn: Average broadcast latency of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

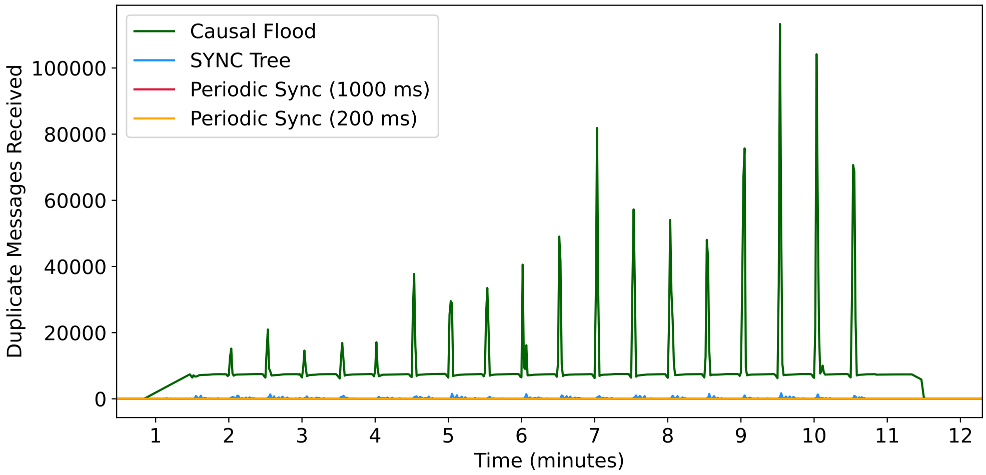


Figure I.2: Churn: Number of duplicate messages of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

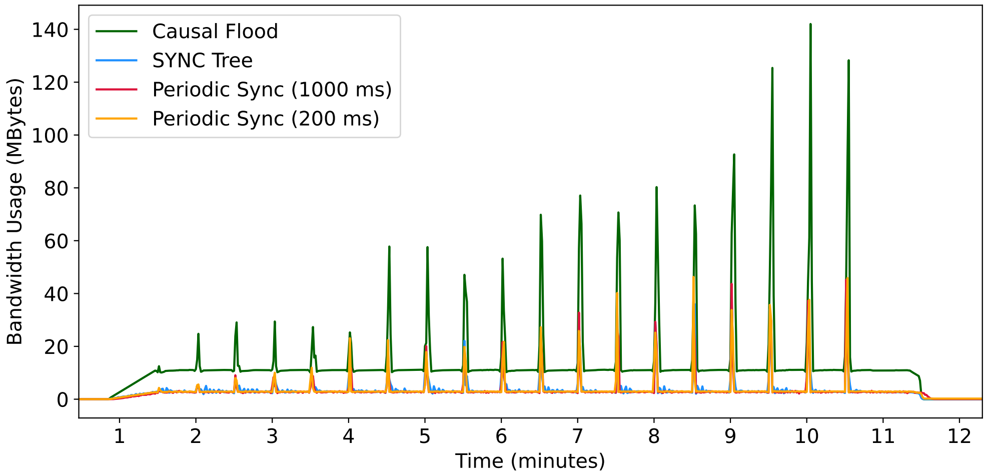


Figure I.3: Churn: Communication cost of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

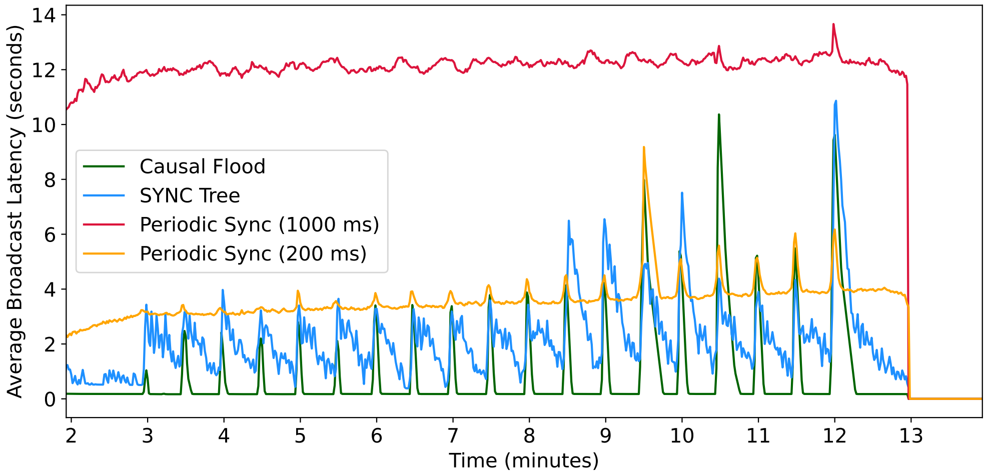


Figure I.4: Churn: Average broadcast latency of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

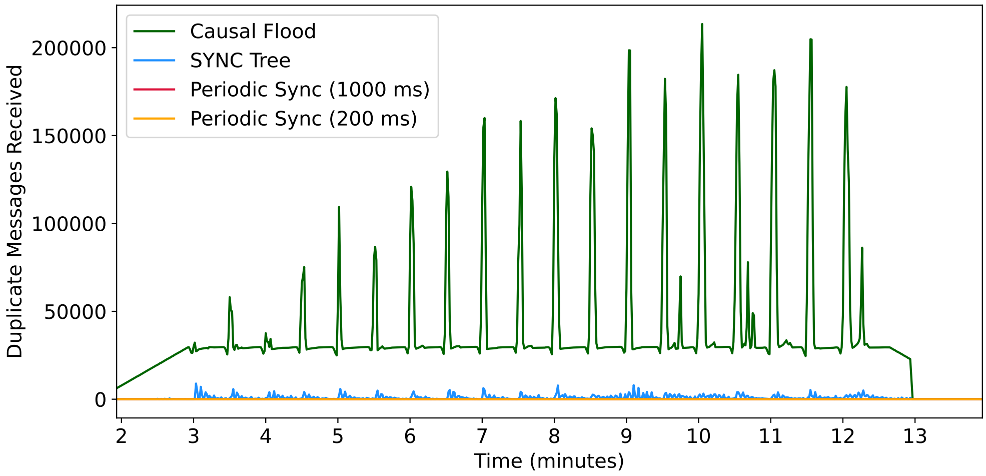


Figure I.5: Churn: Number of duplicate messages of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

