NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

SCALABLE CONSISTENCY FOR DATA REPLICATION

PEDRO FILIPE VEIGA FOUTO

Master in Computer Science and Engineering

DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2024

DEPARTMENT OF
COMPUTER SCIENCE

SCALABLE CONSISTENCY FOR DATA REPLICATION

PEDRO FILIPE VEIGA FOUTO

Master in Computer Science and Engineering

Adviser: João Carlos Antunes Leitão
Associate Professor, NOVA School of Science and Technology of NOVA University Lisbon

Co-adviser: Nuno Manuel Ribeiro Preguiça
Full Professor, NOVA School of Science and Technology of NOVA University Lisbon

DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2024

Scalable Consistency for Data Replication

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.27) [71].

Abstract

Distributed data storage solutions are key components of large-scale Internet services. The consistency guarantees provided by the protocols used to replicate data in such solutions can vary greatly, with some protocols providing strong guarantees while sacrificing availability, and others providing weaker guarantees while allowing higher availability. The choice of the replication protocol to use is often tied to the physical distribution of data replicas. As a general rule for Internet services, data replicated within a single datacenter is often replicated using protocols providing stronger consistency guarantees, while data replicated across multiple datacenters (i.e., geo-replicated) is often replicated using protocols providing weaker consistency guarantees. However, designing performant and fault-tolerant data replication solutions with data consistency that can scale in number of replicas is a challenging task. This is true not only for strong consistency models, but also for weaker consistency models, such as causal (and causal+) consistency.

In this thesis, we propose to address the scalability and fault-tolerance limitations of providing consistency guarantees in data replication solutions, by addressing the entire spectrum of replication deployment scenarios, from single datacenter deployments, to geo-replicated, and edge computing deployments. To this end, we propose three main contributions, one for each of these deployment scenarios. First, we propose a new state-machine replication protocol, based on a new variant of Paxos, which improves on existing solutions by maximizing the throughput while avoiding performance degradation when increasing the number of replicas. Second, we leverage the properties of the proposed protocol to replicate key components in geo-replicated causal consistency solutions, overcoming their fault-tolerance limitations while maximizing their performance. Third, by addressing the challenges of the edge computing environment, we propose a causal consistency data management solution that can efficiently scale to hundreds of edge locations.

Keywords: Data replication, Causal consistency, State machine replication, Edge computing, Geo-replication

Resumo

As soluções de armazenamento de dados distribuídas são um componente crucial para o funcionamento dos serviços de larga escala na Internet. As garantias de coêrencia fornecidas pelos protocolos utilizados para replicar dados nestas soluções podem variar consideravelmente. Enquanto alguns protolocos sacrificam disponibilidade para fornecer garantias de coêrencia fortes, outros fornecem garantias mais fracas, garantindo maior disponibilidade. Tipicamente, a escolha de protocolo de replicação a utilizar está associada à distribuição fisica das réplicas dos dados. Como regra geral para serviços na Internet, a replicação de dados dentro de um único centro de dados frequentemente utiliza protocolos que fornecem garantias de coêrencia mais fortes, enquanto a replicação de dados entre múltiplos centros de dados (i.e., geo-replicação) utiliza protocolos que fornecem garantias de coêrencia mais fracas. No entanto, o desenho de soluções de replicação de dados com alto desempenho e tolerantes a falhas, com garantias de coêrencia que possam escalar em número de réplicas, é uma tarefa desafiante. Isto não só para modelos de coêrencia fortes, mas também para modelos de coêrencia mais fracos, como a coêrencia causal (e causal+).

Nesta tese, propomos abordar as limitações de escalabilidade e tolerância a falhas de soluções de replicação de dados com garantias de coêrencia, abordando todo o espectro de cenários de distribuição de réplicas, desde um único centro de dados, passando por replicação geo-distribuida e terminando em cenários de *edge computing*. Para tal, propomos três contribuições principais, uma para cada um destes cenários. Em primeiro lugar, propomos um novo protocolo de replicação de máquina de estados, baseado numa nova variente de Paxos, que supera as soluções existentes maximizando o número de operações por segundo sem degradar o desempenho ao aumentar o número de réplicas. Em segundo lugar, utilizamos as propriedades do protocolo proposto para replicar componentes chave em soluções de coêrencia causal geo-replicadas, superando as suas limitações de tolerância a falhas, simultaneamente maximizando o seu desempenho. Em terceiro lugar, ao abordar os desafios do ambiente de *edge computing*, propomos uma solução de gestão de dados com coêrencia causal com capacidade para escalar eficientemente para centenas de localizações de *edge*.

Palavras-chave: Replicação de dados, Coêrencia causal, Replicação de máquina de estados, *Edge computing*, Geo-replicação

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Distributed data storage systems are key components of large-scale Internet services, such as commercial platforms, social networks, entertainment distribution platforms, among others. As the name implies, distributed data storage systems rely on a distributed architecture, spreading replicas (i.e., copies) of data across multiple physical servers, or even geographic regions. Data replication provides many important benefits, mainly: fault-tolerance and availability, as it allows the system to continue operating even in the presence of failures; throughput and scalability, as distributing the load of clients across multiple nodes allows the system to handle a larger number of requests; and latency, by replicating data in geographic locations closer to clients.

The physical location of the data replicas is a key aspect when deploying a distributed data storage system. *Co-locating* multiple replicas in the same datacenter (typically on different physical servers) is a common approach that can provide both fault-tolerance and performance benefits. However, for services that serve clients across the world and where user-experience is a key factor, this is not enough. As such, most modern global-scale Internet services rely on *geo-replication*, by employing distributed data storage solutions that replicate data across multiple datacenters in different geographic locations [30, 52]. This allows clients to contact the closest datacenter, minimizing the latency of their operations. More recently, with the emergence of the *edge computing* paradigm [101, 65, 29], application components started being deployed outside cloud datacenters, at the *edge* of the network closer to end users, further enhancing the performance of latency-sensitive applications. In real-world deployments, these *replica distribution levels* (i.e., the different levels of replica distributions across physical locations) are usually combined in a hierarchical way, with geo-replicated deployments usually employing multiple co-located replicas within each datacenter [30, 52, 69, 31, 32, 16], and edge computing solutions relying on small set of (usually geographically distributed) datacenters to support the operation of up to hundreds of edge nodes [1, 111, 12]. This hierarchical structure is illustrated in Figure 1.1.

While bringing many benefits, data replication comes with a main challenge: maintaining the consistency of all data replicas. Each distributed data storage system defines a *consistency model*, which specifies the guarantees it provides to clients regarding the data states they observe. Following the CAP theorem [39, 17], consistency models can be classified in two categories: *Strong consistency* models behave as if there was only a single copy of the data, guaranteeing that clients always observe an up-to-date state, however, this requires frequent and costly coordination between replicas that makes the system unavailable in the presence of failures. Such models sacrifice availability during network partitions in favor of consistency. On the other hand, *weak consistency* models allow replicas to temporarily diverge, possibly exposing inconsistent states to clients, but require less coordination between replicas, providing better performance and allowing the system to operate even in the presence of failures. These models tolerate network partitions by sacrificing consistency in favor of availability.

When considering data storage systems that only require weak consistency models, due to the cost associated with providing any kind of consistency guarantee, most existing solutions choose to minimize this cost by providing the weakest consistency model possible: *eventual consistency* [30, 52]. Eventual consistency is the weakest consistency model, with its only guarantee being that all replicas *eventually* converge to the same state if no new updates are made to the system during a long enough period of time [91]. This means that no guarantees are provided on the state observed by clients, allowing any sort of inconsistency (such as a client observing outdated data or out-of-order operations). Unfortunately, employing eventual consistency puts a high burden on application developers, who have to explicitly deal with consistency anomalies. Being the strongest consistency model that does not compromise availability (i.e., that falls in the category of weak consistency) [6, 73], *causal consistency* provides guarantees which are more intuitive for programmers to reason about their applications, by ensuring that operations that are causally related are made visible to clients respecting their precedence order (i.e., a client can never observe the result of an operation without being able to see the results of all operations that causally precede it). This makes it a balanced middle ground between eventual consistency and strong consistency models.

The *consistency model* of data storage systems and the physical location of data replicas (*replica distribution levels*) are closely related design decisions. Strong consistency models require frequent coordination between replicas that happens in the critical path of executing client operations, which is expensive in terms of latency, and thus are more suitable for datacenter deployments, where replicas are co-located and network latency is low. On the other hand, weak consistency models allow replicas to progress (and respond to client requests) without coordination, being able to tolerate network partitions, making them ideal for geo-replicated and edge computing deployments, especially when supporting user-facing services. This leads to a common approach where data replication solutions employ different consistency models at different *replica distribution levels*. For example, many geo-replicated causal consistency solutions focus on enforcing causal consistency for data being replicated *across* datacenters, while assuming that some sort of strong consistency protocol is used to replicate data between replicas *within* each datacenter [16, 69, 93, 70].

1.1 Motivation

Due to the coordination costs of ensuring consistency between replicas in a distributed data storage system, designing and implementing replication protocols with consistency guarantees that can scale to a large number of replicas (and/or geographic locations) without sacrificing performance, while simultaneously being fault-tolerant, is a challenging task. The challenges to be addressed are different depending on the *replica distribution levels* (i.e., the physical location of the data replicas) being addressed, as each level has its own unique characteristics, such as the typical number of replicas, the latency between them, the frequency of network partitions or even the consistency models that can be practically employed.

- When considering datacenter deployments (where replicas are co-located), most existing solutions opt for strong consistency models [46, 10, 13, 50, 18], typically providing *linearizability* [44], the consistency model with the strongest guarantees, being based on the Paxos protocol [58]. With Paxos being a leader-based protocol, Paxos-based solutions scale poorly with the number of replicas, with their performance decreasing as the number of replicas increases. This happens since these solutions require the leader to coordinate all operations, propagating them to all replicas and collecting their responses. This means that the throughput of the system is limited by the throughput of the leader, whose load increases with the number of replicas. Additionally, when considering read operations, which intuitively should be cheaper than write operations (since they do not modify the state of the system), in Paxos-based solutions, reads are often executed either by using the same (costly) protocol as writes, or by being served from a single replica, again limiting the scalability of such solutions. Finally, when considering fault-tolerance, many existing solutions rely on external coordination services that effectively make the system more vulnerable to network partitions [3], and less available.

- On the other hand, for geo-replicated deployments, where replicas are spread across multiple datacenters, most existing industry solutions employ weak consistency, being based on the eventual consistency model [30, 52]. While these solutions are able to scale well with the number of replicas, they only do so due to their lack of consistency guarantees, making them harder to use for application developers, that need to handle possible inconsistencies that may arise. To address these concerns and provide a more intuitive consistency model for application developers, many geo-replicated solutions providing causal consistency have been proposed [69, 31, 2, 36, 70, 16]. However, due to the increased coordination required to provide causal consistency, these solutions are not able to scale as well as eventual consistency, being limited in two different ways: *i*) within each datacenter, data needs to be replicated across multiple nodes to ensure fault tolerance. To achieve this, existing solutions often assume that some strong consistency protocol is used inside each datacenter to ensure that operations in each partition are linearizable; *ii*) most existing causal solutions that support partial replication [107, 36, 74, 16] (i.e., where datacenters do not host a full copy of the data) employ some sort of *operation sequencer* in each datacenter, which ensures operations are propagated between datacenters and executed in the correct order. This sequencer is a bottleneck for all operations, with its performance being crucial for the performance of the entire system. However, for fault tolerance, this single point of failure also needs to be replicated with a strong consistency protocol. Both of these limitations can only be overcome by leveraging on a high throughput strong consistency protocol that can be used without negatively impacting the performance of the system as a whole.

- Finally, when considering edge computing deployments, a completely new set of challenges arises. In particular, the number of edge locations (numbering in the hundreds for the largest cloud providers [40, 24]) can be orders of magnitude larger than the number of datacenters (at most a few dozen) in a traditional geo-replicated deployment. This, combined with the fact that edge locations are more prone to failures and network partitions, and have lower computing and storage capacity when compared to cloud datacenters, means that existing solutions for geo-replication are simply not adequate for the edge. As far as we are aware, there are no existing solutions with any level of consistency guarantees that are able to scale to a large number of edge locations. This results in the use of edge locations being heavily limited, only serving static content in content delivery networks (CDN) [98, 117] or executing simple computations without manipulating application data in serverless computing [53].

1.2 Thesis Statement

In this thesis, we propose a set of solutions which aim to address the scalability limitations of providing consistency guarantees in distributed data storage systems. The main goal is to allow application developers to build scalable and fault-tolerant applications that can be deployed across the entire spectrum of *replica distribution levels* (i.e., from single datacenters, to geo-replicated, to edge computing), while providing the strongest possible consistency guarantees at each level.

As such, the main question that this thesis aims to address is:

> *Is it possible to build scalable and fault-tolerant distributed data storage systems that extend from the cloud to the edge, while providing the strongest practical consistency guarantees at each level?*

Each *replica distribution level*, however, has its own set of distinctive challenges and requirements, from the level of consistency that can be provided, to the order of magnitude

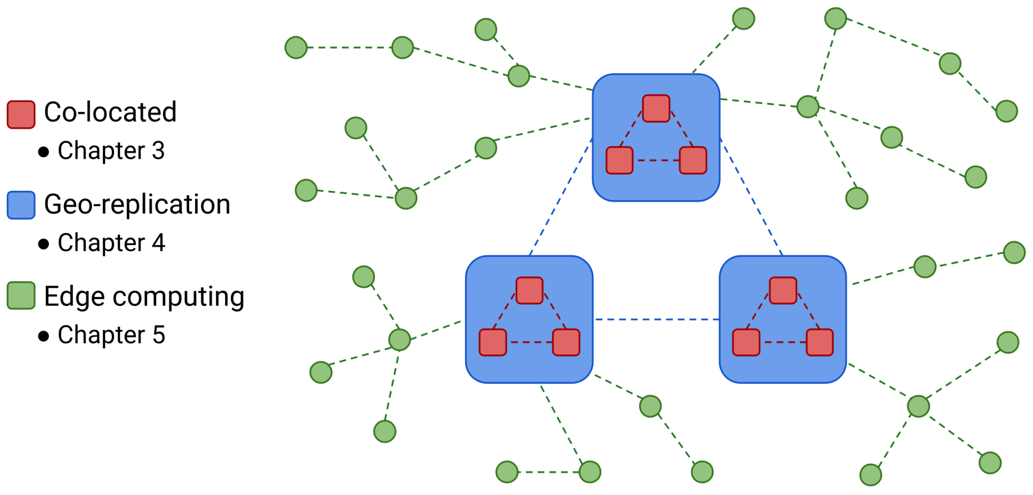


Figure 1.1: Hierarchical organization of *replica distribution levels*.

of the number of replicas, to the required fault tolerance and reconfiguration mechanisms. As such, there is no single solution that is optimal to be used across all deployment locations. Instead, the main contribution of this thesis consists in proposing a set of solutions that were specifically designed to address the challenges and requirements of each of these levels. These solutions can be used together to support distributed data management systems that span multiple deployment locations, but can also be used in combination with existing solutions, depending on the specific requirements of each deployment. Figure 1.1 illustrates the different *replica distribution levels* addressed in this thesis, along with their corresponding chapters.

Taking into account the distinct challenges of each *replica distribution level*, the *scalability* and *fault-tolerance* requirements are different for each one. In the chapters dedicated to each contribution (Chapters 3 through 5), we will provide a more detailed definition of these terms, as well as the specific requirements that each solution aims to address. We note, however, that in this thesis we do not address the problem of tolerating Byzantine faults, restricting our fault-model to crash failures and network partitions (i.e., we assume an asynchronous system model).

1.3 Contributions Summary

Following the thesis statement, this thesis proposes three main contributions, each addressing a different *replica distribution level* in the cloud to edge spectrum. These contributions are:

1.3.1 Scalable State Machine Replication

This contribution addresses the challenges of providing scalable strong consistency in a co-located (datacenter) deployment. It is materialized in a novel state machine replication protocol, called *ChainPaxos*. ChainPaxos relies on organizing replicas in a chain topology, allowing the load of the system to be distributed across all replicas equally. This, combined with executing multiple operations in parallel using a pipeline approach, allows the throughput of ChainPaxos to be higher than existing solutions, while simultaneously avoiding the performance degradation that happens when increasing the number of replicas. Additionally, ChainPaxos includes a novel mechanism for handling linearizable read operations, allowing them to be served from any replica, while scaling in throughput with the number of ChainPaxos replicas. Finally, ChainPaxos includes an integrated reconfiguration mechanism, thus minimizing its vulnerability to network partitions.

1.3.2 Scalable and Fault-Tolerant Geo-Replicated Causal Consistency

The second contribution addresses the scalability and fault-tolerance limitations of existing causal consistency solutions in geo-replicated deployments. It consists in a study of such limitations, while proposing and evaluating solutions to overcome them. In this contribution, we focus on a popular class of solutions based on operation *sequencers* that, while critical for enforcing causal consistency, also represent a single point of failure and contention for an entire region of a geo-replicated deployment. In this contribution, we show that it is possible to deploy causal consistency solutions based on sequencers in a scalable and fault-tolerant manner, which is a key requirement (and current limitation) for their adoption in real-world deployments.

1.3.3 Scalable Causal Consistency for the Edge

The final contribution addresses the challenges of providing causal consistency in a large-scale edge computing environment. It consists of a novel causal consistency data management solution specifically designed for the edge, called *Arboreal*. Arboreal distinguishes itself from existing traditional causal consistency solutions by being able to scale to hundreds of edge locations due to its decentralized hierarchical topology that allows for localized decisions without the need for global or centralized coordination. This topology also allows Arboreal to provide fault tolerance and reconfiguration mechanisms that are specifically designed for the edge. Additionally, Arboreal allows each edge location to dynamically change the set of data objects being replicated, adapting to the access patterns of clients.

1.4 Summary of Results

Considering the contributions summarized above, the main results present in this thesis are:

- Design and implementation of ChainPaxos, a novel state machine replication protocol, and evaluation of its performance in comparison with both existing academic state-of-the-art solutions and an industry solution.

- Design and implementation of a variant of an existing sequencer-based geo-replicated causal consistency solution where the sequencer is replicated with state machine replication, along with a study of its impact on the performance of the system and the visibility times of operations.

- Design and implementation of Arboreal, a new causal consistency data management solution for the edge that can scale to hundreds of edge locations, along with a detailed evaluation of its performance in an emulated large-scale edge environment. Additionally, in this thesis we also present the design and implementation of two exploratory works in the context of edge computing which influenced the design of Arboreal, along with their experimental evaluation.

1.4.1 Publications

The work presented in this thesis has led to the following publications, some of which resulted from collaborations both within my institution and with other research groups:

- Pedro Fouto, Nuno Preguiça, and João Leitão. Large Scale Causal Data Replication for Stateful Edge Applications. *2024 IEEE International Conference on Distributed Computing Systems (ICDCS).* 2024.

- Pedro Fouto, Nuno Preguiça, and João Leitão. High throughput replication with integrated membership management. *2022 USENIX Annual Technical Conference (USENIX ATC).* 2022.

- Pedro Fouto, Pedro Ákos Costa, Nuno Preguiça, and João Leitão. Babel: a framework for developing performant and dependable distributed protocols. *2022 41st International Symposium on Reliable Distributed Systems (SRDS).* IEEE, 2022.

- Miguel Belém, Pedro Fouto, Taras Lykhenko, João Leitão, Nuno Preguiça, and Luis Rodrigues. ENGAGE: Session Guarantees for the Edge. *2022 International Conference on Computer Communications and Networks (ICCCN).* IEEE, 2022.

- Pedro Ákos Costa, Pedro Fouto, and João Leitão. Overlay networks for edge management. *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA).* IEEE, 2020.

- Albert van der Linde, Pedro Fouto, João Leitão, and Nuno Preguiça. The intrinsic cost of causal consistency. *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC).* 2020.

1.5 Outline

The remaining of this document is organized as follows:

Chapter 2 provides fundamental concepts on data replication protocols, which are relevant for the understanding of the contributions presented in this thesis.

Chapter 3 presents ChainPaxos, a novel state machine replication protocol, including its design, implementation, and evaluation.

Chapter 4 presents our study on the performance impact of replicating the operation sequencer present in causal consistency solutions with partial replication, along with our proposed solution to overcome the limitations identified.

Chapter 5 presents Arboreal, a novel causal consistency data management solution for the edge, including its design, implementation, and evaluation.

Chapter 6 presents secondary contributions that were developed during the course of this work, which are related to the main contributions, but do not directly address the main goal of the thesis.

Chapter 7 concludes the document by summarizing the main results and discussing future work.

2 Fundamental Concepts and Research Context

Replicated data management systems are a fundamental building block of many modern applications. Depending on the application requirements, the techniques used to replicate data can be significantly different. In this chapter, we start by introducing the concept of data replication along with its main benefits and challenges.

We then discuss what we consider to be the most important design decisions for data replication protocols in the context of this thesis: we start by discussing the different consistency models that can be provided by a data replication system, the common techniques used to implement them, and the trade-offs between availability and consistency in Section 2.1; in Section 2.2 we discuss the impact of the distribution of the physical location of data replicas, and how user-centric applications can benefit from different distributions; then in Section 2.3 we discuss the trade-offs between full and partial replication, how the latter can be used to improve the performance of a replicated system, and how this becomes an essential requirement when considering edge computing; finally, in Section 2.4 we discuss how these design decisions influence each other, and how they can be combined to design efficient and reliable data replication systems.

Data Replication

The always-increasing popularity of the Internet and its applications has led to growing demands from its users. Such demands are usually related to performance, availability or latency. To cope with these demands, the data storage systems that support these applications resort to some form of data replication, by storing multiple copies of the same data in different locations. While there is a wide range of replication protocols with different characteristics, they usually strive to provide one or more of the following properties:

Fault-Tolerance and Availability By replicating data across multiple replicas, if one replica (or several) becomes unavailable, due to a crash, network partition, or any other reason, the system remains available since there are other (correct) replicas that have copies of the data to ensure that the system can keep its correct operation. Data redundancy, if managed correctly, makes it very unlikely that any data will ever be lost, even if multiple replicas (even an entire datacenter) fail at the same time. While some replication protocols require some sort of reconfiguration from the replicas that remain active after failures, protocols focusing on availability allow clients to continue to access the system without any noticeable impact from failures.

Scalability By having multiple replicas of data, replication protocols usually allow the system to scale with the number of replicas. By distributing client operations across all replicas, the system can handle a larger number of operations per second. This is usually true for operations that do not change the data (i.e., read operations), as they can often be executed in parallel in all replicas. However, state-modifying operations (i.e., write operations) usually require coordination between replicas (at the very least to propagate the operation to other replicas), which can limit the scalability of the system.

Latency While many deployments of replicated systems are co-located (e.g. all replicas are in the same datacenter), replicating data across multiple geographical locations (geo-replication) is a common practice that improves user-experience by reducing the latency of their operations. This is especially important for user-centric global applications, where users are spread all over the world and expect fast response times, which can only be achieved by having replicas close to the users.

Despite all the benefits of data replication, designing and implementing a replicated data storage system also brings multiple challenges. One of the main challenges is data consistency, as when multiple replicas exist, it is impossible to execute an operation atomically in all of them, requiring some form of coordination between replicas to propagate updates. This coordination can be costly, and sometimes it has to be avoided. Additionally, the physical distribution of replicas can also have a significant impact, as the latency and reliability of the network connecting replicas can vary significantly, affecting the coordination between replicas and the availability of the system. Finally, creating multiple full copies of data leads to increased costs in terms of storage, network bandwidth, and processing power, which can be prohibitive for some applications or deployments.

2.1 Data Consistency Models

As previously discussed, replicating data in a distributed system requires coordination between replicas both to propagate updates and to ensure clients observe up-to-date views of the data. This coordination can be costly, and sometimes it has to be avoided, either to improve performance or to ensure availability. However, avoiding this coordination can lead to clients observing inconsistent states (e.g., two clients simultaneously observing different versions of the same data object), which can be problematic for some applications. As such, each replication protocol employs a set of restrictions that limit the states that the system can expose to clients, which implicitly also defines the amount of coordination required between replicas to ensure these restrictions are enforced. These rules are known as *consistency models*.

Data consistency model: A contract between a distributed system and its clients which defines the valid states that the system can expose to clients, given the history of the system and of the client. Stricter (stronger) consistency models require more complex and costly coordination between replicas, but also reduce the possible inconsistencies that can be observed by clients.

This trade-off between having more coordination (preventing clients from observing inconsistent states) or avoiding coordination (achieving better availability or performance) is captured by the CAP theorem [39], which states that it is impossible for a distributed system to simultaneously provide the following three guarantees:

(Strong) Consistency: Allowing clients to always an up-to-date state of the system (i.e., the system behaves as if there was a single replica that is always up-to-date)[1].

Availability: Allowing clients to access the system at all times (i.e., the system eventually responds to every client request).

Partition Tolerance: Ensuring that all properties provided by the system are enforced in the presence of network partitions.

According to the CAP theorem, only two of these guarantees can be provided simultaneously by a distributed system. In practice, however, network partitions are unavoidable in large scale distributed systems (even those that are deployed within a single datacenter) and, as such, CAP effectively implies that a distributed system is unable to provide both (strong) consistency and availability in the presence of network partitions [17]. Due to this observation, most data replication solutions opt to be tolerant to network partitions, and thus, they need to choose between offering consistency or availability.

While multiple consistency models have been proposed (each with many practical materializations), the CAP theorem allows us to classify them into two categories: *strong consistency*, which favors consistency over availability; and *weak consistency*, which favors availability [17][2]. When considering the guarantees provided by replicated data storage

[1]Note that consistency in the context of distributed systems (and particularly the CAP theorem) is different from consistency in the context of the ACID guarantees of databases, where it is defined as database transactions not violating database constraints. In this thesis we do not address transactions, only individual read/write operations over data items.

[2]Replicated storage systems employing strong or weak consistency are also sometimes referred to as being *CP* or *AP*, respectively. In this document, we choose to use the terms *strong consistency* and *weak consistency*.

systems to their clients, weak consistency models guarantee that, even in the presence of network partitions, clients can continue to access the system, at the cost of observing stale or unordered data. On the other hand, strong consistency models guarantee that clients will always observe the most recent value of a data object, however they may be unable to access the system in the presence of network partitions. When considering the practical implementation of the replication protocol itself, this distinction usually comes down to the amount of coordination required between replicas: weak consistency models usually do not require coordination between replicas *before* responding to client requests (which sacrifices data consistency), whereas strong consistency models do (thus sacrificing availability during network partitions).

We now provide a more detailed discussion of the two categories of consistency models, along with specific models for each category and their practical implementations.

2.1.1 Strong consistency

In systems employing *strong consistency*, clients always observe the most recent view of the data, with every client observing the same sequence of updates. Informally, this means that despite the fact that the data is replicated across multiple processes, which can be updated concurrently, clients are given the illusion that there is a single always up-to-date replica.

The main advantage of strong consistency models is that, by providing the illusion of a single replica, developing applications for such systems is simple, as the application developer does not need to reason about the possibility of observing stale or inconsistent data.

On the downside, ensuring strong consistency in a replicated system is not a trivial task, requiring expensive coordination between all replicas to ensure both that all updates are executed in the same order in all replicas, and that all clients observe the most recent state of the system. This coordination is what prevents strong consistency systems from being tolerant to network partitions, as replicas on opposite sides of a partition are unable to coordinate. In these situations, strong consistency systems sacrifice availability, by blocking client requests until the partition is resolved.

In the context of data replication in distributed systems, the most relevant strong consistency models are the following:

Sequential Consistency: Sequential Consistency [54] ensures that all updates to the system state are made visible to all clients in the same order, as if they were executed sequentially in a single replica. This order does not need to be consistent with the real-time order of the updates, meaning that clients may observe updates in a different order than they were executed.

Linearizability: Linearizability [44] is the strongest and most common strong consistency model. In addition to ensuring that all replicas agree on the order in which operations are applied (thus giving the illusion of a single copy of the system state), linearizability further requires that the order in which operations are applied is consistent with the real-time ordering of these operations. This means that if an operation A completes before an operation B starts, then A must be ordered before B, and any subsequent read operation must return the value written by A or a more recent value.

Being the strongest and most common strong consistency model, in this thesis we will focus on linearizability over other strong consistency models. Two main approaches have been proposed in the literature to implement linearizability in a distributed system:

Shared Registers Shared register protocols replicate shared registers across multiple processes for fault tolerance, providing a simple interface with only read and write operations [5, 72]. Such protocols provide a linearizable ordering of operations, however, as each operation defines the entire state of the register, the ordering does not need to be stable. This means that a replica can apply a received write operation without knowing about earlier writes, as the applied operation contains their effects. These earlier writes can then be ignored if they are received later, as they are redundant. Ultimately, the simple interface of shared registers limits their applicability, as they are unable to implement generic computations over the replicated data.

State Machine Replication State machine replication (SMR) [99] is the most popular approach to implementing fault-tolerant services with linearizability, both in industry and academia [46, 10, 13, 50, 18]. In SMR, the state of a service is replicated across multiple replicas, with all operations that modify the state being applied in the same order in all replicas, making them transition through the same sequence of states. Operations in SMR can be any computation over the state of the service, as long as they are deterministic. Contrary to shared registers, SMR requires the ordering of operations to be stable, meaning that each replica must know when the position of an operation has been established and that no earlier operations will be received.

The main challenge of implementing SMR is ensuring that all replicas agree on both which operations to execute and the order in which they are executed. In asynchronous systems, consensus protocols are the most common approach to achieve this, with Paxos [58] (and its numerous variants and optimizations [45, 55, 56, 78, 75, 83]) being the most common one, widely popular both in industry and academia.

Many distributed systems ensure coordination between processes by either relying on external coordination services (such as Zookeeper [46]) that provide linearizability, or by employing their own linearizable (usually based on SMR) solutions. Examples of this include stream processing systems [50], distributed lock services [18], distributed computing frameworks [102], among others. Additionally, many distributed databases leverage strong consistency protocols to maintain data consistency. In particular, relational database management systems ($RDBMS$) require strong consistency in order to provide the $ACID$ guarantees. Even NoSQL databases, which usually focus on weaker consistency models, often support executing individual operations with strong consistency [52].

Being the most generic and widely used approach to provide linearizability, in this work (in particular in Chapter 3) we focus on using SMR to provide strong consistency guarantees.

2.1.1.1 Practical implementations of strong consistency

Being the most common approach to implement state machine replication (and thus, linearizability), Paxos, and its numerous variants and optimizations are the basis for many distributed systems that require strong consistency.

Paxos is a consensus protocol that allows a set of processes to agree on a single value in an asynchronous system where processes can fail, and messages can be lost or delayed. The protocol is divided into two main phases: the first phase is the *prepare* phase, where a process proposes a value to be agreed upon, which must be acknowledged by a quorum; and the second phase is the *accept* phase, where a pre-prepared value is accepted by a quorum of processes. A quorum in Paxos typically consists of a simple majority of processes. The protocol ensures only one value is chosen, and that all (non-failed) processes agree on this value. By executing multiple rounds of Paxos, a sequence of values can be agreed upon, effectively implementing a replicated state machine.

In practice, as all processes can propose values, conflicting proposals cause delays in the protocol execution, limiting its performance. To address this issue, practical implementations instead rely on Multi-Paxos [57]. In Multi-Paxos, after the first phase of the protocol, a leader is elected, which is solely responsible for proposing values. By having a single leader, conflicting proposals are avoided, with other processes simply redirecting their proposals to the leader. After the leader is elected, the fault-free execution of Multi-Paxos is relatively simple: the leader sends an *accept* message to all processes, which then respond with an acknowledgement message, either back to the leader, or broadcasted to all processes. Once a quorum of acknowledgements is received, the value is considered chosen. In the variant where the acknowledgements are sent back to the leader, the leader then needs to inform all processes of the chosen value.

While many industry solutions rely on the classic Multi-Paxos (or slight variants with the same communication patterns) to provide linearizability [46, 52, 50, 18], there is a large body of academic research focused on altering the protocol to improve different aspects of its performance, by trading off different properties of the protocol. This includes optimizations such as minimizing the latency of the protocol by skipping the leader [55, 56, 97], using different topologies to decrease the communication costs [78, 79, 113] or attempting to distribute the role of the leader across multiple processes [83, 75, 34].

2.1.2 Weak Consistency

Following the restriction captured by the CAP theorem (presented in Section 2.1), systems that employ *weak consistency* are those that sacrifice strong consistency guarantees in order to ensure availability in the presence of network partitions. In such systems, clients are allowed to observe outdated views of the data, or even inconsistent views (i.e., different clients may observe different values for the same data object).

The main advantage of weak consistency models is that replicas can progress independently of each other, without requiring the costly coordination between replicas that is required by strong consistency models. This has two main benefits: (1) client requests can be served faster, as they are not blocked by the coordination between replicas; and (2) during network partitions, replicas can continue to serve client requests. These benefits are particularly important in geo-replicated deployments, where the latency between replicas is high, and network partitions are more common.

On the downside, weak consistency models require application developers to reason about the possibility of observing inconsistent or stale data, which can increase the complexity of the application. Additionally, the lack of coordination between replicas can lead to the divergence of the state of the system, requiring some form of conflict resolution mechanism to ensure that all replicas eventually converge to the same state (which can be handled either by the application or by the replication protocol itself).

In the context of data replication in distributed systems, the most relevant weak consistency models are the following:

Eventual Consistency Eventual consistency is the weakest consistency model. The only guarantee provided by this model is that all replicas *eventually converge* to a consistent state, as long as no new updates are made to the system for a long enough period of time. This property is also known as *convergence* [91]. This allows replicas to diverge for an indefinite period of time, allowing clients to observe inconsistent states or even out of order updates.

Despite the lack of guarantees, eventual consistency is a very attractive consistency model for applications in which availability, low latency and performance are critical, as it offers the best performance of all consistency models by minimizing coordination between replicas. However, eventual consistency puts the burden of dealing with inconsistencies on the application developers, who need to be aware of the inconsistencies that may arise and handle them accordingly.

Causal Consistency Causal consistency ensures that clients will always observe a state of the system that respects the *happens-before* relationships between operations [59]. Intuitively, this means that if a client executes or observes the results of an operation A, and then executes an operation B, then it is possible that A caused B (or, in other words, *A happened-before B*). In such a case, systems employing causal consistency ensure that no client will be able to observe the effects of *B* without observing the effects of *A*. On the other hand, operations that are concurrent (i.e., neither one happened before the other) can be observed in any order, for instance if two clients concurrently update the same object in different replicas, then any client may observe either update first.

Causal consistency can alternatively be defined as a combination of the following four session guarantees:

Read Your Writes: A client's read operations will always observe the effects of its previous writes.

Monotonic Reads: Clients always observe an increasingly up-to-date state of the system.

Writes Follow Reads: If a client executes a write operation after a read operation, all other clients will observe the effects of the read before the effects of the write.

Monotonic Writes: If a client executes two write operations in succession, all other clients will observe the effects of the operations in the order they were executed.

Causal+ consistency, coined in [69], combines the properties of causal consistency with the convergence property of eventual consistency. As the *happens-before* relation of causal consistency only orders a subset of operations, some conflict resolution mechanism is still required to ensure convergence of concurrent updates. Thus, causal+ consistency ensures that all replicas will eventually converge to the same state, while preserving the causal ordering of operations. This is also known as *convergent causal consistency*[3].

An important property of causal consistency that makes it an attractive consistency model is that it provides the strongest possible consistency guarantees without sacrificing availability during network partitions [6]. On the other hand, while causal consistency is still available during network partitions, the coordination required to enforce causal dependencies can be costly, limiting its performance and scalability when compared to eventual consistency. In fact, this performance overhead has been cited as the main reason for the slow adoption of causal consistency in the industry [112, 80].

Databases supporting eventual consistency are commonly used by geo-replicated large-scale Internet services, with examples including Cassandra [52], Dynamo [30] and Riak KV [49]. Due to its more complex semantics, causal consistency is less adopted in

[3]While the term causal+ was only recently coined in [69], in practice any distributed data storage system with causal consistency requires the convergence property to be usable (and many previous systems already provided it). In this document, when using the term causal consistency, we assume that it also includes the convergence property.

the industry. However, some solutions have started providing support for it, including MongoDB [81] and Redis [95].

Despite its lower adoption, the consistency guarantees of causal consistency make it a very attractive alternative to eventual consistency, having been the focus of academic research in the context of geo-replicated (and edge) data storage systems [16, 69, 74, 32, 4, 36, 31, 2, 96, 107]. Being the consistency model in the weak consistency category that provides the strongest guarantees, in this work (in particular in Chapters 4 and 5) we focus on using causal consistency when weak consistency models are more appropriate than strong consistency models.

2.1.2.1 Practical implementations of weak consistency

Eventual Consistency. By not enforcing any order over the execution of operations, eventual consistency avoids the need for complex coordination between replicas, allowing the replication protocol of solutions employing it to be much simpler (and therefore performant) than those employing strong or causal consistency. In eventual consistency, replicas are allowed to execute operations immediately upon receiving them (either from a client or from another replica), without any coordination.

To ensure the *convergence* property, operations received from a client are then propagated to all other replicas, either immediately or periodically. Additionally, to ensure that all replicas eventually converge to the same state, solutions employing eventual consistency usually require some form of conflict resolution mechanism. This is required since replicas can execute operations in different orders, leading to different application states in each replica, for instance if two clients concurrently update the same object in different replicas. To resolve these conflicts, a common approach is to use *last-writer-wins* resolution (employed by popular industry solutions such as Dynamo [30] and Cassandra [52]), where each operation is tagged with a unique timestamp, with the operation with the highest timestamp being chosen in case of conflicts. Alternative approaches include requesting the application to resolve the conflict, storing multiple concurrent versions of the same object, or leveraging on conflict-free replicated data types (CRDTs) [100].

Causal Consistency. Implementing causal consistency requires more complex coordination between replicas than eventual consistency, as replicas need to ensure that operations are made visible to clients in an order that respects the *happens-before* relationship. For this, the key is to track the *causal history* of operations. The causal history of an operation is the set of operations on which it is causally dependent, or in other words, that *happened-before* it.

In systems employing causal consistency, operations are usually tagged with causality-tracking metadata, which represents the causal history of the operation, and is used to ensure that operations are made visible to clients in the correct order. While some systems (such as COPS [69] or Eiger [70]) directly track the specific operations on which an operation is causally dependent, a more common approach is to use vector clocks [59] for this purpose. This approach is commonly used in geo-replicated causal consistent systems, where vector clocks contain one entry for each datacenter. An operation is tagged with a vector clock that contains, for each datacenter, the timestamp of the most recent operation originated in that datacenter that the operation is causally dependent on. This approach allows compressing the size of the metadata, at the cost of possibly including unrelated operations in the causal histories. Solutions leveraging vector clocks for causal consistency include ChainReaction [4], SwiftCloud [93], or C3 [36]. Some solutions further compress the metadata by using a single scalar, however such solutions require additional mechanisms to enforce metadata, such as periodic background synchronization across all replicas (as in Cure [2] and GentleRain [31]), or by using specialized overlay networks to ensure that metadata is propagated in causal order (as in Saturn [16]).

There are two common approaches to assigning the causal history of an operation received by a replica: the first is to assume that all operations that are currently visible in that replica are part of the causal history (as the client may have observed any of them before issuing the operation), while the second is to maintain client-side metadata that tracks the causal history of the client itself, and using that metadata to assign the causal history of the operation. The former approach is simpler, but can lead to a large number of unrelated operations being included in the causal history (and is impractical for systems using direct tracking of causality, as it would lead to very large metadata sizes). The latter is slightly more complex, but allows for more accurate causal histories.

Finally, a practical implementation of causal consistency also requires the *convergence* property, achieving causal+ consistency. For this, techniques similar to the ones used in eventual consistency are usually employed.

Discussion

Recalling the goal of this work (in Section 1.2), we aim to make contributions for each *replica distribution level*, while providing the strongest practical guarantees for each level. Since availability is a key requirement for many global applications [30], limiting our focus to strong consistency models would be too restrictive. As such, in this thesis we focus on the two strongest consistency models in each category: linearizability for strong consistency, and causal consistency for weak consistency, relying on the most appropriate model for each *replica distribution level*.

2.2 Replica Distribution Levels

The targeted geographical distribution of data replicas in physical locations is another important design decision for data replication protocols. While some protocols assume that all replicas will be co-located in the same region (e.g. a single datacenter), others are designed to replicate data across multiple regions (i.e. geo-replication or edge computing), taking into account the higher latency and lower reliability of the network. When combined with appropriate consistency models, the geographical distribution of replicas can significantly influence the fault-tolerance, client-perceived latency and scalability of replicated systems.

Replica Distribution Levels: In this document, we define *replica distribution level* as the geographical distribution of data replicas in physical locations. We consider three main *replica distribution levels*: co-located/datacenter replication; geo-replication and; edge computing.

2.2.1 Co-located/Datacenter Replication

A common deployment of replicated data storage systems involves placing multiple replicas of the data in the same datacenter, but in different physical machines. This ensures fault-tolerance, as the failure of a single machine does not result in data loss or affect the availability of the system. Due to the high performance and reliability of datacenter networks, this deployment favors strong consistency models, by minimizing the latency cost of coordination between replicas. Examples of strong consistency solutions that assume data replication in a single datacenter for fault-tolerance include the Paxos protocol [58] and its numerous variants [61, 90, 78, 45] or Chain Replication [113]. These replication protocols are often used as the basis for implementing more complex solutions, such as data storage systems (both relational databases [88] and NoSQL databases [52]), and distributed coordination services (e.g., ZooKeeper [46] and Chubby [18]).

2.2.2 Geo-Replication

Due to the popularity of interactive global Internet services (such as social networks and instant messaging, among others), which are characterized by having users spread across the globe that expect fast response times, deploying data management systems (and thus, application logic) in a single datacenter is not enough. For such services, *geo-replication* has become the standard practice, where data replicas are spread across multiple datacenters in different geographical locations. This deployment allows replicas to be closer to clients, reducing the latency of their operations and improving their experience. Additionally, it also provides stronger fault-tolerance, as the system can tolerate the failure of entire datacenters.

However, just like with every other aspect of replication protocols, geo-replication has a cost. Since replicas are distributed across the globe, the network connecting them usually presents higher latency and is less reliable than networks within a single datacenter, with the latency between datacenters being possibly several orders of magnitude higher (i.e., tens to hundreds of milliseconds when compared to less than a millisecond on a single datacenter). Such latency values mean that employing stronger consistency models is less desirable, as the client would have to wait for the coordination between replicas across datacenters to complete before a response can be issued by the system, defeating the purpose of having geo-replication in the first place. As such, geo-replicated systems usually resort to weaker consistency models, which allow replicas to respond to clients without coordination. Examples of such data storage systems with weak consistency include Dynamo [30] and Cassandra [52], both of which were specifically designed to support large-scale Internet services, providing low latency and high availability in geo-replicated deployments.