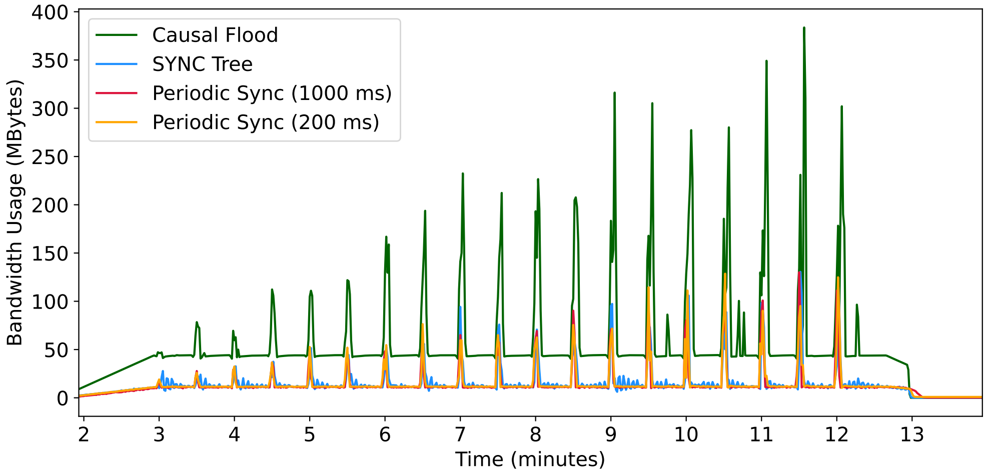
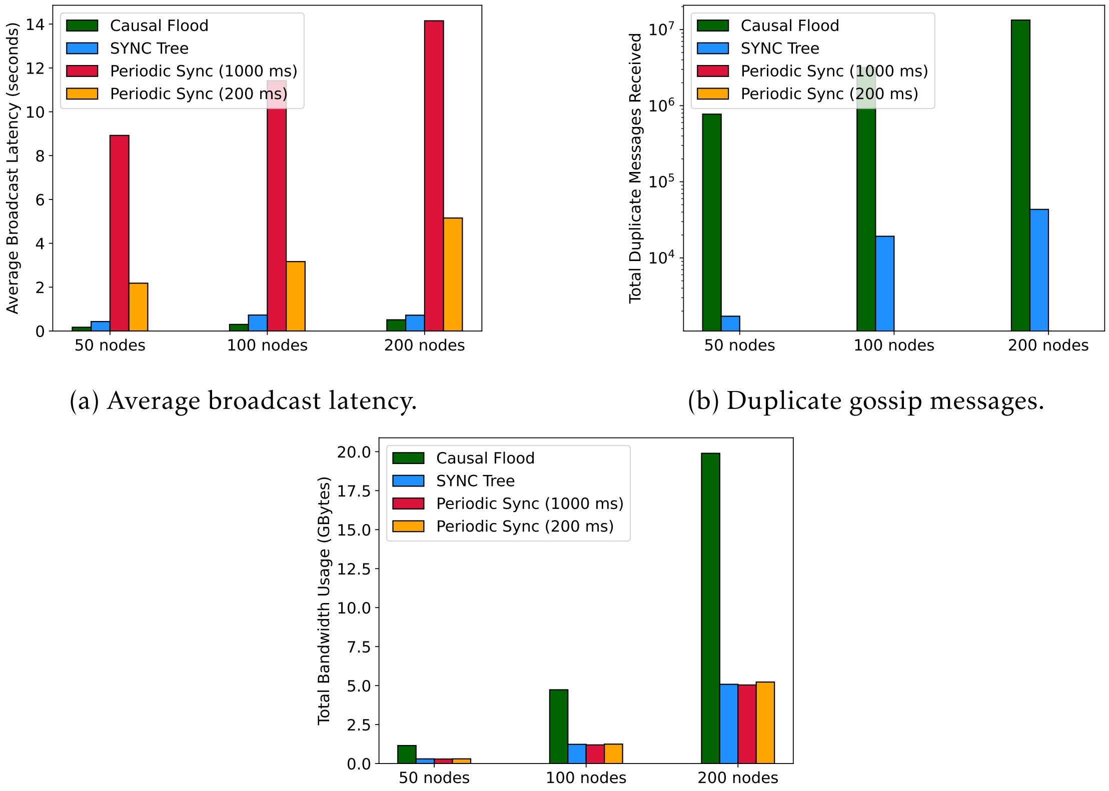


Figure I.6: Churn: Communication cost of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.



(a) Average broadcast latency.

(b) Duplicate gossip messages.

(c) Communication cost.

Figure I.7: Metrics in massive join scenario with $p = 1$, payload of 1024KB, and varying number of nodes.

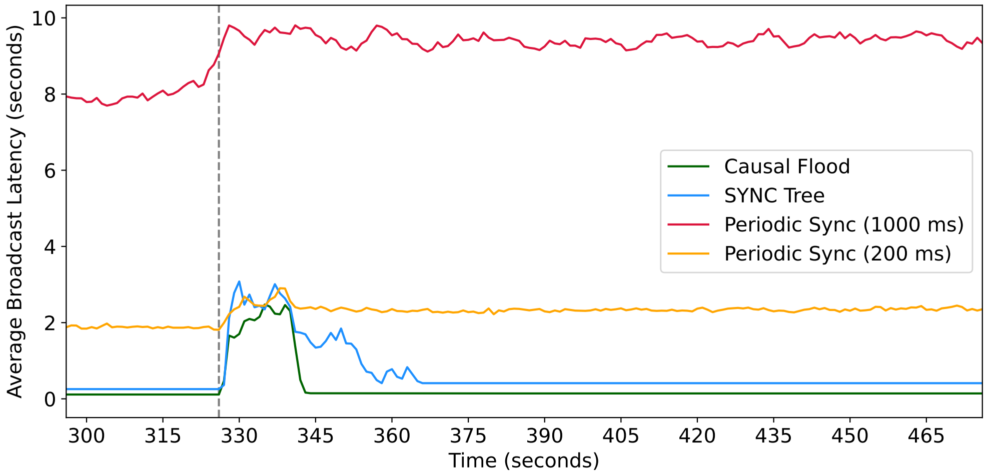


Figure I.8: Massive Join: Average broadcast latency of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

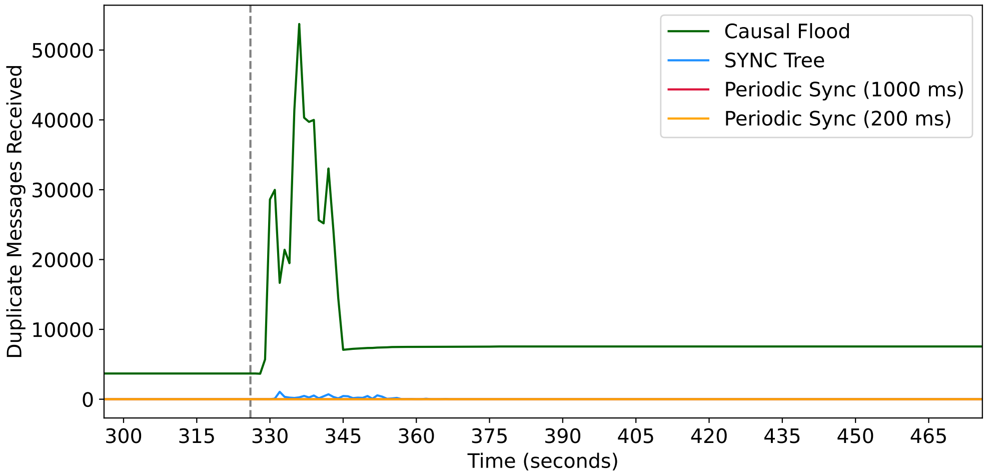


Figure I.9: Massive Join: Number of duplicate messages of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

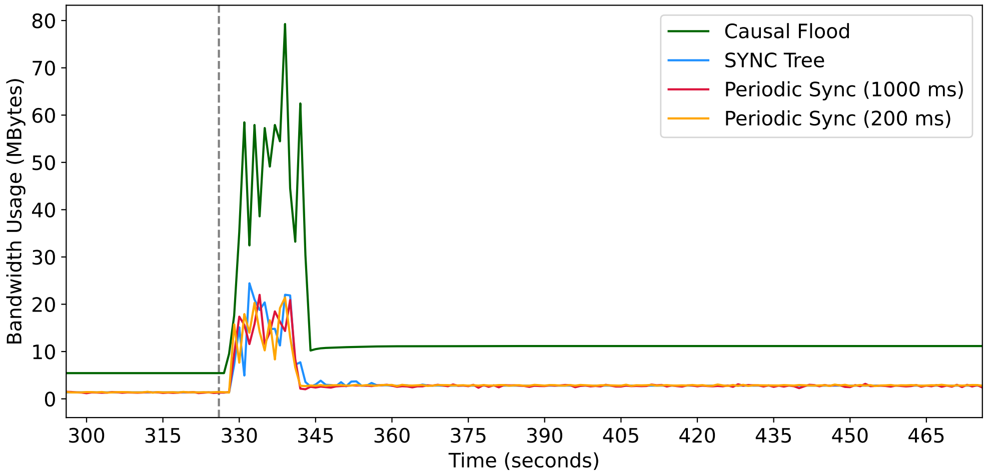


Figure I.10: Massive Join: Communication cost of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

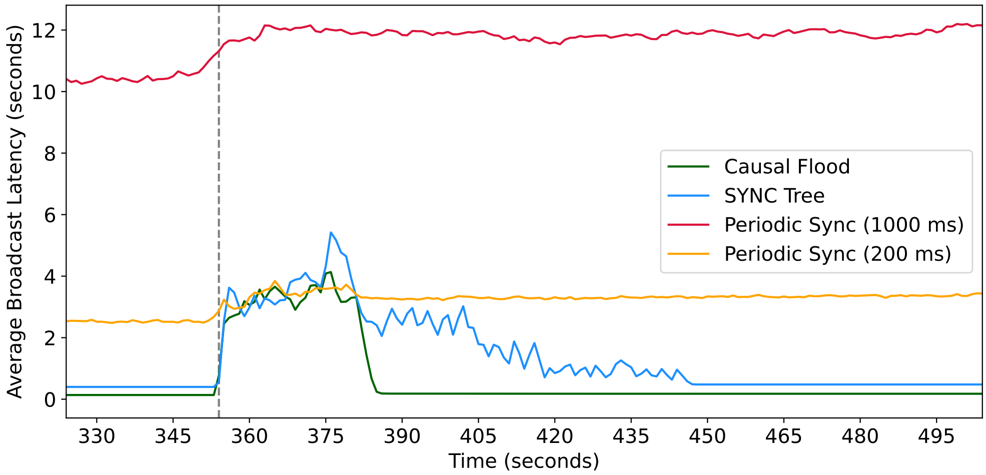


Figure I.11: Massive Join: Average broadcast latency of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