2.2.3 Edge Computing

While geo-replication in datacenters is a common practice nowadays [30, 52, 69, 2, 31, 32, 16], the increasing user demand for low-latency applications has led to the emergence of a new paradigm: *edge computing*. While edge computing can be materialized in different ways [101, 65, 29], it can be generally defined as moving or replicating components of applications (both logic and data storage) beyond the boundaries of datacenters, closer to end-users. This not only allows for a new class of applications that require very low latency, such as AR/VR applications [104], autonomous vehicles [68], live video analytics [14] and location-based video games, but also satisfies the increasing demands of users of everyday large-scale Internet services, such as social networks or online shopping.

In fact, nowadays, cloud providers make use of hundreds of edge locations deployed across the globe, which are both offered to customers as part of their cloud services, and used internally to support their own public-facing services [40, 24]. However, application logic is usually centralized in the data centers, since the application data required to handle client requests is traditionally stored in datastores within those same data centers. This results in the use of edge locations being heavily limited, typically being used to serve static content in content delivery networks (CDN) [98, 117] or to execute simple computations without manipulating application data in serverless computing [53]. For user requests that must manipulate application data, edge nodes simply operate as reverse proxies, redirecting requests to data centers [40, 23], where the data required to process the requests is stored.

The main reason for these limitations is the lack of adequate data replication protocols for edge computing. While edge computing can be seen as an extension of geo-replication, it comes with a new set of challenges that bring additional restrictions to the design of data replication protocols. First, the high number of edge locations means that most traditional geo-replicated solutions, which were designed for a small number of datacenters, are unusable in this scenario. Second, to fully leverage the benefits of edge, applications need to be able to dynamically deploy (and remove) components in different edge locations (e.g., to adapt to the number of users in each location or to the availability of resources in each location). This requires data replication solutions that can similarly create and remove data replicas in edge locations on-demand, which is not supported by most existing solutions that assume a static number of datacenters. Finally, the limited resources of edge locations (e.g., limited storage and bandwidth) requires replication protocols to adapt to the different characteristics of each edge location, whereas most existing protocols assume homogeneous resources across all datacenters.

Discussion

The three *replica distribution levels* that were just presented are not independent, or mutually exclusive. In fact, they can be seen as a continuum, where each level depends on the previous one. For instance, geo-replicated deployments are not composed of a single replica in each datacenter, but rather of multiple co-located replicas in each datacenter, usually employing different replication protocols to replicate data *across* or *within* datacenters. Similarly, edge computing can be seen as an extension of geo-replication, with edge locations usually replicating data to one or more cloud datacenters, which is then replicated to other datacenters.

Following the goal of this work (in Section 1.2), we aim to make contributions for all three *replica distribution levels* that were just presented, each tailored to the specific requirements of each level. The contributions can either be used independently, combined with each other, or even combined with existing solutions that target a specific level.

2.3 Replication Scope

While creating and coordinating multiple replicas of the data that represents the state of a system is the main idea behind data replication, it is not always necessary (or even desirable) to create *complete* copies of the data in every location.

> **Replication Scope:** In this work, we define *replication scope* of a data replication protocol as the distribution of data replicas across the processes executing the replication protocol. We consider three distinct replication scopes: full replication, static partial replication, and dynamic partial replication.

2.3.1 Full Replication

Data replication solutions which assume that every process executing the replication protocol contains a full copy of the data are said to rely on *full replication*. In these solutions, every operation is applied to every replica[4]. While easier to design and implement,

[4]Note that, in the context of full replication (usually in co-located solutions), the term *replica* is often used to refer both to a replica of the data and to a process executing the replication protocol, as there is a one-to-one correspondence between the two.

full replication is not the appropriate solution for every scenario. In particular, when considering global applications with a large number of users and large volumes of data, materialized on geo-replicated deployments, full replication can negatively impact the performance of the system, due to both the storage cost of maintaining numerous full copies of the data, and the network cost of propagating every update operation across datacenters in distant geographic locations.

2.3.2 (Static) Partial Replication

To circumvent the problems of full replication, one can replicate only a subset of the data in each node. Solutions that adopt this approach are said to employ *partial replication*. Such solutions typically divide the data into *data partitions*, with each node storing only a subset of the data partitions[5]. Since only a subset of the data is replicated in each node, each operation only needs to be applied in a subset of nodes, reducing the network cost and overall computing costs of executing operations. This allows partial replication solutions to display better performance and scalability than full replication solutions, at the cost of less reliable fault-tolerance guarantees (as the failure of a subset of nodes may result in data loss).

However, partial replication also comes with its own set of challenges. Tracking which nodes replicate which data partitions requires additional coordination between replicas and metadata overhead, and to provide any sort of (even if weak) data consistency, nodes may need to coordinate with other nodes that have no data partitions in common. Since efficiently addressing these challenges is not trivial, most existing solutions only support *static partial replication*, where both the number of data partitions and the set of nodes replicating each data partition are static and determined when the system is started [52, 16, 93].

2.3.3 Dynamic Partial Replication

While static partial replication is a good solution for traditional geo-replicated deployments, where the number of datacenters is small and stable, it is not suitable for edge computing, where the number of edge locations can reach the hundreds (or even thousands) and be highly dynamic (with edge locations being added or removed on-demand). In such scenarios, hand-picking which data partitions are replicated in each location is not practical, and edge locations may not have the resources to replicate an entire data partition. As such, we argue that edge computing requires solutions that support *dynamic partial replication*, where individual data items can be dynamically replicated in different locations, depending on the access patterns of the clients, and without requiring manual intervention.

[5]While the term *replica* is still (less frequently) used to refer to a process executing the replication protocol in the context of partial replication, this can lead to confusion, since it no longer refers to a full replica of the data. As such, in this document, in the context of partial replication, we use the term *node* or *process* to refer to a process executing the replication protocol, and *replica* to refer to a copy of data replicated by a node

Discussion

For solutions with strong consistency (typically co-located), coordination between all nodes is required even among those who do not replicate the same data partitions, as every operation needs to be ordered with respect to all other operations. As such, these solutions are unable to fully benefit from partial replication, and thus, they usually assume full replication [78, 113, 56].

On the other hand, solutions with weak consistency, which require less coordination between nodes, can leverage partial replication to improve performance. This is particularly important in geo-replicated deployments, where application developers can leverage static partial replication to define which data partitions are replicated in which geographic locations, in order to minimize the latency of client operations. In the context of edge computing, dynamic partial replication is crucial to allow data items to be dynamically replicated in different edge locations, depending on the access patterns of the clients.

Recalling the goal of this thesis (in Section 1.2), we aim to make contributions for each *replica distribution level*. For this, we will leverage all three replication scopes presented in this section, using each when it is more appropriate for the specific requirements of each level, as explained in the following section.

2.4 Final Remarks and Contributions Preview

The previous sections (Sections 2.1 through 2.3) presented what we consider to be the most important design aspects of data replication protocols (consistency models, replica distribution levels, and replication scope) in isolation. As the reader can probably infer from the discussions of these sections, in practice, these aspects interact with and influence each other, with some combinations being more appropriate than others for specific scenarios.

2.4.1 Interactions Between Design Aspects

When considering strong consistency models, their characteristics make them more suitable for co-located deployments (usually in a single datacenter), which are less prone to network partitions and where the latency between replicas is low, allowing for fast coordination and minimizing the impact of the coordination overhead. Furthermore, as every client write operation needs to be ordered with respect to all other operations, strong consistency models are unable to fully benefit from partial replication, with most solutions assuming full replication [78, 113, 56].

On the other hand, weak consistency models, with their lower coordination requirements and tolerance to network partitions, are ideal for large-scale geo-replicated deployments, where the higher latency and lower reliability of the network between datacenters can be problematic for strong consistency models. Additionally, when considering partial replication, employing it is much simpler in weak consistency models when compared to strong consistency models. Particularly in the case of eventual consistency, where no order is enforced between updates, partial replication can be achieved by simply propagating updates only to the interested subset of replicas. While this is trickier in causal consistency, as the *happens-before* needs to be preserved even between updates that are not replicated in the same set of nodes, many recent contributions provide support for partial replication [16, 36, 107, 116].

2.4.2 Replication Hierarchy

Another relevant aspect to consider is that existing data replication solutions do not always restrict themselves to a single *replica distribution level*. As briefly discussed in Section 2.2, each level requires the previous one, with edge computing depending on geo-replication, which in turn depends on co-located replication. This means that solutions that target a specific *replica distribution level* need to either include replication protocol(s) for the previous level(s) (which can have different replication scopes and consistency models), or abstract the previous level(s) by assuming that some other replication protocol with the desired properties is employed.

For instance, many geo-replicated causal consistency solutions [69, 32, 31] propose replication protocols that employ full replication (i.e., each datacenter as a whole contains a full copy of the data) and enforce causal consistency between datacenters, while abstracting replication inside each datacenter, assuming that some replication protocol with partial replication (i.e., each node of a datacenter only replicates a subset of data) and strong consistency is being used within each datacenter.

On the other hand, some geo-replicated solutions [52, 36, 4] employ replication protocols that span both the geo-replicated and the co-located levels, with different replication scopes and consistency models. For instance, C3 [36] employs a causal consistency protocol both for inter-datacenter and intra-datacenter replication, using two different partial replication schemes, with each datacenter replicating a number of data partitions, which are further divided into smaller partitions to be replicated in each node.

2.4.3 Contributions Preview

Recalling the goal of this thesis (Section 1.2), we aim to make contributions for each *replica distribution level*. Following the rationale presented in Section 2.4.1, we will leverage all three replication scopes and two consistency model categories, using each when it is more appropriate for the specific requirements of each level. As such, our three main contributions will be focused on:

- In Chapter 3, we will focus on *co-located* replication, with *linearizability* (strong consistency) and *full replication*, by proposing a new scalable and fault-tolerant state machine replication protocol.

- In Chapter 4, we will focus on *geo-replication*, with *causal consistency* and *static partial replication*, by studying the fault-tolerance and scalability limitations of such protocols, and proposing and evaluating solutions to overcome them.
- In Chapter 5, we will focus on *edge computing*, with *causal consistency* and *dynamic partial replication*, by proposing a new data replication protocol that can dynamically replicate data items in different edge locations, while scaling to hundreds of edge locations.

When considering the discussion in Section 2.4.2, each of our contributions is focused on a single *replica distribution level*, focusing on the specific challenges and requirements of that level, and abstracting the previous ones. While our contributions can be combined to span multiple levels, it may not be the most efficient combination for all scenarios. Being isolated allows each contribution to be used independently, or combined with other existing solutions for different levels, providing a more flexible and adequate solution for each specific scenario.

3 ChainPaxos: Scalable State Machine Replication

This chapter presents the *ChainPaxos* protocol, a novel scalable state machine replication protocol which is able to maximize the throughput of both write and linearizable read operations, while providing an integrated membership management mechanism.

Following the goal of this thesis and the definitions presented in Chapter 2, this contribution addresses the co-located *replica distribution level*, by proposing a replication protocol with strong consistency guarantees (linearizability) that employs full replication.

We start by providing some background on state machine replication and the Paxos consensus protocol, followed by the concrete goals of this contribution. We then present the design and implementation of ChainPaxos, along with its extensive evaluation. The chapter concludes with a discussion on related work, and a final discussion on the most relevant aspects of the contribution.

3.1 Background

Fault-tolerance is a key property for distributed systems, being fundamental to guarantee that they continue to operate despite failures of individual components. To achieve this, the state of the system needs to be replicated over multiple nodes.

A particularly interesting and widely used way of providing fault-tolerance is by using state machine replication (*SMR*) [99, 60], which allows to replicate any service with linearizability. *SMR* replicates the state of a service across multiple replicas by executing the same sequence of deterministic operations on all replicas, ensuring they transition through the same sequence of states. The main challenge of implementing *SMR* is ensuring that all replicas agree on both which operations to execute and the order in which they are executed. In asynchronous systems, where processes can fail, consensus protocols are the most common approach to achieve this, with Paxos [58] (and its numerous variants and optimizations [45, 55, 56, 78, 75, 83]) being the most common one, widely popular both in industry and academia.

Paxos works by having replicas in a distributed system proposing values to be ordered, and then reaching agreement on a single value to be selected. By executing multiple rounds of this process, replicas can agree on a sequence of values to be ordered, effectively implementing a replicated state machine. In classic Paxos, every replica competes to order its own operations, which can cause conflicts that delay the progress of the system. To avoid this, practical implementations employ Multi-Paxos [57], where a single replica is elected as the leader, which is the sole responsible for submitting operations to be ordered. Being the canonical solution to implement SMR solutions (and thus, linearizability), the majority of academic research on this area is based on Multi-Paxos, with a focus on improving multiple aspects of the protocol.

3.2 Motivation and Goals

Regardless of their final purpose, consistency model, or *replica distribution level*, most distributed data storage systems rely on some replication protocol based on SMR, making it extremely relevant to improve the performance of consensus protocols. This includes using such protocols to either directly replicate and maintain the consistency of the data, both in relational database management systems (in order to provide the $ACID$ guarantees) and weak consistency NoSQL databases (which often support executing individual operations with strong consistency [52]), but also indirectly, in the form of external coordination services (such as Zookeeper [46]) or distributed lock services [18], among others.

Despite the numerous state machine replication solutions proposed (most of which consisting of Paxos variants), no existing solution satisfies the goal of this thesis of having a scalable and fault-tolerant solution. We now present our goals for this contribution, in terms of scalability and fault-tolerance, based on the limitations of existing solutions and the characteristics of state machine replication protocols.

Scalability When considering the scalability of a state machine replication protocol, there are two complementary aspects to consider: i) the maximum throughput (in terms of operations per second) that the protocol can achieve, and ii) the impact in throughput as the number of replicas increases.

When considering state-changing (write) operations, the majority of SMR solutions place an uneven load on a single replica (usually the leader), resulting in drastic throughput degradation as the number of replicas (and, consequently, the load on the leader) increases. As we will show in our evaluation (Section 3.4), this drastic degradation occurs with a number of replicas as low as 5. To achieve scalability, our key ideas are: minimizing the number of messages each node processes, and distributing the load uniformly across all replicas. Note that, for the latter goal, the aim is not for the throughput to increase with the number of replicas (as it is not possible to do so without sacrificing consistency), but rather to minimize as much as possible the throughput degradation as the number of replicas increases.

On the other hand, the performance of read operations is also crucial for SMR solutions, as they form the majority of operations in most systems. As these operations do not change the state of the system, intuitively, they should be cheaper to execute than write operations, however this is not always the case. In fact, most SMR solutions either execute reads as regular consensus operations (which is costly), require additional coordination between replicas to ensure linearizability, or limit reads to be executed in a single replica (usually the leader). All of these solutions limit the throughput of read operations. We aim to provide a solution that allows linearizable reads to be executed cheaply, from any replica and without extra coordination, effectively maximizing their throughput and allowing it to scale with the number of replicas.

Fault-Tolerance When considering fault-tolerance, existing solutions are limited by either: i) not supporting membership changes during system operation; or ii) requiring an external coordination service to be able to recover from faults and reconfigure the membership of the system. The latter limitation is particularly problematic, as it can lead to higher operational costs, slower fault handling, and increasing vulnerability to network partitions [3].

Taking this into account, our goal is to design a self-contained solution that is able to both handle and recover from crash failures and to reconfigure the membership of the system without relying on external coordination services, by integrating these mechanisms into the protocol itself.

3.3 ChainPaxos

In this section, we present the design of our solution, ChainPaxos, which aims to achieve the goals of scalability and fault-tolerance previously presented. We start by presenting the system model assumed, followed by an overview of the design of ChainPaxos. We then present the techniques used to handle faults and reconfiguration, and finalize by describing how linearizable reads are executed.

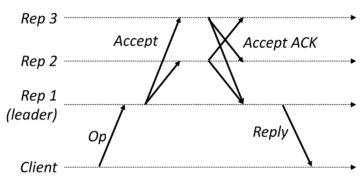
3.3.1 System Model

We assume an asynchronous distributed system with n nodes, connected by a network that can lose, duplicate, and deliver messages out of order. Nodes communicate by exchanging messages over a network with a fair loss model that allows the creation of FIFO channels between any pair of nodes. Nodes can fail by crashing, where they stop sending messages. We do not consider Byzantine faults.

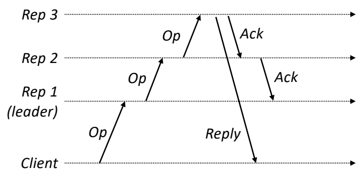
We follow the SMR model [99], in which each replica holds a copy of the system state and there exists a set of deterministic operations that may output a reply. Replicas start in the same initial state and apply the same sequence of operations, thus guaranteeing that all replicas transition through the same sequence of states and output the same results. We defer the processing of read operations to Section 3.3.4.

ChainPaxos is used to order the execution of operations. The system state includes the application state and the membership of the system, with `AddNode(n)` and `RemoveNode(n)` operations, respectively, adding and removing node `n` to the replica-set. These operations execute in the state machine, as other application operations, potentially impacting the quorum size of following operations. For correctness, a node can only decide on a given instance strictly after knowing the decision of all previous instances (and the current membership).

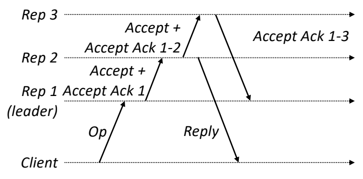
3.3.2 Design Overview



(a) Multi-Paxos



(b) Chain Replication



(c) ChainPaxos

Figure 3.1: Message flow of replication protocols on a fault-free run.

We now present the design of ChainPaxos. In order to better contextualize our design decisions, we start by briefly revisiting two popular strong consistency replication protocols, Multi-Paxos and Chain Replication, both of which served as inspiration for ChainPaxos, using Figure 3.1 as a reference.

Multi-Paxos In Multi-Paxos [58, 57], a distinguished proposer, known as *leader*, prepares multiple Paxos instances in a single step (Phase 1), followed by multiple sequential executions of Phase 2 of Paxos. In a fault-free run (Figure 3.1(a)), the leader sends an *accept* message to all replicas, with each replying to all replicas with an *accept ack* message. Any replica that receives *accept ack* messages from a majority of replicas can decide and execute the request (with the replica that received the operation replying back to the client). With n replicas, the message complexity of the protocol is $O(n^2)$: each replica incurs in $O(n)$ message overhead (the leader sends/receives $2n$ messages). The reply to the client is produced after 2 communication steps between replicas. Alternatively, a replica could send the *accept ack* message only to the leader, which would then forward the decision to all replicas. In that case, the overhead of non-leader replicas decreases to $O(1)$ at the cost of an additional communication step.

Chain Replication Chain Replication [113] leverages a chain topology, forwarding operations from the head to the tail (Figure 3.1(b)). The tail replies to clients after executing an operation, and sends *ack* messages backward, to allow replicas to perform garbage collection. In a fault-free run, each replica incurs in $O(1)$ message overhead, with a reply being produced after $O(n)$ communication steps.

The main design goals of ChainPaxos are: (i) minimize the number of messages each node processes in fault-free runs, making the load uniform and maximizing throughput (both for write and read operations); and (ii) integrate an efficient fault handling scheme into the algorithm, by taking advantage of Paxos messages, avoiding the need to rely on an external service. To achieve the first goal, we leverage the chain topology to combine and forward multiple Multi-Paxos messages in a single ChainPaxos message. For the second goal, by building on Multi-Paxos messages, leader faults can be handled by falling back to the first phase of Paxos.

In ChainPaxos, to execute a write operation in a fault-free run (Figure 3.1(c)), the leader sends the *accept* message, including its *accept ack*, to the following replica in the chain. Upon receiving an *accept* message, a replica forwards the message modified to include its own *accept ack*. When the *accept* message reaches the tail of the chain, it sends a message directly to the head with the *accept ack* of all replicas, guaranteeing that the head learns about the decided value. Additionally, it is necessary to inform the replicas that have not received enough *accept ack* messages to decide the value of the instance. ChainPaxos piggybacks this information in the next *accept* message.

When an *accept* reaches the replica at the middle of the chain, it includes *accept acks* from a majority quorum. Thus, the replica knows that the received request has been decided, and can execute the request and return the result to the client. In the example of Figure 3.1(c), with three replicas, the leader and replica 2 form a quorum, with replica 2 replying to the client.

This message flow achieves the first goal of minimizing the number of messages handled by each replica and keeping the load uniform: a single message is sent and received by every replica. As ChainPaxos is just using a different communication pattern to convey the messages of Multi-Paxos, it can fall back to the regular two phases of Paxos to handle faults. This is the base for achieving the second goal of integrating fault handling in the protocol, which we describe in Section 3.3.3.

By building on this message flow, and leveraging Multi-Paxos messages, ChainPaxos is able to provide high throughput replication by addressing the following challenges: i) support efficient garbage collection of information about decided values, which is a common challenge in many variants of Paxos, rarely addressed in the specification of algorithms (Section 3.3.2.2); ii) optimize fault-handling and integrate membership management by leveraging information about the chain topology, thus avoiding the common vulnerabilities/complexity encountered in systems that rely on external coordination services in the presence of network partitions [3] (Section 3.3.3); and iii) integrate a novel mechanism that leverages the chain topology to enable efficient linearizable read operations handled by a single replica without additional communication (Section 3.3.4).

Next, we detail the operation of ChainPaxos, describing the state maintained by each replica and the operation of the protocol in fault-free runs and during reconfiguration.

3.3.2.1 Protocol State

Algorithm 1 presents the state of each replica. The first variable group is related with the organization of the system and includes: the members and their order in the chain (`chain`); the identity of the local node (`self`); the next node in the chain that is not marked for removal (c_{nextok}); the currently supported leader ($c_{sleader}$); and the replicas for which a `RemoveNode` has been received but not yet decided (`marked`).

Algorithm 1 State of ChainPaxos nodes.

```
1:  chain : array of nodes
2:  self : node                                        ▷ local node identifier
3:  $c_{nextok}$ : node                                ▷ next (unmarked) node in the chain
4:  $c_{sleader}$ : node                               ▷ supported chain leader
5:  marked : set of node                               ▷ nodes marked for removal (init : $\emptyset$)

6:  $np_{leader}$ : int                                ▷ special prepare number of the leader
7:  inst : map int × PaxosInst                         ▷ PaxosInst : ($n_a$, val, $n_{acpts}$, decided)

8:  submitted : set of requests                        ▷ requests submitted by the client
9:  pending : set of requests                          ▷ requests waiting to execute (leader only)

10: $max_{ack}$ : int                                  ▷ highest instance acknowledged
11: $max_{acpt}$ : int                                 ▷ highest leader initiated instance (leader only)
12: amLeader : bool                                    ▷ true if current leader
```

The second group maintains the information to run Paxos instances. This includes the prepare number (np_{leader}) that the leader can use to bypass the first phase of Paxos. Each replica also maintains a map (`inst`) with the information of Paxos instances including, for each instance, the highest prepare number (n_a) used by a leader to accept a value (`val`), the number of nodes that accepted `val` with n_a (n_{acepts}), and a boolean indicating if the instance was decided (`decided`).

The third group is used for managing client requests. It consists of two sets: `submitted` stores requests received from clients and not yet decided, and `pending` contains the requests received by the leader (redirected from itself or other replicas) but not yet submitted for ordering.

The final group of variables is used for clarity of presentation and stores information that could be derived from other variables, including the highest instance started by the leader (max_{acpt}), and the highest instance known to have a decided value accepted by all nodes (max_{ack}). Each node also keeps track of whether it is the current leader in `amLeader`.

3.3.2.2 Fault-free Execution

We now present the protocol in fault-free runs, assuming that all nodes agree on the configuration of the system. Algorithm 2 presents the ChainPaxos algorithm, with auxiliary functions detailed in Algorithm 3. The highlighted lines represent the logic used in faulty scenarios that require reconfiguration, which are detailed in the next section.

Algorithm 2 Paxos algorithm: message flow.

```
1:  procedure RECEIVE(<NEW_REQUEST,req>)
2:      $\texttt{submitted} \leftarrow \texttt{submitted} \cup \{req\}$
3:      SEND($c_{sleader}$,<REDIRECT_REQUEST,req>)
4:
5:  procedure RECEIVE(<REDIRECT_REQUEST,req>)
6:      if $\texttt{self} = c_{sleader}$ then                          ▷ Even if there is no quorum yet
7:          $\texttt{pending} \leftarrow \texttt{pending} \cup \{req\}$
8:
9:  function STARTINSTANCE
10:     $max_{acpt} \leftarrow max_{acpt} + 1$
11:     SEND(self,<ACCEPT,$max_{acpt}$,self,$np_{leader}$,pending,0,$max_{ack}$>)
12:     $\texttt{pending} \leftarrow \emptyset$
13:
14: procedure RECEIVE(<ACCEPT,$n_i$,ldr,$n_a$,val,$n_{acpts}$,$m_{ack}$>)
15:     if $np_{leader} \leq n_a$ then                               ▷ Has not seen higher prepare
16:         UPDATELEADERINFO(ldr,$n_a$)                          ▷ If a prepare was missed
17:         if $\nexists$inst[$n_i$] $\vee$ inst[$n_i$].$n_a < n_a$ then
18:             inst[$n_i$] $\leftarrow$ ($n_a$,val,$n_{acpts} + 1$,false)
19:         else                                                ▷ Repeated accept
20:             inst[$n_i$].$n_{acpts} \leftarrow$ MAX($n_{acpts} + 1$,inst[$n_i$].$n_{acpts}$)
21:         if inst[$n_i$].val = RemoveNode(node) then
22:             MARKFORREMOVAL($n_i$,node)
23:         if ISQUORUM($n_{acpts}$) $\wedge \neg$inst[$n_i$].decided then
24:             DECIDE($n_i$)
25:         DECIDEANDGCUPTO($m_{ack}$)
26:         FORWARD($n_i$)
27:
28: procedure RECEIVE(<ACCEPT_ACK,$n_i$>)
29:     DECIDEANDGCUPTO($n_i$)
```

Requests from clients can be received by any replica, and are redirected to the leader (Alg. 2, line 1), which stores them in a set of pending requests (Alg. 2, line 5). The leader, upon receiving a new request, starts a new instance by increasing the instance number and generating a new *accept* message (Alg. 2, line 9). The *accept* message contains the following information: (i) the instance number, which the leader tracks in max_{acpt}; (ii) the id of the leader; (iii) the prepare number, np_{leader}, used by the leader in its previous prepare message; (iv) the client request (i.e., operation); (v) the number of nodes which have accepted the value (n_{acepts}), initialized to 0; and (vi) the highest instance for which the decided value is known to have been accepted by all replicas ($\texttt{max}_{\texttt{ack}}$).

The leader is the first to handle the *accept* message of each instance, as it starts a new instance by sending the *accept* to itself (Alg. 2, line 11). Upon receiving an *accept* message for an instance (Alg. 2, line 14), a node stores the information for the instance, increasing the value of $\texttt{n}_{\texttt{acpts}}$ to indicate the node itself is accepting the value. If $\texttt{n}_{\texttt{acpts}}$ is greater than $n/2$, the message has already been accepted by a majority of nodes, and its value can be decided (Alg. 2, line 23). Otherwise, the value will be decided (and garbage-collected) when an *accept* message is received with $\texttt{m}_{\texttt{ack}}$ greater or equal to its instance number. This is performed in function `DecideAndGCUpTo` (called in Alg. 2, line 25 and defined in Alg. 3, line 39). This function traverses every (non-garbage-collected) instance up to instance $\texttt{max}_{\texttt{ack}}$, marking them as decided (if they were not yet), and garbage-collecting the information about them after their execution. This is safe since all instances up to $\texttt{max}_{\texttt{ack}}$ have been accepted by every node in the chain.

Finally, the node forwards the *accept* message (with the incremented $\texttt{n}_{\texttt{acpts}}$) to the next node in the chain. If the replica is the last node in the chain, it sends an *accept ack* message to the leader, signalling that every node in the chain has seen and accepted the instance. Upon receiving this message, the leader executes `DecideAndGCUpTo`, increasing its $\texttt{max}_{\texttt{ack}}$ which leads subsequent *accept* messages to trigger `DecideAndGCUpTo` in every node across the chain.

The nodes in the second half of the chain (starting from the $n/2^{th}$ node) can decide instances as soon as they receive the *accept* message, while the first $n/2$ nodes only decide (and execute an operation) after receiving an acknowledgement (the leader via an *accept ack* message, and the other nodes via the $\texttt{max}_{\texttt{ack}}$ value piggybacked in subsequent *accept* messages).

In a fault-free run, our protocol simply encodes the messages of Multi-Paxos in ChainPaxos messages. A ChainPaxos *accept* message sent by node `n` encodes the Multi-Paxos *accept* message and the *accept ack* messages of `n` and all nodes that precede it in the chain. It also encodes the *accept ack* messages of all nodes in the chain for all instances up to $\texttt{m}_{\texttt{ack}}$. A ChainPaxos *accept ack* message encodes the Multi-Paxos *accept ack* messages of all nodes in the chain.

3.3.3 Faults and Reconfiguration

To describe how faults and membership reconfigurations are handled in ChainPaxos, we begin by describing the mechanisms used by replicas to suspect other nodes (i.e., fault detection) and then discuss the steps taken by ChainPaxos to reconfigure the system, either keeping the current leader or when the leader is suspected. The main challenge faced by ChainPaxos is that, when using a chain topology, the failure of a single node leads the chain to break, making it impossible for messages to keep flowing along the chain, resulting in a system halt.

Algorithm 3 ChainPaxos algorithm: auxiliary functions.

```
1: function MarkForRemoval($n_i$,node)
2:     marked ← marked ∪ {$node$}
3:     if node = $c_{nextok}$ then                          ▷ We marked the closest unmarked node
4:         $c_{nextok}$ = NextNodeNotMarked(self, marked)
5:         for n ← $max_{ack}$ + 1, $n_i$ − 1 do                   ▷ Re-propagate accepts
6:             Forward(n)
7:
8: function Forward($n_i$)
9:     if $c_{nextok}$ = leader then
10:        Send($c_{nextok}$,<ACCEPT_ACK, $n_i$>)
11:    else
12:        Send($c_{nextok}$,<ACCEPT, leader, $n_i$, inst[$n_i$].$n_a$,
               inst[$n_i$].val, inst[$n_i$].$n_{acpts}$, $max_{ack}$>)
13:
14: function UpdateLeaderInfo(leader,$n_p$)
15:    if $np_{leader}$ < $n_p$ then
16:        amLeader ← false
17:        pending ← ∅
18:        $c_{sleader}$ ← leader                                  ▷ Set new leader
19:        $np_{leader}$ ← $n_p$                   ▷ Set the prepare number for the leader
20:        for req ∈ submitted do                   ▷ Redirect requests to new leader
21:            Send($c_{sleader}$,<REDIRECT_REQUEST, req>)
22:        marked ← {}
23:        $c_{nextok}$ = NextNodeNotMarked(self, marked)
24:
25: function Decide($n_i$)
26:    inst[$n_i$].decided ← $true$
27:    if inst[$n_i$].val = RemoveNode(node) then
28:        marked ← marked \ {$node$}
29:        chain ← chain \ {$node$}
30:    else if inst[$n_i$].val = AddNode(node) then
31:        chain ← chain ∪ {$node$}
32:        $c_{nextok}$ = NextNodeNotMarked(self, marked)
33:        if $c_{nextok}$ = node then                     ▷ Was added right next to me
34:            StateTransfer($c_{nextok}$, $n_i$)
35:    else
36:        SMRExecute(inst[$n_i$].val)
37:        pending ← pending \ {inst[$n_i$].val}
38:
39: function DecideAndGCUpTo($n_i$)
40:    for i ∈ inst ∧ $i \leq n_i$ do                    ▷ sequential iteration up to $n_i$
41:        if ¬i.decided then
42:            Decide($n_i$)
43:        inst ← inst \ {$i$}
44:    $max_{ack}$ ← $n_i$
```

3.3.3.1 Fault Detection:

We have implemented two mechanisms for fault suspicion. To pinpoint faults in the chain, each replica expects to receive periodic keep-alive messages from the following node in the chain. If a node does not receive the keep-alive for a configurable period of time, it suspects the node, and requests the leader to remove it, triggering a *Reconfiguration not involving the leader* (Section 3.3.3.2). In case the tail suspects the failure of the leader (which is its next node), it starts the process of taking leadership (Phase 1 of Paxos), and then starts the process of removing it. This effectively triggers a *Reconfiguration involving the leader* (Section 3.3.3.3).

We note that, as we assume an asynchronous system, suspecting a node does not necessarily mean that it failed, but rather that there is a chance it might have, as it can just be temporarily slow [92]. However, since a single failed (or just slow) node can block progress in the whole chain, the keep-alive mechanism is important to allow quick removal of suspected nodes, minimizing their negative impact on the overall throughput of the chain. Incorrectly removed replicas can later rejoin the system.

The second mechanism is based on the continuous flow of *accept* messages. If a replica does not receive an *accept* for a configurable period of time, it assumes that the leader is faulty and attempts to take leadership. If, during this process, the new leader cannot establish a connection to some other node (to send them the *prepare* message), it suspects and starts the process of removing them. To make sure this mechanism operates correctly even if the system is subjected to a low load, the leader issues periodic *accept* messages for a special `NoOP` operation if there are no client requests.

3.3.3.2 Reconfiguration not involving the leader:

We now explain how ChainPaxos reconfigures the chain by removing a suspected node that is not the leader.

When the leader is notified that node `n` is suspected, it starts an instance with `RemoveNode(n)` operation to remove node `n` from the chain. When the instance is decided, `n` is removed from the chain, updating the variables with the local configuration of the chain (`chain` and `marked`).

When a `RemoveNode` operation is being propagated, two actions need to be taken to guarantee correctness and progress: *i*) guarantee that all previous *accept* messages that might have been lost due to the failure of the node are forwarded to the next correct node (to reestablish the flow of those accept messages); and *ii*) guarantee that all subsequent *accept* messages are forwarded through the chain despite faulty nodes, until the `RemoveNode` operation is decided, removing the faulty node, and repairing the chain.

The former is implemented in `MarkForRemoval`, executed when processing an *accept* message for a `RemoveNode` operation (Alg. 2, line 22). The node to be removed is added to the set of marked nodes (Alg. 3, line 2). If the node to be removed is the next node that was not previously marked, it is possible that it failed to propagate previous messages

Algorithm 4 ChainPaxos algorithm: leader election.

```
1:  function TryToBecomeLeader
2:      $n_p$ = NextPrepareNum($np_{leader}$)
3:      Send($\forall n \in chain$, <PREPARE, $max_{ack} + 1, n_p$>)
4:
5:  procedure Receive(<PREPARE, $n_i, n_p$>)r
6:      if $np_{leader} \leq n_p$ then                         ▷ Has not seen higher prepare
7:          UpdateLeaderInfo(leader,$n_a$)                     ▷ New accepted leader
8:          $insts_{accepted}$ = GetAcceptedInstsFrom($n_i$)
9:          Send(node,<PREPARE_OK, $n_i, n_p, insts_{accepted}$>)
10:
11: procedure Receive(<PREPARE_OK, $n_i, n_p, insts_{accepted}$>)rep
12:     if $np_{leader} \leq n_p$ then                         ▷ Has not seen higher prepare
13:         RegisterPrepareOk($n_i, n_p, insts_{accepted}$)
14:         if HasPrepareOkQuorum($n_i$) then                  ▷ Became leader
15:             amLeader ← $true$                              ▷ Can now start new instances
16:             for ($a_{ni}, a_{na}, a_{val}$) ∈AcceptedInstsFrom($n_i$,$n_p$) do
17:                 Send(self,<ACCEPT, $a_{ni}$, self, $n_p$, req, 0, $max_{ack}$>)
18:                 $max_{acpt}$ ← $a_{ni}$
```

through the chain. Thus, the node sends to the next non-marked node any *accept* messages (or *accept ack* if the next non-marked node is the leader) for instances that have not yet been garbage collected (Alg. 3, line 5). This guarantees that, when healing the chain by bypassing faulty nodes, all *accept* messages will be received by all nodes that will not be removed from the chain, somewhat falling back to the pattern of Multi-Paxos[1].

The latter guarantee is provided by the `Forward` function (Alg. 3, line 8). This function forwards the *accept* message for a given instance to the next node. When one or more of the following nodes are marked to be removed (because a `RemoveNode` operation has been received, but has not yet been decided), the function forwards the *accept* message to the next non-marked node. This guarantees that a node that is to be removed in instance n_i will not vote for instances $n > n_i$.

3.3.3.3 Reconfiguration involving the leader:

ChainPaxos supports changing the leader by having a node become the leader at a given instance for that and all following instances by executing the first phase of Paxos.

This process is initiated in function `TryToBecomeLeader` (Alg. 4, line 1). The node selects a prepare number higher than any prepare number already seen in any instance, and sends a *prepare* message for instance $max_{ack} + 1$ directly to all nodes. Although this prepare is for a given instance, it will make the node leader of all instances from that point onward – thus, the prepare number must be larger than any previously used by any

[1]Note that this might lead to nodes receiving multiple *accept* messages for the same instance with the same prepare number from different nodes. This is addressed by considering the highest observed number of acks reported in these messages. This is safe because the forward process employed during recovery never generates cycles.

replica. We use $\texttt{max}_{\texttt{ack}} + 1$, since it guarantees that previous instances have already been accepted by every node, and all messages regarding those instances can be discarded. As such, nodes only need to maintain the single highest prepare number $\texttt{np}_{\texttt{leader}}$ ever received (instead of keeping a prepare for each instance). Since *prepare* messages need to have unique prepare numbers, this number includes an identifier of the node which is used to make sure that no two *prepare* messages from different nodes have the same np.

A *prepare* message for a given instance is rejected if the node has already seen a higher prepare number for any instance (either on *prepare* or *accept* messages). Otherwise, the usual Paxos logic is executed for this and all higher instances, with the corresponding *prepare ok* message being returned, which includes all previously accepted values (and corresponding prepare numbers) for the instance indicated in the *prepare* message and all following instances (Alg. 4, line 9). This is necessary as a successful prepare also makes the node the leader of all future instances. From this point until a *prepare* with a higher prepare number is received, the sender of the *prepare* message will be set as the supported leader $\texttt{c}_{\texttt{sleader}}$ and all pending and future client requests will be redirected to it (Alg. 3, line 14).

Upon reception of a quorum of *prepare ok* messages (Alg. 4, line 14), the node considers itself the new leader. It then executes the regular Paxos logic, but for multiple instances: for all instances for which accepted values exist, it uses the value with the highest associated prepare number as its proposal for that instance, and forwards the corresponding *accept* message over the chain. The regular protocol execution then resumes.

When a leader change occurs while a RemoveNode operation for a node *r* is being propagated through the chain, it is possible that the operation, while observed by a minority of replicas (that add *r* to their marked set), is not decided. Following the regular behavior of Multi-Paxos, the new leader might issue a different operation for that instance. (if the quorum of *prepare ok* messages gathered by the new leader did not include any of the replicas that had seen the operation to remove *r*). Such operations should be sent to *r* to ensure correctness. To do so, when a replica learns about the new leader it removes all nodes from the marked set and updates the $\texttt{c}_{\texttt{nextok}}$ variable, ensuring that messages flow across all nodes. (Alg. 3, line 14).

3.3.3.4 Adding a new replica:

For adding a node *n* to the chain, *n* sends a request to a replica with AddNode(n) operation as its value. The leader processes this request by starting an instance that is executed as any other instance of ChainPaxos

When the instance is decided, the node is added to the tail of the chain updating the local chain configuration (variables chain and $\texttt{c}_{\texttt{nextok}}$). Once the new node is added, it requests the current state (history of operations or snapshot) from another node at the instance in which the operation to add the node was decided. While this state transfers in the background, the new node can already participate in the following instances actively forwarding messages (although it can only locally execute and garbage collect operations after the completion of the state transfer).

3.3.4 Local Linearizable Read Operations

Providing cheap linearizable reads from any replica in an asynchronous system vulnerable to network partitions is one of the main goals of ChainPaxos. Being a challenging task, most SMR protocols only support linearizable reads by executing them as normal consensus operations or, in some cases [19], by contacting a quorum of replicas (and falling back to executing the read as a normal operation when conflicts occur). We now discuss how we leverage on the chain topology and our integrated membership management to provide linearizable reads without any added communication cost.

To provide linearizable reads, it is necessary to guarantee that the result of a read reflects a state that, at the moment the read is received, is at least as recent as the most recent state for which any node has returned a result (either for a read or for a write). The base intuition of our proposal is that a node can guarantee this property by waiting for a message to loop around the entire chain, making sure that the local node is as up-to-date as any node was at the moment the message started looping around the chain.

Based on this intuition, our solution for linearizable reads works as follows. Clients issue read operations to any replica in the chain. Upon receiving the operation, the replica locally registers that the operation depends on the lowest unseen consensus instance (but no information is sent to other nodes). For instance, if the highest instance that the replica has seen so far is 6 (regardless of it being decided or not), the read operation will depend on instance 7. Upon receiving the *accept ack* message to the consensus instance for which the operation depends on, the read operation is performed locally in the local committed state and the reply is sent to the client.

This protocol implements linearizable reads by enforcing the following properties:

1. A read r returns a value that is at least as recent as any value outputted by the protocol at the moment the read was received. By waiting that the following consensus instance is acknowledged and executing the read in the current local state, a replica is assured that the result of any read that was returned at any replica before the reception of r cannot be more recent than the result that will be returned for r – this follows from the properties of ChainPaxos, which guarantee that as messages loop the chain they make the state of replicas advance, so that the following replica in the chain is in a state that is at least as recent as the previous replica. As such, if some replica has already returned a value for state s_i, by waiting for the following consensus message to be acknowledged, which requires a full loop of an operation through the chain, the local replica state will be at least as recent as s_i. Due to the same reason, the result of a read will also reflect the result of any committed write operations, at the moment the read was received;

2. Upon a reconfiguration, a node that is partitioned from the chain will not return stale values. When a replica loses connection to the others, either it is eventually removed from the chain, preventing it from ever replying to client read operations, or it eventually reconnects to the other nodes, allowing it to continue responding to read requests. In the latter case, since the replica was not removed from the chain, no progress was made while it was partitioned, thus linearizability is not lost.