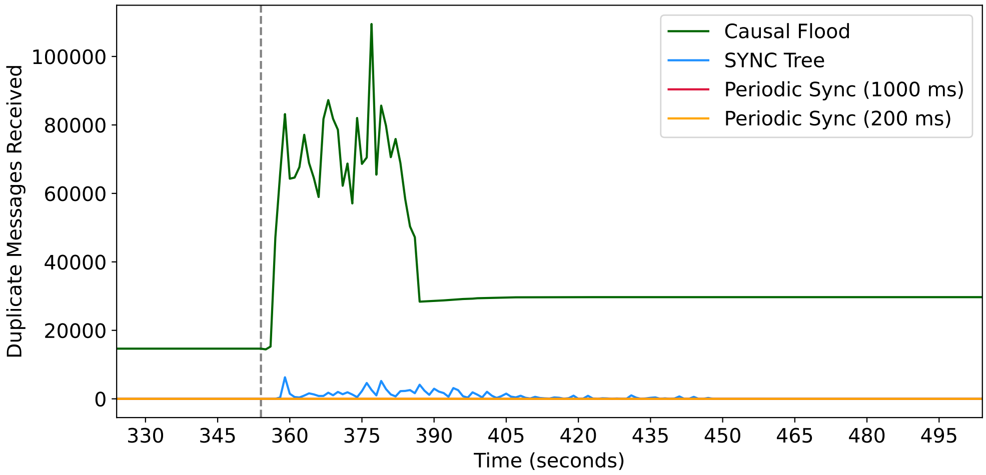


Figure I.12: Massive Join: Number of duplicate messages of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

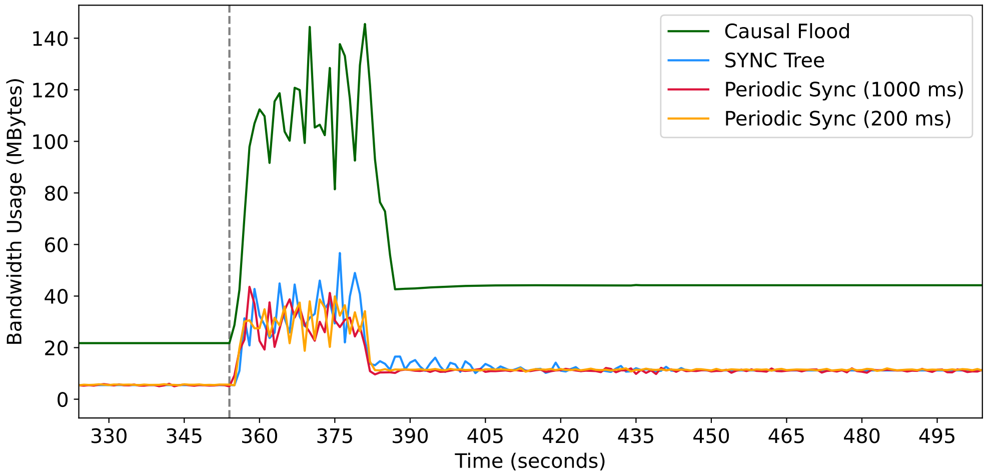
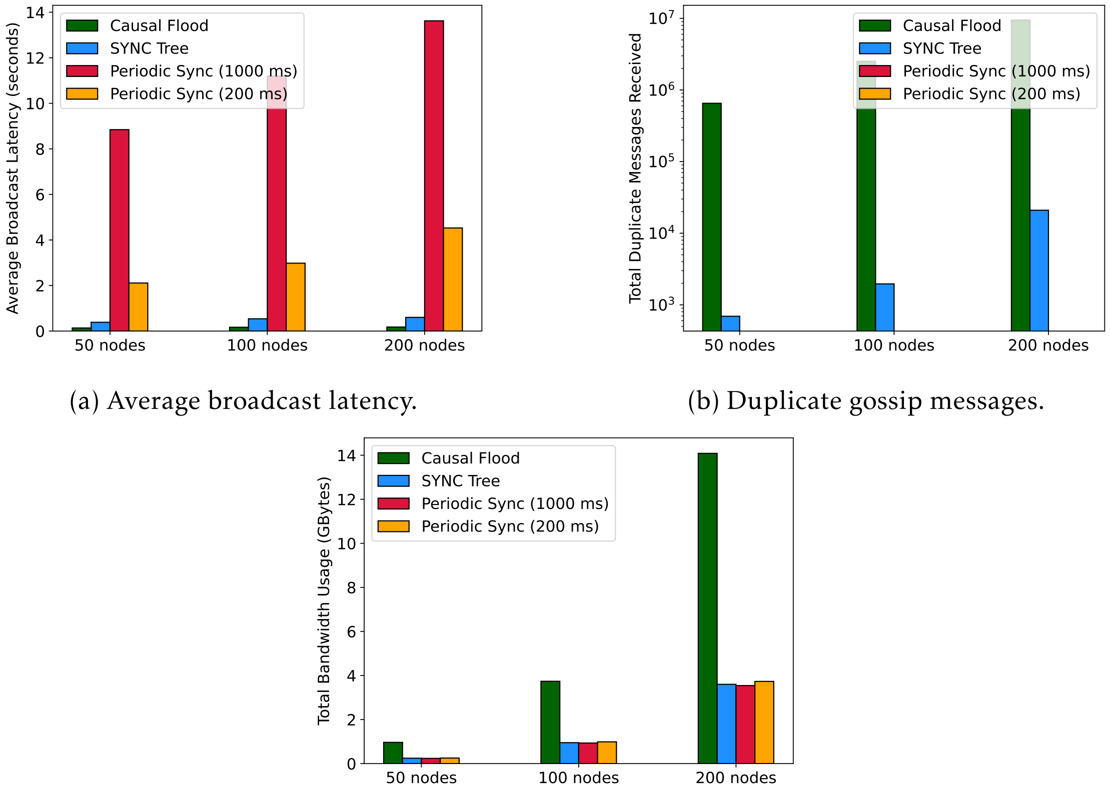


Figure I.13: Massive Join: Communication cost of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.