Our proposal trades a potentially higher latency (compared to executing a read as a normal operation) for the possibility of processing a read locally at any node, without additional consensus instances or communication steps. This leads to lower communication and processing overhead, and allows to balance the load of read operations across all replicas, leading to better overall performance. Under low load, write operations may be less frequent, which could delay read operations. We note however, that the head of the chain issues periodic *NoOP* operations if no write is received, as to show to the other replicas that the head is still correct, hence the maximum latency of reads in scenarios with a low load will be controlled by the frequency of these *NoOP* operations. Alternatively, to ensure faster read processing, a replica processing a read which has not received the message for the next consensus instance after some configurable timeout can forward the read to the leader to be executed as a normal operation (which in turn will allow other pending reads to complete).

3.4 Evaluation

This section reports the experimental evaluation of ChainPaxos in a broad range of scenarios. We start by assessing the performance and scalability in CPU-bound and network-bound settings (Section 3.4.2), and the impact of our novel read protocol (Section 3.4.3), using a replicated key-value store application under the YCSB workload [26], when compared with other consensus protocols. Then, we report the results of integrating ChainPaxos with ZooKeeper [46], by replacing the Zab [48] replication protocol (Section 3.4.4). Despite not being its main target deployment, we then study how ChainPaxos behaves in a geo-replicated setting (Section 3.4.5) Finally, we test the impact of reconfigurations in our integrated membership when compared to using an external coordination service (Section 3.4.6).

We have implemented a prototype of ChainPaxos in Java, using a framework for building distributed protocols, Babel [38], which relies on the Netty [89] framework for the communications module. Similarly to other authors [34, 83], to guarantee fairness in our comparisons, the other consensus protocols were implemented using the same codebase as ChainPaxos. This guarantees that the results are not influenced by specific implementation aspects, such as the programming language, client communication patterns or differences in optimizations (such as batching). Each protocol was implemented following the description presented in their respective publications as well as available code bases for EPaxos [82] and Ring Paxos [76]. For the latter, as proposed by the authors, we limit the number of concurrent instances the leader can start as a form of flow control to mitigate the loss of multicast messages and include a mechanism for recovering from lost messages.

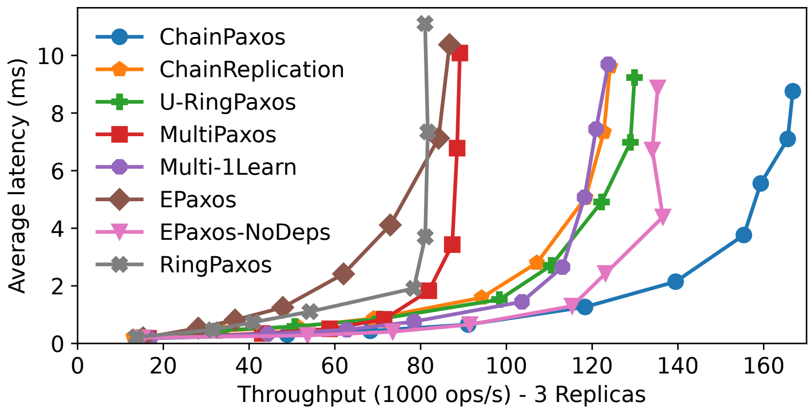
Each replica includes: an application (either the replicated key-value store or Zookeeper), which receives client requests, submits them for ordering, and replies to the client when the operation is executed; a proxy, serving as the intermediary between the application and the consensus protocol, also redirecting operations to the consensus leader when applicable and; the consensus solution itself, which receives operations from the proxy and notifies it once their ordering is decided.

3.4.1 Experimental Setup and Parameters

The experiments were conducted on the Grid5000 testbed [11], using a cluster of machines with an Intel Xeon Gold 5220 CPU with 18 cores and 96 GiB DDR4 RAM. Machines are connected through a 25 Gbps Ethernet-switched network. Each replica executes in its own machine, and clients (running YCSB [26]) execute on 3 independent machines (with multiple client threads per machine). Each client thread connects to a replica for executing operations in a closed-loop.

Every protocol is executed in similar conditions, except for Chain Replication that uses Zookeeper as the external management service (following [113]). For all protocols, the leader is elected at the start of the experiment and the protocols run multiple consensus instances in parallel. All results are the average of 5 independent runs, discarding the start and end periods of each experiment. In all results presented, the standard deviation between runs is always below 10%.

In addition to ChainPaxos, Chain Replication [113], and Ring Paxos [78], we report as: *EPaxos*, the execution of EPaxos [83] in a workload where all operations conflict (which is the same case of other baselines); and *EPaxos-NoDep*, the execution of EPaxos in a workload where no two operations conflict, which is equivalent to running multiple independent Paxos instances in parallel. We note that this is an unrealistic workload, as it would require all operations to be independent of each other, being presented only to provide the best (theoretical) results for a protocol following the strategy of EPaxos. *MultiPaxos* refers to the variant of Multi-Paxos [57] where acceptors forward their *accept ack* messages to all replicas, whereas *Multi-1Learn* represents the variant where acceptors only send the *accept ack* to the leader that, upon collecting a quorum of replies, issues a *decided* to all replicas - this protocol has a message flow equal to Raft [90] in the normal case. *U-Ring Paxos* [79] is a variation of Ring Paxos, using unicast instead of multicast, with a message flow similar to our solution and Chain Replication.



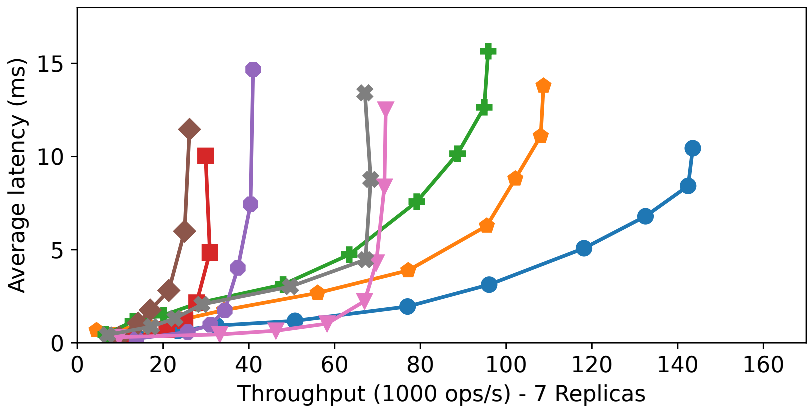


Figure 3.2: Performance for operations with 128 bytes (CPU bottleneck)

3.4.2 Performance in a Single Data Center

This section reports the results obtained in a single data center, running the YCSB benchmark with a replicated key-value store application. We study scenarios that attempt to saturate the CPU and the available bandwidth, by varying the size of the data stored in the key-value store.

3.4.2.1 CPU Bound

Figure 3.2 shows the performance of each protocol in a CPU-bound scenario. For this experiment, clients execute small (128 bytes) operations and no batching is employed (i.e., each operation is executed in an individual consensus instance). Clients connect uniformly at random to a replica, and receive a reply after the operation is executed in that replica. While this does not provide optimal latency for some solutions, it maximizes throughput by distributing the load of handling client requests as much as possible.

These results show that, by pipelining a single message per each operation through all replicas, ChainPaxos minimizes CPU usage, achieving the best performance and scalability. While leveraging on similar topologies, Chain Replication and U-RingPaxos perform worse, as they propagate some extra messages: acknowledge messages in the former, and proposals being propagated to the leader through the chain in the latter. These messages could be batched or piggybacked with a small penalty to latency. We note that the throughput of ChainPaxos with 7 replicas, which tolerates 3 faults, is higher than that of Chain Replication with both 3 and 7 replicas, which tolerate 2 and 6 faults, respectively.

For both versions of Multi-Paxos, the leader (and all replicas in regular Multi-Paxos) transmits and receives messages from, at least, a majority of replicas, resulting in higher CPU usage and lower performance. The impact of this effect increases with the number of replicas. For EPaxos, when all operations need to be ordered, the algorithm requires two rounds of communication, leading to a higher number of messages and lower throughput. The execution of *EPaxos-NoDeps* is similar to executing multiple parallel Multi-Paxos instances, distributing the load among replicas. This leads to a higher throughput than Multi-Paxos and EPaxos which, unlike ChainPaxos, also decreases with the number of replicas, as more messages need to be processed. Furthermore, we note that *EPaxos-NoDeps* is not totally ordering all operations, as other protocols do. RingPaxos is tricky to tune, as a single lost multicast message can stall the entire system. Even for our best configuration (with 150 simultaneous consensus instances), RingPaxos performance is worse than U-RingPaxos.

Overall, these results show that protocols using chain-based topologies (ChainPaxos, Chain Replication and U-RingPaxos) are able to lower the number of messages processed by each replica, allowing them to achieve higher throughput with a negligible latency overhead. Furthermore, this throughput degrades very slowly when increasing the number of replicas, while the throughput of other protocols with communication patterns based on Multi-Paxos degrades quickly, as the number of messages processed by each node depends on the number of replicas in the system. This is relevant for supporting critical systems with high throughput and availability requirements.

3.4.2.2 Network Bound

Figure 3.3 presents the performance in a network-bound scenario. For this experiment, the bandwidth of replicas is limited to 1Gbps, with clients issuing 2048 byte operations, saturating the bandwidth of the replicas without saturating their CPU. For saving the bandwidth consumed in redirects and maximizing the bandwidth available for the consensus protocol, all clients connect directly to the leader/head (uniformly distributed in both EPaxos variants).

Results show that ChainPaxos, ChainReplication, and U-RingPaxos, by only receiving and transmitting each operation once, achieve maximum use of available bandwidth. For these solutions, the replicas were consuming approximately 900 Mb/s of both inbound and outbound bandwidth. This allows the system to maintain its performance with an increasing number of replicas. For Multi-Paxos, since the leader needs to transmit each

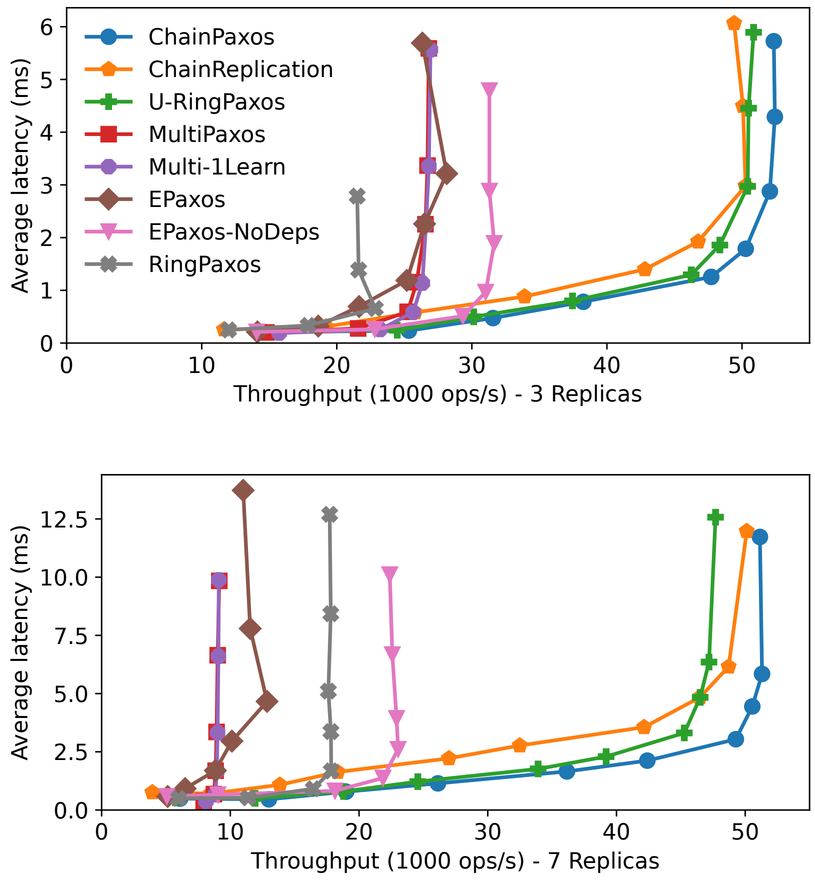


Figure 3.3: Performance with network bottleneck.

operation to all other replicas, its bandwidth usage is disproportionately higher than that of other replicas, limiting their throughput. Furthermore, the throughput decreases with the number of replicas. EPaxos versions suffer from the same issue, but since EPaxos uses multiple leaders, it distributes the load of the leader across all nodes, leading to better scalability than Multi-Paxos. *EPaxos-NoDeps* requires fewer communication steps, having higher throughput, but still far from ChainPaxos. For RingPaxos, the higher message size results in more frequent message losses. Even configuring the number of concurrent instances to 20 as to achieve the best results, the performance is substantially lower than that of ChainPaxos and Chain Replication.

3.4.2.3 Latency with a fixed throughput

Figure 3.4 shows the latency with a fixed load – clients execute 9000 operations per second, using payloads of 128 bytes. In this experiment, clients are setup to minimize latency: in RingPaxos and Multi-Paxos clients connect directly to the leader; in EPaxos clients connect to all replicas uniformly; in Chain Replication and U-RingPaxos clients connect to the tail; and in ChainPaxos to the replica in the position $n/2 + 1$. Error bars

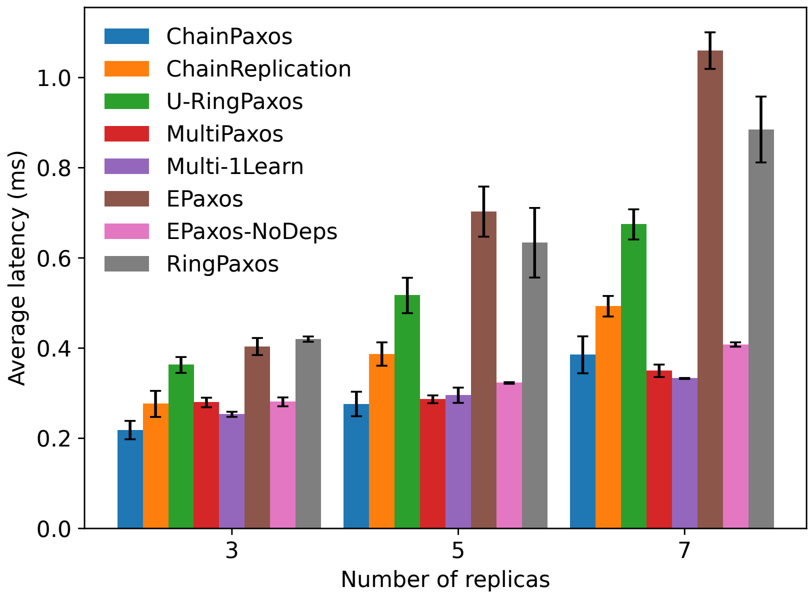


Figure 3.4: Latency under low load.

present the standard deviation of the results.

The results show that, with 3 replicas, ChainPaxos and Multi-Paxos variants exhibit the lowest latency, since they can respond to client requests after a single communication step. With increasing numbers of replicas, the latency of ChainPaxos increases, while the latency of both Multi-Paxos variants remains mostly unaffected. Since both U-Ring Paxos and Chain Replication require more communication steps until a reply is generated, their latency is always higher than ChainPaxos. We note that ChainPaxos with 5 replicas presents a similar latency to Chain Replication with 3 replicas, in which case both tolerate the failure of 2 replicas. In EPaxos, since conflicts lead to extra communication rounds, the variant where all operations conflict (*EPaxos*) naturally shows higher latency. For RingPaxos, multicast message drops (which happen even without saturation) and retransmissions lead to higher latency. Even with 7 replicas, ChainPaxos presents a latency that is only slightly higher than that of Multi-Paxos (which shows the minimum possible latency).

3.4.3 Performance of Read Operations

Figure 3.5 shows the impact of ChainPaxos's novel linearizable read approach in workloads with different read ratios and payloads of 128 bytes. The throughput of executing reads as normal (consensus) operations (*Chain Reads*) is constant, regardless of the ratio of read operations. For our novel approach (*Local Read*), the throughput is much higher than executing reads as normal operations, and the throughput scales both with the ratio of read operations and with the number of replicas. This is explained by the fact that as reads impose no overhead to the consensus protocol and the load is distributed evenly among replicas, more replicas can process more reads. The high throughput of

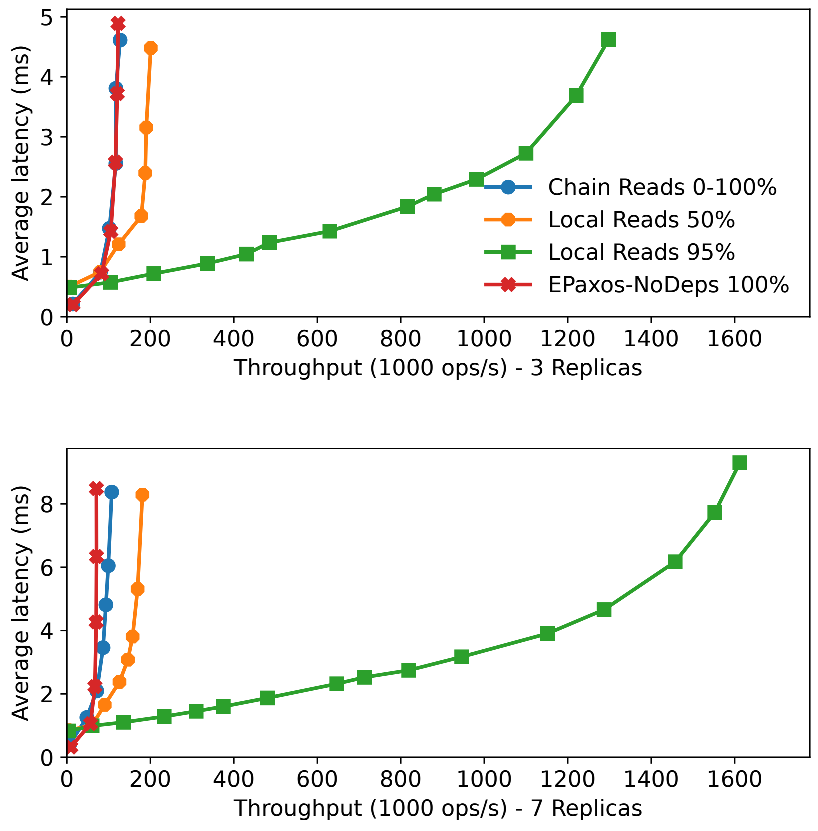


Figure 3.5: Performance with read operations.

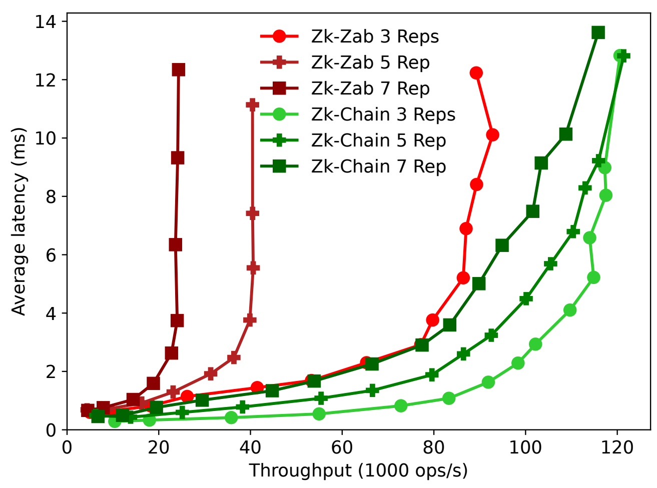
ChainPaxos's local linearizable reads comes at the cost of a small additional latency under low load.

For comparison, we include the results of *EPaxos-NoDeps*, the Paxos-based protocol with the best performance in the previous results. The results show that ChainPaxos achieves significantly higher throughput than *EPaxos-NoDeps*. We further evaluate the impact of our linearizable reads approach in a practical scenario in the following section.

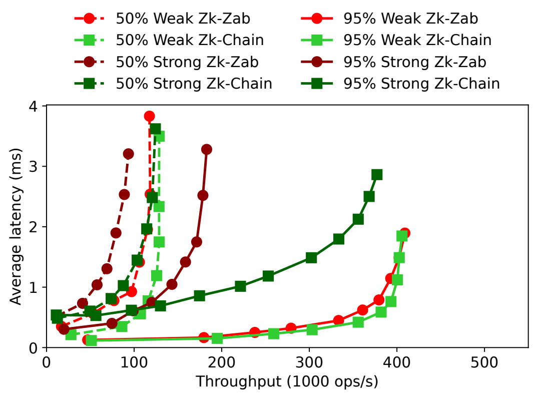
3.4.4 Zookeeper case-study

To evaluate the performance of our protocol in a more realistic scenario, we adapted ZooKeeper [46], a very popular distributed coordination service, to use ChainPaxos as its consensus protocol, instead of Zab [48]. While some features were not implemented, such as ephemeral nodes, our implementation fully supports creating, updating, and retrieving *znodes* (the base data structure of Zookeeper). We evaluated the performance of our implementation (*ZK-Chain*) against the original ZooKeeper using Zab (*ZK-Zab*), in a setup similar to the CPU bound scenario of Section 3.4.2. The results are presented in Figure 3.6.

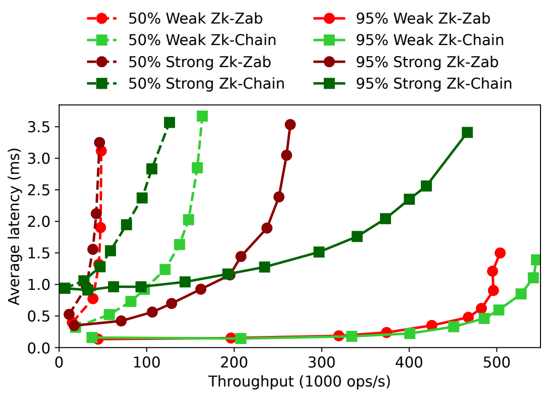
For a write-only workload (Figure 3.6(a)), the results show that ChainPaxos achieves



(a) Write-only workload



(b) Mixed workload with 3 replicas



(c) Mixed workload with 7 replicas

Figure 3.6: Performance of Chain-based Zookeeper vs original Zookeeper.

higher throughput than the original Zookeeper, with the difference increasing drastically with the number of replicas. These results are consistent with the results ChainPaxos and Multi-Paxos in the CPU-bound scenario, as Zab uses a communication pattern similar to Multi-Paxos.

Figures 3.6(b) and 3.6(c) present mixed workloads (50% and 95% of read operations), with both weak and strong reads. *Weak reads* represent the regular reads of ZooKeeper, where a replica replies with its current state, allowing for stale data to be served (e.g., with late replicas and under network partitions). *Strong reads*, in our solution, are executed using linearizable local reads. While ZooKeeper does not support linearizable reads, the authors suggest issuing a *sync* operation before a read as a close approximation of linearizability in most cases. The results show that, unlike with Zab, the strong reads with ChainPaxos scale to a throughput similar to executing weak reads. Overall, the throughput with ChainPaxos is higher than with Zab for the same setting, and the difference increases with the number of replicas.

3.4.5 Performance in a Geo-Replicated Setting

While not the main target deployment type of ChainPaxos, for completeness, we evaluate how the protocol behaves in a geo-replicated setting by emulating an environment with 5 sites. Using the Linux `tc` command, we limited the bandwidth to 1Gbps, and increased latency to the values shown on Table 3.1 (in milliseconds), extracted from a service which provides real-time information about the latency between AWS datacenters [25].

In these experiments, we did not use Ring Paxos, since IP multicast is typically unavailable across data centers. The replica in site A is always the leader/head. Experiments with 3 replicas use sites A, B, and C. Clients connect to the replica that leads to the best performance: the leader for Multi-Paxos; evenly distributed for EPaxos, and; the tail for chain-based solutions.

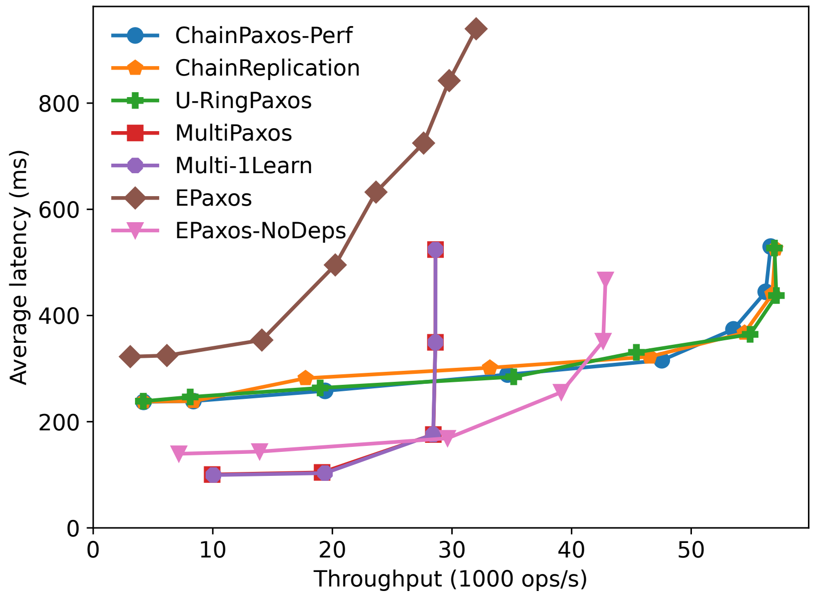
Table 3.1: Latencies between data centers

Sites	A	B	C	D	E
North Virginia (A)	-	92	127	204	186
Frankfurt (B)	88	-	210	288	279
São Paulo (C)	122	207	-	338	359
Sydney (D)	211	292	325	-	161
Seoul (E)	188	287	309	156	-

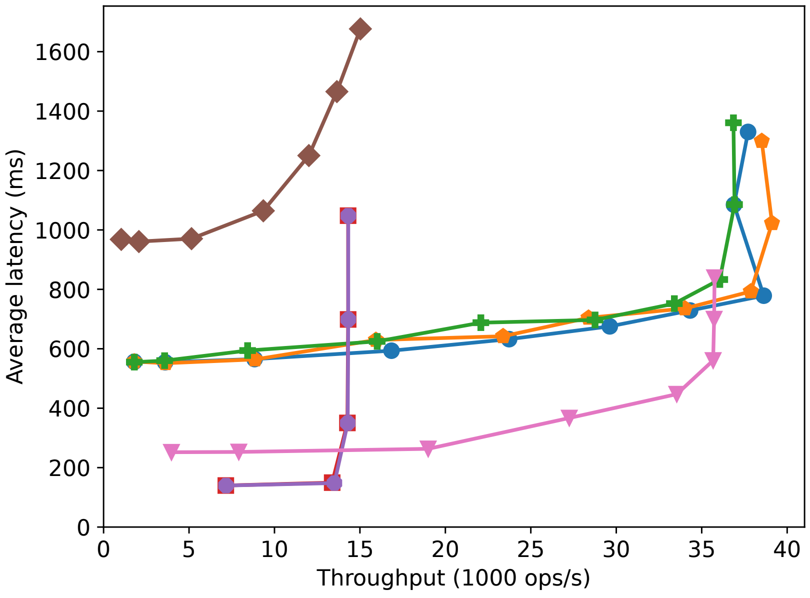
Figure 3.7 presents the throughput and latency when using all available bandwidth. As within a single data center, ChainPaxos, Chain Replication, and U-Ring Paxos are able to make optimal use of available bandwidth, providing higher throughput than other protocols that order all operations. The EPaxos variant without inter-operation dependencies is able to maintain its throughput with a varying number of replicas, since the cost of transmitting each operation to all nodes is divided among the multiple leaders. However, we remind the reader that this configuration of EPaxos provides weaker guarantees than the other alternatives. As for latency, the latency of the chain-based solutions degrades as the number of replicas increases, as it takes longer for the messages to traverse the chain. For a high number of replicas, Multi-Paxos variants provide lower latency, as communication between the leader and other replicas proceeds in parallel, although with, at most, half the throughput of ChainPaxos.

3.4.6 Impact of Reconfiguration

In our final experiment we evaluate the impact of reconfiguration, comparing ChainPaxos that uses its own integrated management mechanism and Chain Replication that



(a) 3 Replicas



(b) 5 Replicas

Figure 3.7: Performance in geo-replicated setting.

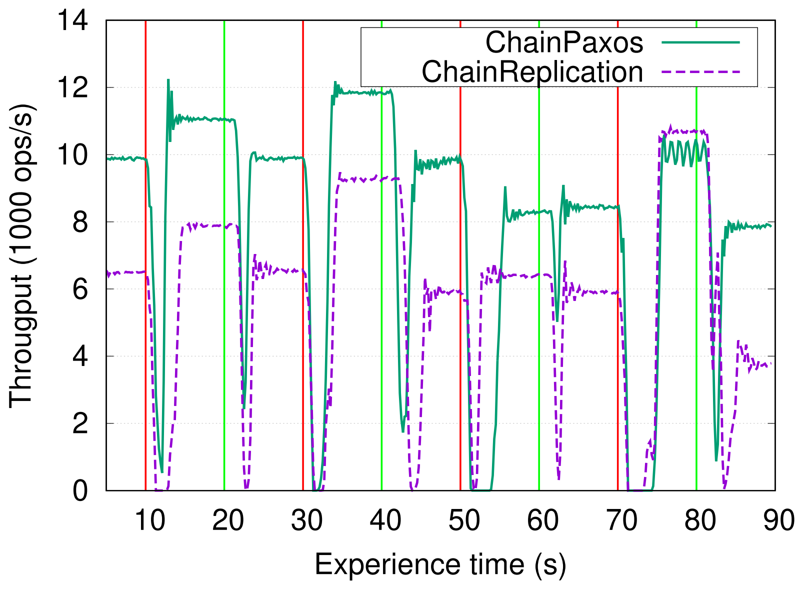


Figure 3.8: Reconfiguration.

uses an external management scheme based on Zookeeper (executing on dedicated machines). We conduct these experiments in the geo-replicated scenario with independent Zookeeper instances at sites A, B, and D. This distribution minimizes latency for replicas without a local Zookeeper replica. We used 1s timeouts to suspect the failure of another node (both in ChainPaxos and in ZooKeeper).

Experiments run for 90 seconds. Every 10 seconds the following reconfiguration events occur (denoted by vertical red lines for replica failures and green lines for replica additions): 10s) the tail node fails; 30s) the middle node fails; 50s) the head/leader fails; 70s) the head and middle replicas fail simultaneously. Replicas are added at 20s, 40s, 60s, 80s, in sites where a replica had previously failed. Clients issue operations to a random active replica to distribute the load.

Figure 3.8 shows the throughput observed during the experiments. Despite Chain Replication using additional resources (3 extra machines executing Zookeeper), it takes more time to perform a reconfiguration than ChainPaxos, particularly when adding replicas to the set (green vertical lines). This happens because any reconfiguration has to be coordinated through Zookeeper. However, ChainPaxos takes longer to perform the reconfiguration when the leader fails because it resorts to the regular communication pattern of Paxos, whereas Chain Replication only fetches the new leader from Zookeeper. When the leader and middle nodes fail simultaneously (70s), both solutions take the same time to perform reconfiguration because ChainPaxos can handle both reconfigurations in parallel (albeit using two operations), whereas Chain Replication performs two sequential reconfiguration steps with Zookeeper. In general, ChainPaxos handles reconfiguration faster than Chain Replication without the cost of requiring additional machines to run the external management system, while avoiding the vulnerabilities to network partitions that can compromise the safety of the system [3].

3.5 Related Work

Many state machine replication protocols have been proposed over the years, many of which are based on the Paxos algorithm [20, 61, 83, 90, 78, 77]. In this section, we discuss the most relevant works that have influenced the design of ChainPaxos, grouping them into categories based on which aspects they aim to optimize: minimizing latency (Section 3.5.1), reducing communication cost (Section 3.5.2), distributing the load (Section 3.5.3), and supporting linearizable reads (Section 3.5.4).

3.5.1 Minimizing latency

Multiple Paxos variants enforce a variety of techniques to optimize the latency of executing operations. In FastPaxos [55], clients send Accept messages directly to multiple acceptors, skipping the leader. Generalized Paxos [56] extends FastPaxos by additionally allowing non-interfering requests to execute in different orders. While in the optimal case these protocols can achieve low latency, collisions in client requests result in additional round trips, hindering performance and resulting in the opposite effect. SwiftPaxos [55] improves on this by decreasing the number of round trips required to execute operations in the presence of conflicts. Flexible Paxos [45] uses different quorum sizes for executing operations (akin to read-write quorum systems [115]), reducing the size of accept quorums and decreasing the latency of accepting operations in the fast path, however, this results in decreased fault-tolerance. While ChainPaxos only matches the fast path latency of these protocols when configured to tolerate a single fault, it requires each replica to handle only one message per operation. In contrast, the $O(n)$ message complexity of Paxos leader and learners in these solutions results in much lower throughput and scalability, as we showed in our evaluation.

3.5.2 Communication cost

Similar to ChainPaxos, some variants employ chain (or ring) topologies to decrease communication cost. Ring Paxos [78] sends *Accept* messages to all replicas using IP-multicast, with responses being propagated through a ring. IP-multicast limits the operation of the protocol across data centers and negatively impacts the performance under high load when messages are lost. Chain Replication [113] is an *SMR* algorithm, developed for synchronous systems, where replicas are organized in a *chain* and write operations are forwarded from the head to the tail, with acknowledge messages travelling the opposite way. This approach has the advantage that all replicas send and receive the same number of messages for executing an update. Similarly, U-Ring Paxos [79] propagates messages in a ring topology, with acknowledge messages being forwarded from the tail to the head. These solutions require an external coordination service (e.g., Zookeeper [46]) to reconfigure the system when faults occur, leading to higher operational costs, slower fault-handling, potentially lower fault-tolerance (dependent on that of the external service), and vulnerability to network partitions [3]. ChainPaxos, while having a similar communication pattern, further reduces the number of messages processed by each replica, while handling reconfigurations and faults in an integrated and efficient way.

3.5.3 Distributing the load

Other variants of Paxos try to distribute the load across replicas, by allowing all replicas to behave as a leader. Mencius [75] pre-assigns the leader of each instance to a different node. While providing better throughput, the overall availability suffers since the failure of *any* replica will cause the system to stop until another replica takes over. In Egalitarian Paxos (EPaxos) [83], any replica can commit operations and non-conflicting operations execute in different orders. When there is no conflict, operations commit in a single communication round. Multi-Ring Paxos [77] (based on Ring Paxos) uses a similar approach, while taking advantage of the ring topology to minimize communication. Such solutions, however, relax the consistency guarantees of the system, by assuming that some operations are independent and can be executed in different orders. When operations are not independent and conflict (which is often the case in SMR), these protocols require extra rounds of communication, resulting in worse throughput and latency. Atlas [34] improves on this by allowing some conflicting operations to execute in a single round, while Tempo [33] relies on a decentralized timestamping mechanism to avoid ordering conflicts.

Despite trying to distribute the load across all replicas, these protocols still require nodes to send and receive $O(n)$ messages (even when considering independent operations). In contrast, ChainPaxos minimizes the overall load imposed by the protocol, having $O(1)$ message complexity, while also distributing the load across replicas.

3.5.4 Linearizable reads

While the previous solutions focus on optimizing the performance of write operations, in replicated systems, where reads are more frequent than writes, it is important to reduce the cost of read operations to improve overall performance. In this section we discuss the most relevant works that proposed techniques to improve the performance of read operations, starting by discussing why solutions designed for synchronous systems do not work in asynchronous environments, and then presenting the most relevant solutions for asynchronous systems.

Synchronous systems. Chain Replication [113] proposes to execute linearizable reads by contacting a single node: the tail of the chain. When the tail fails, as detected by an external coordination service, clients fall back to the previous node in the chain to continue reading the system state. This solution, however, was designed for a synchronous model where failures can be reliably detected. In an asynchronous system, linearizability can be violated, as the tail can become isolated and be excluded from the chain without knowing, while still serving (outdated) reads. To avoid this, for each read, either the tail or the client would need to contact the external coordination service to verify the current configuration of the chain, which is too expensive. In [113], the authors mention that the coordination mechanism needs to stop clients during reconfigurations, which is unfeasible under network partitions.

Asynchronous systems. Due to a similar reason, solutions based on Paxos cannot execute read operations by simply contacting any replica (including the leader), usually requiring to run a consensus instance for ordering read operations or, in special cases, to contact a quorum of replicas. However, some alternative read schemes to improve replication performance have been proposed. In Smarter [15], reads execute on a single replica, but require a special *whats_my_view* message to be sent to all replicas to gather a majority of replies confirming that no reconfiguration took place concurrently with the read operation. In [41], reads are executed on a single replica, however at the cost of requiring writes to execute in two phases. CRAQ [109] improves reads in Chain Replication by allowing to read from any replica in an asynchronous model, however it only provides per-object linearizability, and for SMR it would require all reads to contact the tail whenever there is a write executing.

In contrast with all these solutions, ChainPaxos includes a novel technique to execute linearizable reads on a single replica, in an asynchronous environment, without ever requiring any additional communication costs. Furthermore, it allows any replica to process reads, thus distributing the read load across all replicas.

3.6 Discussion

When considering the goals of this contribution (Section 3.2), ChainPaxos successfully achieves all of them. By minimizing the number of messages processed by each replica, and distributing the load of both read and write operations across all replicas, ChainPaxos achieves higher than existing state-of-the-art solutions, while simultaneously scaling better with the number of replicas: for write operation the throughput degradation is minimal compared to other solutions and; for read operations, the throughput increases with the number of replicas, while providing full linearizability. On the other hand, when considering fault tolerance and reconfigurations, ChainPaxos is able to handle both aspects in an integrated and efficient way, without the need for external coordination services.

3.6.1 Trade-offs and Limitations

As the reader may have noticed by now, when designing replication protocols, there is never a one-size-fits-all solution that is optimal for all scenarios, with ChainPaxos being no exception. While ChainPaxos provides a significant improvement in performance and scalability over existing solutions, there are some trade-offs that were made in the design of the protocol.

Latency While the chain topology of ChainPaxos is key to its performance, both for write and read operations, it also means that the latency of operations increases with the number of replicas, as messages need to traverse at least half of the chain. For scenarios where the throughput is not the main concern, and SMR is only employed for fault-tolerance, for instance in distributed lock services [18], ChainPaxos may not be the best solution. On the contrary, for scenarios where high throughput is required, such as in distributed databases or distributed messaging systems, the performance of ChainPaxos pays off the latency overhead, which is not very significant, especially in co-located deployments.

Effects of Replica Failures Another trade-off in ChainPaxos is the effect of failed replicas on the availability of the system. Having all operations go through the chain one replica at a time, while maximizing throughput, also means that the failure of a single replica can stall the system temporarily, as the chain is broken. In contrast, in solutions based on the communication pattern of Multi-Paxos, the failure on non-leader replicas does not affect the system, as long as a majority of replicas are still available. This, however, does not mean that ChainPaxos is not fault-tolerant, as its integrated reconfiguration mechanism is able to remove failed replicas in a timely manner, as shown in our evaluation.

Summary

In this chapter, we presented ChainPaxos, a novel state machine replication protocol that relies on a chain topology to maximize the throughput of both write and read operations. ChainPaxos features an integrated fault-tolerance and reconfiguration mechanism, without the need for external coordination services. Contrary to existing solutions, increasing the number of replicas in ChainPaxos does not result in a significant decrease in performance, with the throughput of linearizable read operations actually increasing with the number of replicas. ChainPaxos contributes to the goal of this thesis by providing a scalable and fault-tolerant solution for the co-located/datacenter *replica distribution level*.

In the next chapter, we focus on the geo-replication level, by studying the limitations of existing solutions and proposing and evaluating solutions to address them.

Publications and Artifacts

This contribution resulted in the publication of a research paper in the 2022 USENIX Annual Technical Conference (USENIX ATC 22) [37]. Additionally, a fully functional implementation of ChainPaxos was made available as open-source software, and is available at `https://github.com/pfouto/chain`.

4 Scalable Geo-Replicated Causal Consistency

This chapter presents the second contribution of this thesis. Contrary to the other contributions, we do not propose a new replication protocol, but rather study the limitations of existing geo-replicated causal consistency solutions, and propose solutions to overcome them. The main goal of this chapter is to study the performance impact of replicating the centralized sequencers of existing causal consistency solutions, in order to ensure fault-tolerance and scalability with minimal performance impact.

Following the goal of this thesis and the definitions presented in Chapter 2, this contribution addresses the geo-replication *replica distribution level*, focusing on causal consistency with (static) partial replication.

We start by providing some background on causal consistency enforcement in geo-replicated systems, followed by the goals of this contribution. We then present the use-case chosen for this work, followed by the design and implementation of the solution. We follow by presenting the experimental results and discuss the implications of our findings. Finally, the chapter concludes with a summary of the related work, and a final discussion on the results obtained.

4.1 Background

When considering modern large-scale Internet services, that focus on serving users across the globe with low latency and high availability, it is common to deploy data replication solutions that span multiple datacenters in different geographic locations. Due to the prohibitive coordination cost of ensuring strong consistency across a geo-replicated large-scale system, such services often rely on weaker consistency models, mainly eventual consistency. In fact, nowadays, all main cloud providers offer fully-managed eventually consistent data stores to their clients, such as Cassandra [52], DynamoDB [30], and BigTable [21].

Due to providing the strongest possible consistency guarantees without sacrificing availability during network partitions [6], causal consistency has been the focus of multiple research efforts in the last few years, with many geo-replicated causal consistency solutions being proposed [36, 93, 4, 70, 74, 69, 32, 2]. However, industry adoption of this consistency model has been slow, with only a few services, such as MongoDB [112] and Redis [95] providing support for it. The most likely reason for this slow adoption, as claimed by both industry [112] and academia [80] authors, is the performance overhead of enforcing causal consistency. While the consistency guarantees of causal consistency are desirable, performance is often the deciding factor when choosing weaker consistency models, with occasional cases of incorrectness being an acceptable trade-off.

The main challenge of causal+ consistency is tracking and enforcing the *happens-before* relation between operations. While existing solutions employ drastically different approaches, the goal is always to ensure that no client will observe the effects of an operation before the effects of its causal dependencies. For this, data replication solutions with causal consistency are usually implemented in the following manner:

- Each node (or process) maintains metadata that represents the causal history of the operations it has made visible to clients (i.e., its current state).
- When a client operation is received in a node, it is tagged with metadata that represents the causal history of the operation, which can either be derived from the metadata of the node or from metadata kept by the client representing its causal history (usually the operations it has seen). This metadata is then propagated to other nodes along with the operation.
- When an operation is received by a node (from another node), it can only be made visible to clients (i.e., applied to the local data replica) after all the operations in its causal history have been made visible. This is usually enforced by comparing the metadata with which the operation was tagged with the metadata of the node. If the operation can not be made visible, the node simply waits until the missing operations are received (and made visible)[1].

Solutions providing causal consistency utilize a wide range of techniques to track causal dependencies between operations, with different trade-offs between performance, false dependencies, and remote visibility times. As the number of nodes in each datacenter of a geo-replicated system can be in the hundreds or more, the cost of coordination between all nodes can be prohibitively high. As such, many solutions restrict the communication between nodes in different datacenters, by employing two different replication protocols: one between nodes in the same datacenter, and another between different datacenters[2].

[1]Some solutions employ a different approach, where all operations received are immediately applied to the local data replica. Causal dependencies are then checked when a client executes a read operation, which may cause the read to be blocked until the missing operations are received. Solutions following this *optimistic* approach are less common [80, 106], and are discussed in Section 4.7.

[2]As previously discussed in Section 2.4.2, some solutions even abstract the intra-datacenter replication completely [69, 32, 31, 16], focus solely on the inter-datacenter replication

Such solutions require a way to gather and propagate updates from the local datacenter to the others. For this, the common approach is to employ a centralized component in each datacenter, which is responsible for ordering operations, controlling when they can be executed, and propagating them to other datacenters. This component is often called a *sequencer*. The main advantage of this approach is that it reduces the coordination cost of the protocol, by avoiding the coordination and metadata overheads of tracking causal dependencies directly between the hundreds or thousands of replicas in modern geo-replicated deployments, instead relying on a single component in each datacenter.

In this work, we chose to focus on this class of solutions for two main reasons: *i*) existing causal consistency solutions that support partial replication (which is a key requirement for geo-replication) rely on either on sequencers [36, 16] or on global stabilization [116, 107] (discussed in Section 4.7) and *ii*) sequencer-based solutions provide a good trade-off between metadata size and remote visibility times, typically by relying on vector clocks [35] (as opposed to solutions based on global stabilization, which suffer from high visibility times).

Despite their popularity and advantages, solutions based on sequencers have a critical limitation: the sequencer is a single point of failure in the entire datacenter. As such, it is crucial to replicate this component using state machine replication to ensure fault-tolerance, preventing the entire datacenter from halting. However, it is also a single point of contention for all operations, meaning that it needs to be replicated using a high throughput protocol to avoid becoming a performance bottleneck for the entire system. This is a challenging problem, as most existing state machine replication protocols do not scale well with the number of replicas, displaying significant performance degradation.

4.2 Motivation and Goals

While different solutions leveraging *sequencers* employ different techniques to ensure causal consistency (e.g. in Saturn [16] the sequencer is not on the critical path of local client operations, while in C3 [36] it is), the sequencer always ends up being a single contention point for inter-datacenter propagation of operations. While replicating the sequencer using *SMR* is commonly mentioned as a requirement for fault-tolerance in such solutions [4, 93, 36, 16], the practical implications of doing so are rarely (if ever) studied, with experimental evaluations relying on a single non-replicated sequencer in each datacenter.

Having this in mind, we focus on studying the performance implications of replicating the sequencer of geo-replicated causal consistency solutions, both in terms of throughput and remote operation visibility times (i.e., the time it takes for an operation to be made visible across all remote datacenters). For this purpose, we use C3 [36] as a case study, and in particular we replicate its sequencer using two different state machine replication protocols, and study the performance of the resulting solutions (compared to the original non-replicated solution).

The goal is to demonstrate that it is possible to deploy causal consistency solutions based on sequencers in a scalable and fault-tolerant manner, which is a key requirement (and current limitation) for their adoption in real-world deployments.

Considering the goals of this thesis, of building scalable and fault-tolerant distributed data storage systems with consistency guarantees, we now formalize these aspects in the context of geo-replicated causal consistency solutions:

Fault-tolerance Replicating the sequencer to ensure it is fault-tolerant, avoiding halting the entire datacenter in case of a single failure, is a crucial requirement for the practical deployment of any geo-replicated causal consistency solutions based on sequencers. While state machine replication is commonly mentioned as the solution for this problem, implementing it may not be trivial, especially when considering the performance implications of doing so, leading us to the next point.

Scalability Being a single point of contention for all operations, the performance of the sequencer has a direct impact on the performance of the entire system. In particular, we are interested in two metrics: (i) the throughput of local operations (i.e., operations that were issued by clients in the local datacenter), which is directly impacted by the throughput of the sequencer when it is in the critical path of client operations; and (ii) the remote visibility times of operations (i.e., the time it takes for an operation to be made visible in remote datacenters), which is directly impacted by the performance of the sequencer, as it is always responsible for ordering, propagating and executing operations that originated in other datacenters.

To optimize both metrics, we must minimize the performance impact of replicating the sequencer, by ensuring that a performant and scalable state machine replication protocol is used, by designing the integration between the sequencer and the state machine replication protocol in a way that minimizes overhead, and by leveraging on SMR optimizations to maximize throughput, such as batching.

Additionally, this contribution also has a secondary goal: it validates the applicability of our SMR protocol (ChainPaxos), that resulted from the first contribution, as a building block to provide performant and scalable fault-tolerance to other data replication solutions.

4.3 Baseline

The first step in this contribution is to choose a causal consistency geo-replicated baseline solution, based on sequencers, to use as a case study. For this, we picked C3 [36].

C3 is a geo-replicated data storage system that provides causal consistency while supporting partial replication. Its main goal is to provide a balanced trade-off between the throughput of local operations and the remote visibility times of operations. For this, it relies on a sequencer component in each datacenter, encoding causality information in the form of vector clocks, which allows concurrency in the execution of remote operations, decreasing visibility times. Additionally, C3 is built on top of Cassandra [52], a popular open-source NoSQL database, providing causal consistency with the sequencer approach by intercepting all operations issued by clients to Cassandra nodes, enriching them with causal metadata, and ensuring they are executed in a correct causal order.

The main reason for choosing C3 as a baseline is that it is the only geo-replicated causal consistency solution that we are aware of that matches all the following criteria:

- It is a sequencer-based solution.
- It supports partial replication, which is a key requirement for geo-replication.
- The sequencer component is in the critical path of local operations, allowing us to study the performance impact of replicating the sequencer on the throughput of such operations.
- Its design and causality tracking metadata representation (vector clocks) allow concurrency in the execution of remote operations (as opposed to solutions that rely on a total order of operations), which puts a higher demand on the performance of the sequencer, as it must be able to handle a higher throughput of operations.
- It contains an open-source implementation that includes the integration with an industry-standard datastore (Cassandra), allowing us to evaluate the performance of the solution in a realistic environment. [3]

4.3.1 Architecture of C3

To better understand the work presented in this chapter, it is important to first understand the architecture of C3 and how write operations are processed in the system.

Figure 4.1 shows an overview of the architecture of C3, with the yellow boxes representing Cassandra nodes, and the green circles representing the sequencer component. Note that this architecture is similar to other sequencer-based solutions. C3 leverages the data partitioning and replication mechanisms of Cassandra. Data is divided into *partitions*, with each datacenter being responsible for a subset of the partitions (partitions can be replicated in multiple datacenters). Inside each datacenter, Cassandra uses consistent hashing to distribute the individual data items of each partition across the nodes in the datacenter, replicating each item in multiple nodes to ensure fault-tolerance. As such, a write operation over a data item is always replicated to multiple nodes in the local datacenter, and to multiple nodes in each remote datacenter where the partition to which the data item belongs is replicated.

In C3, every Cassandra node in a datacenter is always connected to the local sequencer. Operations in C3 are propagated between Cassandra nodes directly (both intra and inter datacenter), however, they are never executed directly upon reception. Instead, the

[3]Additionally, as the authors of C3, we know the implementation details of the solution, facilitating its adaptation to our needs.

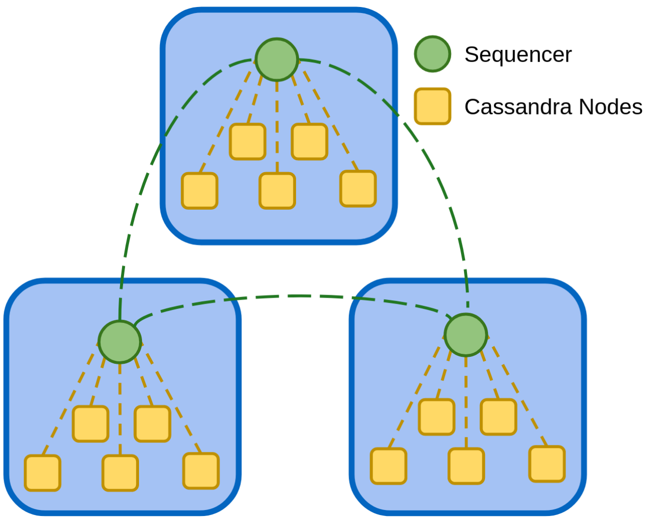


Figure 4.1: Architecture of C3 (similar to most sequencer-based solutions)

Cassandra node on which the operation was issued sends a *label* (i.e., metadata about the operation including a unique identifier) to the local sequencer, which is then responsible for ordering local operations, instructing the local Cassandra nodes on when to execute them, and propagating the metadata to the sequencers in other datacenters, all while ensuring that the causal dependencies between operations are respected.

4.3.2 Write Execution in C3

We now present how write operations are processed in C3, specifically focusing on the role of the sequencer, as this is the component that we will replicate in this contribution.

Algorithm 5 presents the state of each sequencer in C3. The first two variables (lines 1-2) represent the sequencer's view of the system, with Π being the set of all sequencers (one per datacenter), and `LocalDC` being the name of the datacenter in which the sequencer is located.

The next three variables (lines 3-5) are the key to tracking causality in C3. Each operation received from the local Cassandra nodes is tagged with a unique and incremental identifier, which is assigned by the sequencer using the `WriteCounter` variable. The `ExecutingClock` and `ExecutedClock` represent vector clocks, with one position per datacenter, tracking, per datacenter, the highest operation identifier that is currently executing and has already executed, respectively, in the local datacenter, being used to tag received operations with their causal dependencies, and to calculate when an operation can be executed in the local datacenter.

The variable `WaitingLabels` (line 6) stores the labels of operations that have been received by the sequencer (both local and remote) but can not yet be executed in the local datacenter due to missing causal dependencies. Since (unfortunately), operations are not executed instantaneously and atomically in all nodes of a datacenter, the sequencer must also keep track of operations that are currently being executed (i.e, that the sequencer has already instructed the local Cassandra nodes to execute, but have not finished being executed), which is done using the `CurrentWrites` variable (line 7). Finally, as C3 allows for concurrency in the execution of both local and remote operations, the sequencer may receive acknowledgments of operations that finished executing in an order different from the order defined by the unique identifiers assigned to them by the sequencer[4]. To handle this, the sequencer tracks such operations using the `AheadExecutedOps` variable (line 8), which is then used to update the `ExecutedClock` vector clock.

Algorithm 5 State of C3 Sequencer

```
1: Π : set of processes            ▷ Represents the sequencer processes in the system (one per dc)
2: LocalDC : string                ▷ Name of this node's datacenter

3: WriteCounter : int              ▷ Counter used to assign identifiers to write operation labels
4: ExecutingClock : map string × int   ▷ Tracks the highest locally executing write for each dc
5: ExecutedClock : map string × int    ▷ Tracks the highest locally completed write for each dc

6: WaitingLabels : map string × queue of labels   ▷ Stores labels to be executed from each dc
7: CurrentWrites : map writeId x label            ▷ Tracks currently executing writes in the local dc
8: AheadExecutedOps : set of labels               ▷ Tracks operations that finished executing out of order
```

Algorithm 6 presents the pseudo-code of a simplified version of the sequencer in C3, focusing on the processing of write operations, where the main goal is to ensure that operations are only executed by the local Cassandra nodes after all their causal dependencies have been executed. When a client executes a write operation in a Cassandra node of C3, the execution proceeds as follows:

1. The Cassandra node propagates the operation to all other nodes that replicate the data item, both in the local datacenter and in remote datacenters. The operation is not executed immediately in any node. Simultaneously, it generates a label, `DatastoreWriteLbl`, which includes a unique identifier for the operation (assigned by Cassandra), and the list of datacenters and nodes within them that must execute the operation. The label is then sent to the local sequencer.

2. Upon receiving the label from the Cassandra node (line 1), the sequencer assigns its own incremental unique identifier to the operation (`lblId`), and tags it with the current value of the `ExecutingClock` vector clock, which represents the operation's causal dependencies. The sequencer then sends a `WriteLbl` message to all sequencers in the other datacenters that must execute the operation, and a `ClockLbl` message to all other sequencers. It then processes the label itself.

[4]Note that this out-of-order execution does not violate causality, as operations that execute concurrently in C3 are those that are not causally dependent, allowing them to be executed in any order.

Algorithm 6 Execution of writes in C3

```
1: procedure RECEIVE(<DatastoreWriteLbl,writeId,targets>)
2:     lblId ← ++WriteCounter
3:     lblDeps ← ExecutingClock
4:     for all p ∈ Π do
5:         dc ← GetDatacenter(p)
6:         if dc ∈ targets then
7:             SEND(p,<WriteLbl,LocalDC,lblDeps,lblId,writeId,targets[dc]>)
8:         else
9:             SEND(p,<ClockLbl,LocalDC,lblDeps,lblId>)
10:    Trigger Receive(<WriteLbl,LocalDC,lblDeps,lblId,writeId,targets[LocalDC]>)
11:
12: procedure RECEIVE(<WriteLbl,sourceDC,lblDeps,lblId,writeId,localTargets>)
13:     enqueue(WaitingLabels[sourceDC],
            {WriteLbl,sourceDC,lblDeps,lblId,writeId,localTargets})
14:     Trigger CheckWaitingLabels
15:
16: procedure RECEIVE(<ClockLbl,sourceDC,lblDeps,lblId>)
17:     enqueue(WaitingLabels[sourceDC], {ClockLbl,sourceDC,lblDeps,lblId})
18:     Trigger CheckWaitingLabels
19:
20: function CHECKWAITINGLABELS
21:     for all queue ∈ WaitingLabels do
22:         {lblType, sourceDC, lblDeps, ...} ← peek(queue)
23:         while SmallerOrEqual(lblDeps, ExecutedClock) do
24:             if lblType = WriteLbl then
25:                 Trigger ExecuteWrite(sourceDC, lblId, writeId, localTargets)
26:             else
27:                 Trigger ExecuteClock(sourceDC, lblId)
28:             dequeue(queue)
29:             {lblType, sourceDC, lblDeps, ...} ← peek(queue)
30:
31: function EXECUTEWRITE(sourceDC, lblId, writeId, localTargets)
32:     ExecutingClock[sourceDC] ← lblId          ▷ lblId is always ExecutingClock+1
33:     CurrentWrites[writeId] ← {sourceDC, lblId, localTargets}
34:     for all target ∈ localTargets do
35:         SEND(target, <ExecuteWrite, writeId>)
36:
37: procedure RECEIVE(<WriteAck, writeId>)
38:     if WriteCompleted(CurrentWrites[writeId]) then          ▷ Executed on a quorum
39:         {sourceDC, lblId, localTargets} ← CurrentWrites[writeId]
40:         Trigger FinishedExecuting(sourceDC, lblId)
41:
42: function EXECUTECLOCK(sourceDC, lblId)
43:     ExecutingClock[sourceDC] ← lblId          ▷ lblId is always ExecutingClock+1
44:     Trigger FinishedExecuting(sourceDC, lblId)
45:
46: procedure FINISHEDEXECUTING(sourceDC, lblId)
47:     if ExecutedClock[sourceDC]+1 ≠ lblId then
48:         AheadExecutedOps[sourceDC]←AheadExecutedOps[sourceDC]∪{lblId}
49:     else
50:         ExecutedClock[sourceDC] ← lblId
51:         ExecutedClock[sourceDC] ← CheckAheadOps(AheadExecutedOps[sourceDC])
52:         Trigger CheckWaitingLabels
```

3. Upon receiving either a `WriteLbl` (line 12) or a `ClockLbl` (line 16) message, the sequencer enqueues the label in the `WaitingLabels` queue, and triggers the function `CheckWaitingLabels`.

4. The `CheckWaitingLabels` function (line 20) is responsible for checking if the sequencer can execute any of the labels (which represent write operations) that are currently in the `WaitingLabels` queue. As labels for each datacenter are received in the order they were generated, labels in a queue will never depend on labels that are behind them in the same queue. As such, this function only checks if the first label on each queue can be executed, moving to the next label in the queue if it can. A label can be executed if all its causal dependencies have already been executed, which is checked by comparing the label's dependencies with the `ExecutedClock` vector clock (line 23). If a label can be executed, the sequencer triggers the `ExecuteWrite` or `ExecuteClock` functions, depending on the type of the label.

5. The `ExecuteWrite` function (line 31) starts by incrementing the `ExecutingClock` vector clock for the datacenter that generated the label. This leads to any subsequent local operation to depend on the current operation. It then updates the `CurrentWrites` variable with the label, and sends a `ExecuteWrite` message to all Cassandra nodes in the local datacenter that must execute the operation.

6. When a Cassandra node finishes executing an operation, it sends a `WriteAck` message back to the sequencer. Upon receiving it (line 37), the sequencer checks if the operation was executed by a quorum of local Cassandra nodes. If it was, the `FinishedExecuting` function is triggered. The node additionally sends an acknowledgment to the Cassandra node to which the operation was issued by the client, allowing it to respond to the client, and to garbage collect any metadata related to the operation.

7. The `FinishedExecuting` function (line 46) is responsible for updating the vector clock `ExecutedClock`, taking into account operations that finished executing out of order. After updating the vector clock, it triggers the `CheckWaitingLabels` function, which will check if any other operation can be executed.

4.4 Replication Architecture

Having chosen the baseline, the next step is to replicate the sequencer component of C3. The main idea is to provide (non-byzantine) fault-tolerance to the sequencer by replicating it using the state machine replication approach, by applying every state-changing operation to all replicas in the same order, ensuring that all replicas transition through the same sequence of states.

However, adapting an existing solution to use a replicated sequencer is a non-trivial task, as there are many possible design choices that can be made which can have an impact on the performance of the system. Additionally, we want to study the performance implications of replicating the sequencer of geo-replicated causal consistency solutions, and not specifically C3, meaning that the integration between the (replicated) sequencer and the state machine replication protocol must be as generic as possible, without relying on any C3-specific optimizations, in order to be applicable to other solutions.

We now present and justify the design choices made in the materialization of the replicated sequencer component of C3, taking into account that the goal is to minimize the overhead of SMR on the performance of the sequencer:

Communication between sequencer and datastore nodes When considering the communication between the multiple replicas of the sequencer and the datastore nodes (Cassandra nodes in the case of C3), the simplest solution would be to have all datastore nodes connected to a single sequencer replica, the leader of the SMR protocol, being responsible for both receiving and sending messages to all datastore nodes.

However, this would put a significant load on the leader, as it would have to handle the same load as a non-replicated sequencer, plus the overhead of the SMR protocol. As such, in our solution, we chose to split the datastore nodes uniformly among the sequencer replicas. This has two main advantages: i) it splits the costs of communicating with the datastore nodes equally among all sequencer replicas, and ii) it allows each sequencer replica to batch operations independently, before propagating them through the SMR protocol, which is a common optimization in SMR protocols which can significantly increase throughput at the cost of slightly increasing latency.

Intra-datacenter communication Following the state machine replication approach, replicas of the sequencer in each datacenter do not explicitly communicate with each other. Instead, they are maintained synchronized by the SMR protocol, which ensures that all labels submitted to all sequencer replicas are ordered and executed by that order in all replicas.

As such, every label received in a sequencer replica from a datastore node is not immediately handled, but instead is first submitted to the SMR protocol, being processed only after being ordered by the protocol. This ensures that all sequencer replicas handle labels in the same order, which results in all replicas tagging each label with the same causal metadata, and in no label being lost in case of a replica failure.

When labels related to operations that must be executed in nodes of the local datacenter are processed by a sequencer replica, that replica is responsible for informing the local datastore nodes which are connected to it when the operation can be executed. Those nodes then execute the operation and send an acknowledgment back to the sequencer replica, which is again submitted to the SMR protocol before being processed.

Similarly to local operations, remote operations received by a sequencer replica from a remote datacenter go through the state machine replication protocol, ensuring that all replicas in the remote datacenter handle them in the same order.

In summary, this approach consists of having sequencer replicas submit all received messages to the SMR protocol before processing them. This can result in the SMR protocol being forced to order a large number of messages, which can impact the performance of the sequencer, as a single write operation can result in multiple messages being ordered (the operation itself plus all acknowledgments from the local datastore nodes). This effect can, however, be successfully mitigated by employing batching, as we show further ahead in the evaluation section.

Inter-datacenter communication When considering the communication between replicas of the sequencer in different datacenters, we chose to follow an approach similar to the non-replicated sequencer, where a single sequencer replica in each datacenter is responsible for communicating with the sequencer replicas in the other datacenters. As most SMR protocols are leader-based, we opted to use the leader of the SMR protocol as the sequencer replica responsible for this communication.

This approach has the disadvantage of putting a higher load on the leader compared to the other sequencer replicas, however, this load is not different from having a single (non-replicated) sequencer. Additionally, employing an alternative solution where all sequencer replicas in all datacenters communicate with each other would not only require a much more complex coordination protocol, but would also force messages received by the sequencer replicas of a datacenter to be manually reordered before being processed. This is because many sequencer-based solutions rely on the order in which messages are propagated between datacenters to determine the order in which they must be executed [36]. In fact, in some systems, the metadata of an operation is not enough to determine its causal dependencies [16], making it impossible to reorder received messages without additional information.

Figure 4.2 shows the resulting architecture of C3 with a replicated sequencer. Following the design choices made, the figure displays the Cassandra nodes (in yellow) distributed among the sequencer replicas (in green) of their local datacenter, and a single sequencer replica in each datacenter responsible for communicating with the sequencer replicas in the other datacenters.

4.5 Implementation

In order to be able to study the performance implications of replicating the sequencer in causal consistency solutions, we implemented a replicated version of the sequencer

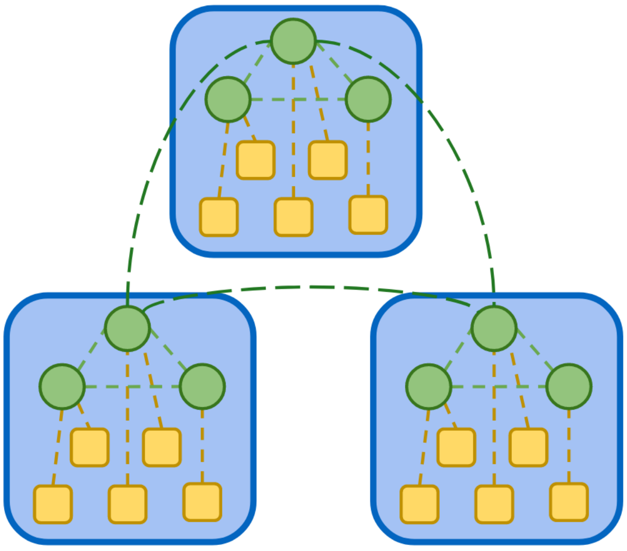


Figure 4.2: Architecture of C3 with replicated sequencer

component of C3. This implementation is based on the open-source code of C3 [36], and is integrated with 2 different state machine replication protocols: ChainPaxos (the result of the first contribution) and Multi-Paxos, which is the most common SMR solution used in production systems [52, 18, 46]. For both SMR protocols, we used the open-source artifact resulting from the first contribution of this thesis.

Our implementation includes all the necessary components to deploy and evaluate the system, including:

- The replicated sequencer component of C3, which follows all the design choices discussed in the previous section, and includes both the ChainPaxos and Multi-Paxos implementations.
- An adapted version of Cassandra, which integrates with the replicated sequencer.
- Client libraries that integrate with the YCSB [26] benchmarking tool, to allow evaluating the system.
- Fault tolerance and recovery mechanisms, leveraging on the SMR protocol, and on the mechanisms provided by Cassandra itself (such as retrying operations in case of failures).

4.5.1 Replicated Sequencer

Being the main goal of this contribution to study the performance implications of replicating the sequencer of causal consistency solutions, in this section we will focus on the integration of the sequencer with the SMR protocol, and the execution of write operations. For this, we will use the pseudo-code of the non-replicated sequencer of

C3 as a baseline (algorithms 5 and 6), showing the differences in the execution of write operations when replicating this component.

Algorithm 7 State of each C3 sequencer replica

```
1: Π : set of processes               ▷ Represents the sequencer processes in the system (multiple per dc)
2: AmLeader : boolean                 ▷ Whether this replica is the leader of the local datacenter
3: DatastoreNodes : set of processes  ▷ Stores the set of connected datastore (Cassandra) nodes
4: LocalDC : string                   ▷ Name of this node's datacenter

5: WriteCounter : int                 ▷ Counter used to assign identifiers to write operation labels
6: ExecutingClock : map string × int  ▷ Tracks the highest locally executing write for each dc
7: ExecutedClock : map string × int   ▷ Tracks the highest locally completed write for each dc

8: WaitingLabels : map string × queue of labels  ▷ Stores labels to be executed from each dc
9: CurrentWrites : map writeId x label          ▷ Tracks currently executing writes in the local dc
10: AheadExecutedOps : set of labels            ▷ Tracks operations that finished executing out of order
```

Algorithm 7 presents the state of each sequencer replica in the replicated version of C3. The highlighted lines represent the new or modified variables when compared to the non-replicated version. The first difference is that the variable Π (line 1) now contains multiple processes per datacenter, representing all sequencer replicas in each. In our implementation, there is a static set of sequencer replicas, which are known by both Cassandra nodes and other sequencers, however, in a real-world deployment, a coordination service such as Zookeeper [46] could be used to manage and discover the set of sequencer replicas and current leader of each datacenter. A new variable, `amLeader` (line 2), is used to determine if the replica is the current leader of the local datacenter (i.e., the replica responsible for communicating with the leader replica of the other datacenters). The variable `DatastoreNodes` (line 3) stores the set of datastore (Cassandra) nodes connected to the sequencer replica, as they are split among the sequencer replicas.

Note that, due to the usage of state machine replication, every other variable in the state of the sequencer will always be synchronized between all replicas, even without explicit communication between them. This is because every state-changing operation is submitted to the *SMR* protocol, ensuring that all replicas transition through the same sequence of states.

Algorithm 8 presents the (simplified) pseudo-code of the execution of write operations in the replicated version of the sequencer of C3. The most significant difference when compared to the non-replicated version (algorithm 6) is that every message received both from the datastore nodes and from other sequencer replicas is first submitted to the *SMR* protocol, by calling the `Order` function, before being processed (lines 1 to 11). This ensures that all replicas handle the same sequence of messages, progressing through the same sequence of states. Handling the messages is done by the `Process` functions, which are triggered by the *SMR* protocol after the messages are ordered. The `Process` functions are similar to the original `Receive` functions, with two main differences:

Algorithm 8 Execution of writes in replicated C3

```
procedure RECEIVE(<DatastoreWriteLbl,writeId,targets>)
    ORDER(<DatastoreWriteLbl,writeId,targets>)

procedure RECEIVE(<WriteLbl,sourceDC,lblDeps,lblId,writeId,localTargets>)
    ORDER(<WriteLbl,sourceDC,lblDeps,lblId,writeId,localTargets>)

procedure RECEIVE(<ClockLbl,sourceDC,lblDeps,lblId>)
    ORDER(<ClockLbl,sourceDC,lblDeps,lblId>)

procedure RECEIVE(<WriteAck, writeId>)
    ORDER(<WriteAck, writeId>)

procedure PROCESS(<DatastoreWriteLbl,writeId,targets>)
    lblId ⟵ ++WriteCounter
    lblDeps ⟵ ExecutingClock
    if AmLeader then
        for all p ∈ leaders(Π) do
            dc ⟵ GetDatacenter(p)
            if dc ∈ targets then
                SEND(p,<WriteLbl,LocalDC,lblDeps,lblId,writeId,targets[dc]>)
            else
                SEND(p,<ClockLbl,LocalDC,lblDeps,lblId>)
    Trigger Receive(<WriteLbl,LocalDC,lblDeps,lblId,writeId,targets[LocalDC]>)

procedure PROCESS(<WriteLbl,sourceDC,lblDeps,lblId,writeId,localTargets>)
    enqueue(WaitingLabels[sourceDC],
        {WriteLbl,sourceDC,lblDeps,lblId,writeId,localTargets})
    Trigger CheckWaitingLabels

procedure PROCESS(<ClockLbl,sourceDC,lblDeps,lblId>)
    enqueue(WaitingLabels[sourceDC], {ClockLbl,sourceDC,lblDeps,lblId})
    Trigger CheckWaitingLabels

function EXECUTEWRITE(sourceDC, lblId, writeId, localTargets)
    ExecutingClock[sourceDC] ⟵ lblId            ▷ lblId is always ExecutingClock+1
    CurrentWrites[writeId] ⟵ {sourceDC, lblId, localTargets}
    for all target ∈ localTargets do
        if target ∈ DatastoreNodes then
            SEND(target, <ExecuteWrite, writeId>)

procedure PROCESS(<WriteAck, writeId>)
    if WriteCompleted(CurrentWrites[writeId]) then        ▷ Executed on a quorum
        {sourceDC, lblId, localTargets} ⟵ CurrentWrites[writeId]
        Trigger FinishedExecuting(sourceDC, lblId)
```

- When a `DatastoreWriteLbl` message is received (line 13), only the current leader replica propagates the message to the other leader replicas in the other datacenters.
- When an operation is ready to be executed (line 33), each sequencer replica is responsible for instructing the (possibly empty) subset of Cassandra nodes connected to it to execute the operation.

Note that some functions that were present in the non-replicated version of the sequencer (Algorithm 6 - page 61) were omitted, as no changes were necessary to adapt them to the replicated version.

4.6 Evaluation

We now present the experimental evaluation of our implementation of C3 with the replicated sequencer component. In this section, we study the performance impact of replicating the sequencer, when compared with the non-replicated version, and how different state machine replication protocols can impact the performance. To better understand the results, we measure not only the throughput of the system, but also the latency perceived by clients and the visibility times of operations in remote datacenters, which are affected by the queuing of remote operations when the sequencer is overloaded with operations. We also study how the number of replicas, the use of batching with varying sizes, and the load in the system impact these key performance metrics.

4.6.1 Experimental Setup

Our experiments were conducted in a cluster of 8 machines, each having 2 AMD EPYC 7343 processors with 64 threads and 128 GB of memory, connected by a 20 Gbps network. We deployed a docker swarm across all machines, with an overlay network connecting all containers. To emulate a geo-replicated deployment, we emulated 7 datacenters, where each is composed by 7 Cassandra nodes, and a varying number of sequencer replicas. Each physical machine represents an entire datacenter, hosting 7 docker containers, each executing a Cassandra node, and possibly a sequencer replica. Each docker container is assigned 8 CPUs and 16 GB of memory. The remaining physical machine is used both to coordinate the experiments and to run the clients, using the YCSB [26] benchmarking tool to issue operations across all datacenters.

As for the data model, data is split into 7 partitions, one for each datacenter, with each partition being replicated across 5 of the 7 datacenters. Inside each datacenter, the individual data items are replicated in 3 of the 7 Cassandra nodes, using the consistency hashing mechanisms provided by Cassandra.

Since reads in C3 are executed by communicating directly with the Cassandra nodes in the local datacenter, without the need to go through the sequencer, we focus our evaluation on write operations, which need to go through the sequencer to be executed both locally and in remote datacenters. Therefore, in our experiments, a varying number of clients in each datacenter issues write operations to data items in the partition of that datacenter, in a closed-loop fashion (i.e., each client issues a new operation as soon as the result from the previous one is received).

To properly evaluate the overheads in client-perceived latency and visibility times of remote operations, in these experiments we do not emulate network latencies between datacenters, as it would not allow us to isolate the effects of replicating the sequencer. Regardless, we discuss further ahead how the latency from clients to datacenters and between datacenters can affect the overall performance of the system.

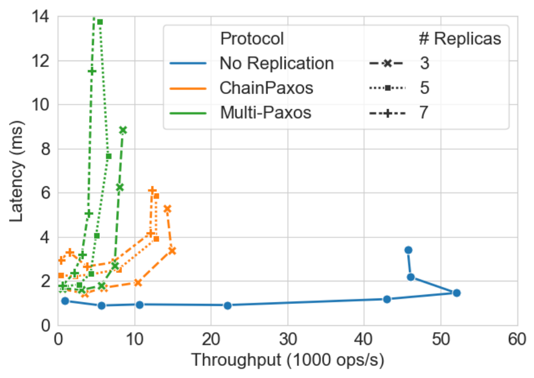
4.6.2 Single datacenter

We start by evaluating a simpler scenario, with a single datacenter composed of 7 Cassandra nodes. In this experiment, there is only a data partition. Each write operation is propagated to the sequencer by the Cassandra node which receives it from the client, which then propagates it to 3 Cassandra nodes, and receives an acknowledgment from each of them. When using a replicated sequencer, both the message with the operation and the acknowledgment messages are submitted to the SMR protocol, resulting in a total of 4 messages being ordered for each client write operation.

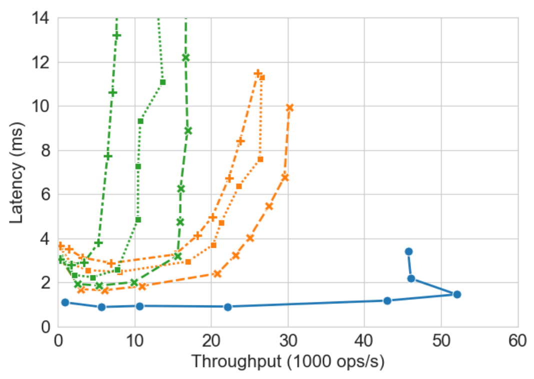
Figure 4.3 shows the results of this experiment. Each figure shows the throughput and latency of operations as perceived by clients, with each point of each line representing an increasing number of clients, which increases the load in the system. In each figure, we show as a continuous blue line the performance of the sequencer without replication, with the dotted lines showing the replicated sequencer with a varying number of replicas (three, five, and seven) and using two different state machine replication protocols, ChainPaxos and Multi-Paxos. The different figures show the results for different batch sizes, ranging from no batching to a batch size of 10.

The first figure (Figure 4.3(a)) shows that, without using batching (i.e., each operation issued to the SMR protocol results in an individual consensus round), the throughput of the sequencer suffers significantly from the overhead of the SMR protocol, regardless of the number of replicas or the protocol used. This is expected, as it means that executing a single client operation requires 4 consensus rounds, resulting in a significant processing overhead for the sequencer.

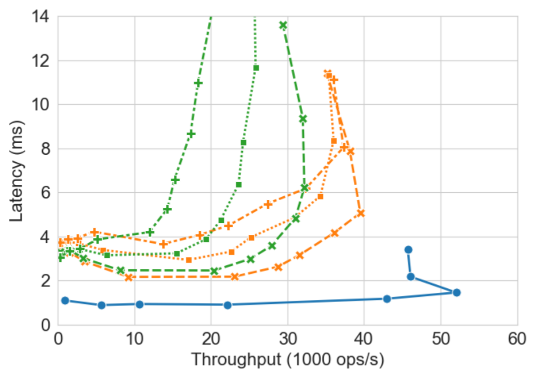
However, as we increase the size of batches (figures 4.3(b) through 4.3(d)), we can see that this throughput overhead is mitigated, with the throughput of the replicated sequencer approaching the throughput of the non-replicated solution. This is because batching operations allows to amortize the cost of the SMR protocol, by ordering multiple operations in a single consensus round. While the throughput of the replicated sequencer is still lower than the non-replicated solution, with a batch size of 10, ChainPaxos is able to provide a throughput of around 80% of the non-replicated solution. Note that batching is a parameter of the SMR protocol, having no effect on the non-replicated solution.



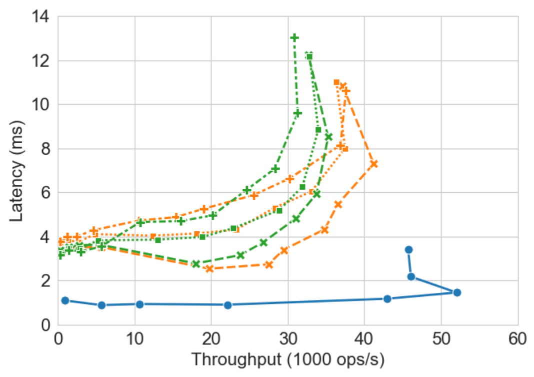
(a) No batching



(b) Batch size of 2



(c) Batch size of 5



(d) Batch size of 10

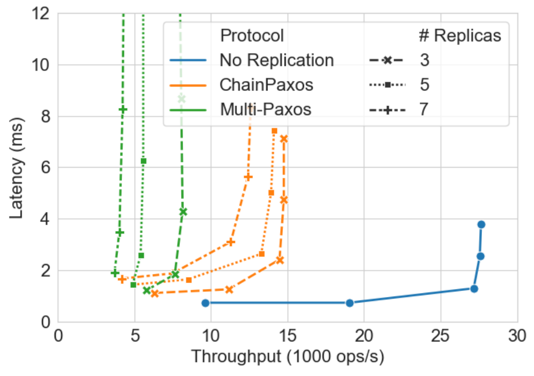
Figure 4.3: Performance of the replicated sequencer in a single datacenter.

While allowing the replicated solution to approach the performance of the non-replicated solution, batching has the disadvantage of increasing the latency of operations, as the sequencer replicas must wait for a batch to be filled before starting the ordering process. This effect is visible in the results we obtained, where the overall client-perceived latency increases as the batch size increases. However, this latency increase is not significant, being mostly under 10 milliseconds before the system reaches saturation. In the real world, the latency of client operations will likely be dominated by the round-trip time between the client and the datacenter, with the overhead of the sequencer becoming negligible.

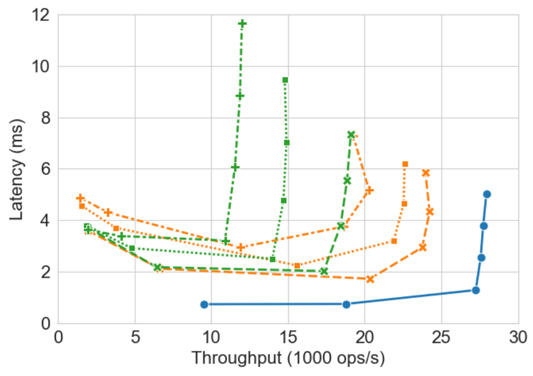
When comparing the performance of ChainPaxos against Multi-Paxos, we see that ChainPaxos is able to provide a much higher throughput, especially with higher numbers of replicas. This difference, however, becomes less significant as we increase the batch size since, as discussed above, it masks the cost of the *SMR* solutions albeit with a small penalty to the client-perceived latency.

4.6.3 Multiple datacenters

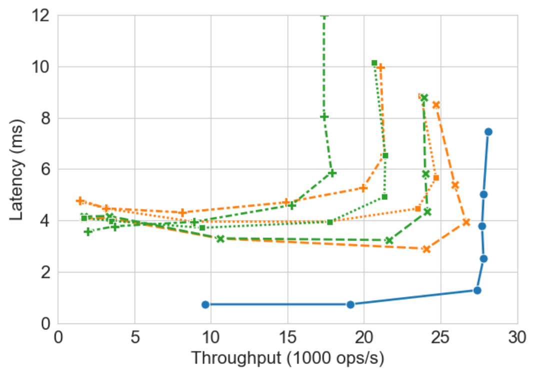
While the previous experiment provides a good understanding of the performance implications of replicating the sequencer of C3 in a simple scenario, we are interested in understanding its performance in a more realistic geo-replicated environment with multiple datacenters. To this end, in the following experiment, we use all 7 datacenters, as explained in the experimental setup (Section 4.6.1). In this scenario, operations issued by clients in a datacenter must not only execute in the local datacenter (just as in the previous experiment), but are additionally propagated to the sequencer replicas in other datacenters, which deliver them to the local Cassandra nodes in those datacenters, and wait for their acknowledgment.



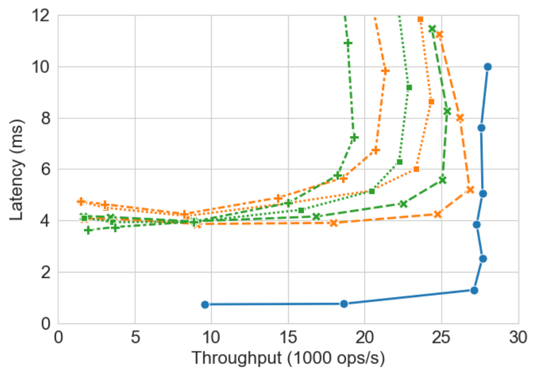
(a) No batching



(b) Batch size of 5



(c) Batch size of 20



(d) Batch size of 50

Figure 4.4: Performance in a geo-replicated environment.

Figure 4.4 shows the results of this experiment. The figures show the same metrics as the previous experiments, with the continuous blue line representing the performance of the non-replicated solution (which is not affected by batching), and the dotted lines showing the performance of the replicated solution with a varying number of replicas (three, five and seven) and using either ChainPaxos or Multi-Paxos. Overall, these results show a very similar trend to the single datacenter experiment, with the throughput penalty of the replicated solution being mitigated by increasing the batch size.

Despite the similarities, there is an important result to highlight: the throughput of ChainPaxos, with 3 replicas and a big enough batch size (20 or 50), is able to nearly match the throughput of the non-replicated solution, with even the throughput with 7 replicas not being significantly lower. This is an important result, as it shows that, by using the right state machine replication protocol, we can provide fault-tolerance to the sequencer without a significant performance impact.

The trade-off between throughput and latency is also present in this experiment (and can be seen more clearly in these figures), with the latency of operations increasing as the batch size (and therefore, performance) increases.

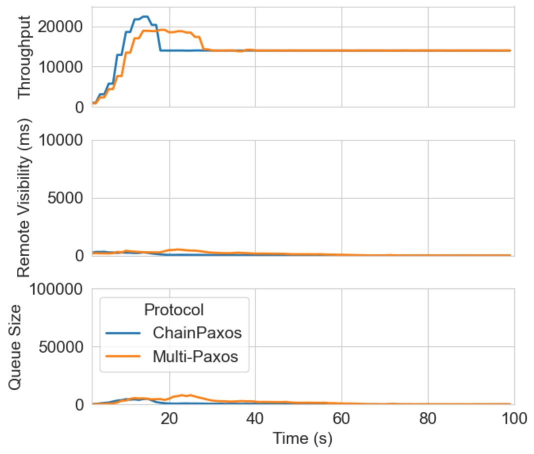
4.6.4 Visibility times and operation queuing

While the previous experiment resulted in very positive results, there is an important (and often hidden, and thus, overlooked) limitation of causal consistency solutions that needs to be considered in order to fully understand their performance: the trade-off between the visibility times of operations (i.e., how long it takes for an operation to become visible in a remote datacenter) and the maximum rate of operations executed in each datacenter. Even though this trade-off is not limited to sequencer-based causal consistency solutions, studying it is crucial to ensure the practical viability of replicating the sequencer in such solutions.