(a) Average broadcast latency.

(b) Duplicate gossip messages.

(c) Communication cost.

Figure I.14: Metrics in catastrophic failure scenario with $p = 1$, payload of 1024KB, and varying number of nodes.

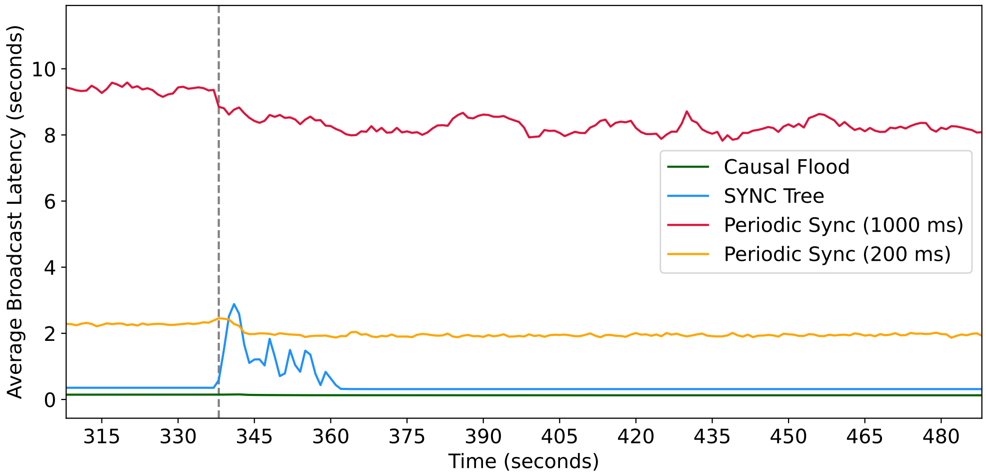


Figure I.15: Catastrophic Failure: Average broadcast latency of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

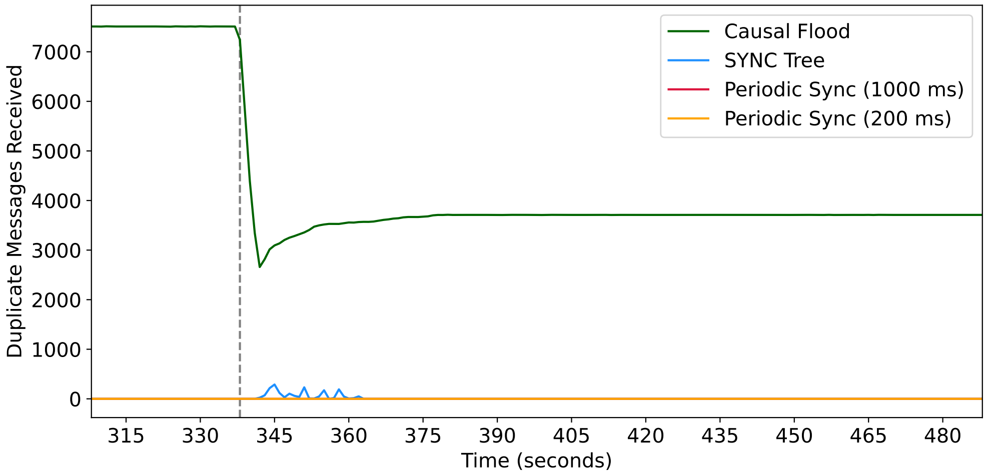


Figure I.16: Catastrophic Failure: Number of duplicate messages of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

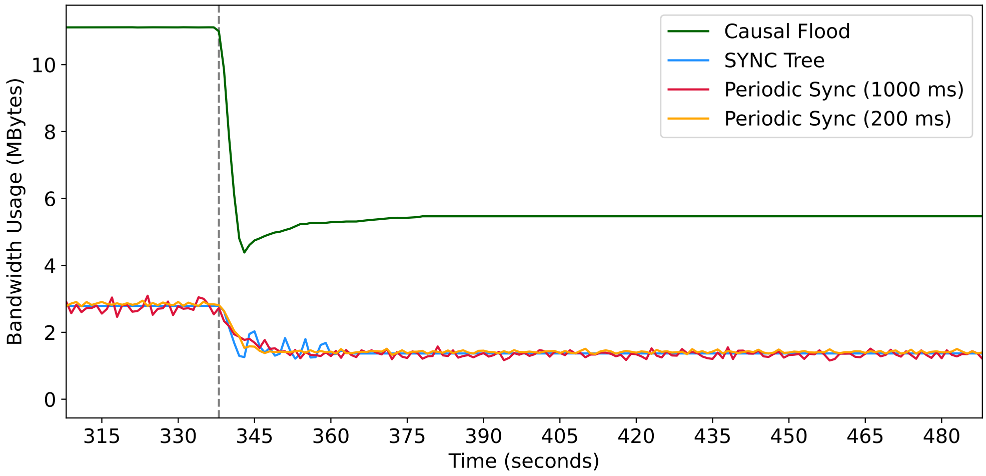


Figure I.17: Catastrophic Failure: Communication cost of the 4 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

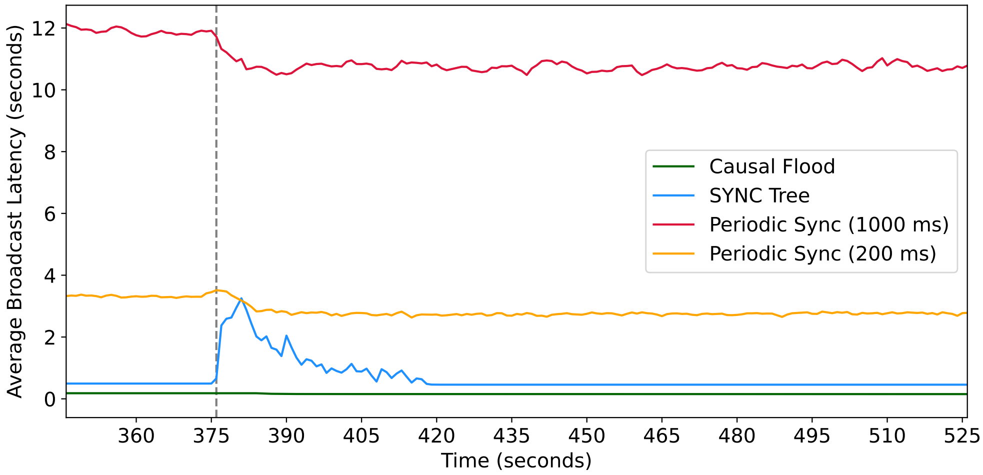


Figure I.18: Catastrophic Failure: Average broadcast latency of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