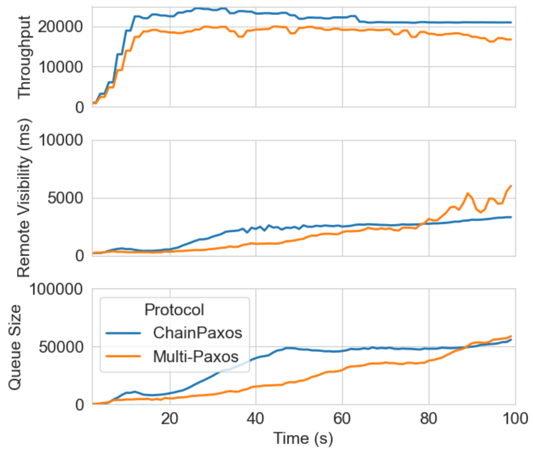
To understand this limitation, consider a scenario with two datacenters, where each can execute up to x operations per second. If the application data is replicated across both datacenters, then all operations issued by clients in one datacenter must be executed in the other. This means that each datacenter can only accept client operations at a rate of $x/2$ operations per second (in order to be able to execute the operations received from the other datacenter). Adding another datacenter does not help this limitation, as it does not increase the rate at which each datacenter can execute operations, but instead decreases the rate at which each datacenter can accept operations from clients to $x/3$ operations per second.[5]

In the previous experiment, each datacenter simply accepted client-issued operations as they arrived, effectively ignoring this limitation. The results of this, which are not directly perceivable by clients, is that the queue of operations received and to be processed in each datacenter from remote datacenters increases indefinitely, as the rate at which operations are received is higher than the rate at which they are executed. This results in an also indefinitely increasing visibility time of remote operations, meaning that clients executing reads operations in a datacenter will not see the results of operations executed in remote datacenters until a long time after they were issued. Note that this occurs regardless of using a replicated sequencer or not.

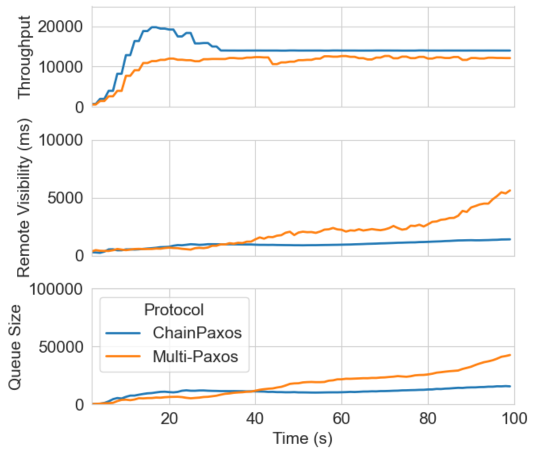
[5]We refer the interested reader to the work in [9] for a more detailed discussion on this limitation.



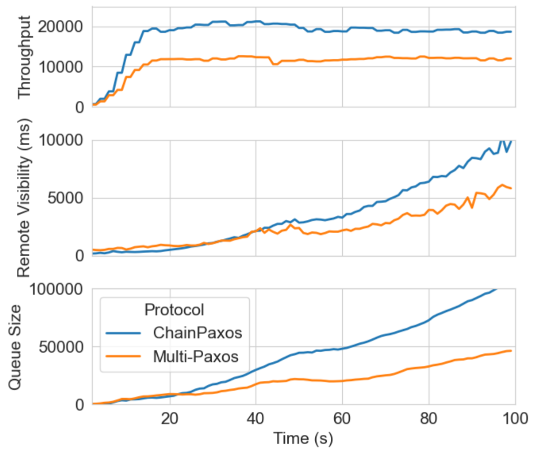
(a) 3 replicas with 2000 throughput



(b) 3 replicas with 3000 throughput



(c) 7 replicas with 2000 throughput

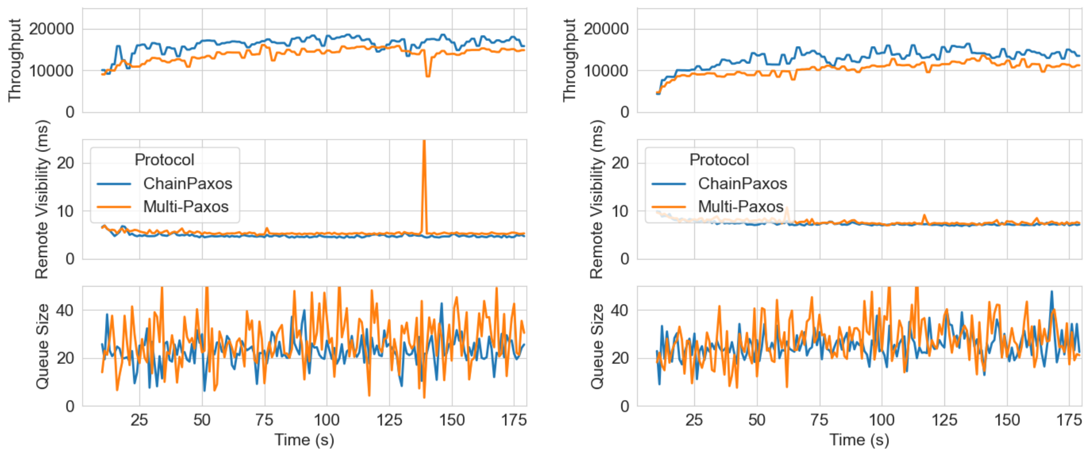


(d) 7 replicas with 3000 throughput

Figure 4.5: Throughput vs visibility times

Figure 4.5 visibly captures this effect. In this experiment, we use a batch size of five, with 20 clients in each datacenter, limiting the number of operations that can be executed in each datacenter to 2000 and 3000, while varying the number of sequencer replicas (three and seven). The figure shows, for each configuration, the total throughput of the system, the average visibility times of operations in remote datacenters, and the average queue size of remote operations in the sequencer of each datacenter over time. Looking at the results with three replicas (the topmost figures), we can see that with 200 operations per datacenter (Figure 4.5(a)), as datacenters are not saturated, remote operations are executed as soon as they are received, resulting in very low visibility times, and no operations being queued. However, as we increase the throughput to 3000 operations per datacenter (Figure 4.5(b)), datacenters are no longer able to keep up with the rate at which operations are received, which results in an endless (as long as new operations are being submitted) increase in the size of the queue of remote operations in the sequencer of each datacenter, which in turn results in an increase in the visibility times of remote operations over time. The results with 7 replicas (the bottom figures) show a similar trend, with the difference that even with 2000 operations per datacenter, Multi-Paxos is not able to keep up with the rate at which operations are received, while ChainPaxos is.

Essentially, this means that the throughput results from the previous experiment (Figure 4.4) are misleading. While operations are indeed being executed in the local datacenter at the shown throughput, the visibility times and the queue of remote operations in each datacenter are increasing indefinitely, which is not sustainable in a real-world deployment. As such, in this work, we are interested in finding out what is the actual maximum throughput that can be achieved both by the replicated and the non-replicated sequencer, while keeping visibility times and operation queuing at acceptable levels. For this, instead of limiting the clients to a fixed throughput, we change their behavior to only issue a new operation after the previous one has been executed in all datacenters. This effectively limits the rate at which operations are issued to the maximum sustainable rate at which they can be executed in all datacenters (since if operations are queued, clients will block until they are executed), without requiring setting an artificial limit on the throughput of the system.



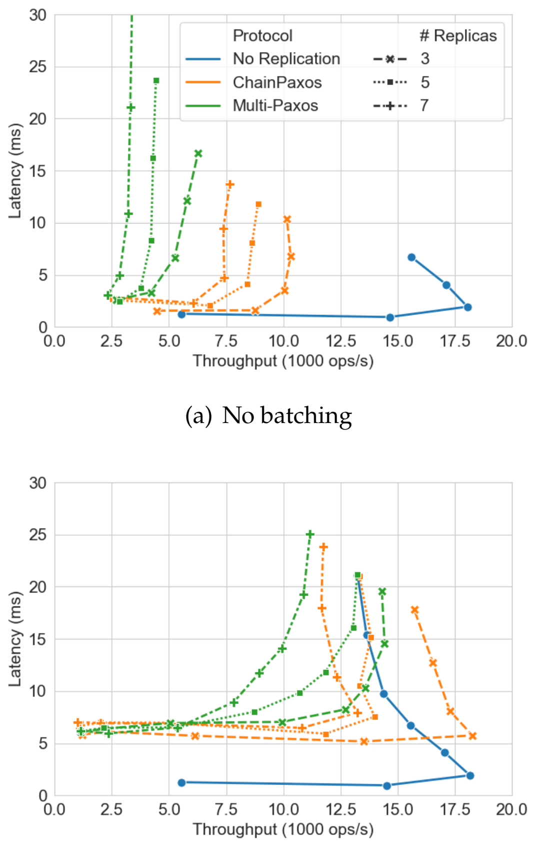
(a) 3 replicas (b) 7 replicas

Figure 4.6: Sustainable throughput vs visibility times

Figure 4.6 shows the results of this experiment, with the same configuration as the previous one, but with clients issuing operations only after the previous one has been executed across all datacenters. As we can see in the figure, this allows us to find the actual maximum sustainable throughput of the system, as both the remote visibility times and the queue of remote operations are kept at constant low levels.

4.6.5 Multiple datacenters - revisited

Taking into account the discussion and results of the previous section, we now revisit the experiments from Section 4.6.3, using the same configuration, but with clients issuing operations only after the previous one has been executed in all datacenters, effectively achieving the actual maximum sustainable throughput of the system.



(a) No batching

(b) Batch size of 5

(c) Batch size of 20

(d) Batch size of 50

Figure 4.7: Sustainable performance in a geo-replicated environment.

Figure 4.7 shows the results of this experiment. The figure presents the same metrics as Figure 4.4, with results also being similar. The most relevant difference is that the maximum throughput of the system is significantly lower (both for the replicated and non-replicated versions), as this time remote operations are being executed in a sustainable way, instead of being indefinitely queued. Again, these results also show that by using big enough batch sizes, ChainPaxos is able to achieve a throughput that is very close to the non-replicated solution, with negligible latency overhead.

4.7 Related Work

In the work reported in this chapter, we focused on the performance implications of replicating the sequencer component of sequencer-based causal consistency solutions. However, there is a significant amount of research contributions in the area of causal consistency, with many geo-replicated solutions being proposed that use drastically different approaches to track and enforce causal dependencies between operations.

In this section, we group such solutions into categories, discussing their main characteristics, how they differ from sequencer-based solutions, their pros and cons, and some of the most relevant contributions in each category. Other than *sequencer-based solutions*, the other categories are: *explicit dependency checking*, *global background stabilization* and *optimistic approach*.

4.7.1 Sequencer-based

This class of solutions, which was studied in this chapter, and includes C3 [36], focus on compressing metadata when tracking causal dependencies between operations across datacenters. For this, they rely on a centralized component in each datacenter, the *sequencer*, which is responsible for gathering and assigning unique identifiers to all operations that are issued by clients in that datacenter. The sequencer then propagates these operations to the sequencers in other datacenters, which are responsible for executing them in an order that respects the causal dependencies between operations. This approach greatly simplifies the tracking of causal dependencies, as it avoids the need for a complex distributed protocol among the possibly hundreds of nodes in a geo-replicated system.

While sequencer-based solutions all leverage this centralized component to propagate operations and track causal dependencies between datacenters, they do so in different ways. When operations are issued by clients in a datacenter, some solutions (such as Saturn [16] and ChainReaction [4]) allow their execution in the local datacenter to proceed without requiring the sequencer to order them. For this, such solutions require that both the clients themselves store metadata representing their causal past, and that metadata is stored along with data objects in the datastore. Other solutions (such as C3 [36]) avoid any form of client-side metadata, instead requiring the sequencer to order all operations before they are executed locally.

As sequencer-based solutions rely on a sequencer per each datacenter, the intuitive way to compress causality tracking metadata is to use a vector clock of scalars, with one entry per datacenter. This is the common approach of sequencer-based solutions, including C3 [36], ChainReaction [4] or SwiftCloud [93]. As seen in C3, the usage of vector clocks allows some concurrency in the execution of remote operations, as they allow identifying causally independent operations, which can be executed in parallel and in any order. Other solutions, such as Saturn [16], further compress metadata to a single scalar. While this approach reduces the overhead of metadata, it requires operations received by a sequencer to be executed in the same order that they were received, greatly limiting the parallelism in the execution of remote operations.

As discussed and studied throughout this chapter, the main drawback of sequencer-based solutions is that the sequencer represents a single point of failure, with its performance directly impacting the performance of the entire datacenter.

4.7.2 Explicit dependency checking

Solutions based on explicit dependency avoid any form of centralized component by having datastore nodes explicitly check the dependencies of each operation before executing them. These solutions leverage sharding inside datacenters, by assuming that application data is split into multiple partitions, with each partition being replicated across multiple datastore nodes in each datacenter.

In this approach, each operation is associated with a set of dependencies (which can be a list of operations, versions of data objects, or matrix clocks). These dependencies are tracked by a client-side library, which attaches them to operations before issuing them to a datastore node in the local datacenter that replicates the target data partition. The operation is then propagated to the nodes that replicate the same partition in remote datacenters. In each datacenter, the node responsible for the partition first checks if all dependencies are satisfied locally, by communicating with the nodes responsible for the partitions that are dependencies of the operation.

Examples of solutions that follow this approach include COPS [69], where clients track their nearest dependencies by storing the versions of the data objects they have read, and Eiger [70], a similar system which improves on COPS by tracking dependencies on operations instead of versions of data objects. Both systems assume that operations executed in each datacenter are linearizable, and that clients are bound to a single datacenter. Karma [74] aims to improve on the deployment model of COPS and Eiger by allowing partitioning data across multiple datacenters, by creating *consistent hashing rings* that span multiple datacenters. It additionally addresses limitations of Cops and Eiger by avoiding the need to have linearizability in each ring, and allowing clients to execute operations in any ring. Finally, Orbe [32] employs dependency matrices to track causal dependencies in clients, with the dimensions of the matrix being the number of data partitions and the number of datacenters.

When compared to sequencer-based solutions, the main drawback of solutions based on explicit dependency checking is that they do not support partial replication, as the mechanism used to check if dependencies are satisfied requires all data partitions to be present in all datacenters.

4.7.3 Global background stabilization

These solutions employ a global background stabilization mechanism to enforce causality between operations. Similar to solutions based on dependency checking, these solutions assume that data is split into partitions, with replicas of each partition spread across multiple datacenters. Client operations are sent to the partition in the local datacenter, which tags them with a timestamp representing their causal dependencies before propagating them to the other replicas of the partition in remote datacenters. These solutions usually employ multiversion datastores, with operations received by a partition replica from a remote datacenter creating new versions of the data objects, without removing the old ones.

The background stabilization mechanism is responsible for ensuring that updates are only made visible after all their causal dependencies have been made visible. For this, replicas of the same partition in all datacenters keep track of the highest timestamp seen from each other. Inside each datacenter, a background process periodically runs to determine the minimum timestamp seen by each partition, which represents the current *stable time* of the system. Updates with a timestamp smaller than the current stable time can then be made visible.

Solutions employing this approach include GentleRain [31], which uses physical clocks as scalar timestamps to tag operations and compute the current global stable time, Cure [2], which employs vector clocks instead of scalar timestamps to support transactions, while still using physical clocks, and CausalSpartan [96], which uses Hybrid Logical Clocks [51], a combination of logical clocks with physical clocks. The work from Xiang and Vaidya [116] shows that the use of global background stabilization techniques can be used to provide partial replication, and PaRiS [107] expands on this by supporting generic transactions with partial replication.

While having the advantage of being fully decentralized, solutions based on global background stabilization suffer from high remote visibility times, as they require costly coordination among all partitions of all datacenters to determine the current stable time. There is a trade-off between the throughput of the system and the visibility time of remote operations, with solutions that use more metadata to track dependencies (such as vector clocks [2]) being able to provide lower visibility times at the cost of lower throughput. Additionally, while increasing the frequency of the background stabilization process can reduce visibility times, this also increases the overhead of the process, resulting in a decrease in the throughput of the system.

4.7.4 Optimistic Approach

All the solutions discussed so far enforce causality by controlling the execution of write operations, ensuring that they are only made visible in a datacenter after all their causal dependencies have also been made visible. Solutions based on the optimistic approach take a different approach, by immediately applying all received write operations, and enforcing causality only when clients execute read operations. For this, clients keep track of metadata representing their causal history, tagging their read operations with this metadata. When the read operation is received by a server, it checks if all dependencies are satisfied, delaying the response if they are not.

Solutions employing this approach are less common. Occult [80] relies on this approach to avoid the problem of *slowdown cascades*, where a single slow replica can cause a slowdown in the whole system. However, it relies on a master-slave replication model, where updates can only be executed in the master of each partition, which delays client operations. POCC [106] differs from Occult by allowing clients to execute operations in any replica, however at the cost of only supporting read-only transactions, as opposed to Occult which supports generic transactions.

While solutions following this approach have the advantages of avoiding expensive coordination mechanisms (such as background stabilization) and being resilient to slow replicas, they have three main drawbacks: *i*) solution employing this approach do not support partial replication; *ii*) client read operation may block indefinitely, sacrificing availability; and *iii*) this approach shifts the burden of enforcing causality to read operations, which are the majority of operations in most systems.

4.8 Discussion

When considering the goals of this contribution (Section 4.2), our prototype shows that, by leveraging performant state machine replication protocols and their common optimizations, we can provide fault-tolerance to sequencer-based causal consistency solutions with a minimal performance impact. Additionally, the results also validate the applicability of our first contribution, ChainPaxos, in a practical scenario, showing that it can provide a significant performance improvement over solutions with the communication pattern of Multi-Paxos.

4.8.1 Trade-offs and Limitations

As the objective of this contribution was to study the performance impact of making the sequencer of sequencer-base causal consistency solutions fault-tolerant, we did not focus on some aspects that are crucial for the practical deployment of such solutions.

One such aspect is keeping track of the current set of sequencer replicas in each datacenter, allowing both datastore nodes and sequencers from other datacenters to discover and communicate with them. In our implementation, this was done by having a static list of sequencer replicas in each datacenter, and making this list available both to datastore nodes (Cassandra) and sequencers. However, in a real-world deployment, this would likely be done by leveraging a distributed coordination service such as Zookeeper [46].

Another crucial aspect is recovering from failures of sequencer replicas. This has two challenges: *i*) reconnecting datastore nodes to sequencers, and reconnecting sequencers across datacenters; and *ii*) re-propagating any operations that were lost due to the failure. For the first challenge, in our implementation, both datastore nodes and sequencers simply pick another sequencer replica from the list of sequencers of that datacenter to connect to.

In a real-world deployment, this would likely be done by once again leveraging on the distributed coordination service, which would inform about sequencer replica failures, the set of alive replicas, and could even be used to elect a new leader replica in case the leader fails. For the second challenge, datastore nodes can simply re-propagate pending operations when connecting to a new sequencer. However, when the sequencer replica responsible for propagating and receiving operations from other datacenters fails, there needs to be a synchronization mechanism to ensure that the new leader replica has all requests from other datacenters that were sent to the failed sequencer replica but not submitted to the SMR protocol.

We note that addressing these fault-recovery challenges is highly dependent on the implementation details of the specific solution being used. While our implementation addresses these challenges in a simplified way that works for C3, other solutions may require different approaches. Regardless, the focus of this contribution was to show that it is possible to replicate the sequencer in sequencer-based solutions without a significant performance impact, and not to design fault-recovery mechanisms that are applicable to all solutions.

Summary

In this chapter, we studied the scalability and fault-tolerance limitations of existing causal consistency solutions, by focusing on a specific class of solutions, based on operation sequencers. For this, we implemented a fault-tolerant version of the sequencer of one such solution, C3, replicating it using two different state machine replication protocols, ChainPaxos and Multi-Paxos. Our evaluation showed that, by using the right state machine replication protocol, along with common techniques such as batching, we can provide fault-tolerance to the sequencer without sacrificing performance, and with negligible latency overhead. This work contributes to the goal of this thesis by proposing a solution that allows causal consistency solutions in the geo-replicated *replica distribution level* to be fault-tolerant, while still providing good scalability.

In the next chapter, we focus on the edge-computing level, by proposing a novel distributed data management system designed to provide large-scale causal consistency in edge-computing environments.

Artifacts

A fully functional implementation of C3 with replicated sequencers using both ChainPaxos and Multi-Paxos was made available as open-source software, and is available at `https://github.com/pfouto/c3-sequencer`.

5

Arboreal: Scalable Causal Consistency for the Edge

This chapter presents *Arboreal*, the third main contribution of this thesis. Arboreal is a novel distributed data management system designed for the edge. Its main goal is to allow stateful edge applications to be deployed with full local access to data. Its replication protocol allows to dynamically replicate data across edge locations, while ensuring global causal+ consistency, by leveraging on a decentralized hierarchical topology.

Following the goal of this thesis and the definitions presented in Chapter 2, this contribution addresses the edge computing *replica distribution level*, by proposing a replication protocol with causal consistency guarantees that employs dynamic partial replication.

We start by providing some background on the role of edge computing in modern Internet services, along with its current limitations. We then present the goals of this contribution, by presenting the stateful edge computing model and the challenges it poses. We follow by presenting the design and implementation of Arboreal, along with its extensive evaluation. This chapter concludes with a discussion on related work, and a final discussion on the most relevant aspects of the contribution.

5.1 Background

As was discussed in the previous chapters, modern Internet services play a central role in society, providing a wide range of services to millions of users worldwide, such as social media, instant messaging or online shopping. These services are typically deployed in multiple datacenters across the globe, to ensure low latency and high availability to users. To support these services, distributed data management systems are deployed along with the application logic in these datacenters, ensuring that data is replicated and kept up-to-date across all locations. For this, user-facing services typically employ NoSQL databases [30, 52], which provide high availability and scalability while relaxing consistency guarantees.

The always increasing need (and client demand) for low application response times has led to the rise of the *edge computing* [101, 65] model. Edge computing can be broadly defined as the paradigm of bringing computations and data outside of datacenters, closer to end users, in order to reduce the latency of applications. While edge computing enables a wide range of novel applications, such as augmented reality, location-based video games or autonomous vehicles, it is also extremely relevant for improving the latency of the aforementioned "traditional" Internet services, which is the focus of this contribution.

In fact, many of the largest Internet services already operate with hundreds of edge nodes mediating the access of clients to datacenters [7, 40]. However, since application data is traditionally stored in cloud databases designed to be deployed in datacenters [30, 52, 27], the application logic of these services (which requires access to the application data to process client requests) is also typically only deployed in datacenters.

The result of this deployment model is that clients always need to access datacenters to interact with the applications, with edge nodes having a limited role in the processing of client requests. These days, edge nodes are mostly used for: (1) accelerating the delivery of static content, such as images or videos, to clients, by serving as Content Delivery Networks (CDN) [8, 24, 98, 117]; (2) serving as reverse proxies that simply forward client requests to the closest datacenter, where they are processed [40, 23]; or (3) executing simple computations over client requests that do not require access to application data, such as collecting metrics, filtering requests or injecting advertisements [53].

Simply deploying the application logic components in edge nodes has little advantage if processing client requests requires the retrieval of application data from the cloud. To fully exploit the potential of edge computing, it is necessary to extend the cloud data storage solutions to the edge, allowing edge nodes to access and update application data locally. However, doing so while maintaining the same guarantees provided by cloud data storage solutions is not a trivial task, due to the unique challenges posed by edge environments. First, edge nodes may have limited resources, only being able to store a subset of an application's data. This, combined with the fact that the data required at each edge location may change over time, makes traditional static and coarse-grained data partitioning techniques unsuitable for the edge. Second, the number of edge locations can be orders of magnitude larger than the number of data centers (hundreds or thousands instead of a few dozens), and edge nodes may be more prone to failures, with entire edge locations becoming partitioned, rendering traditional solutions based on a fixed number of stable cloud datacenters unsuitable for the edge.

5.2 Motivation and Goals

The prevailing deployment model for Internet services, which involves running the main application components in cloud infrastructures, accessing data managed by (potentially geo-replicated) storage systems deployed in these same datacenters [98, 23], greatly limits the potential of edge computing.

To fully realize the potential benefits of edge computing to provide low latency to clients and reduce the load in centralized components, it is necessary to adopt a different model, where edge nodes process all client requests. Doing this efficiently requires the application logic running in edge nodes to have unrestricted access to local replicas of the application data. The key for supporting this *stateful edge applications* model consists in expanding storage systems from datacenters to edge locations, addressing challenges that depart from those addressed by traditional cloud storage systems [30, 52, 4, 69].

As such, the goal of this contribution is to design, implement and evaluate a distributed data management system that allows stateful edge applications to be deployed at the edge, by providing these applications with full local access to data with consistency guarantees. To achieve this goal, we identify three main requirements that the proposed solution must satisfy:

Dynamic Partial Replication Applications with large user bases are expected to leverage on a considerable number of edge locations, largely exceeding the typical count of datacenters employed nowadays. While it is imperative for application components to have unrestricted data access while maintaining consistent guarantees, regardless of executing in the cloud or at the edge, edge locations are characterized by having fewer resources compared to cloud datacenters, not being able to store the entire dataset of an application. To overcome this limitation, the proposed solution must support fine-grained *partial and dynamic replication*. This means that the set of data objects replicated at each edge location needs to evolve over time to reflect the access patterns of clients accessing the applications in that location. This poses a challenge as the replication protocol must dynamically adapt to ensure timely propagation of data updates to the correct locations, preventing components and clients from encountering stale data, and ensuring their operations become visible across the rest of the system.

Large-scale Consistency Enforcing some form of consistency across all edge locations is crucial to the ensure correctness of applications. While strong consistency simplifies application logic by avoiding data anomalies, it proves impractical for edge settings due to latency and availability issues arising from coordinating numerous replicas. In edge environments, it is more suitable to rely on a weak consistency model, which relaxes consistency for improved availability and response times. To mitigate anomalies in eventual consistency models, many solutions adopt *causal+ consistency* [69, 59] which provides the strongest consistency guarantees while allowing the system to remain available when some replicas are unreachable [6].

Traditional causal consistency solutions, however, are designed with a small and static set of cloud datacenters in mind, and are not suitable for the edge. This includes solutions that rely on vector clocks [12, 36, 2, 80], which grow linearly with the number of replicas and data and/or data partitions, leading to significant metadata overhead. Other solutions, such as the ones based on *background stabilization* [31, 2] assume a small and static set of data partitions and employ all-to-all communication, both of which are not suitable for the edge.

As such, supporting stateful edge applications requires a novel scalable solution for tracking causal dependencies across write operations that can *scale to hundreds of locations*, in a context involving dynamically replicated data objects, and in a way that can deal with replica set changes.

Client Mobility Applications benefiting most from the stateful edge applications model often involve numerous users dispersed across different locations, accessing data based on their location (e.g., in collaborative applications or multiplayer mobile games). In these applications, where low response times are crucial, users may change locations while using the application. When doing so, users should be able to migrate from interacting with an application component on one edge location to a closer one seamlessly, without encountering data consistency anomalies. This requires mechanisms to support such migrations efficiently and without sacrificing consistency.

In addition to these requirements, and considering the goals of this thesis of building scalable and fault-tolerant distributed data storage systems with consistency guarantees, a satisfactory solution also needs to address the following requirements:

Scalability Where in traditional cloud deployments, the number of datacenters is at most in the order of a few dozen, in edge computing scenarios, the number of edge locations is expected to be at least in the order of hundreds [40, 7]. This poses challenges that are not addressed by existing cloud-based solutions. While we already discussed the need for a scalable causality tracking protocol, both the replication protocol (i.e., how we propagate updates across the system) and the dynamic partial replication mechanism must also take into account the large number of edge locations to avoid performance degradation as the system scales. Similarly to the causality tracking protocol, this requires designing novel replication protocols, as the typically employed solutions where every datacenter has full knowledge about the subset of data objects replicated in every other datacenter and updates are propagated in an all-to-all manner do not scale.

Fault Tolerance While fault tolerance is a key requirement for any distributed system, deploying distributed data management systems in the edge poses new challenges. In particular, while typical geo-replicated solutions assume a static set of datacenters that can be partitioned but never fail (only individual nodes in each datacenter fail), in the edge, entire edge locations may fail (by crashing or becoming partitioned) or be decommissioned at any time, and new locations may be added dynamically. This requires fault tolerance mechanisms that can recover from the failure of entire edge locations, while ensuring that consistency guarantees are maintained, and, importantly, doing so in a scalable manner by avoiding both centralized components or coordination between all edge locations.

5.3 Arboreal

In this section, we present the design of Arboreal, a distributed data management system designed for the edge that addresses all the requirements presented in the previous section. We start by presenting the system model where Arboreal operates, followed by the novel replication protocol that Arboreal employs to provide causal consistency with dynamic partial replication in a decentralized and scalable manner. We then present the fault tolerance and recovery mechanisms of Arboreal, finalizing with the protocol to support client mobility.

5.3.1 System Model

Our solution assumes an existing deployment of a service executing in a set of cloud datacenters spread across different geographic regions, in which a distributed (geo-replicated) NoSQL database is deployed. Deployment specifics of this database (e.g., replication protocol, partitioning scheme) are orthogonal to this work. We consider a set of *edge locations* equipped with computational resources, controlled by a single organization (e.g. edge locations offered by cloud providers). Arboreal extends the database from the cloud datacenters to edge locations within each individual geographic region (i.e., each edge location is assigned a datacenter based on its location). For this, an instance of Arboreal is deployed both in each datacenter and each edge location. We assume that an edge location may consist of one or multiple physical nodes, however, Arboreal always treats an edge location as a single node, with a single instance of Arboreal being deployed in each edge location. As such, in this section, the term *edge node* refers to an instance of Arboreal deployed in an edge location. No assumptions are made about replication and data partitioning schemes between physical nodes within each edge location, with the focus being instead on data replication across edge locations. To accommodate diverse scenarios and applications, we assume that, at any time, Arboreal can be dynamically deployed, along with application components, in new edge locations and that edge locations may fail, or Arboreal (and the application) may be decommissioned from them. Applications deployed on edge nodes rely on Arboreal for consistent local data access to support the processing of client operations that can both access and modify application data. Clients, potentially mobile, can migrate between edge locations at any time, and have dynamic workloads, with the set of accessed data objects possibly changing over time.

In respect to the data model offered by Arboreal, it provides a key-value store interface, akin to other highly available distributed NoSQL databases[30, 52], where each data object is identified by a unique key. Clients issue read or write operations over any individual data object without constraints, and Arboreal ensures that: (1) data objects are replicated transparently and on-demand to the edge locations where they are accessed; (2) write operations are propagated to all locations currently replicating the modified data object; and (3) clients always observe a state respecting a causal order of operations. Arboreal makes no assumptions on how data is stored in each node. For convergence of concurrent update operations over the same data object, a *last-writer-wins* policy is used, relying on operation timestamps.

5.3.2 Replication Model

The main challenge of this work is scaling the replication protocol to hundreds of edge locations, while providing causal consistency and supporting dynamic partial replication. The key ideas to achieve this are, in one hand, to avoid the use of metadata (both for causality tracking and to track the set of replicated data objects in each location) that grows linearly with the number of edge locations, and, in the other hand, to enable edge locations to synchronize directly with each other while avoiding expensive all-to-all communication between edge locations, which becomes impractical as the number of locations grows.

5.3.2.1 Hierarchical Approach

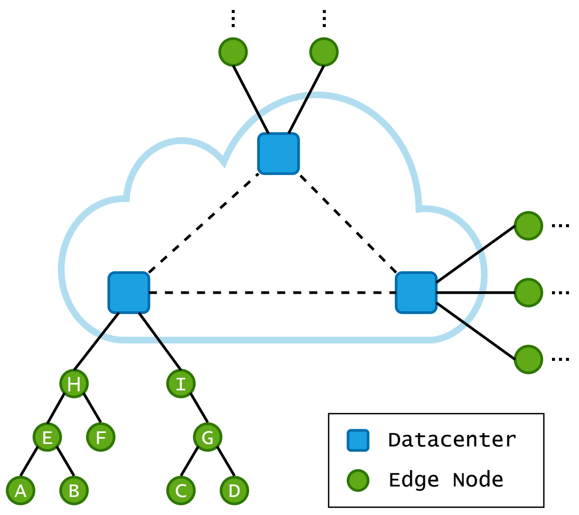


Figure 5.1: Design of Arboreal, with 3 geographic regions.

To address these challenges, Arboreal employs a hierarchical design, with each edge location hosting an instance of Arboreal. These edge locations form a tree structure rooted at their regional data center, establishing what we call the region's *control tree*. Figure 5.1 shows a geo-distributed example of an Arboreal deployment with 3 datacenters, each having its *control tree* composed of the edge locations of that region. Recall that our system model assumes an existing geo-replicated database deployed in the datacenters, which is responsible for replicating data across datacenters (represented by the dotted lines in the figure). As such, each *control tree* is responsible for replicating between the edge locations and the datacenter of a single geographic region. To avoid the bottleneck of a centralized component, the management of the *control trees* is fully decentralized, with nodes communicating solely with their parent and children. Nodes only retain detailed information about their (direct) children and minimal information about their ancestors (i.e., nodes in the path between itself and the root of the *control tree*), and are not aware of the existence of any other nodes. This decentralized structure allows for latency-sensitive tasks, such as creating replicas of data objects (5.3.2.2), failure recovery (5.3.3), and mobile client handling (5.3.4) to be performed in a localized fashion, involving as few nodes as possible, and thus reducing the amount of metadata that needs to be stored in each node. Note that, apart from being the root of the *control tree*, the datacenter node does not have any special role in the replication protocol of Arboreal, and is treated as any other node in the tree. Consequently, it only stores information and is only aware of the existence of its direct children.

To establish the *control tree*, when an instance of Arboreal is deployed on an edge location, it uses a configurable heuristic to select the most suitable existing instance in the *control tree* of its region to be its parent, basing this decision on its knowledge of the existing nodes in the region. Note that the goal of Arboreal is to replicate data to applications running on edge locations, and is not an orchestrator that decides where or when to deploy the application. To accommodate diverse edge scenarios, both the heuristic defining the *control tree* and the information used by the heuristic are configurable and left to the application developer to define. Additionally, Arboreal does not provide mechanisms for joining nodes to gather information about the existing ones. Different techniques can be used to achieve this, based on the requirements of the application and the edge environment where Arboreal is deployed, such as using a pre-existing centralized directory service, using decentralized gossip protocols, or simply using the regional datacenter as a central point of contact.

Given the significance of geographic locality in edge computing, our Arboreal implementation employs geographic distance between edge locations as the primary metric for forming the *control tree*, with edge nodes forming a decentralized overlay network to exchange information about themselves. For this, nodes joining the tree only take into consideration as possible parents nearby nodes that are closer to the datacenter than themselves. However, depending on the application, various metrics can be used, such as latency between edge nodes, client locations, or even predicting future demand for the application. Section 5.4.1 provides an insight into how the *control tree* is formed in our implementation.

As we will see in the following sections, the hierarchical topology of Arboreal is key to implement all the features of Arboreal in a scalable and decentralized manner.

5.3.2.2 Dynamic Partial Data Replication

A critical limitation of edge computing is that, unlike cloud datacenters, edge locations can not be expected to have resources to replicate the entire dataset of an application. As such, partial replication is a key aspect of Arboreal. Moreover, to support a wide range of applications, Arboreal must adapt not only to changes in client access patterns but also to mobile clients that can change the edge location to which they are connected at any time. This means that, unlike cloud-based data management systems that typically use static data partitions, Arboreal needs to allow edge nodes to dynamically change the set of replicated data objects at any time with fine granularity. However, keeping track of which nodes replicate which data objects across a large-scale system can be costly and require substantial metadata propagation, especially with dynamic sets of nodes and data objects. Fortunately, we can rely on the hierarchical topology of Arboreal to address this challenge.

In Arboreal, instead of the traditional approach of splitting the dataset of an application into fixed coarse-grained partitions, data objects are replicated individually across the *control tree*. New replicas of data objects are always created by replicating from a parent node to a child node, resulting in the replicas of each data object forming a subtree of the *control tree*, rooted at the datacenter. We refer to this subtree as the *replication tree* of the data object. Figure 5.2 illustrates a simple example of this, showing the evolution of the *replication trees* of two data objects, α and β, across the *control tree* of a region. The lifetime of the *replication trees* of an object is dictated by the access patterns of clients, with nodes replicating data objects that are currently being accessed by their clients, and releasing data objects that are no longer being accessed.

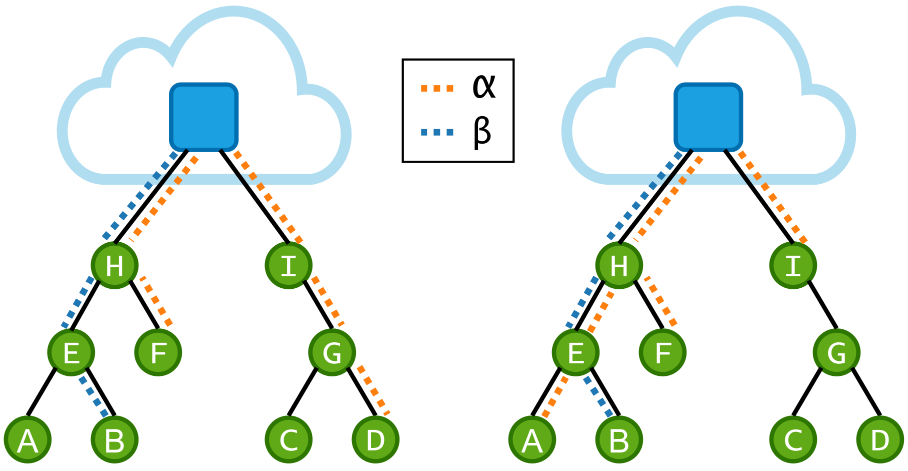


Figure 5.2: Dynamic partial replication in Arboreal.

The main advantage of this design is that we avoid the need for expensive coordination between all edge nodes to track the set of replicated data objects. Instead, each edge node only needs to keep track of the data objects that itself and its direct children replicate.

This is possible since Arboreal guarantees that the parent of a node always replicates a superset of the data objects replicated by its children. This also means that edge nodes can only replicate data objects that are replicated in their regional datacenter. This mechanism assumes that storage and computation capacity increases moving up the *control tree*, closer to cloud datacenters, which we believe to be a reasonable assumption for an edge environment [65].

While this restriction in data object replication may seem limiting, forcing edge nodes to replicate data objects that their current clients may not be interested in, it is actually a very beneficial design decision that allows efficient solutions for the other mechanisms of Arboreal (that will be presented in the following sections): i) as a subtree of the *control tree*, an object's *replication tree* inherits its causal dissemination guarantees, providing causal consistency globally across all objects (detailed in Section 5.3.2.3); ii) when the *control tree* is repaired after node failures (detailed in Section 5.3.3), the *replication trees* involving the faulty nodes are also repaired, enabling the system to quickly recover from failures with minimal client impact; and iii) when mobile clients move between edge nodes, even if their new node does not replicate the required data objects, there is a high chance that one of its nearby ancestors does, allowing the client to quickly resume its operation (detailed in Section 5.3.4).

We now present the mechanisms that control to creation and removal of replicas of data objects in Arboreal.

Replica creation Replicas of data objects in Arboreal are only created on-demand, by clients accessing the data object in an edge node. This process is illustrated in Algorithm 9 and works as follows. When a client requests a data object (line 1) not replicated in its connected edge node (line 6), the process of expanding the *replication tree* of that object to include the client's node is initiated (if it was not already in progress). The client's node sends a request to its parent node, asking to be added to the object's *replication tree* (line 12). If the parent node is part of it (line 15), it sends the current object version to the child node and keeps track that this child node is now replicating the object, propagating any future updates over the object to the child. If the parent node is not part of the *replication tree* (line 19), it forwards the request to its own parent (line 25), and so on, until the request reaches a node replicating the object (or the cloud datacenter, which replicates all objects). This node then sends the object to its child node (registering that the child node now replicates the object), and the process repeats (line 27) until the object reaches the node that initially requested it. When this process finishes, it is possible that multiple replicas of the object are created, as the object must always be replicated in all nodes in the path between the client's node and the datacenter.

A simple example of this process is illustrated in Figure 5.2, where a client connected to node A requests data object α, which results in the creation of replicas of α both in node A and in node E.

Algorithm 9 Replica creation in Arboreal

```
procedure RECEIVE(<ClientWriteOp,key,value> from client)
    metadata ← generateOpMetadata()
    if key ∈ LocalObjects then
        Trigger ApplyWrite(key, value, metadata)
        Trigger PropagateWrite(key, value, metadata)
    else
        if PendingObjects[key] then
            {writes, reads, children} ← PendingObjects[key]
            writes ← writes ∪ {<key, value, metadata>}
        else
            PendingObjects[key] ← {{<key, value, metadata>}, ∅, ∅}
            SEND(parent, <RequestObject, key>)

procedure RECEIVE(<RequestObject,key> from child)
    if key ∈ LocalObjects then
        {value, metadata} ← RetrieveObject(key)
        SEND(child, <ReplicateObject, key, value, metadata>)
        ChildObjects[child] ← ChildObjects[child] ∪ {key}
    else
        if PendingObjects[key] then
            {writes, reads, children} ← PendingObjects[key]
            children ← children ∪ {child}
        else
            PendingObjects[key] ← {∅, ∅, child}
            SEND(parent, <RequestObject, key>)

procedure RECEIVE(<ReplicateObject, key, value, metadata> from parent)
    Trigger ApplyWrite(key, value, metadata)
    {writes, reads, children} ← PendingObjects[key]
    for all child ∈ children do
        SEND(child, <ReplicateObject, key, value, metadata>)
        ChildObjects[child] ← ChildObjects[child] ∪ {key}
    for all write ∈ writes do
        Trigger ApplyWrite(key, value)
        Trigger PropagateWrite(key, value)
    PendingObjects[key] ← {}
    LocalObjects ← LocalObjects ∪ {key}
```

Garbage Collection Due to the possible large number of data objects and limited storage capacity in edge nodes, Arboreal nodes must be able to remove replicas of data objects that are no longer useful, to free up storage space for new objects. To achieve this, Arboreal employs a *garbage collection* process. Each node keeps track of the last time each currently replicated data object was accessed by any client. Periodically, objects that have not been accessed for a configurable duration are removed. When removing a data object, a node must inform its parent that it no longer replicates that object, to prevent it from forwarding future updates over that object. Note that, due to the nature of the *replication trees*, nodes can only garbage collect objects that are not being replicated in any of their children, otherwise the *replication tree* would break.