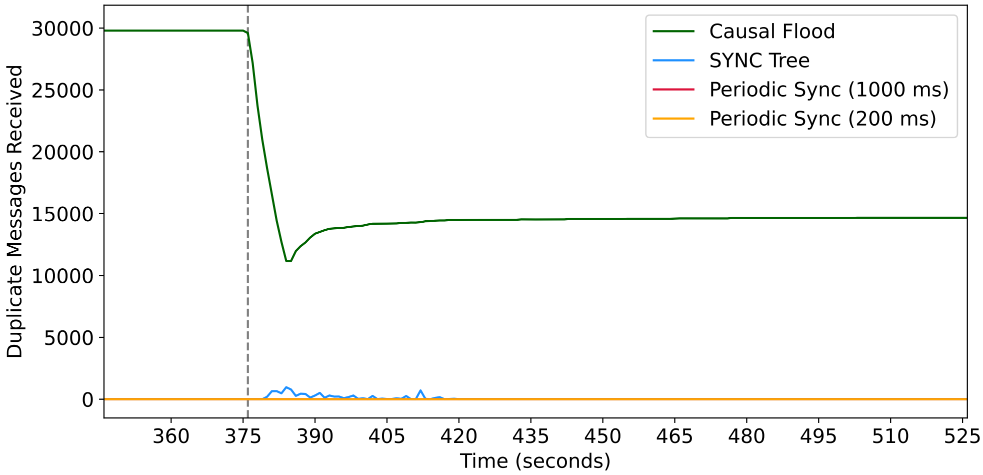


Figure I.19: Catastrophic Failure: Number of duplicate messages of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

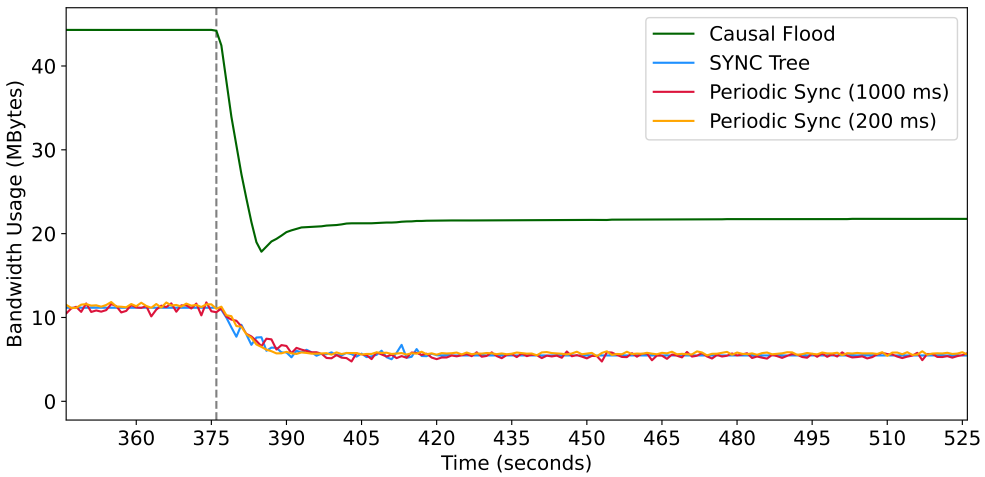


Figure I.20: Catastrophic Failure: Communication cost of the 4 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

II

ECO SYNC Tree: Additional Results

Figures II.1, II.2, II.3, II.4, II.5, and II.6 complement the results presented in Figures 5.15, 5.16, and 5.17 from Section 5.4.2, when varying the number of nodes in the churn scenario.

Figure II.7 complements Figure 5.18 of the same section by showing the disk usage of the two protocols over time when we fix p to 1, the payload size to 1024KB, and the number of nodes to 200, for the churn, massive join, and catastrophic failure scenarios.

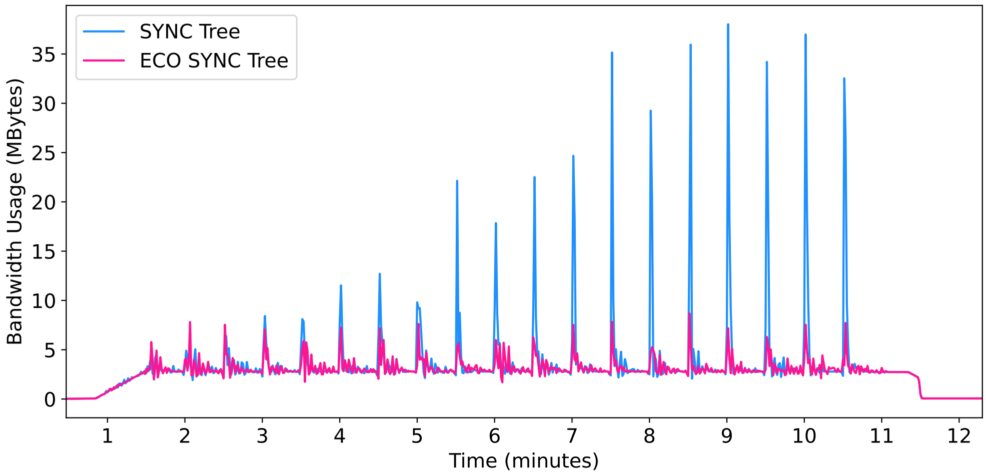


Figure II.1: Churn: Communication cost of the 2 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

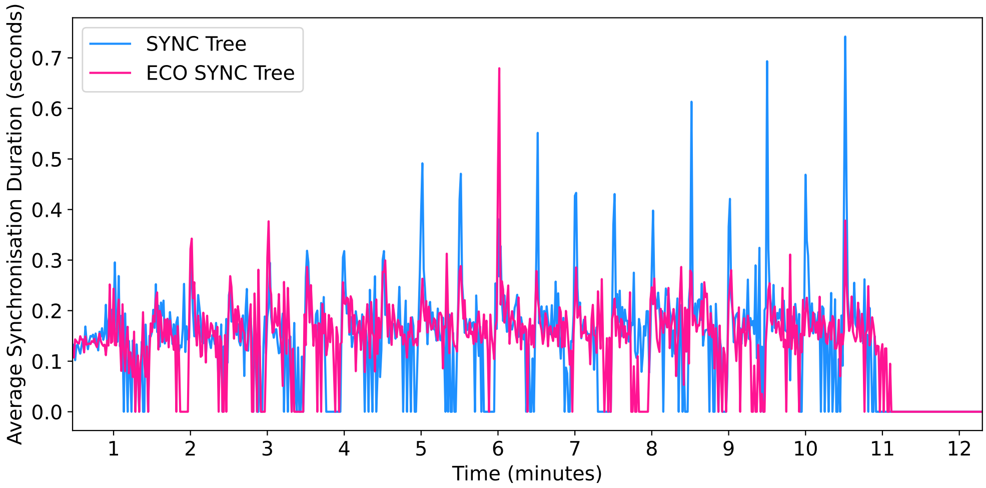


Figure II.2: Churn: Average synchronisation duration of the 2 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

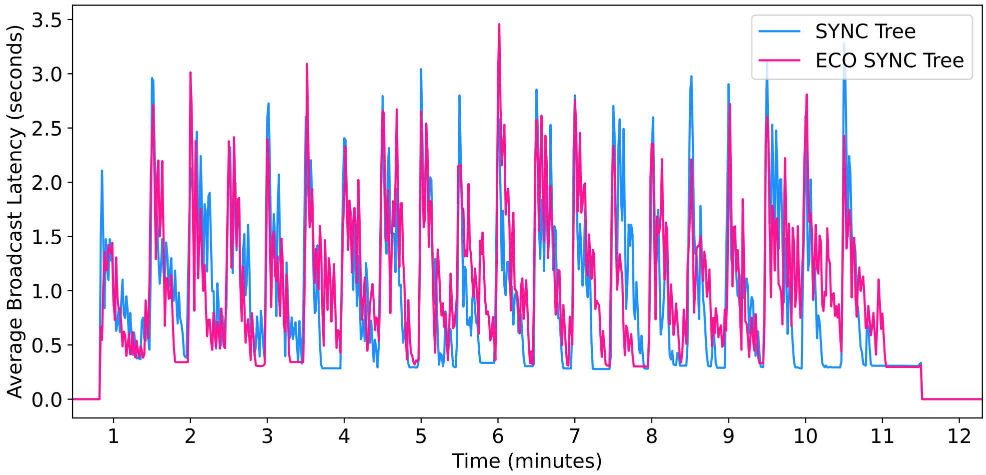


Figure II.3: Churn: Average broadcast latency of the 2 protocols over time with 50 nodes, $p = 1$, and payload of 1024KB.

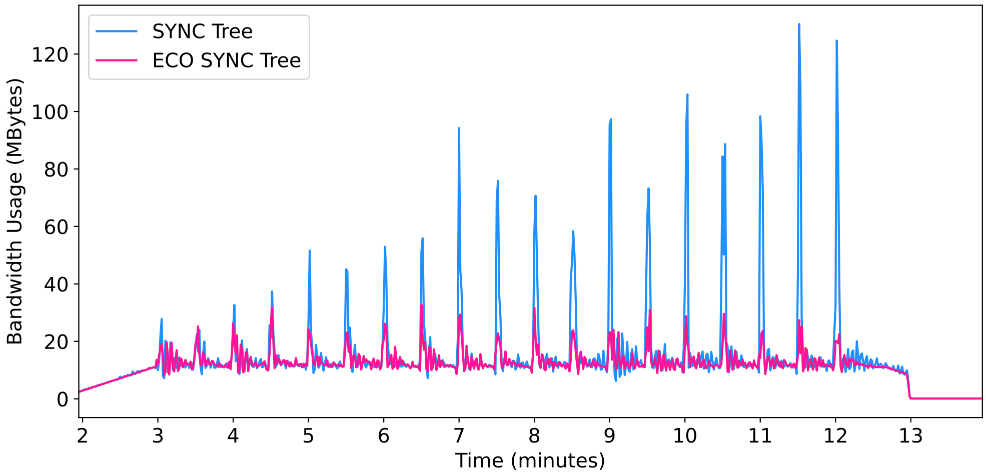


Figure II.4: Churn: Communication cost of the 2 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