Figure 5.2 shows an example of this mechanism. Object α is no longer being accessed by clients in nodes D, leading to it garbage collecting the object and being removed from the *replication tree* of α. In the example, node G is also removed from the *replication tree* of α, meaning that its clients were not accessing the object, and it was only being replicated since it was in the path between D and the datacenter. On the other hand, node I is still replicating the object, meaning that its clients are still accessing it.

The decentralized design of Arboreal's replication protocol allows each node to only track metadata proportional to the number of objects it replicates, avoiding linear growth with the total number of edge nodes and data partitions/objects. In turn, this allows Arboreal to scale to a large number of edge nodes while supporting fine-grained replication of data objects.

5.3.2.3 Enforcing causality

The main challenge in providing causal+ consistency in a large-scale edge environment is tracking and enforcing causal dependencies without incurring in prohibitive metadata or communication costs. In particular, we need to avoid using solutions where the size of metadata grows linearly with the number of edge locations, such as vector clocks, as these would lead to performance degradation as the system scales. For this, once again, the key is to leverage the hierarchical topology both to disseminate operations to all nodes in the *replication tree* of each object and to aggregate metadata, decreasing the amount of metadata that each node needs to store and transmit. Arboreal employs two complementary mechanisms to achieve this: *causal dissemination* and *timestamping* (illustrated in algorithms 10 and 11):

Causal Dissemination: We start by leveraging the hierarchical topology of Arboreal, which allows achieving causal consistency without requiring any metadata [16, 67]. For this, nodes form the *control tree* (and, consequently, all *replication trees*) by establishing FIFO links to their parent and children. As seen in Algorithm 10, when a node receives a write operation (line 13) from a link (i.e., from a parent or child), it atomically executes the operation locally and puts it on the outgoing queue of every other link that is part of the *replication tree* of that object. Additionally, local operations from clients (line 1) are atomically added to the outgoing queues of all links of the object's *replication tree* (line 6). As the *replication trees* are simply subtrees of the *control tree*, this ensures that operations are always propagated (and thus, executed) after their causal dependencies, even if the operations are over different data objects.

While causal dissemination allows Arboreal to provide causal consistency without any additional metadata costs, it only works under the assumption that clients always issue operations to the same node (i.e., there is no client mobility) and that the *control*

Algorithm 10 Causal propagation in Arboreal

```
 1: procedure RECEIVE(<ClientWriteOp,key,value> from client)
 2:     ts ← getNextTimestamp(localTimestamp)
 3:     Trigger ApplyWrite(key, value, ts)
 4:     Trigger PropagateWrite(key, value, ts, _)
 5:
 6: function PROPAGATEWRITE(key, value, ts, from)
 7:     for all child ∈ myChildren() do
 8:         if parent ≠ from then
 9:             SEND(parent, <WritePropagation, key, value, ts>)
10:         if child ≠ from and key ∈ ChildObjects[child] then
11:             SEND(child, <WritePropagation, key, value, ts>)
12:
13: procedure RECEIVE(<WritePropagation,key,value,ts> from node)
14:     localTimestamp ← merge(localTimestamp, ts)
15:     Trigger ApplyWrite(key, value, ts)
16:     Trigger PropagateWrite(key, value, ts, node)
17:
```

tree is stable (i.e., there are no failures or node additions), which are both unrealistic assumptions for the edge. As such, Arboreal relies on an additional timestamping mechanism to enforce causal consistency in all scenarios.

Timestamping: For causality enforcement, in addition to causal dissemination, each node in Arboreal keeps track of two additional pieces of information: a local timestamp (TS) based on the physical time of the node, and the current *Branch Stable Time* (BST).

Whenever a client's write operation is received by a node, the current TS of the node is incremented, and the operation is tagged with this timestamp (as seen in Algorithm 10 - line 2). As the operation is propagated through its *replication tree*, the timestamp is propagated with it, being stored with the object data in each node. Additionally, if any node receives an operation with a timestamp higher than its current TS, it updates its TS to the timestamp of the operation (line 14), ensuring that any future operations will have a higher timestamp. As the TS is based on the physical time of the node, it also increases as time passes, even if no operations are being executed. This simple mechanism ensures that the timestamp of an operation is always higher than the timestamps of their causal dependencies.

In addition to the TS, Arboreal also employs a mechanism which we call *Branch Stable Time* (BST), illustrated in Algorithm 11. The BST of a node represents the minimum timestamp in that node's branch (i.e., the node itself and all of its descendants) of the *control tree* (line 7). As nodes in Arboreal are only aware of their direct children, the BST of a node is computed in a recursive and decentralized manner, by using the BST of its children. This mechanism works as follows:

1. The BST of a leaf node is always its current TS (as it has no children).
2. Periodically (line 1), each node sends its current BST to its parent (line 3).

3. Upon receiving the *BST* of a child (line 13), a node stores it, using it in any future computations of its own *BST*.

In addition to propagating the current *BST* to its parent, each node also periodically sends its *BST* and the *BST* of all ancestors (i.e., all nodes in the path between itself and the root of the *control tree*) to its children (line 4). The result is that each node in Arboreal always knows the *BST* of its direct children, its own *BST*, and the *BST* of all ancestors.

Algorithm 11 Branch Stable Time computation in Arboreal

```
 1: function OnBSTTimer
 2:     localBST ← computeLocalBST()
 3:     Send(parent, <UpstreamMetadata, localBST>)
 4:     for all child ∈ myChildren() do
 5:         Send(child, <DownstreamMetadata, localBST, ancestorBSTs>)
 6:
 7: function ComputeLocalBST
 8:     localBST ← localTimestamp
 9:     for all child ∈ myChildren() do
10:         localBST ← min(localBST, childBSTs[child])
11:     return localBST
12:
13: procedure Receive(<UpstreamMetadata,childBST> from child)
14:     childBSTs[child] ← childBST
15:
16: procedure Receive(<DownstreamMetadata,parentBST, ancestorBSTs> from parent)
17:     ancestorBSTs ← parentBST ∪ ancestorBSTs
```

The main advantage of the *BST* mechanism is that it provides a lower bound on the timestamps of future operations that can be generated in a branch of the *control tree*. For instance, a node with a *BST* of 10 knows that it has already received all operations with timestamps lower or equal to 10 from its descendants, and that no future operation with a timestamp lower than 10 will be generated by any node in its branch.

While this timestamping mechanism can be implemented by directly using the physical time of the nodes, this can lead to issues where client operations are being issued at a very high rate. In such cases, and taking into account that in Arboreal operations issued to a node must be tagged with increasing timestamps, if operations are issued faster than the physical clock of the node can advance, there are two options: (1) the node delays operations until its physical clock advances, which can lead to performance degradation; or (2) the node tags operations with timestamps higher than its current physical time, which would lead to other nodes that receive these operations having to do the same, leading to significant differences between the timestamps of different nodes, which would negatively impact the *BST* mechanism.

As such, instead of directly using physical time, Arboreal employs Hybrid Logical Clocks ($HLCs$) [51]. $HLCs$ combine physical time for monotonic advancement in each node with logical clocks, leveraging the key benefits of both. This allows Arboreal to capture causal dependencies between operations by using the logical part of the HLC, without sacrificing the monotonic advancement of physical time. In turn, this decreases the vulnerability to time synchronization issues, which could have a negative impact on the duration of migrations and fault recovery. In practice, $HLCs$ multiple operations to be tagged with the same physical time, using the logical part to order them correctly.

As we will see in the following sections, the use of these timestamping mechanisms is the key to allow Arboreal to provide causal consistency in every scenario, including failure recovery (discussed in Section 5.3.3) and mobile clients (discussed in Section 5.3.4).

We note that both the causal dissemination and the timestamping mechanisms require the hierarchical tree topology to function correctly. For the causal dissemination, the acyclic nature of the tree is key to ensure operations are always propagated after their causal dependencies, as using other topologies would require additional metadata (e.g., vector clocks). For the timestamping mechanism, the BST is directly tied to the tree structure, as it represents a lower bound on the future timestamps that can be generated in an entire branch of the tree. Using alternative topologies, such as unstructured peer-to-peer networks, would incur in significant communication and metadata overhead to ensure causal consistency [67], negatively impacting the scalability of the system.

5.3.3 Fault Tolerance

Unlike cloud environments, where individual nodes can fail, but it is unlikely that an entire data center does, in edge environments, we need to assume that entire edge locations can fail or become partitioned at any time. Therefore, Arboreal must not only be capable of recovering from failures but also provide data persistence and consistency guarantees when they occur.

5.3.3.1 Data Persistence

Arboreal provides a mechanism allowing applications using it to specify the *persistence level* of write operations. Effectively, this allows applications to specify how many nodes upstream in the *replication tree* a write operation must reach before it is considered to be persisted and hence having a reply being returned to the application. This mechanism is especially beneficial for applications requiring data persistence guarantees in more volatile edge locations. Its design prevents scalability issues by avoiding extra communication steps between nodes and the need for nodes to track the origin of each operation. The mechanism works in two steps: assigning a special *persistence identifier* to write operations as they are propagated upstream (i.e., from child to parent) and periodically sending *persistence level notifications* downstream (i.e., from parent to children) to inform them about the number of nodes that have persisted each operation.

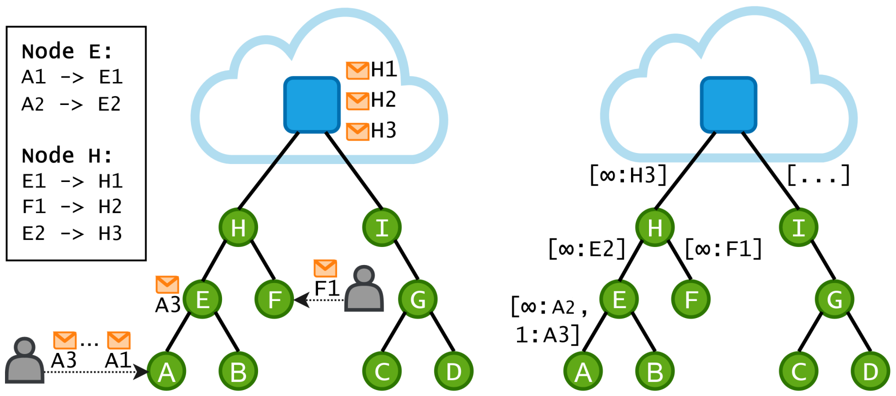


Figure 5.3: Data persistence in Arboreal.

Persistence identifier: The mechanism to assign *persistence identifiers* to write operations is illustrated in Algorithm 12, which is an incremental version of Algorithm 10. Arboreal nodes tag each write operation received from a client with a *persistence identifier* (line 3). This identifier is simply a counter local to each node which is incremented every time a node assigns it to an operation. Whenever a node receives a write operation from one of its children (line 10), it replaces the *persistence identifier* of the operation with its own (using its local counter), storing a mapping between the child's *persistence identifier* and its own, before propagating the operation to its parent (and possibly other children). Additionally, each node stores the highest *persistence identifier* received from each of its children (line 11). Both operations from clients and from children use the same local counter to assign *persistence identifiers*. In practice, this mechanism defines a total order of all operations received by a node, which is the same order in which they are propagated upstream. Operations propagated downstream (from a parent to a child) are not affected by this mechanism, not being assigned a *persistence identifier* (line 15).

The left side of Figure 5.3 illustrates an example of this mechanism in action. A client issued three write operations in node A, which were tagged with the *persistence identifiers* $A1$, $A2$, and $A3$. From these, the first two were propagated all the way to the datacenter, while the last one only reached node E. Additionally, an operation executed in node F also reached the datacenter.

The figure also depicts the mappings by each intermediate node. We can see that node E received the first two operations from node A, mapping them to its own *persistence identifiers* $E1$ and $E2$. The third operation from A has no mapping, as it was not yet propagated upstream. Node H received the two first operations originating

Algorithm 12 Persistence identifier assignment in Arboreal

```
procedure RECEIVE(<ClientWriteOp,key,value> from client)
    ts ← getNextTimestamp(localTimestamp)
    persistenceId ← ++persistenceIdCounter
    Trigger ApplyWrite(key, value, ts)
    Trigger PropagateWrite(key, value, ts, persistenceId, _)

procedure RECEIVE(<WritePropagation,key,value,ts, persistenceId> from node)
    localTimestamp ← merge(localTimestamp, ts)
    Trigger ApplyWrite(key, value, ts)
    if node ∈ myChildren() then
        highestChildId[node] ← persistenceId
        localPersistenceId ← ++persistenceIdCounter
        PersistenceMapping[localPersistenceId] ← {node, persistenceId}
        Trigger PropagateWrite(key, value, ts, localPersistenceId, node)
    else
        Trigger PropagateWrite(key, value, ts, persistenceId, node)
```

in A from E, however, it also received an operation from F between them, tagging all three operations according to this order. Note that, as these three operations are propagated from H to the datacenter, the datacenter has no information about the origin of each operation, only that they originated somewhere in the branch of H. By using this mechanism, Arboreal nodes do not need to track the origin of each operation, which enables it to scale to a large number of edge nodes without incurring prohibitive metadata costs.

Persistence level notifications: To determine the current persistence level of operations, each node periodically sends to all its children a *persistence level notification*, consisting of a list of pairs $(persistenceLevel, persistenceID)$, where each pair in the list signifies that all child operations tagged with *persistence identifiers* up to *persistenceID* have been persisted in *persistenceLevel* nodes upstream. This protocol is depicted on the right side of Figure 5.3, and presented in Algorithm 13, and works as follows:

1. The datacenter periodically sends a *persistence level notification* to all its children (line 2) with a single entry $(\infty, persistenceID)$, where *persistenceID* is the highest *persistence identifier* received from that child (i.e., the identifier of the most recent received operation). We use ∞ to represent the highest persistence level, as all operations reaching the datacenter are considered persisted. This can be seen in the figure, where the datacenter sends a notification to node H with the entry $(\infty, H3)$, meaning that all operations up to $H3$ have been persisted in the datacenter.
2. Upon receiving the *persistence level notification* from its parent, a node maps the *persistenceID* of each pair to the *persistenceIDs* of its children, increasing the *persistenceLevel* of each pair by one (line 8). If any operations from a child were not included in the received list, the node adds an entry with the highest of

Algorithm 13 Persistence level notifications in Arboreal

```
1: function OnPersistenceTimer
2:     if AmDatacenterNode() then
3:         for all child ∈ myChildren() do
4:             Send(child, <PersistenceMsg, {inf, highestChildId[child]}>)
5:     else
6:         for all child ∈ myChildren() do
7:             childPairs ← ∅
8:             for all {persistenceLevel, persistenceId} ∈ LastParentPersistence do
9:                 childId ← highestChildMapping(PersistenceMapping, persistenceId)
10:                childPairs ← childPairs ∪ {persistenceLevel + 1, childId}
11:            childPairs ← childPairs ∪ {1, highestChildId[child]}
12:            Send(child, <PersistenceMsg, childPairs>)
13:
14: procedure Receive(<PersistenceMsg,persistencePairs> from parent)
15:     LastParentPersistence ← persistencePairs
16:     Trigger CheckClientPersistence(LastParentPersistence)
```

those operations and a *persistenceLevel* of 1, signifying that the operation has not yet been persisted in any node upstream (line 11). This can be seen in the figure, where node H receives the notification from the datacenter, mapping the received *persistenceID* $H3$ to $E2$ and $F1$, which are the highest *persistence identifiers* of operations received from E and F, respectively. Node E executes a similar mapping, however, as it also received the operation $A3$, which was not included in the notification from H, it adds a new entry with a *persistenceLevel* of 1 to the notification it sends to A.

3. Additionally, upon receiving the *persistence level notification* from its parent (line 14), if any operations that originated in the node are included in the list, with a *persistenceLevel* equal or higher than the persistence level requested by the application, the node can reply to the client informing that the operation has been persisted.

In order to reduce communication cost, *persistence level notifications* are periodically sent to children piggybacked on the *BST* messages. Upon receiving confirmation that an operation has been persisted in the datacenter (i.e., with a *persistenceLevel* of ∞), nodes can forget all persistence information related to that operation. Regardless of the requested persistence level, this mechanism is always active for all operations to ensure no loss of operations during fault recovery (detailed in the next section).

We note that this mechanism is conservative, as we only consider the ancestors of a node to determine the persistence level of an operation. For instance, in the figure, the executions of operations originating from A in nodes B and F are not considered for the persistence level of these operations. Additionally, nodes only received persistence information after multiple communication rounds (e.g., A only knows its operations are persisted in the datacenter after 3 rounds), which can delay the reply to the client. However, we believe that this is an acceptable trade-off, since avoiding these limitations would require additional metadata and communication costs that would hinder the scalability of the system.

5.3.3.2 Fault Handling and Recovery

Due to the nature of edge environments, decentralized fault handling and recovery are essential for Arboreal. In this context, three main challenges need to be addressed: (1) rebuilding the *control tree* after node failures; (2) ensuring causal consistency during the rebuilding process; and (3) maintaining data persistence through failures and reconfiguration.

To address the first challenge, each node independently rebuilds the *control tree* after detecting that its parent failed. As mentioned in the previous sections, each node in Arboreal is aware of its ancestors (i.e., the nodes in the path between itself and the datacenter). Using this information, nodes attempt to connect to their grandparent if they suspect their parent has failed, falling back to the great-grandparent and so on, until reaching a non-faulty node, or the datacenter. Since every ancestor of a node replicates a superset of its data (as previously seen in Figure 5.2), this simple approach is enough to automatically repair not only the *control tree*, but also any *replication trees* that the node is part of. We note that, as we assume an asynchronous system, suspecting a node does not necessarily mean that the node has failed, but rather that there is a chance it might have, as it can just be temporarily slow. However, since a single failed (or just slow) node can block operations from being propagated through the *replication trees*, this mechanism minimizes the impact of such nodes, as when their children change parents, they become leaf nodes that cannot negatively affect other nodes. Simultaneously, this same mechanism can be used to avoid nodes in higher levels (i.e., closer to the root) of the *control tree* from being a bottleneck if they become overloaded due to a large number of children, as in such cases, (some) children can disconnect and reconnect to an ancestor.

For the second challenge, when connecting to a new parent, Arboreal employs a 3-step protocol to synchronize with its new parent, ensuring no violations of consistency guarantees. This protocol works as follows:

1. The (to-be) child node sends a *Sync Request* to the (to-be) parent node, which includes its current BST, and the list of the names of the data objects that it currently replicates (excluding their data), along with the timestamp of the last update of each object. Recall that the parent node will always replicate all objects that the child replicates;

2. The parent node adds the child as a new child, registering its BST. It then compares the timestamps of the objects received from the child with its own, checking for any outdated objects. It then replies with a *Sync Response* containing the child's new ancestor list, the BST of each ancestor, and a list of updates for the child's outdated objects (including the timestamp and the data of the object).

3. Upon receiving this message, the child updates its ancestor list and BSTs and updates its outdated objects with the received versions. It sends to the parent requests for any pending replica creation requests and any client write operations that were received while the node was disconnected from the tree. Finally, the child propagates a *Reconfiguration Message* to its children, containing their new ancestor list and BSTs, which is propagated through its branch.

Addressing the third challenge requires restarting the data persistence mechanism after a failure. For this, after the node whose parent has failed reconnects to the *control tree*, both the node and every node in its branch (after receiving the *Reconfiguration Message*) re-propagate any local write operations which were not yet persisted in the datacenter.

Although this synchronization protocol may seem complex, it allows each node to reconnect itself to the *control tree* without any centralized coordination. This decentralized approach enables Arboreal to recover from multiple failures in parallel, without any effect on the rest of the tree. We study the benefits of this mechanism in Section 5.4.

While so far we focused on the effects of node failures on the *control tree*, it is also important to discuss how clients themselves are affected by node failures. For this, there are three possible scenarios:

- Clients that are connected to a node which is not a descendant of a failed node do not experience any impact from the failure, as neither the failure nor the recovery process affect their connected node;

- If a client is connected to a node that remains operational, but one of its ancestors fails, the impact is limited. The client can continue to issue operations normally, which will be propagated through all the nodes of the disconnected branch. However, operations executed in the connected portion of the tree will not be visible by the client until the branch reconnects to the *control tree*. Additionally, if the client operations require persistence, the confirmations may be delayed until the branch reconnects;

- Finally, if the node to which the client is connected fails, the client must migrate to a new node (detailed in the following section - 5.3.4). Operations that were completed with persistence confirmation are guaranteed to be visible in the new node, but those without *may* have been lost, affecting the causal session of the client. We note that this case is not unique to Arboreal, as in any system with causal consistency, the failure of the node to which a client is connected to will possibly result in the client losing its causal session. The client can then re-execute lost operations or perform read operations in its new node to verify the persistence of those operations.

5.3.4 Client Mobility

In edge application scenarios with mobile clients, such as users with smartphones, Arboreal must seamlessly support clients moving from an edge node to a closer one while maintaining consistency guarantees (e.g., to support mobile AR applications [104]). This is a non-trivial task in a decentralized system like Arboreal, where the new node may not have any information about the existence of the client's previous node, which may even have failed. Therefore, in order to support client mobility in Arboreal, we need to design a protocol that allows clients to migrate between nodes while maintaining causal consistency without requiring those nodes to communicate with (or even know about) each other.

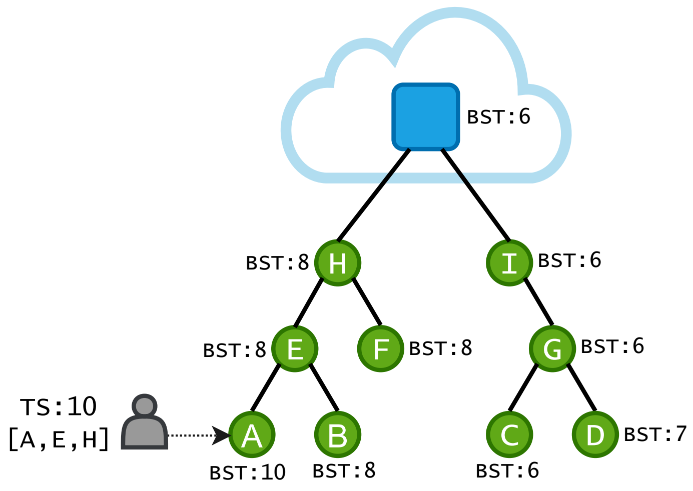


Figure 5.4: Client mobility in Arboreal.

Client state: To support this protocol, clients of Arboreal track two critical pieces of metadata: (1) the list of ancestors of its current node and; (2) a timestamp with the client's current causal dependencies. This timestamp is updated upon receiving responses to operations, and always contains the highest timestamp seen. Read operations return the timestamp of the object read, while write operations return the timestamp assigned to them by the client's node. Figure 5.4 shows the *BST* of nodes and the timestamp and list of ancestors of a client in an example deployment.

Mobility procedure: The combination of the client-side timestamp and list of ancestors with the node-side *BST* is what enables Arboreal to support quick client migrations without compromising causality. When a client connects to a new node (either due to client mobility or because its old node failed), it sends a migration request to the new node, containing its current timestamp and list of ancestors. The new node then compares the received list with its own list of ancestors, leading to one of two cases:

1. If the new node is not in the client's list of ancestors, this signifies a *horizontal migration* to a new branch of the *control tree*. In this case, the new node identifies

its closest ancestor that is also in the client's list of ancestors. It responds to the client only after receiving a BST from that ancestor greater than the client's timestamp. This ensures that the new node responds to the client only when it is certain that it has observed all operations the client depends on. Using Figure 5.4 as an illustrative example, if the client wishes to migrate from node A to node F, then it will need to wait until the BST of node H (the closest common ancestor) reaches a value of at least 10 (the client's timestamp) and that BST is propagated to node F. In practice, what this process ensures is that any operation that the client has seen or executed in its old node will be visible in the new node when it receives the BST from the common ancestor.

2. If the new node is in the client's list of ancestors, this indicates a *vertical migration* to an ancestor. In this case, the new node identifies which of its children is an ancestor of the client's old node (or which of its children is the old node itself). It responds to the client once it receives a BST from that child that is equal to or greater than the client's timestamp. In Figure 5.4, if the client migrates to E, it waits until E receives a BST from A with at least 10 (which should be quick, as the BST of A is already 10). In cases where the client migrates to the parent of a failed node, the new node immediately accepts the migration as there is nothing to wait for.

After the migration procedure is completed, the client receives an updated list of ancestors. Note that the metadata stored in the client only needs to be readable by Arboreal and may be opaque (e.g., encrypted) to the client itself, preventing information leakage about the internal organization of the system.

The duration of the migration process increases as the client moves farther from its old node, requiring the new node to wait for a BST from a more distant node in the *control tree*. However, in typical scenarios, we expect clients to mostly migrate to close-by nodes, resulting in swift migrations. We evaluate this in Section 5.4.

This mechanism might be overly cautious. For instance, in Figure 5.4, if the client aims to migrate to F, it depends on the BST of H, which, in turn, relies on the BST of B. However, node B might not have participated in any operation observed by the client, causing potential delays in migration completion. While we recognize this cautious approach may introduce unnecessary delays, alternatives that accelerate migrations typically involve additional metadata or explicit migration messages sent through the tree. Such approaches could compromise Arboreal's scalability and fault tolerance.

5.4 Evaluation

This section reports our extensive evaluation of Arboreal, studying the impact of its mechanisms on an emulated edge environment. We start by discussing the details of the implementation of our prototype (Section 5.4.1). We then present the experimental setup used to evaluate Arboreal (Section 5.4.2). We follow with a series of experiments evaluating multiple aspects of Arboreal, including its performance and scalability (Section 5.4.3), fault tolerance (Section 5.4.4), dynamic replication (Section 5.4.5), and client mobility (Section 5.4.6).

5.4.1 Implementation

Our Arboreal prototype, supporting all features outlined in Section 5.3, is implemented in around 4000 lines of code of Kotlin, leveraging on Netty [89] for network communication. It is open-source and extensible, comprising four modules: (1) the main Arboreal module manages the *control tree* and *replication trees*, handling message propagation and all other aspects detailed in 5.3; (2) the storage module allows the use of different storage backends. For our evaluation, we implemented an in-memory key-value store; (3) the client module serves as a proxy to allow clients to interact with the system; and (4) the management module allows nodes to collect information about each other, determining the initial *control tree* topology. Modules communicate asynchronously through message passing.

As we briefly discussed in Section 5.3.2.1, we leave to application developers the responsibilities of defining the best heuristic to shape the *control tree* and choosing the most suitable mechanisms to allow edge nodes to find each other and exchange information about themselves. This can be achieved through a variety of techniques, such as using centralized coordination services [46] or decentralized management overlay networks [28]. In our prototype, these responsibilities are delegated to the management module, which controls the lifecycle of edge nodes. In line with decentralization, in our prototype, nodes establish an overlay network using HyParView[64], with the datacenter node as a contact point. Upon entry, nodes utilize the overlay to propagate their information, including geographic location, while simultaneously collecting information about others. Using this data, the management module of each node employs a configurable heuristic to select an optimal parent, shaping the *control tree*.

In our implementation, we employ two heuristics to shape the *control tree* layout, both using the following equation:

$$cost = dist(self, candidate) + w \times dist(candidate, dc)$$

which assigns a $cost$ to each possible parent node $candidate$, based on its geographical distance $dist()$ to the node $self$ and the datacenter dc. This equation prioritizes parent proximity while also considering their distance to the datacenter, with a weight w. We used the value $w = 0.75$ in our experiments, as it resulted in the most balanced trees. The first heuristic simply selects the node with the lowest $cost$ as the parent, favoring nearby nodes which are in the general direction of the datacenter. This results in *control trees* with deep branches and a small number of children per node. The second heuristic limits the depth of the tree by assigning a *level* to nodes based on their distance to the datacenter. It only considers nodes with a level closer to the datacenter as potential parents, leading to wider layouts with less deep branches and a wider range in the number of children per node. The impact of these tree layouts, referred to as *deep* and *wide*, is evaluated in the following sections.

Along with the fully working prototype, we implemented a client driver in Java. Extending the codebase of YCSB [26], it allows evaluating Arboreal in a variety of scenarios, allowing clients to move between edge nodes due to node failures or to capture client mobility, select diverse persistence levels, and dynamically modify their workload, adapting to varying data access patterns.

5.4.2 Experimental Setup

We conducted experiments in a cluster of 10 machines, each having two AMD EPYC 7343 processors with 64 threads and 128 GB of memory, connected by a 20 Gbps network. We deployed a docker swarm across all machines, with an overlay network connecting all containers. A container with no resource restrictions on one machine models the datacenter, while up to 200 containers distributed across the others represent edge nodes. To better represent an edge environment, the computational resources of edge nodes were restricted to two virtual cores and 4 GB of memory. Each container executes an instance of Arboreal, as presented in Section 5.4.1.

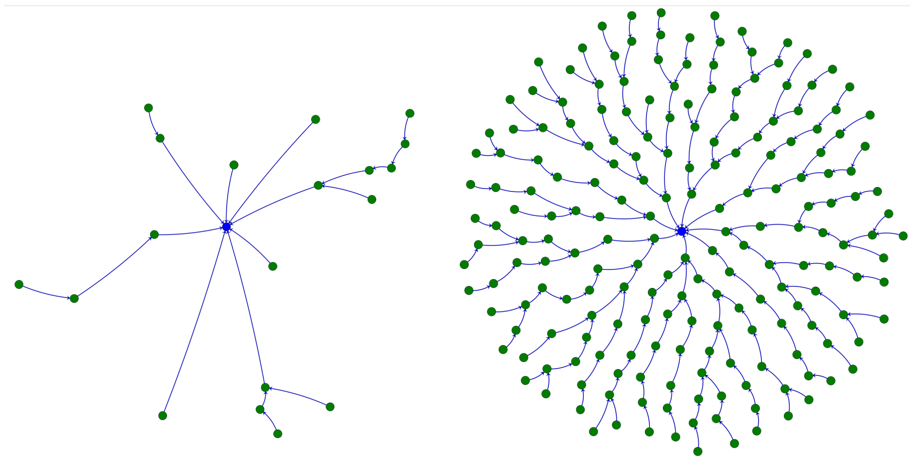

(a) 20 nodes (b) 200 nodes of 2

Figure 5.5: Example node distributions in a geographic region

To emulate a geographic region, we randomly distributed the 200 nodes across a virtual 2-dimensional space, representing edge locations, with the datacenter at the center. We used Linux *tc* to emulate latency between nodes based on their Euclidean distance, with a maximum latency of 150ms from an edge node to the datacenter. Experiments were conducted five times on three such distributions using either 20 or all 200 nodes. Figure 5.5 shows an example distribution with the formed *control tree*, using the *deep* layout. Additionally, we deployed 200 client containers, distributing them across the same virtual space and setting the latency between each client and edge node using the same method, with a minimum latency of 10ms.

Using this setup, we compare Arboreal with other solutions that provide causal+ consistency on the edge, namely a decentralized solution using vector clocks (Engage [12]) and solutions using a centralized topology to enforce causality (Gesto [1] and Colony [111]). For the former we use the provided code, while for the latter, as the code is not available, we mimic their centralized topology by implementing a version of Arboreal, named *centralized*, where all edge locations connect directly to the datacenter. We also compare Arboreal against Cassandra [52], for being an industry-standard cloud database that due to its configurable partial replication and peer-to-peer synchronization can be used in edge settings, having served as a baseline in contributions designed for both edge [43] and IoT [103].

5.4.3 Performance and Scalability

In our performance benchmarks, we evaluate the performance of Arboreal in terms of operation throughput and visibility times, comparing it against other causal+ solutions, Cassandra, and using different *control tree* layouts.

In these experiments, clients connect to their closest edge node and perform operations on data objects based on their location. The geographic region is divided into 8 equal segments, with each being assigned a data partition. Clients perform operations on data objects in their segment and the two adjacent segments. This setup assesses the performance of Arboreal with data locality, where clients are more likely to access nearby data objects. As the replication of data in Arboreal is dynamic and based on client access patterns, this results in each edge node replicating data objects from at least three partitions, with the data center replicating data objects from all 8 partitions. More complex scenarios with dynamic access patterns and client mobility are explored in Sections 5.4.5 and 5.4.6.

For Cassandra, as partial replication is based on a static placement, we configured each edge node as an independent cluster (*datacenter* in Cassandra terminology) and created partitions (*keyspaces* in Cassandra terminology) so that each edge node replicates all data objects from the three partitions accessed by clients, while the data center replicates all eight partitions. A similar approach was used to distribute partitions in Engage.

5.4.3.1 Throughput

Figure 5.6 shows the throughput of Arboreal compared to Engage and a centralized topology solution, varying the number of nodes and the number of distinct data partitions. We show the throughput of write operations only, as read operations execute locally in all solutions. As weak consistency models (which include causal consistency) allow nodes to respond to clients without coordination with other nodes, measuring throughput on the clients is unreliable, as operations may be processed in their local nodes at a higher

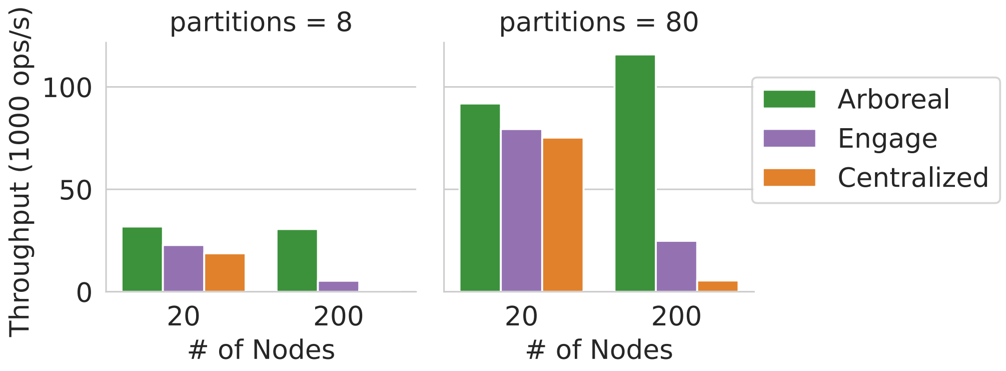


Figure 5.6: Throughput evaluation of Arboreal against other causal solutions

rate than they are replicated to other nodes. As such, the values displayed represent the maximum throughput measured in the data center node for each solution.

Two main conclusions can be drawn from these results: (1) Unlike existing causal+ solutions, Arboreal scales to hundreds of nodes without performance degradation. When compared to Engage, this difference is due to its use of vector clocks, which requires each individual operation to be tagged with a vector clock with the same size as the number of nodes, and each edge location to fully process that vector clock to determine the causal dependencies of the operation, which becomes prohibitively expensive as the number of nodes increases. When compared with centralized topologies (as the ones used in [1] and [111]), while it could be expected that a data center with high computing power could handle a centralized solution with a large number of edge nodes, this is not the case, as causality enforcement requires some form of (partial) serialization of operations, limiting parallelism; (2) Increasing the number of data partitions (reducing the number of data objects accessed by each client) and nodes (increasing the number of replicas for each data object) increases the throughput of Arboreal. This happens since each added node removes load from existing ones, allowing more operations to be processed in parallel. This is a key advantage of Arboreal's dynamic replication mechanism over state-of-the-art solutions.

Figure 5.7 shows the throughput achieved by Arboreal and Cassandra as observed by clients, varying the read/write ratio and number of nodes. Note the Y axis is in logarithmic scale. For more reliable measurements, we use ∞ persistence in Arboreal and *quorum* consistency in Cassandra, ensuring that writes are only acknowledged to clients after being replicated to the data center and a majority of edge nodes, respectively, preventing artificial throughput inflation. *Quorum* consistency in Cassandra also ensures clients observe the latest value of data objects, providing some consistency guarantees (although different from Arboreal, as it can violate causality across objects), making the comparison with Arboreal more fair.

In write-only scenarios (0% reads), Arboreal achieves higher throughput by leveraging its hierarchical topology and persistence mechanism. Nodes in Cassandra must send each

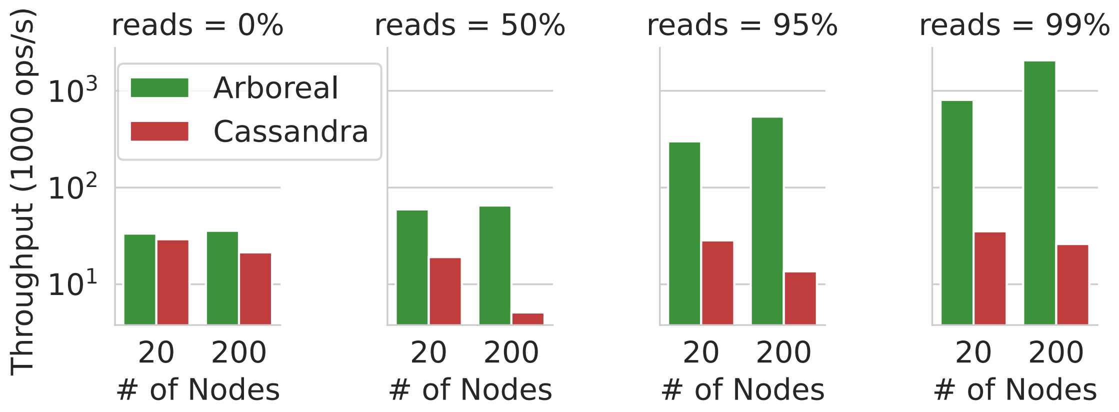


Figure 5.7: Throughput evaluation of Arboreal against Cassandra

write operation to all other nodes replicating the data object, then await acknowledgement from a quorum. In contrast, Arboreal sends write operations only to the parent (and any children replicating the object), and waits for the persistence acknowledgement. This greatly reduces message complexity, allowing higher throughput. With increased read operations, by processing reads locally, Arboreal outperforms Cassandra, which requires coordinating with a quorum before responding. While Cassandra can avoid coordination, sacrificing data consistency guarantees, to increase its throughput, Arboreal can process reads locally while providing causal+ consistency. Regardless, Figure 5.7 demonstrates that the hierarchical topology of Arboreal is more suitable for edge environments than traditional solutions designed for cloud environments.

5.4.3.2 Visibility Times and Tree Layouts

In this section, we explore the impact of different *control tree* layouts in Arboreal's hierarchical topology on the visibility times of operations and compare them with a centralized topology.

Utilizing the same setup as in the previous experiment, with 200 nodes and only write operations, Figure 5.8 presents the results in the form of a boxplot, where each box shows the distribution of the visibility times of operations in the different topologies. *Arboreal deep* and *Arboreal wide* represent the layouts presented in Section 5.4.1, with the *wide* layout limited to a depth of 4. The figure depicts visibility times for the closest remote node (1), the 5th closest, and all nodes. The values for 1 and 5 are crucial in an edge environment as clients in geographical proximity, connected to different but close-by edge nodes, are likely to access the same data objects.

Results indicate that utilizing a hierarchical topology with the *deep* layout is optimal for achieving low visibility times, enabling rapid operation propagation to nearby nodes. However, reaching all nodes requires traversing a significant number of hops, resulting in higher global visibility times. In contrast, the centralized topology always requires

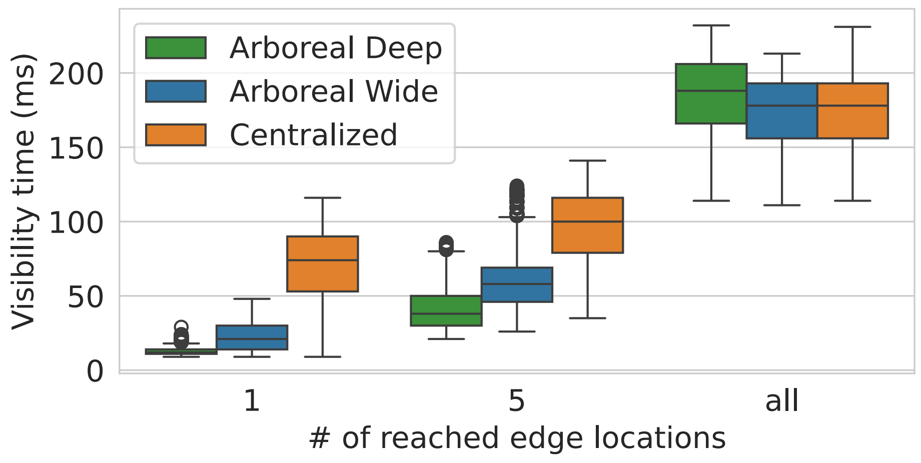


Figure 5.8: Visibility times of different topologies

propagating operations directly to the data center, resulting in much higher visibility times. The hierarchical *wide* layout serves as a balanced compromise, enabling slightly slower propagation to nearby nodes while matching the speed of the *centralized* topology in reaching *all* nodes. Overall, these results show that a hierarchical topology is better suited for edge environments than a centralized one. In Sections 5.4.5 and 5.4.6, we further explore the benefits of Arboreal's hierarchical topology.

5.4.4 Fault Tolerance

In these experiments, we examine the resilience of Arboreal to the failure of edge locations, exploring different persistence levels, failure rates, and failure patterns. As reads are processed locally, all clients execute only write operations. The *deep* layout is used for the *control tree* in all experiments, allowing a wider range of persistence levels to be evaluated.

5.4.4.1 Operation Persistence

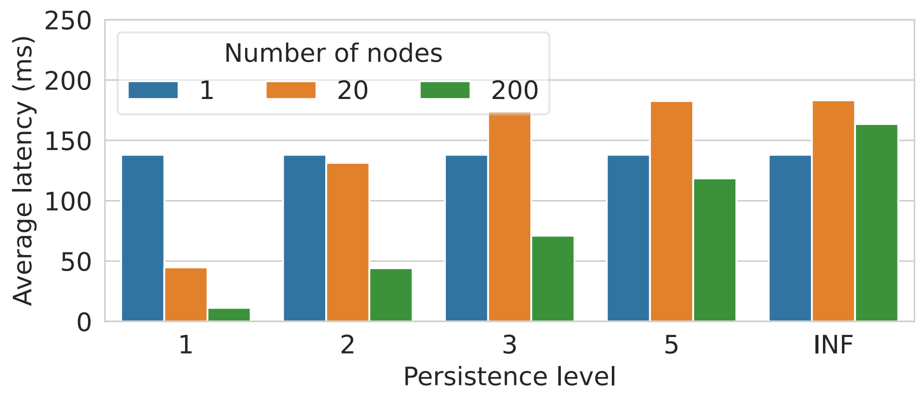


Figure 5.9: Latency impact of persistence levels

We start by evaluating the latency penalty of different persistence levels in Arboreal. Figure 5.9 shows the latency of write operations when varying node numbers and persistence levels. In this experiment, persistence messages are propagated from nodes to their children every 20ms. The configurations with one node and those with a persistence level of 1 are used as baselines for comparison, representing the latency of operations if clients were communicating directly with the data center and without the persistence mechanism, respectively. As expected, as the persistence level increases, so does the latency of operations, requiring more nodes to execute and acknowledge the operation before it is completed. In the 20 nodes experiments, as most nodes are within three hops from the data center, only persistence levels up to two provide latency benefits, with levels of three or above having similar latency to ∞. With 200 nodes, however, latency benefits are observed for persistence levels up to five. This means that, even in very unstable edge environments where data persistence is crucial, using Arboreal can be beneficial when compared with directly contacting the data center, particularly in a real-world scenario where only a small fraction of data objects might need high levels of persistence.