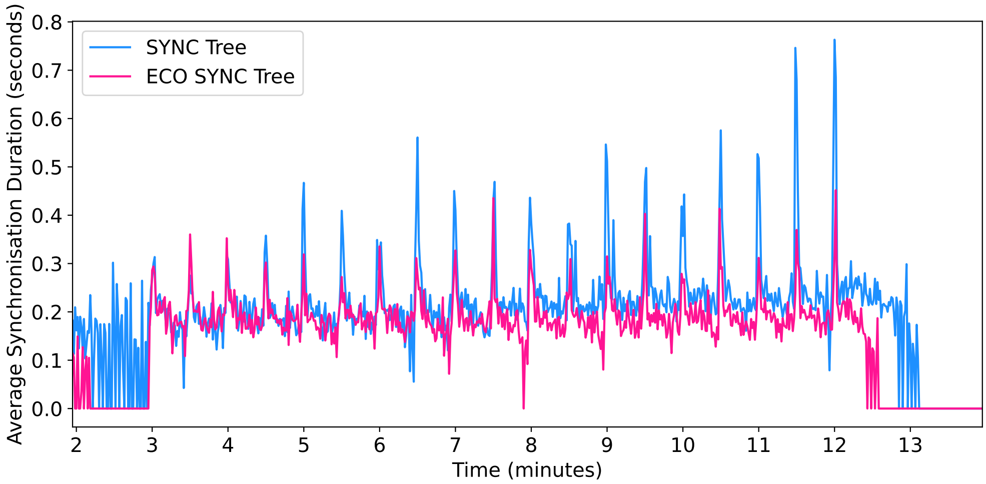


Figure II.5: Churn: Average synchronisation duration of the 2 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.

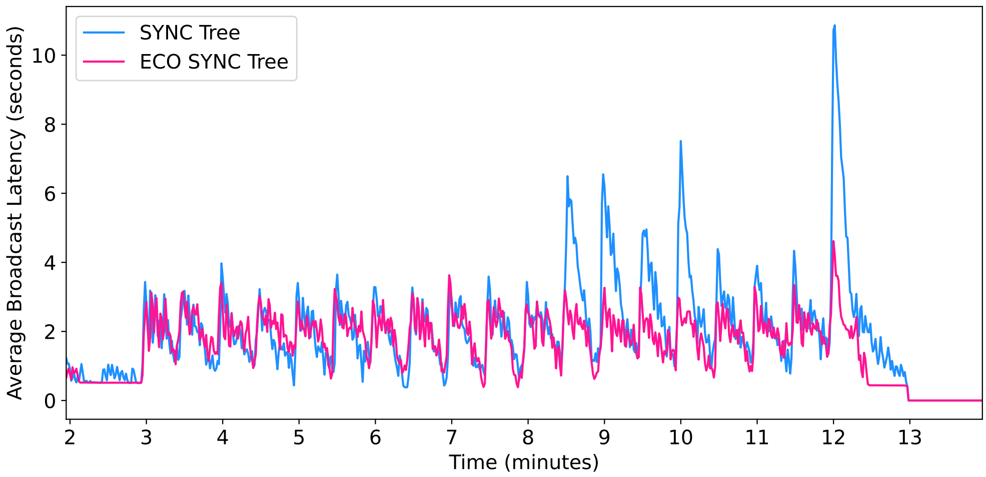
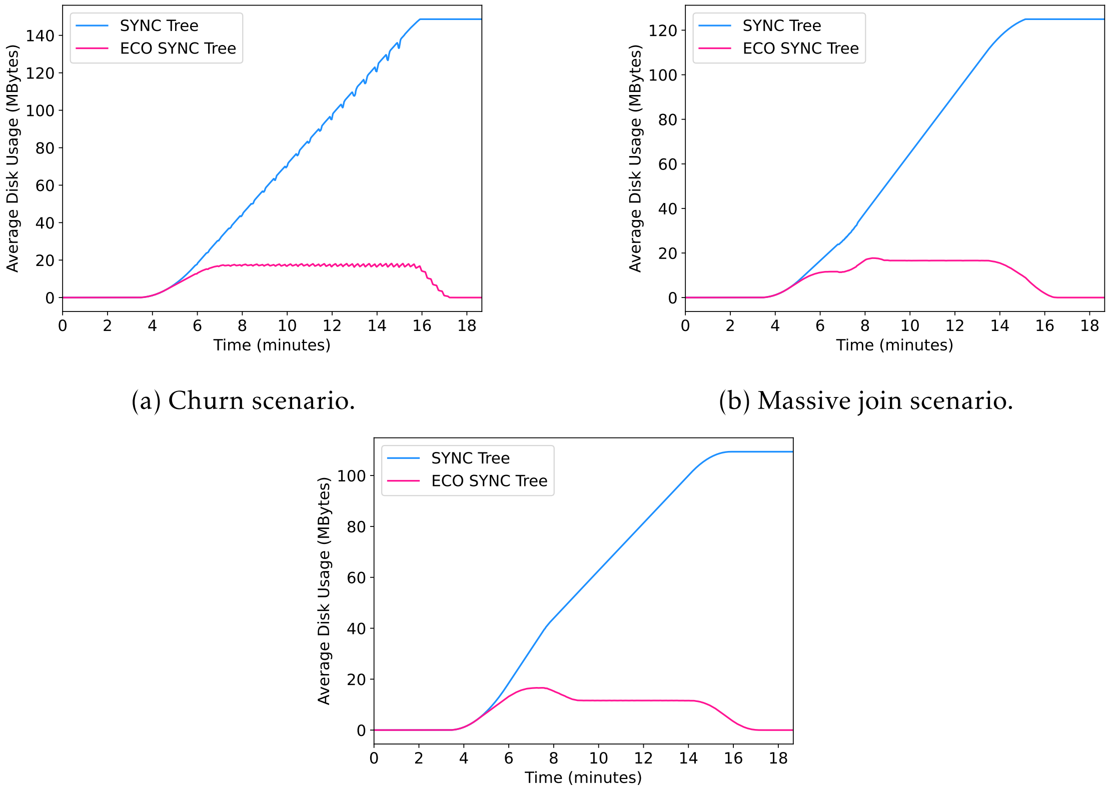


Figure II.6: Churn: Average broadcast latency of the 2 protocols over time with 100 nodes, $p = 1$, and payload of 1024KB.



(a) Churn scenario.

(b) Massive join scenario.

(c) Catastrophic failure scenario.

Figure II.7: Disk usage of the 2 protocols over time with 200 nodes, $p = 1$, and payload of 1024KB.