5.4.4.2 Effects of failures

Figure 5.10 shows the average latency over time as perceived by clients when 50% of all edge nodes fail within a 10-second period (30-40s in the figure), with persistence levels of 1 and ∞ and with both 20 and 200 nodes. The shaded area above each line represents the 99th percentile of latency (P99). In the results with a persistence level of 1, the average latency barely increases during the failure event, as clients that are connected to non-failing nodes remain unaffected even if their node is temporarily disconnected from the *control tree*. However, after each node failure, some clients must reconnect to a new node. As outlined in Section 5.3.4, the locality aspect of the *control tree* makes this process quick, with even the P99 not being significantly impacted. With a persistence level of ∞, the latency increase is more noticeable. This occurs because clients connected to nodes that did not fail, but whose ancestors did, will stop receiving replies to their operations until their branch of the *control tree* is repaired. As this process is fast and happens concurrently, the latency increase is minimal and does not accumulate over time.

Figure 5.11 shows the effects of a large percentage of edge locations (25% and 50%) failing simultaneously, with clients using ∞ persistence level. The results highlight the crucial role of a decentralized solution for efficient failure recovery. In both experiments, average latency increases significantly during the failure events with larger failure rates causing higher latency penalties, as more clients are affected. However, the *control tree* recovery time remains mostly consistent, regardless of the initial number of nodes and the percentage of failures. Again, this happens since disconnected nodes can reconnect to the *control tree* in parallel and without coordination, enabling Arboreal to swiftly recover from failure events in highly unstable edge environments with numerous edge locations.

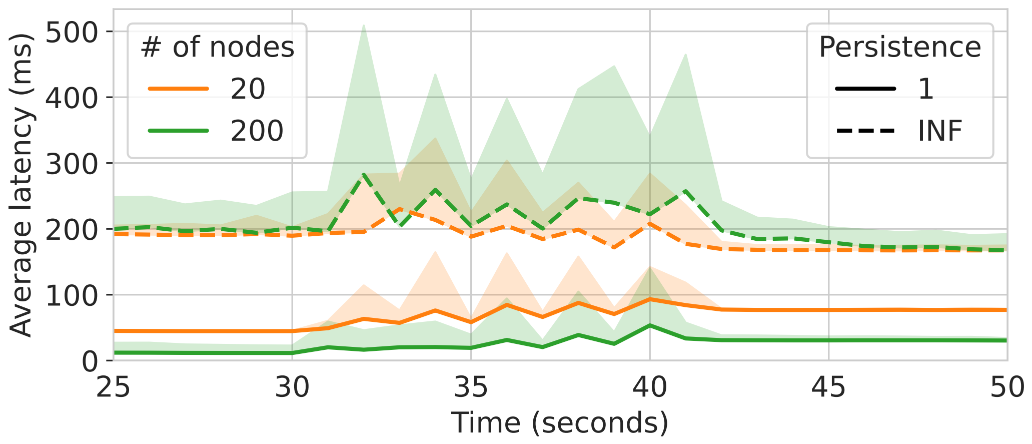


Figure 5.10: Continuous node failures (50%)

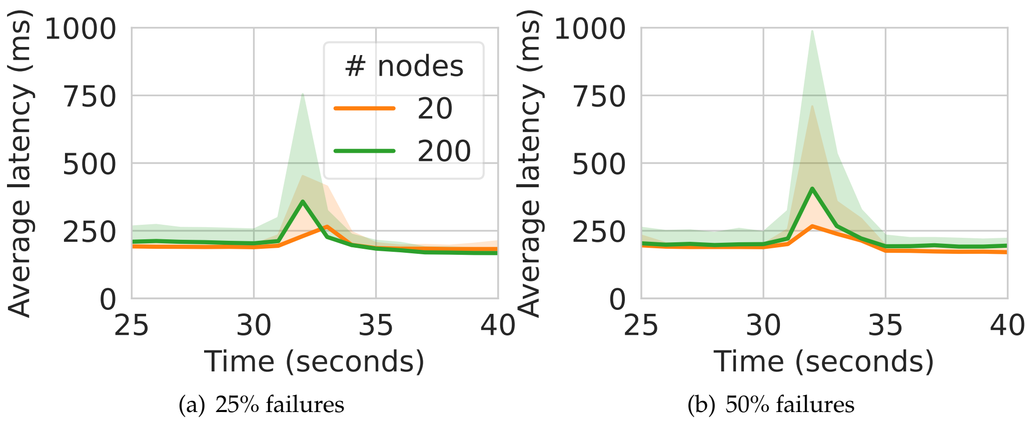
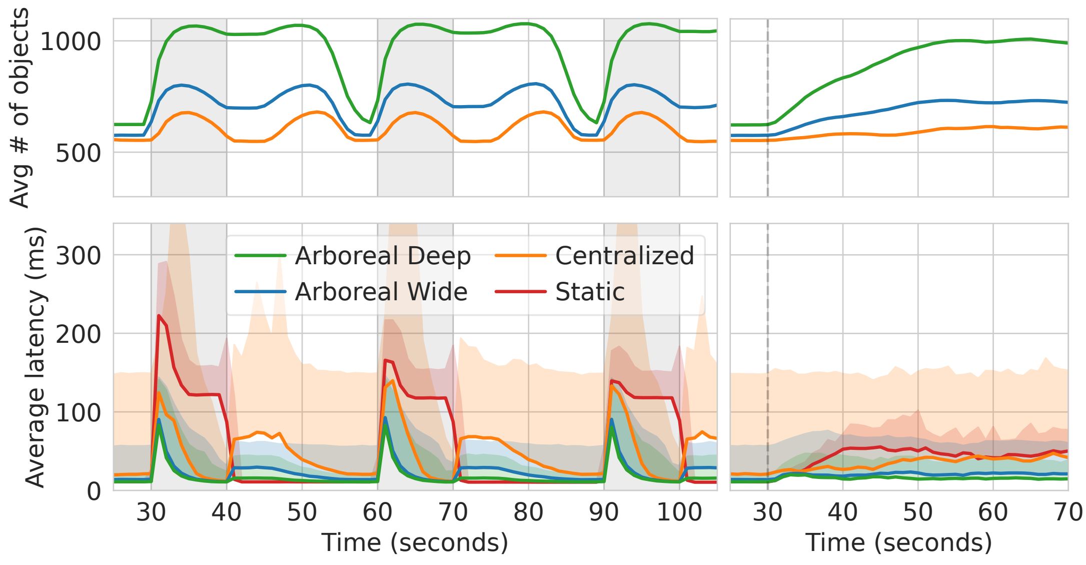


Figure 5.11: Simultaneous node failures with ∞ persistence

5.4.5 Dynamic Replication

While data locality is a primary assumption for edge computing (i.e., data is generated and consumed in the same geographical region), this does not mean that data is never accessed by far away clients. As an example, consider a social network application where users from a region might suddenly become interested in something that happened in another region. We explore how Arboreal adapts to such situations, analyzing the impact of different topologies on latency and object replicas, and comparing with solutions featuring static replication. Using the same data distribution as in Section 5.4.3, where data partitions align with geographical segments, we modify client behavior to simulate the social network scenario. Clients predominantly access local data with a workload comprising 99% reads, occasionally expressing interest in non-local partitions. Figure 5.12 reports the results of this experiment, using 200 nodes and a persistence level of 1, showing client-perceived latency over time (bottom plots) and the average number of replicated objects in all edge nodes. Two scenarios are considered: one where all clients synchronously change their interests to a new partition and later return to their original interests (5.12(a)), and another where clients randomly change their interests (5.12(b)). Similar to Section 5.4.5, we compare different layouts of Arboreal's hierarchical topology against a centralized topology, all using dynamic replication, and against a traditional static replication solution (Cassandra).



(a) Coordinated Pattern (b) Random Pattern

Figure 5.12: Dynamic client access patterns

In the coordinated scenario (Figure 5.12(a)), shaded gray areas represent periods when clients access data from remote partitions. While latency always increases in these periods, in static replication solutions this increase is more pronounced and remains high until clients return to their original interests (i.e., access patterns). On the other hand, the dynamic replication mechanism of Arboreal results in a smaller latency increase that quickly returns to regular values. Considering different tree layouts, there is a trade-off between the number of replicas of data objects and latency. While the *deep* layout results in lower latency, *wide* seems to provide the best trade-off, having a number of replicas close to the *centralized* and latency close to the *deep* layout.

In the random scenario (Figure 5.12(b)), clients start accessing data from remote partitions at the 30 seconds mark. In Arboreal, the resulting latency increase is minimal, as the dynamic replication mechanism is reactively creating local replicas of data objects (as suggested by the increase in the P99). Using static replication, the average latency remains high as there are always clients accessing data from remote partitions. The *wide* layout again demonstrates an effective trade-off between latency and the number of object replicas.

These findings show that Arboreal is able to efficiently handle dynamic client access patterns, proving suitable for edge applications even when data locality is not guaranteed. Its hierarchical topology enables more efficient adaptation to such scenarios compared to solutions with centralized topologies.

5.4.6 Client Mobility

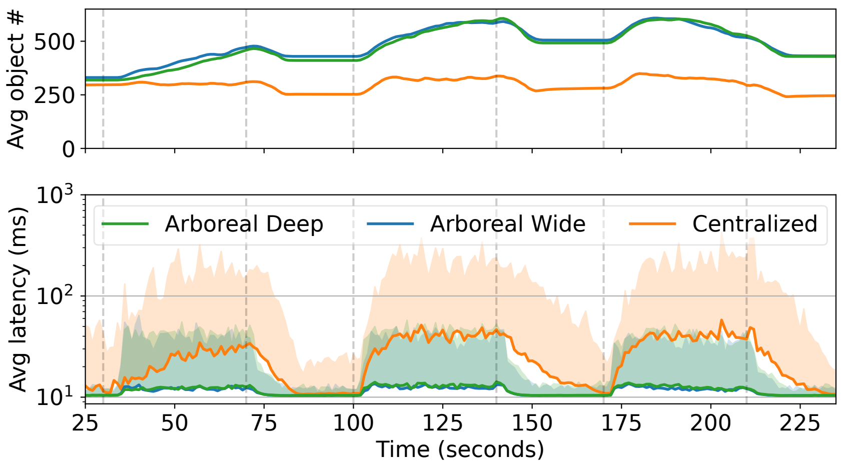
A key challenge for Arboreal is supporting mobile clients that change their connected edge location over time as they move. While crucial for applications such as location-based games or autonomous vehicles, this is also relevant for any Internet service (e.g., social networks), as clients may move in their day-to-day lives. In this section, we evaluate how Arboreal's dynamic replication mechanism adapts to various client mobility patterns, comparing different layouts of its hierarchical topology with a centralized topology.

For this, we modeled clients that move around the geographical space, changing their connected edge node whenever a closer one is available, but never changing the data objects they are interested in. This forces Arboreal to continually create new replicas of the objects that clients are interested in and garbage-collect non-useful replicas. Figure 5.13 reports the results of this experiment, showing, on the top plots, the average number of objects replicated in each edge location and, on the bottom plots, the client-perceived latency across different mobility scenarios, using 200 nodes with a persistence level of 1, and a workload composed of 99% reads. Replicas of objects are garbage-collected after five seconds without being accessed.

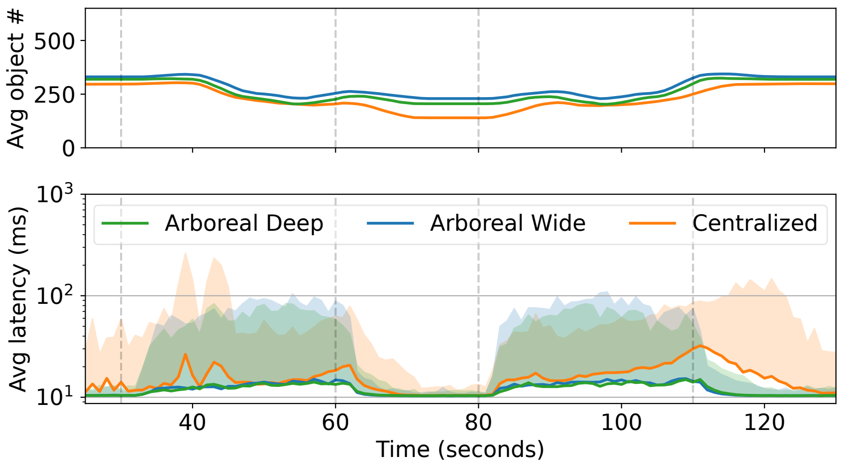
We modeled three mobility patterns: (1) a random pattern (Figure 5.13(a)), where clients move randomly for 40 seconds, stop for 30 seconds, and repeat the process two more times; (2) a commute pattern (Figure 5.13(b)), where clients move toward one of 5 fixed hot-spots (from second 30 to 60), and then return to their original position (from second 80 to 110); and (3) a mobile application pattern (Figure 5.13(c)), where every 30 seconds, clients move together to a new hotspot in the geographical space, emulating location-based AR applications.

The results show that the dynamic replication mechanism of Arboreal is quick to adapt to new client positions, creating new replicas of data objects while garbage-collecting unneeded replicas. Due to the location-based topology of the *control tree*, when a client moves to a new edge node, both nodes likely share a close ancestor replicating the data objects of interest. This allows the new node to quickly create local replicas without being required to contact the cloud data center, avoiding high latency spikes, especially in the random and commute patterns. This latency advantage comes at the cost of a higher number of object replicas being created, as Arboreal must replicate objects across all ancestors of each node. The mobile application pattern (Figure 5.13(c)) shows the worst-case scenario for Arboreal, as all clients move together to the same small subset of edge nodes, the advantages of having a hierarchical topology are less relevant, with the *centralized* topology showing lower latency values.

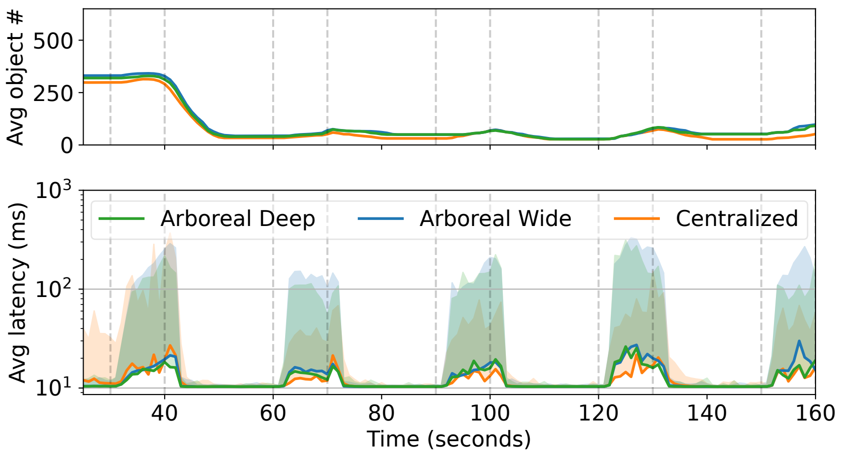
Whenever clients stop moving, latency quickly drops, showing that their accessed data objects have been replicated to their current edge location. This shows that Arboreal can adapt and converge in any mobility pattern, and its dynamic replication mechanism can



(a) Random Pattern



(b) Commute Pattern



(c) Mobile Application Pattern

Figure 5.13: Mobile clients with different mobility patterns

efficiently handle mobile clients.

5.5 Related Work

Numerous data management solutions have been proposed over the years, with a variety of consistency models and replication strategies. Being the most relevant to our work, in this section, we discuss traditional cloud-based solutions that provide causal+ consistency, discussing their limitations in edge environments, and solutions designed for edge environments, employing different consistency models.

5.5.1 Cloud-based Causal Consistency

Multiple solutions for data replication with causal+ consistency have been proposed in the past. However, a great majority of them assume datacenter deployments, with a limited number of replicas, and without supporting partial replication. Such solutions include COPS [69] and Eiger [70], which track causality using explicit dependencies. Other solutions employ vector clocks, resulting in their metadata growing proportionally to the number of replicas. These include, Orbe [32] and Cure [2], among others. ChainReaction [4] uses vector clocks with a size associated with the number of datacenters and not individual replicas, but still this results in excessive metadata overhead in large scale edge scenarios. As we showed in our evaluation, the metadata overhead of these solutions is prohibitive in edge scenarios. Some Causal+ replication solutions have been proposed that aim to limit metadata overhead, by using fixed-sized metadata. Examples include Saturn [16] and GentleRain [31]. However, be it due to the communication overhead or the lack of fault-tolerance mechanisms, these solutions are not suitable for edge environments. In conclusion, while providing causal+ consistency with a variety of strategies, the assumption of a small fixed and stable set of cloud datacenters makes existing cloud-based solutions unsuitable for edge environments, by lacking crucial features such as fault-tolerance handling mechanisms, partial replication, or scalable causality tracking.

5.5.2 Data Replication in the Edge

With the rise of edge computing, multiple solutions for data management on the edge have been proposed.

5.5.2.1 Eventual Consistency

Several solutions to replicate data on the edge with eventual consistency have been proposed. PathStore [84] was designed to allow short-lived stateless functions to access application data. For this, data is replicated on demand to edge nodes, which form a hierarchical topology rooted in a datacenter. SessionStore [86] extends PathStore to support session guarantees when clients move between edge nodes in [87] Feather [85] also uses a hierarchical topology, where data only flows upstream. For data freshness, queries are propagated to a configurable subset of edge nodes. DataFog [43] assigns spatial attributes to data, distributing data to edge nodes according to that data. While the hierarchical topology of some of these solutions is similar to Arboreal, they all lack consistency guarantees, providing only eventual consistency with, at most, some form of session guarantees for individual clients.

5.5.2.2 Strong Consistency

On the other hand, some solutions attempt to provide strong consistency. However, due to the highly geo-distributed nature of the edge, providing global strong consistency is impractical. As such, these solutions limit those guarantees to individual smaller, well-connected groups of nodes, relaxing consistency across groups. Examples include EdgeKV [105], FogStore [42], Colony [111], and DAST [22]. Despite being mitigated by the smaller size of groups and high connectivity assumptions, these solutions suffer from a lack of availability and fault-tolerance, as most strong consistency solutions do.

5.5.2.3 Causal Consistency

As for causal consistency on the edge, few solutions have been proposed. However, all of them have limitations that reduce their applicability in edge environments. Gesto [1] relies on a single centralized component in each region to enforce causal consistency, while Colony [111] (which also supports global transactional causal consistency along with strong consistency within groups) relies on the datacenter to validate all transactions. In these solutions, edge nodes (or groups in the case of Colony) cannot cooperate, requiring the cloud to mediate all interactions and disallowing nodes from progressing independently of the cloud. As we showed in our evaluation, this negatively impacts not only performance but also edge-specific aspects such as dynamic replication and client mobility. Engage [12] provides decentralized global causality in the edge, but relies on vector clocks which greatly limits its scalability.

5.6 Discussion

When considering the goals of this contribution (Section 5.2), Arboreal successfully achieves all of them. By leveraging on a hierarchical topology, Arboreal employs a causality tracking and enforcing algorithm which, contrary to state-of-the-art solutions, allows it to scale to hundreds of edge locations without performance degradation. Additionally, the fault-tolerance and reconfiguration mechanisms of Arboreal allow it to handle failures in a decentralized manner, quickly recovering from them with minimal impact on clients and the system as a whole. The dynamic replication mechanism of Arboreal is crucial for an edge environment where edge nodes are not able to store the entire dataset, adapting in a timely manner to client mobility and access patterns.

5.6.1 Trade-offs and Limitations

Just like with any other data replication protocol, Arboreal is not an ideal solution for all scenarios, having its own limitations and trade-offs. Note, however, that the different features of Arboreal were designed to work together, taking advantage of each other. For example, the fault recovery and client mobility mechanisms were designed to work with the dynamic replication and causality tracking mechanisms, which in turn require the hierarchical topology of the *control tree*. As such, some trade-offs come from this combination of features, and changing the design of one feature to improve them might negatively affect the others, or may require a complete redesign of the system.

Metadata compression and visibility times One of the main trade-offs in Arboreal (and every other causal consistency solution) is the size of causality tracking metadata. In order to allow Arboreal to scale to hundreds of edge locations, it was imperative to keep it fixed, avoiding alternatives such as vector clocks. This, however, results in limited parallelism in the execution of remote operations. While operations coming from different nodes (e.g., two different children) are guaranteed to be causally concurrent, and can be executed in parallel, operations coming from the same node (e.g., two operations from the parent) must in most cases be executed serially, as there is no way to determine if they are causally concurrent or not[1]. The impact of this is two-fold: (1) The visibility times of remote operations when the system is overloaded can be higher than in solutions with vector clocks, as operations must be executed in a serial manner, being delayed by the execution of unrelated operations; (2) As discussed in Section 5.3.4 when clients migrate between nodes, they must wait for the BST of the common ancestor to be updated, which depends on the timestamp (and execution of operations) from *all* nodes in that ancestor's branch, potentially causing higher latency for client mobility. Avoiding these limitations would require a causality tracking mechanism with metadata size proportional to the number of nodes (such as vector clocks), which would prevent Arboreal from scaling to the number of nodes it currently supports. As such, we believe this trade-off is the best compromise for the goals of this thesis.

Hierarchical topology and edge node resources As we mentioned in Section 5.3.2.1, the main goal of Arboreal is to provide a scalable and fault-tolerant data replication solution for the edge, and not to be an orchestrator for edge nodes. As such, while limiting the topology to a tree rooted at the datacenter, Arboreal does not decide what is the best distribution of nodes in the tree (i.e., which nodes should be parents of which). This decision requires knowledge of the available edge nodes, their geographical distribution, the historical or predicted access patterns of clients, and

[1] In fact, we can use the timestamp to allow some parallelism. If a node receives an operation followed by another with a smaller timestamp, it is guaranteed that they are causally concurrent. However, if the second operation has a higher timestamp (which is the most likely case), there is no way to know if they are concurrent or not, and must be executed in sequence.

the resources available in each node, among other factors, being a complex problem out of the scope of this thesis. Despite this, by employing a hierarchical topology and, more importantly, a dynamic replication mechanism based on subtrees of the *control tree*, Arboreal is implicitly assuming that leaf nodes have fewer resources, while nodes closer to the root have more resources. If this assumption is not followed, performance might be impacted, as nodes with fewer resources might struggle to keep up with the number of operations they must process. While such scenarios are best avoided, if they happen, as discussed briefly in Section 5.3.3, an overloaded node will likely be suspected of failing by its children, causing them to connect to its parent, which will redistribute the load. Regardless, just like with any other data replication protocol, proper configuration of the system (in this case, proper distribution of nodes in the *control tree*) is crucial for optimal performance.

Summary

In this chapter we presented Arboreal, a novel data management system for enabling stateful edge applications. Arboreal addresses the key challenges for supporting this model, being the first solution to provide global causal+ consistency while scaling to a large number of nodes, efficiently supporting client mobility and a novel dynamic partial replication mechanism that adapts to the access patterns of clients. This unique combination of features makes Arboreal suitable for many edge computing scenarios. Unlike existing causal consistency solutions, Arboreal operates in a fully decentralized manner, without relying on cloud infrastructure to mediate interactions between edge nodes. This allows Arboreal to take full advantage of the edge computing paradigm. Arboreal contributes to the goal of this thesis by providing a scalable and fault-tolerant solution for the edge computing *replica distribution level*.

In the next chapter, we present the secondary contributions of this thesis, which, while not directly contributing to the main goal, were crucial building blocks for the main contributions.

Publications and Artifacts

This contribution resulted in the publication of a research paper in the 2024 International Conference on Distributed Computing Systems (ICDCS 2024). Additionally, a fully functional implementation of Arboreal was made available as open source software, and is available at `https://github.com/pfouto/edge-tree`.

6 Other Contributions

During the course of this thesis, apart from the three main contributions discussed in the previous chapters, three other related contributions were made, in collaboration with other PhD students and research groups. In this chapter we discuss these contributions, presenting the context in which they were made and their relevance to the main contributions. The first one, Babel (Section 6.1), is a Java framework for implementing distributed protocols, which was used to develop the prototypes of all the main contributions of this work. The other two, *Bias Layered Tree* (Section 6.2) and *Engage* (Section 6.3), are two exploratory works on the context of edge computing which were important to better understand the challenges of this computing paradigm, influencing the design of the third main contribution of this thesis (Arboreal). At the end of each section, we discuss in more detail the relevance of each work to the main contributions of this thesis.

6.1 Babel

Babel is a novel Java framework that provides a programming environment for implementing distributed protocols and systems. Babel provides an event-driven programming and execution model that simplifies the task of translating typical specifications and descriptions of algorithms into practical prototypes with no significant performance bottlenecks. The abstractions provided by Babel are sufficiently generic to enable its usage for implementing a wide variety of distributed protocols and abstractions. Furthermore, Babel allows protocols to be implemented independently of the networking abstractions used to support the communication between different processes, while shielding the programmer from dealing with potentially complex concurrency aspects within, and across, distributed protocols. This does not mean that Babel is only useful for prototyping, or that it is non-performant. On the contrary, as we show in our evaluation, Babel was implemented with performance in mind, allowing developers to implement efficient, production-ready protocols.

6.1.1 Context and Motivation

Distributed systems are being increasingly used to support distinct activities in our everyday life. With the transition to digital platforms, distributed systems continue to grow in scale and consequently in complexity. This, in turn, increases the pressure for developing, validating, and testing different alternatives for building dependable abstractions that support the operation of many of these systems. This often requires comparing different alternatives found in the literature, which entails creating concrete and correct implementations of solutions described in research papers and books to be able to compare their performance under different conditions. This is the case for researchers developing novel solutions that wish to compare their proposals with the existing state of the art; for practitioners, that want to make informed decisions regarding the design and implementation of systems under development and operation; and even in classrooms, where students being exposed to dependable algorithms or large-scale systems can significantly benefit from the opportunity to implement and operate practical solutions.

Taking this into account, the motivation for this contribution comes from the observation (and personal experience) that, while many distributed protocols are conceptually simple to describe (for instance agreement protocols [58], state-machine replication [99], or large-scale peer-to-peer protocols [64]), implementing them tends to be error-prone and time-consuming, negatively affecting the activities of researchers, practitioners, and professors. This is not always due to the complexity of the protocols themselves, but due to the fact that implementing them requires dealing with several low-level aspects such as the use of communication primitives, scheduling and handling timed/periodic actions, multiplexing events and dealing with eventual concurrency issues that naturally arise when implementing complex systems. This usually results in implementations requiring a large effort from the programmer due to the amount of boilerplate code, which is error-prone and hard to maintain, which in turn leads to the actual protocol logic, usually described in pseudo-code in research papers and text books, to be lost within the implementation details.

6.1.2 Design

Babel is a framework that aims to simplify the development of distributed protocols within a process that executes in real hardware. A process can execute any number of (different) protocols that communicate with each other and/or protocols in different processes. Babel simplifies the development by enabling the developer to focus on the logic of the protocol, without having to deal with low level complexities associated with typical distributed systems implementations. These complexities include interactions among (local) protocols, handling message passing and communication aspects, handling timers, and concurrency-control aspects within, and across protocols, by dealing with network communication. For the latter, Babel hides communication complexities behind abstractions called *channels* that can be extended or modified by the developer, with Babel already offering several common alternatives that capture different capabilities (e.g., P2P, Client/Server). Babel is implemented in Java, taking advantage of its inheritance and polymorphism mechanisms, such that developers extend abstract classes provided by the framework to develop their own protocols and solutions. The strong typing provided by Java allows the framework to easily enforce expected behavior, while at the same time offering enough flexibility for the developer to implement any type of distributed protocol or system.

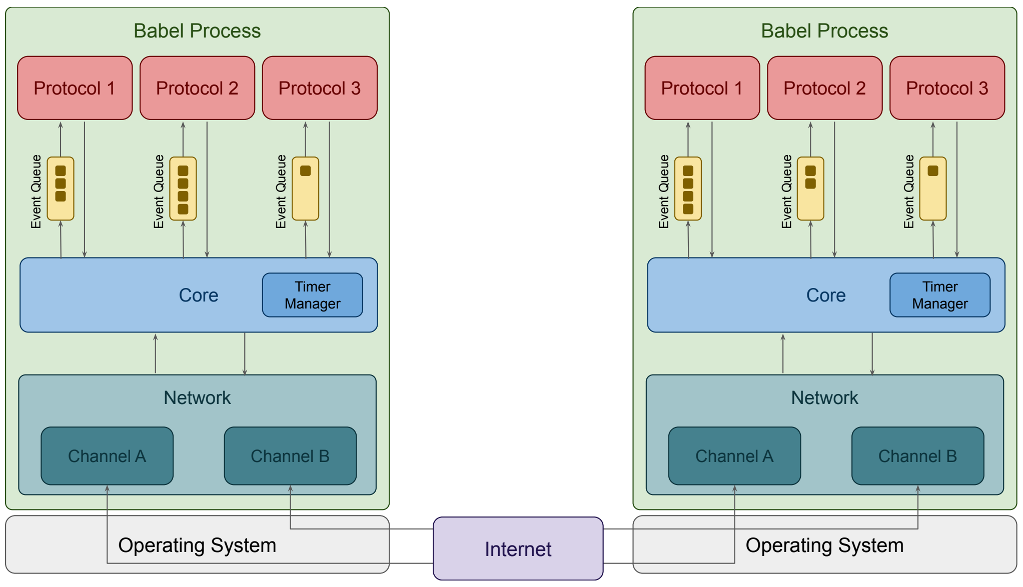


Figure 6.1: Architecture of Babel

Figure 6.1 presents the architecture of Babel. In the example, there are two processes executing Babel, each process being composed by three protocols and two network channels for inter-process communication. Naturally, any distributed system operating in the real world will be composed by more than two processes. The Babel framework is composed by three main components, which we now detail:

6.1.2.1 Protocols

Protocols (illustrated in red in Figure 6.1) are implemented by developers (i.e., the users of the Babel framework), and encode all the behavior of the distributed system being designed. Each protocol is modeled as a state machine whose state evolves by the reception and processing of events. For this purpose, each protocol contains an event queue from which events are retrieved. These events can be *Timers*, *Channel Notifications* from the network layer, *Network Messages* originated from another process, or *Intra-process* events used by protocols to interact among each other within the same process. Each protocol is exclusively assigned a dedicated thread, which handles received events in a serial fashion by executing their respective callbacks. In a single Babel process, any number of protocols may be executing simultaneously, allowing multiple protocols to cooperate (i.e., multithreaded execution), while shielding developers from concurrency issues, as all communication between protocols is done via asynchronous message passing.

From the developer's point of view, a protocol is responsible for defining the callbacks used to process the different types of events in its queue. The developer registers the callback for each type of event and implements its logic, while Babel handles the events by invoking their appropriate callbacks. While relatively simple, the event-oriented model provided by Babel allows the implementation of complex distributed protocols by allowing the developer to focus almost exclusively on the actual logic of the protocol, with minimal effort on setting up all the additional operational aspects.

6.1.2.2 Core

The Babel core is the central component which coordinates the execution of all protocols within the scope of a process. As illustrated in Figure 6.1, every interaction in Babel is mediated by the core component, as it is this component's responsibility to deliver events to each protocol's event queue. Whenever a protocol needs to communicate with another protocol, it is the core that processes and delivers events exchanged between them. When a message is directed to a protocol in another process, the core component delivers it to the network channel used by the protocol, which then sends the message to the target network address. That message is then handled by the core on the receiving process that ensures its delivery to the correct protocol. Besides mediating interaction between protocols (both inter and intra process), the core also keeps track of timers setup by protocols, and delivers an event to a protocol whenever a timer set up by the protocol is triggered.

6.1.2.3 Network

Babel employs an abstraction for networking which we name *channels*. Channels abstract all the complexity of dealing with networking, with each providing different behaviors and guarantees. Protocols interact with channels using simple primitives (`openConnection`, `sendMessage`, `closeConnection`), and receive events from channels whenever something relevant happens (e.g, a connection being established). These events are channel-specific and are handled by protocols just like any other type of event (i.e., by registering a callback for each relevant channel event). For instance, the framework provides a simple `TCPChannel`, useful for peer-to-peer protocols, which allows protocols to establish and accept TCP connections to/from other processes. This channel generates events whenever an outgoing connection is established, fails to be established, or is disconnected, and also whenever an incoming connection is established or disconnected.

The framework also allows developers to design their own channels if they need to enforce some specific behavior or guarantee at the network level for a protocol to function correctly. Network channels are implemented using Netty [89], which is a popular Java networking framework. A protocol can use any number of channels, and a channel can be shared by more than one protocol. In the example of Figure 6.1, two channels were instantiated by Babel. Channels within a Babel process are instantiated on demand by the Babel core when protocols are instantiated. Upon protocol instantiation, protocols define the channels they will be using, instructing the Babel core to prepare and instantiate the necessary network channels.

6.1.3 API

Babel is provided as a Java library. Protocols in Babel are developed by extending an abstract class - `GenericProtocol`. This class contains all the required methods to generate events and register the callbacks to process received events. Each protocol is identified by a unique identifier, used to allow other protocols to interact with it. There is also a special *init* event that protocols must implement, which is usually employed to define a starting point for the operation of the protocol. The API can be divided in three categories: timers, inter-protocol communication (within the same process), and networking.

6.1.3.1 Timers

Timers are essential to capture common behaviors of distributed protocols. They allow the execution of periodic actions (e.g., periodically exchanging information with a peer), or to conduct some action a single time in the future (e.g., define a timeout).

In Babel, using timers can be achieved as follows. First, the developer needs to create a Java class that represents a timer, with a unique type identifier and extending the generic `ProtoTimer` class. Additionally, a timer might have any number of fields or logic as the developer needs. Listing 1 shows an example of the usage of timers in a protocol, with a timer that contains a counter that can be decremented (lines 1-9). To use the timer in a protocol, a callback method must be defined to be executed once the timer expires (lines 16-18). This callback is registered by calling the method `registerTimerHandler`, which takes as arguments the unique type identifier of the timer, and the callback itself (line 13). After registering the handler, any number of single-time or periodic timers can be setup using the methods `setupTimer` or `setupPeriodicTimer`, respectively (line 14). Cancelling timers is also possible – for this, we simply call the method `cancelTimer` with the identifier of a previously setup timer as parameter (line 17).

6.1.3.2 Inter-protocol communication

As Babel supports multiple protocols executing concurrently in the same process, we offer mechanisms for these protocols to interact with each other, allowing them to cooperate or delegate responsibilities. For instance, we could create a solution where a message dissemination protocol takes advantage of a peer-sampling protocol to obtain samples of the system's membership. Listing 2 depicts such an example.

For this, Babel provides two types of communication primitives: one-to-one requests/replies and one-to-many notifications. Similarly to timers, requests, replies, and

Listing 1 Babel Timer Example

```java
public class CounterTimer extends ProtoTimer {
  public static final short TIMER_ID = 101;
  int counter;

  public CounterTimer(int initialValue) {
    super(TIMER_ID);
    counter = initialValue;
  }

  public int decCounter() { return --counter; }
}

public class CountdownProtocol extends GenericProtocol {
  public CountdownProtocol() { super("CountdownProtocol", 100); }

  public void init(Properties props) {
    registerTimerHandler(CounterTimer.TIMER_ID, this::handleCounterTimer);
    setupPeriodicTimer(new CounterTimer(10), 1000, 300);
  }

  private void handleCounterTimer(CounterTimer timer, long timerId) {
    if(timer.decCounter() == 0) cancelTimer(timerId);
  }
}
```

notifications need to extend a generic class. They can also have any extra arbitrary state and/or logic. Again, similarly to timers, we need to register a callback for each type of communication primitive (as seen in lines 4, 5, and 19). The callbacks all have the same parameters: the object that was sent and the identifier of the protocol that sent it. In order to send requests and replies, the methods `sendRequest` and `sendReply` are used. These methods take as parameters the Request or Reply to be sent and the destination protocol (lines 6 and 22). For notifications however, the method `triggerNotification` (line 25) does not require a destination, instead, every protocol that subscribed to that type of notification receives it (as exemplified in line 4).

6.1.3.3 Networking

Naturally, as a framework for distributed protocols, Babel also provides abstractions to deal with networking (including management of connections). For this, we provide different network channels with different capabilities. The interaction of protocols with channels is (mostly) similar across different channels. Listing 3 presents an example of a protocol that sends a ping message to a random peer, waits to receive a pong response message, and then chooses a new random peer to repeat the same behavior endlessly.

To use a channel, we start by setting up the properties for that channel. In the case of the `TCPChannel`, shown in the example, the required properties are the binding address and port for the listen socket (lines 6 to 8); other channels can consider different properties. Next, we create the channel by calling the method `createChannel`, passing the name of the channel and the properties object (line 9), and receiving an identifier representing the created channel. This identifier is useful if a protocol uses multiple

Listing 2 Babel Inter-protocol Communication Example

```java
1  public class DisseminationProtocol extends GenericProtocol {
2    public DisseminationProtocol() { super("DisseminationProto", 200); }

3    public void init(Properties props) {
4      subscribeNotification(MembershipChangeNot.ID, this::onMembershipChange);
5      registerReplyHandler(MembershipReply.ID, this::onMembershipReply);
6      sendRequest(new MembershipRequest(), MembershipProtocol.PROTOCOL_ID);
7    }

8    private void onMembershipChange(MembershipChangeNot not, short emitter) {
9      updateMembership(not.getChanges());
10   }

11   private void onMembershipReply(MembershipReply reply, short from) {
12     setMembership(reply.getNodes());
13   }
14 }

15 public class MembershipProto extends GenericProtocol {
16   public static final short PROTOCOL_ID = 300

17   public MembershipProto() { super("MembershipProto", PROTOCOL_ID); }

18   public void init(Properties props) {
19     registerRequestHandler(MembershipRequest.ID, this::onMembershipRequest);
20   }

21   private void onMembershipRequest(MembershipRequest request, short from) {
22     sendReply(new MembershipReply(currentMembership), sourceProto);
23   }

24   private void onConnectionEstablished(Peer peer){
25     triggerNotification(new MembershipChangeNot(peer));
26   }
27 }
```

channels simultaneously, to be able to select which channel to use to send a specific message, and to register different callbacks for different channels. Similarly to timers and requests/replies, we also need to create a class for each network message to be sent through a channel. Besides extending a generic class, and having a unique type identifier, the developer must also define a serializer for each message to enable the message to be encoded and decoded into network buffers (exemplified in lines 36 to 44). The serializer can be any function that converts Java objects to/from byte arrays, so one could use, for instance, a JSON library or Java's own serializer. We register the serializer for each message in a channel (since we could use different serializers for the same message across different channels) by using the method `registerMessageSerializer` (lines 10-11). We also need to register the callback for when a message is received, by using the method `registerMessageHandler` (lines 12-13). Different callbacks for the same message across different channels are supported. The callbacks to handle received messages take as arguments the message itself, the sender's IP address and port, the sender's protocol identifier, and the channel from which the message was received (lines 17 and 20).

A message can be sent using the `sendMessage` (lines 18 and 25) method, which takes

Listing 3 Babel Networking Example

```
public class PingPongProto extends GenericProtocol {
  List<Host> peers;

  public PingPongProto() { super("PingPong", 400); }

  public void init(Properties props) throws HandlerRegistrationException {
    peers = readPeersFromProperties(props);
    Properties channProps = new Properties();
    channProps.setProperty(TCPChannel.ADDRESS_KEY, props.getProperty("addr"));
    channProps.setProperty(TCPChannel.PORT_KEY, props.getProperty("port"));
    int cId = createChannel(TCPChannel.NAME, channProps);
    registerMessageSerializer(cId, PingMsg.ID, PingMsg.serializer);
    registerMessageSerializer(cId, PongMsg.ID, PongMsg.serializer);
    registerMessageHandler(cId, PingMsg.ID, this::uponPing);
    registerMessageHandler(cId, PongMsg.ID, this::uponPong);
    registerChannelEventHandler(cId, OutConnectionUp.EVENT_ID, this::uponOutConnUp);

    openConnection(chooseRandomPeer(peers));
  }

  private void uponPing(PingMsg msg,Host from,short sourceProto,int cId) {
    sendMessage(new PongMsg(msg.getValue()), from, TCPChannel.CONNECTION_IN);
  }

  private void uponPong(PongMsg msg,Host from,short sourceProto,int cId) {
    closeConnection(from);
    openConnection(chooseRandomPeer(peers));
  }

  private void uponOutConnUp(OutConnectionUp ev, int cId) {
    sendMessage(new PingMsg(System.currentTimeInMillis()), ev.getPeer());
  }
}

public class PingMsg extends ProtoMessage {
  public final static short ID = 401;
  private final long timestamp;

  public PingMsg(long timestamp) {
    super(ID);
    this.timestamp = timestamp;
  }

  public int getTimestamp() { return timestamp; }

  public static ISerializer<PingMsg> serializer = new ISerializer<>() {
    public void serialize(PingMsg msg, ByteBuf out) throws IOException {
      out.writeLong(msg.timestamp);
    }
    public PingMsg deserialize(ByteBuf in) throws IOException {
      long timestamp = in.readLong();
      return new PingMsg(timestamp);
    }
  };
}
```

as arguments the message to be sent, the destination address/port, the channel identifier (if more than one channel is being used) and optionally, the destination protocol. An additional parameter representing the connection to use can be passed. The interpretation of this parameter is however, channel dependent. Finally, each channel is responsible for generating notifications for relevant events that occur in it, for which the protocol can register callbacks. In the given example, the events generated by the `TCPChannel` (in this case we are only interested in a single event) are registered using the method `registerChannelEventHandler` (line 14).

6.1.4 Evaluation

The evaluation of Babel is based on two case studies of two popular distributed applications, where we implemented them using Babel, and compare the implementation effort and performance with other existing implementations. The case studies are:

P2P application We implemented a simple peer-to-peer application where two protocols, HyParView [64] and a Flood Dissemination protocol, interact with each other through the inter-protocol communication mechanisms of Babel to disseminate messages (via the flood protocol) through an overlay network (maintained by HyParView).

SMR application The state machine replication application is an in-memory key-value store that submits (client issued) operations to a consensus protocol for ordering. The consensus protocol exposes a notification that notifies the application of the execution order of operations. For the consensus protocol, we have implemented two variants of Multi-Paxos: a classic Multi-Paxos variant where the acceptors inform all learners of their accepted values, and; a distinguished learner Multi-Paxos variant, where the acceptors inform the leader (i.e., current proposer) of the accepted value, who then informs all learners.

Our evaluation is divided in two parts. In Section 6.1.4.1, we perform a code review of publicly available Multi-Paxos protocols and provide a comparison with our Multi-Paxos implementation with Babel. In Section 6.1.4.2 we present a performance evaluation on both case study applications.

6.1.4.1 Code Review

In this section we compare our implementation of Multi-Paxos with Babel with two publicly available Multi-Paxos implementations, MyPaxos[1] and WPaxos[2], chosen for having the highest number of stars on GitHub. These two implementations provide an interesting comparison since, while implementing very similar Multi-Paxos variants, are

[1]https://github.com/luohaha/MyPaxos
[2]https://github.com/wuba/WPaxos

Table 6.1: Multi-Paxos Implementations Lines of Code

Implementation	Component	Java Lines of Code
Babel-MultiPaxos-Classic	**Total**	735
	Main Protocol Logic	235
	Events	250
	Utils	250
Babel-MultiPaxos-DistLearner	**Total**	787
	Main Protocol Logic	247
	Events	290
	Utils	250
MyPaxos	**Total**	1814
	Main Protocol Logic	683
	Messaging	306
	Utils	487
	Other	338
WPaxos	**Total**	22909
	Main Protocol Logic	2283
	Messaging	862 (+78 Protobuf)
	Networking	1376
	Support	536
	Utils	497
	Other	17891

completely different in terms of complexity. MyPaxos presents a very simple but naive implementation, while WPaxos presents a more complex and performant implementation.

The only considerable design difference between the two implementations is that MyPaxos uses the classic message flow of Multi-Paxos, while WPaxos uses the distinguished learner variant. Table 6.1 summarizes the number of lines of code for each implementation, where it is noticeable that our implementations have significantly fewer lines of code. We now detail implementation aspects that were found while reviewing the code of each solution.

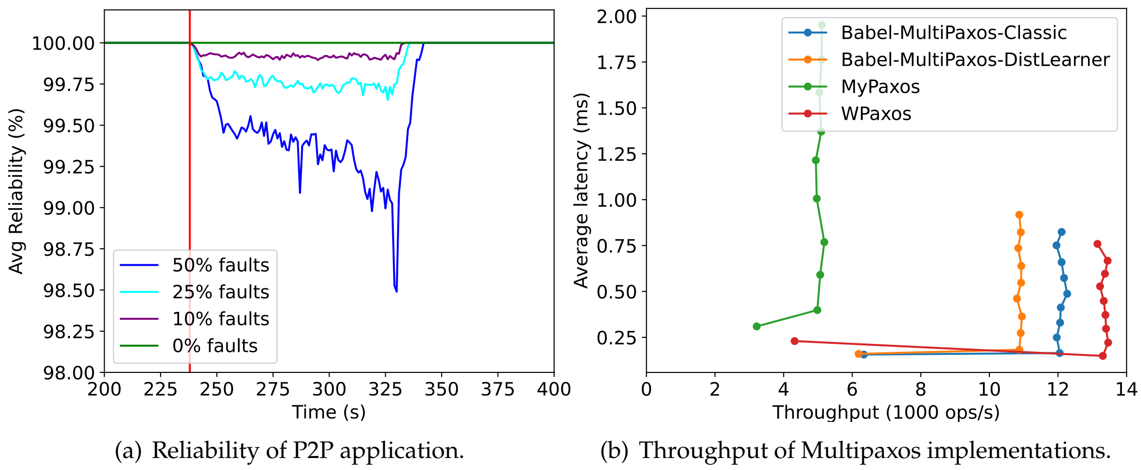
MyPaxos This implementation sacrifices performance and optimizations in favor of a small, simple-to-understand, and clear code structure. Each node has three main threads, corresponding to the three Paxos roles. These communicate by sending messages via TCP. The implementation uses a simple networking library relying on Java's native object serialization. When testing this implementation we observed two critical issues. The first was a concurrency error resulting in multiple threads accessing the same data structure. The second was caused by the protocol spawning a new thread for each Paxos instance that eventually crashes the JVM due to lack of memory. Both issues are directly addressed and avoided by using Babel: Babel avoids concurrency issues by having sequential execution of events within a protocol, and its timer management allows protocols to register timers without the overhead of creating and managing extra threads. As seen in Table 6.1, MyPaxos, while being simpler (and less stable), requires more lines of code to implement than any of our

Babel Multi-Paxos implementations. This is mainly due to having to deal with the execution support code (e.g., threading, messaging, timers) that is handled by Babel.

WPaxos This implementation is much more robust and performant than MyPaxos. It uses a custom communication layer, leveraging on the Netty [89] network framework, and serializing messages using Protocol Buffers [94]. While it also splits the Paxos roles in different Java classes, it handles the interactions between roles within the same process more efficiently than MyPaxos, by using direct method invocations. We were able to deploy and test this implementation without any problems. In comparison with our Babel implementations, it is clear that the use of Babel can greatly reduce the effort of programmers in creating working prototypes of distributed applications. While a considerable number of lines deal with features not implemented in our prototypes (mostly the 16275 lines of *Other* reported in Table 6.1) such as persistency, there is still a substantial difference in the number of lines for the main protocol logic. This difference comes mostly from the logic dedicated to creating and handling timers, multiplexing received messages and triggered timers, as well as making sure received messages are handled sequentially. In Babel, none of these aspects have to be handled by the developer.

6.1.4.2 Performance Evaluation

In this section we describe our performance evaluation using the two case study applications. Our goal was, on the one hand to verify the correctness of our implementations and how they behave under faults (P2P application), and on the other hand to provide insights about the relative performance of these implementations when compared with standalone ones (SMR application).



(a) Reliability of P2P application.

(b) Throughput of Multipaxos implementations.

Figure 6.2: Performance evaluation results

P2P Application Our experiments with the P2P application showcase a typical P2P evaluation with a network with a considerable number of nodes (in this case, 100). In our experiments we measure the average reliability of message delivery in four different scenarios: *i*) a fault-free scenario; and fault scenarios where different fractions of nodes fail simultaneously *ii*) 10% faults; *iii*) 25% faults; and *iv*) 50% faults.

Figure 6.2(a) presents the results of all scenarios, with a vertical red line indicating the point of the experiments where faults were introduced. The x-axis represents the time of the experiment when messages were being disseminated. The y-axis represents the average reliability of message delivery. After the introduction of faults in the experiment, the average reliability drops marginally (up to 1%), this is because the overlay maintained by HyParView is very stable and maintains (high) connectivity. Nevertheless, there are a few nodes that become temporarily disconnected and may not receive messages until they reconnect. The highest reliability drop in our experiment is when we fail 50% of the nodes in the network. These results are aligned with the expected behavior of the protocol, as described in the original paper, showing that Babel can be used to implement correct and reliable distributed protocols.

SMR Application Our experiments with the SMR application present an experimental comparison of different Multi-Paxos implementations. In more detail, we compare our Multi-Paxos implementations with the public ones described previously. To make comparisons fair, we adapted our key-value store application to work with MyPaxos and WPaxos. For this experiment, we execute 3 replicas of the key-value store, while increasing the number of client threads, which issue operations (that must be ordered by Multi-Paxos) in a closed-loop with a 50% read/write ratio.

Figure 6.2(b) presents the average throughput (in the x-axis) against the average latency (in the y-axis) of operations for each implementation. We note that MyPaxos is the implementation with the least throughput. This is most likely due to the serialization used by the implementation which is known to have non-negligible overhead. In contrast, Babel Multi-Paxos implementations show comparable performance results with the significantly more complex WPaxos implementation, which is able to achieve higher throughput due to the optimizations performed in the code. This small difference in performance when compared with the much higher development effort and complexity of WPaxos supports our claim that Babel can be used to implement efficient and performant distributed protocols with relatively low effort.

6.1.5 Discussion and relevance

Babel was designed with the goal of simplifying (both in terms of complexity and effort) the development of distributed protocols. While the evaluation of Babel shows these goals were achieved, Babel has also proven to be a valuable tool in multiple practical contexts. More specifically: *i*) the prototypes of all three main contributions of this thesis were developed using Babel, resulting in a significant reduction in development time and effort, while proving that it can be used for implementing reliable and performant distributed systems; *ii*) multiple research papers (from PhD and Msc students) and MSc thesis in NOVA used Babel to implement their prototypes; *iii*) Babel has been successfully used in laboratory classes in the context of the Algorithms and Distributed Systems course at NOVA, allowing students to easily implement and test distributed protocols and; *iv*) the TaRDIS project [108], a research project funded by the EU Horizon Europe program, with the goal of easing the complexity and reducing the effort of building correct and efficient heterogeneous swarms, relies on (and extends) Babel for implementing its decentralized algorithms and protocols.

Publications and Artifacts

The source code for Babel is open source and fully available at `https://github.com/pfouto/babel-core`. It resulted in the publication of a research paper in the 2022 41st International Symposium on Reliable Distributed Systems (SRDS) [38].

6.2 Bias Layered Tree

Bias Layered Tree is a novel decentralized overlay network that builds a robust hierarchical tree-based topology connecting and allowing the management of large numbers of nodes across the cloud and edge environments. Contrary to previous proposals, that assume the edge environment to be composed of mostly homogeneous devices, this contribution embraces existing heterogeneity and exploits the location of computational resources to devise a (partially) self-organizing overlay network that can be exploited both to provide membership information to applications, but also to efficiently disseminate management information across edge locations.

6.2.1 Context and Motivation

Edge computing has emerged as a solution to address existing limitations of cloud computing for bandwidth-heavy and time-sensitive applications, by moving (some) computations from bandwidth saturated cloud infrastructures closer to client devices, where data is effectively produced and consumed. However, existing materializations of the edge computing paradigm take limited advantage of computational and storage power that exists in the edge and between client devices and the cloud. Most of these leverage static hierarchical topologies (e.g., Fog Computing) to pre-process data before sending it to the Cloud, which limits the advantages that can be extracted from the edge computing paradigm. The challenges of managing large scale systems in a (mostly) decentralized way have been previously tackled in the context of peer-to-peer systems, leading us to believe that leveraging peer-to-peer solutions can enable novel applications to fully exploit the potential of the edge.

As such, the goal of this work is to devise a specialized overlay network that can be used to manage large numbers of computational resources across the cloud and edge environments. For this, and contrary to previous proposals that assume the edge environment to be composed of mostly homogeneous devices, we aim to embrace existing heterogeneity and exploit the location of computational resources to devise a (partially) self-organizing overlay network that can be exploited both to provide membership information to applications, but also to efficiently disseminate management information across edge devices.

6.2.2 Design

Bias Layered Tree is a decentralized overlay network that we have designed to cope with the requirements of a resource management platform to support edge computing. The protocol is inspired in HyParView [64] by maintaining two partial views of the system (active and passive). *Bias Layered Tree* manages the contents of the active view to build and maintain a tree topology across nodes in the system by leveraging a *level* property, that encodes the distance of a given node to the cloud infrastructure as well as serving as an indicator of its capacity.

6.2.2.1 Protocol State

An $activeView$ encodes nodes with whom the local node exchanges information frequently. This partial view is divided into three components. The first component has at most a single node, which is the parent node, and must be on a lower level (i.e., closer to the cloud) than the local node. The second component contains a fixed amount of peers that have the same level as the local node, which may or may not share the same parent. The third component maintains the identifiers of children nodes (which are on a higher level, farther from the cloud), having no strict limit to its size. The first and third components are symmetrical between peers, meaning that if a node is in the first component of the active view of another node, that other node must be in the third component of the active view of the first node.

The $passiveView$ contains information about peers across different levels that is used to recover from faults and as a source of potential candidates to improve the tree topology maintained by *Bias Layered Tree*. Each node strives to have a given number of identifiers for peers in the same level, and the number of identifiers for the remaining levels decreases as the distance of that level increases.

Notice that *Bias Layered Tree* strives to also bias the topology of the generated tree so that links are preferably established among nearby peers. To this end, we rely on the IP address common prefix as distance criteria, with nodes keeping the best peers according to that criteria in their active and passive views. To effectively build the tree, nodes are fully delegated to select their parent according to their local view of the system.

6.2.2.2 Building the tree

Nodes in *Bias Layered Tree* start the process of joining the tree by sending a *Join* message to the root node (in level 0), which is assumed to reside in a cloud datacenter. Upon receiving this message, the root node creates a *ForwardJoin* message, whose goal is to gather information about adequate peers for the new node, populating its passive view and allowing it to select a parent node. For this, this message is propagated across the overlay through a biased random walk, being propagated to multiple nodes in each level until reaching the node in the new node's level which is closest to it according to the IP address distance criteria. This node then forwards the message to the new node. Because the *ForwardJoin* carries information about the network topology, each node that processes the message takes advantage of this information to update its local views. When receiving the *ForwardJoin* message, the new node populates its passive view, and selects the best parent node from a lower level, again using the IP address distance, adding it to its active view, and being added as a child in the parent's active view.

6.2.2.3 Maintaining the tree

To maintain the tree connected, *Bias Layered Tree* relies on periodic information exchanges across nodes to keep their passive views up-to-date with the closest node. This is achieved via a *Shuffle* message, that is periodically sent to nodes in both views. When sending this message to a peer, the local node selects a sample of nodes that are closer to the chosen peer both in its active and passive views. Upon receiving the *Shuffle* message, the peer updates its passive view with the information enclosed in the message, always aiming to keep the closest nodes (considering the IP address distance) in its passive view. The peer then replies with a *ShuffleReply* message, containing its own sample of nodes, which then updates the passive view of the local node.

When a node fails, the tree defined by *Bias Layered Tree* may lose its connectivity. To recover from this, nodes that had the failed node in their active view must select a replacement. Both failed parents and siblings are replaced by selecting the closest node in, respectively, a lower level and the same level, from the passive view. No action is taken when a child crashes. Even if no node fails, nodes periodically check if the passive view (which evolves with the exchange of *Shuffle* messages) contains closer nodes, replacing the parent and siblings if necessary.

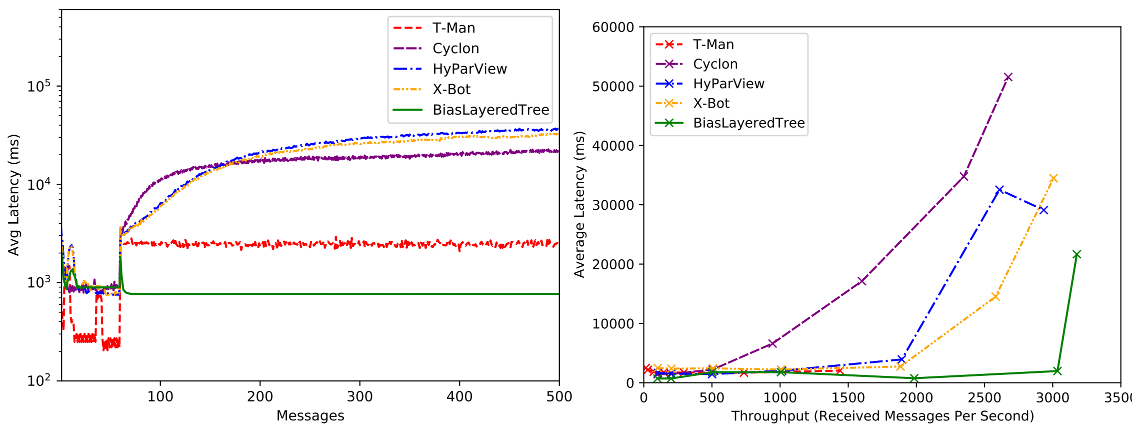
6.2.3 Evaluation

Bias Layered Tree was evaluated in an emulated cloud-edge environment, using a network composed of 100 docker containers. Containers were assigned IP addresses and a level, with nodes in smaller levels being assigned more CPU quota and network bandwidth than nodes in higher levels. Latencies between nodes were emulated according to the IP address distances.

We compare *Bias Layered Tree* against relevant baselines implemented in the same code base: HyParView [64], X-Bot [62], Cyclon [114], and T-Man [47]. We evaluate the overlays by disseminating data across all nodes in the network, using two different dissemination protocols: a simple Flood Gossip protocol, and the more efficient Plumtree protocol [63]. Being the most efficient, in this document we focus on the results of the Plumtree protocol.

6.2.3.1 Performance Evaluation

We start by presenting the performance evaluation in fault-free scenarios, measuring the average latency for delivering messages and the dissemination throughput (delivered messages per time unit).



(a) Latency using Plumtree. (b) Throughput vs Latency Plumtree.

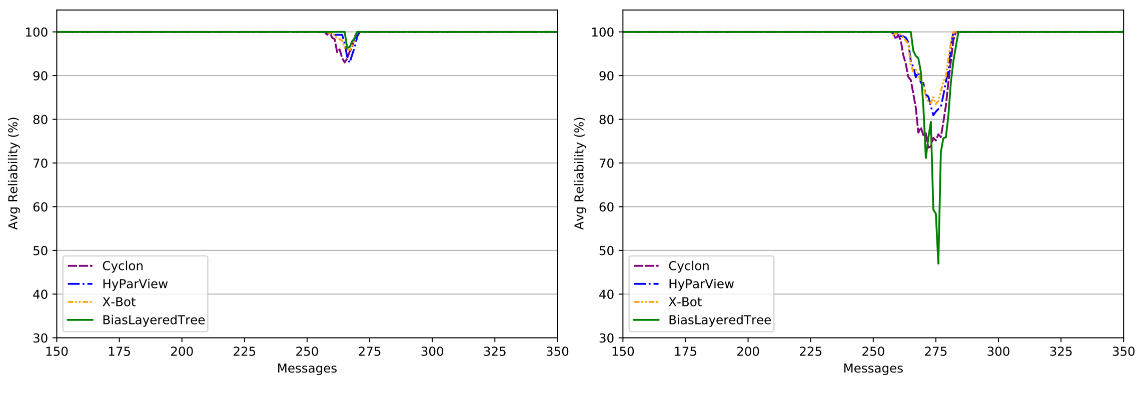
Figure 6.3: Performance of protocols under heavy load

Latency under Load Figure 6.3(a) reports the average latency (in the y-axis in logarithmic scale) measured for each message (in the x-axis) for each protocol under heavy load. In this experiment, messages are generated with a frequency of 1 message per second. During the first 60 seconds, only a single node generates messages, this is required to allow Plumtree to converge to an adequate configuration. Afterward, all nodes start to send messages for 440 seconds. The size of the payload of each message is 20.000 bytes. In the figure, we can see that the latency of most protocols increases over time, as the network bandwidth becomes saturated and messages start to be queued in the operating system. We note that T-Man presents lower latencies, but only because it is not able to deliver all messages. However, our solutions, by establishing an overlay with nodes with higher capacity at higher levels, is able to make a better use of the available network bandwidth, and therefore, present the lowest latency (and without increasing over time) among all solutions.

Throughput Figure 6.3(b) reports the (maximum) throughput (in the x-axis) and latency (in the y-axis) observed for each protocol for increasing rates of broadcast messages being generated by second, varying from 1 (first point on each line) to 100 messages per second per node (last point on each line). The payload size of each message is 1000 bytes. These experiments run for 4 minutes, and we collect metrics of received messages and their average latency every 10 seconds. Each point represents the maximum number of messages received with minimum average latency. The figure shows the results when using Plumtree to disseminate messages. In this figure we can see that, while other solutions start saturating at around 20 messages per second, our solution is able to keep a higher throughput and lower latency until approximately 50 messages per second, when it also starts saturating. As explained previously, this happens because the overlay created by our solution takes into account the bandwidth of nodes (encoded on the level), leading nodes with less bandwidth to become leaves in the Plumtree dissemination tree, preventing them from becoming bottlenecks.

6.2.3.2 Fault-Tolerance Evaluation

Considering fault-tolerance, we tested how the various protocols react to different failure scenarios, by failing 10%, and 50% of all nodes in the network simultaneously. For these experiments, we measured reliability per message achieved by each protocol during the failure, evaluating the ability of the different overlays to reconfigure themselves in the presence of node failures. Messages are broadcast every second for 400 seconds.



(a) Plumtree 10% Faults.

(b) Plumtree 50% Faults.

Figure 6.4: Performance of protocols under heavy load

Figure 6.4 shows the results for these experiments, showing that with a low level of failures (Figure 6.4(a)), *Bias Layered Tree* is able to recover faster and regain 100% reliability faster. This can be explained by the limited impact of these faults on the tree topology. However, higher numbers of faults (statistically) affect an increasing fraction of interior nodes, leading the tree to break with a greater negative impact in the reliability of the dissemination protocol (Figure 6.4(b)). However, the recovery speed of our tree is on par with the other protocols, returning to 100% reliability just as fast.

6.2.4 Discussion and relevance

Bias Layered Tree was developed with the goal of exploring the potential of using hierarchical peer-to-peer decentralized overlay networks to manage a large number of nodes across cloud and edge environments. This work was developed in collaboration with another PhD student, and was my first contribution in the context of edge computing. As the envisioned use of this work is to disseminate control and management information across edge locations, it does not directly address the goals of this thesis, as it does not include data replication (and, consequently, does not address data consistency). Regardless, the hierarchical overlay topology devised in this work showed to be a valuable approach to connect and manage a large number of nodes in the edge and, as such, it was fundamental as the first step towards the development of the third main contribution of this thesis (Arboreal), being the inspiration for the hierarchical overlay topology used in that work.

Publications and Artifacts

The prototype of *Bias Layered Tree*, along with the other baselines is open-source and available at `https://github.com/pedroAkos/EdgeOverlayNetworks`. This work resulted in a research paper that was published in the 2020 IEEE 19th International Symposium on Network Computing and Applications (NCA) [28].

6.3 Engage

Engage is a storage system that offers efficient support for session guarantees in a partially replicated edge setting. Its goal is to minimize the visibility times of operations, allowing clients to quickly migrate between edge locations to access data that is not in their local replica, while simultaneously optimizing the use of the limited bandwidth of edge nodes. For this, *Engage* combines the use of vector clock with a distributed metadata propagation service, leveraging the hierarchical nature of the metadata service to offload the cost of message dissemination to the high-performance datacenter links.

6.3.1 Context and Motivation

Nowadays, numerous applications have clients running on the edge of the network relying on cloud infrastructures for computations and storage. Unfortunately, the high network latency between clients and data centers can impair novel latency-constrained applications. Edge computing has emerged as a potential solution to circumvent this problem. To unleash its full potential, edge nodes must replicate data that is frequently used. However, because edge nodes have limited resources, full replication is infeasible. Therefore, any edge storage service needs to support partial replication. Additionally, the bandwidth of network links at the edge can be highly variable, being much lower than the bandwidth of the dedicated links that connect datacenters [101]. As such, edge applications should take these limitations into account to efficiently disseminate information across all replicas. Furthermore, to avoid the overhead and latency induced by the coordination mechanisms of strong consistency models, which becomes impractical when the number of replicas grows, edge storage should support weak consistency models.

For the latter, session guarantees [110] can be employed to allow configurable consistency guarantees in scenarios where a client may access different replicas of a weakly replicated system. In such scenarios, employing session guarantees prevents clients that need to execute operations in a different replica from observing a state inconsistent with their causal past, by forcing clients to wait for the necessary operations to be applied at the destination replica. While systems with session guarantees have been proposed by relying on vector clocks [110], these are only applicable in full replication settings. When adopting partial replication, the execution of an operation may be delayed when it depends on operations that will not be received, requiring frequent metadata exchanges between replicas to avoid this problem, which can lead to high visibility times.

To address this problem, we can leverage on a distributed metadata service to instruct replicas regarding the order by which they should apply remote updates without violating a given consistency criteria. While this approach has been used for causal consistency [16], providing individual session guarantees can prevent clients from suffering unnecessary delays when accessing remote replicas. As such, the goal of this work is to combine vector clocks with a distributed metadata propagation service that optimizes the use of the bandwidth of edge nodes, achieving low visibility times and efficiently supporting clients performing remote operations with different session guarantees.

6.3.2 Design

Engage is a system that aims at combining low visibility times *and* support for session guarantees, while optimizing the use of available bandwidth in edge nodes. It does so by combining, in a synergistic manner, the use of vector clocks to keep track of the read set and write set of clients, and the use of metadata services to speed up the propagation of updates and decrease bandwidth usage for edge nodes.

Figure 6.5 depicts the architecture of *Engage*. We consider a set of edge servers, fog nodes or cloudlets, that are used to replicate data. We assume the number of cloudlets to vary from a few dozen to one hundred. The system uses (static) partial replication, we assume that some data placement policy is in place and that the assignment of data objects to cloudlets is known by all cloudlets. *Engage* clients have a preferred cloudlet,

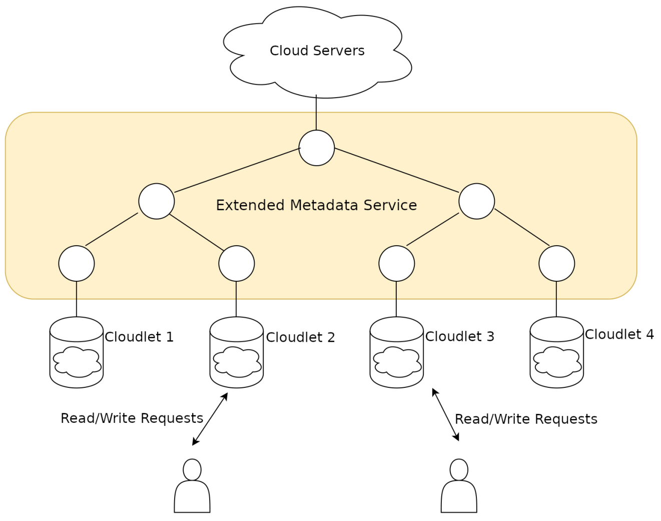


Figure 6.5: *Engage* Architecture

to which they forward all requests. If the preferred cloudlet does not replicate the target data object of an operation, the client must send the request to the nearest replica that does (client migration). If clients are mobile, they may change their preferred cloudlet on the fly. We now present the main aspects of the design of *Engage*.

6.3.2.1 Extended Metadata Service (Overview)

Cloudlets are attached to an extended metadata service that propagates updates to other replicas in an order that does not violate session guarantees. Contrary to other systems [16], where the metadata service is used exclusively to propagate metadata, in *Engage* this service supports the propagation of the update payloads in a manner that takes into consideration the topological properties (e.g., bandwidth) of the edge network. The extended metadata service is implemented by a set of *brokers*, organized in an acyclic graph (or tree). The leaf brokers of the tree are deployed at the edge cloudlets. The brokers that run as inner nodes of the tree are placed strategically to optimize aggregation and data dissemination. They can be deployed in cloud datacenters, cloudlets with more resources, or even in some edge cloudlets that are well-placed in the network (in terms of connectivity to other cloudlets).

6.3.2.2 Metadata

We assume that each cloudlet is linearizable [44]. Thus, each cloudlet keeps a sequence number that is used to uniquely identify updates that are performed locally on behalf of clients; this sequence number is shared by all objects. *Engage* uses vector clocks to keep track of the updates that are observed by clients. Multiple vector clocks, with one entry per cloudlet, are maintained by *Engage* as follows:

- For each object o that is replicated in a cloudlet i, a vector clock V_o^i is stored with the object. The vector clock captures all updates in the causal past of that object.
- Each cloudlet i also keeps a *cloudlet vector clock* V_*^i that captures the state of the local database. This clock encodes the highest sequence number of update operations executed from each remote cloudlet.
- Finally, each client c keeps two vector clocks: V_c^R, that captures the past of all objects the client has read, and V_c^W, that captures all write operations it has performed.

6.3.2.3 Performing Read and Write Operations

When a client performs a read or a write operation it can specify one or more session guarantees. *Engage* supports the following session guarantees: *Read Your Writes* (RYW), *Monotonic Reads* (MR), *Writes Follow Reads* (WFR), and *Monotonic Writes* (MW). Note that combining all session guarantees is equivalent to supporting causal consistency.

When performing a read or write operation on data object o using cloudlet i, the client c provides its own V_c^R, V_c^W, and the desired session guarantees for the operation. The cloudlet holds the request until it is *safe* to execute. In order to check if the cloudlet is in a state that is consistent with the guarantees specified by the client, the cloudlet compares the value of its own vector clock V_*^i with the values of V_c^R and V_c^W based on the requested session guarantees. If the client requests WFR or MR, it is safe to execute the operation if $V_*^i \geq V_c^R$. If the client requests MW or RYW, it is safe to execute the operation if $V_*^i \geq V_c^W$.

In the case of a read operation, the cloudlet sets $V_c^R = \textsc{max}(V_c^R, V_o^i)$, and returns the new value of V_c^R to the client, along with the object data. For a write operation, the cloudlet assigns a sequence number *snb* to the update, by incrementing the local counter that serializes all updates. Then, it creates a temporary *update vector clock* V^{up} that has all entries set to 0 except its own entry i, which is set to *snb*. Next, it updates several clocks as follows:

- It updates its own vector clock with the value of *snb*: $V_*^i = \textsc{max}(V_*^i, V^{up})$.
- It updates the object's vector clock with *snb* and the values of the client clocks: $V_o^i = \textsc{max}(V_o^i, V^{up}, V_c^R, V_c^W)$.
- It updates the client's write vector clock with the value of *snb*: $V_c^W = \textsc{max}(V_c^W, V^{up})$.

After these updates, it returns the new value of V_c^W to the client. In parallel, it schedules the update, tagged with V_o^i, to be sent through the extended metadata service to the other cloudlets that replicate o. Note that, in practice, the client only needs to provide its vector clocks when executing the first operation in a new cloudlet after migrating. After the first operation, session guarantees are assured without the client sending its vector clock.

6.3.2.4 Applying Remote Updates

Remote updates are applied in causal order based on *vector clock stability*, which is a common technique employed in causal replication systems [2, 4, 36, 110], where each replica keeps a vector clock, with one position per replica. This vector clock is used to verify if all dependencies of a remote operation have been executed locally, in order to make that operation visible and/or to check if all operations in the causal history of a client have been applied locally. In *Engage*, when an update performed at a replica *orig*, tagged with vector clock V_o^{orig}, is received at some other replica *dest*, it is put in a list of pending updates, remaining there until: i) it is the oldest (i.e., with the lowest sequence number) unexecuted update received from *orig*, and ii) the vector clock of *dest* ($V_*^{dest}[i]$) is greater than or equal to V_o^{orig}. When these conditions are met, the update is applied to the object and the cloudlet *dest* updates both its vector clock (V_*^{dest}) and the vector clock stored with the object (V_o^{dest}) with the received vector clock (V_o^{orig}).

However, using vector clock stability to apply remote updates is not enough under partial replication, as operations received may depend on operations over objects that are not locally replicated, blocking them from being applied. To solve this problem, nodes can, periodically, send to each other the values of their cloudlet vector clocks. We call these *metadata flush* (MF) messages. These messages, necessary both to apply remote updates and to support client migrations, generate additional traffic and make the remote update latency a function of the period used to exchange them. In *Engage*, the extended metadata service is used to efficiently propagate these MF messages.

6.3.2.5 The *Engage* Extended Metadata Service

Engage connects all cloudlets through a distributed extended metadata service, materialized by a set of servers, denoted *brokers*, distributed in different locations of the network that interconnects the replicas, and can either be co-located with cloudlets and data centers, or execute independently (as seen in Figure 6.5). The brokers are organized as an acyclic graph and each cloudlet is connected to one of these brokers. *Engage* propagates two types of messages through this metadata service: *update notifications*, that carry update operations with their associated vector clocks, and *metadata flush* messages, which only carry the vector clocks associated with update operations.

Update notifications are tuples associated with a concrete update. They include the following fields $\langle \text{UN}, src, snb, oid, payload, V_{oid}^{src} \rangle$, where *src* is the identifier of the cloudlet where the update was originated, *snb* is the sequence number assigned to the update, *oid* is the identifier of the object that has been updated, *payload* is the update operation itself and, finally, V_{oid}^{src} is the vector clock assigned to the update. Metadata flush messages are tuples that include the following fields $\langle \text{MF}, V_{\text{MF}} \rangle$, where V_{MF} is a vector clock that will be used to update the cloudlet vector clocks.

When a cloudlet receives a write request from a client, it creates an update notification message that it delivers to the local broker. When a broker receives an update notification, it performs the following actions for all tree edges *e* (except for the incoming edge):

- If the edge *e* is in the path from *src* cloudlet to another cloudlet that replicates *oid*, it forwards the update notification eagerly on that edge. If there is a MF message pending on that edge, it is piggybacked with the update notification, cancelling any timeout associated with it.
- Otherwise, it transforms the update notifications into a MF message, preserving the vector clock. The MF message is then scheduled to be propagated on that edge. If there is already another MF message pending on the same edge, both MF messages are merged into a single one, with a vector clock that has the maximum of both clocks. If there was no other MF message pending on edge *e*, a timer is started to propagate the MF message.
- When the timer associated with an edge expires, the broker forwards the pending MF message.

When a MF message is received by a cloudlet *dest*, either isolated or piggybacked with some update notification message, the cloudlet *dest* uses V_{MF} to update $V_*^{dest} = \text{MAX}(V_*^{dest}, V_{\text{MF}})$. When an update message is received by a cloudlet, the acyclic layout of the extended metadata service guarantees that every operation that could be a dependency has already been delivered locally and, as such, the update can be applied immediately.

The design of *Engage* allows it to support two different strategies to propagate the update payload:

Broker-Assisted Payload Propagation: In this mode, which we use as the default one, the payload of an update is propagated in the metadata broker network as part of the update notification message. Using the metadata service to propagate update operations (and not only their metadata) allows *Engage* to optimize bandwidth usage in edge scenarios.

Direct Propagation: *Engage* can also be applied to a more traditional geo-replicated scenario, where replicas are datacenters connected by a dedicated high-performance network. In this scenario, propagating the payload of update messages directly between datacenters can decrease visibility times, while the extended metadata service ensures all replicas can progress even with partial replication.

6.3.3 Evaluation

We implemented *Engage*, Bayou [110], and Saturn [16] over the Cassandra [52] distributed database, which only offers eventual consistency. Our implementation of *Engage* uses the Broker-Assisted Payload Propagation. In Saturn, update labels are propagated using the metadata service, while update payloads are disseminated directly between replicas. Our implementation of Bayou, while following the original article [110], was

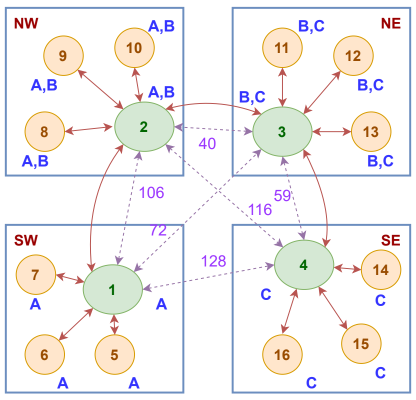


Figure 6.6: Experimental Setting

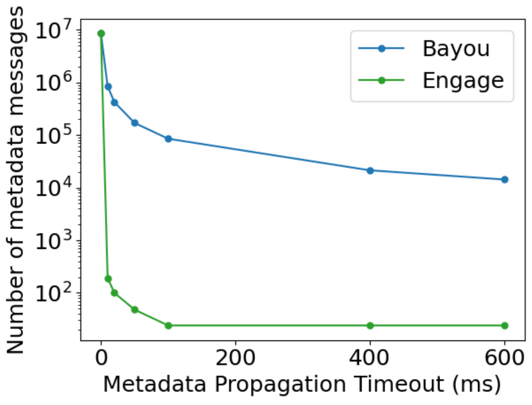


Figure 6.7: Signaling Impact: *Engage* vs Bayou

modified to support partial replication, by having replicas periodically send each other metadata messages containing their vector clock values (as discussed in Section 6.3.2.4). A timeout value is used to control the frequency of control information: for Bayou the timeout controls how often each node broadcasts a metadata message and, in *Engage* the timeout is used to control for how long a broker holds a MF message. Except in experiments where we vary it, this timeout value is set to 100 ms.

Figure 6.6 illustrates the setting used in the experimental evaluation. We considered an edge scenario, where replicas are distributed in four geographic locations, with a main datacenter (numbered 1 to 4) and three edge replicas (5 to 16) per location. The bandwidth of edge replicas is limited to 1Gbps of download and 500Mbps of upload. The dotted lines represent the latency between replicas, with the latency from the edge nodes to their local datacenter being $10ms$. The connections used by the *Engage* and Saturn metadata services are represented by the solid lines. We partition the data across replicas, with each data partition being represented by a letter.

6.3.3.1 Signaling Overhead

We start by comparing the signaling overhead of *Engage* and Bayou. Both systems require the exchange of control messages to keep the vector clocks of each cloudlet up-to-date. In systems such as Bayou, vector clocks can be updated via the periodic exchange of metadata messages, that carry the vector clock position of the sender[32]. *Engage* uses metadata flush (MF) messages for the same purpose. Unlike Bayou, in *Engage* a MF message from a cloudlet can be piggybacked on the update messages sent from other cloudlets. This often prevents *Engage* from sending signaling messages just to update vector clocks.

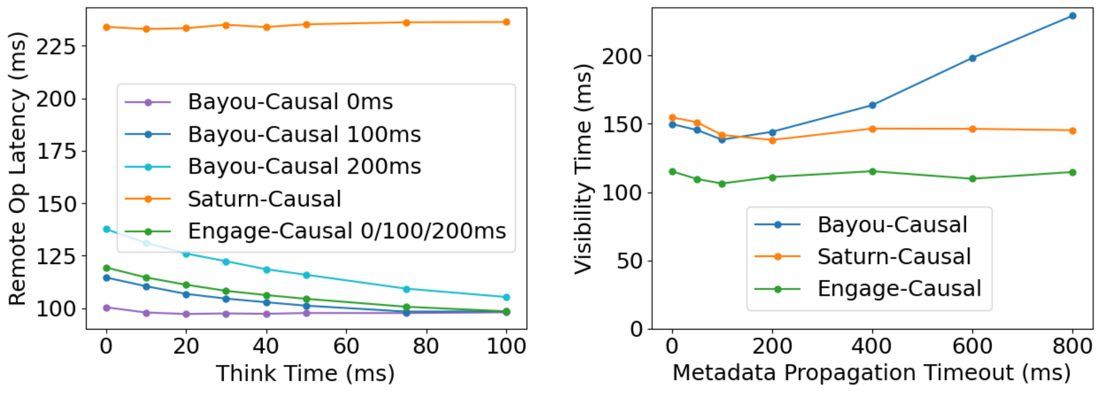
Figure 6.7 shows the number of control messages exchanged, by both systems, as a function of the timeout value. In the x axis we vary the timeout value and in the y axis we depict the number of control messages per second (in logarithmic scale). In *Engage*, we only count the MF messages that were not piggybacked with update messages. Obviously, the larger the timeout, the smaller the signaling overhead is, given that control messages are only sent when the timeout expires. While in Bayou, the number of control messages is simply a function of the timeout values used, the piggyback mechanism of *Engage* allows it to benefit much more from larger timeouts. In fact, it can be observed that, just for a timeout of $5ms$, the number of control messages propagated by Bayou is already multiple orders of magnitude higher than *Engage*, and after a timeout of $100ms$, all the *Engage* MF messages can be piggybacked in some update message with high probability. This shows that the piggyback mechanism of *Engage* is highly effective.

6.3.3.2 *Engage* vs Bayou and Saturn

In this section, we compare the performance of *Engage* against Bayou and Saturn, in terms of the client latency and the visibility latency incurred by these systems.

Figure 6.8(a) shows the latency experienced by clients when they perform a remote operation. In this experiment, clients executed 10 operations on their local replica, followed by an operation in a random remote replica. We vary the timeout value used to control the frequency of metadata messages in Bayou, and the timeout of MF messages in *Engage* and, in the x axis, we vary the clients' think time, *i.e.*, the time between a client receiving an operation response and sending its next operation. The larger the think time, the more likely it is that updates in the causal past of the client have already been applied when it performs a remote operation. As Saturn does not track the causal past of clients, they always depend on the last operation executed in the local cloudlet when migrating, leading to (constant) high remote operation latency. In contrast, *Engage* and Bayou rely on vector clocks with fine-grained information about the updates observed by clients, replying faster. The latency values of *Engage* do not depend on the metadata propagation timeout since, as long as there are write operations being propagated, MF messages are piggybacked before their timeout expires. In contrast, the metadata timeout is a key parameter for the remote operation latency in Bayou. We can see that with a timeout value of $0ms$, remote operations have low latency, however this incurs in a high number of transmitted control messages. With a timeout value of $100ms$, the latency of both Bayou and *Engage* become similar and, with a higher timeout value of $200ms$, Bayou starts showing high latency values for small think times, while still requiring a high number of control messages.

Figure 6.8(b) shows the visibility times of operations, which is an important metric in edge systems with partial replication as it influences both the migration time of clients, and data freshness, i.e., if clients are accessing up-to-date data. In the x axis, we vary the timeout value used to control the frequency metadata messages in Bayou, and the timeout of MF messages in *Engage*. Since regions with shared data partitions are only one hop away, by funneling all inter-region data through the high-bandwidth datacenter connections, *Engage* achieves lower visibility times, while in Bayou and Saturn each edge node needs to propagate the payload of operations to multiple nodes, decreasing their efficiency. Since Bayou needs to receive updates from remote cloudlets before it can apply



(a) Remote Op. Latency (b) Visibility Time

Figure 6.8: *Engage* vs Bayou and Saturn

a remote update, the visibility times of operations rise sharply when the timeout value increases. In contrast, *Engage* and Saturn rely on a metadata service to apply updates, resulting in constantly low visibility time.

6.3.4 Discussion and relevance

Engage is a data replication solution which was developed with the goal of supporting session guarantees (and, consequently, causal consistency) in a partially replicated edge scenario by using vector clocks and a distributed metadata service. This work was developed in collaboration with researchers from the INESC-ID research group, and it was my second work in the context of edge computing. Similarly to the previous contribution, it also exploits the hierarchical nature of the edge to optimize communications. However, while the previous work focused on the dissemination of control information, *Engage* focuses on data replication with causal consistency, being more closely aligned with the goals of this thesis. Unfortunately, the design choices made in *Engage* resulted in a system which is unable to scale to our envisioned number of edge locations (in the order of hundreds), being limited by the use of vector clocks. Furthermore, it relies on static partial replication, and does not integrate mechanisms to deal with node failures in the metadata service, both of which are important aspects which need to be addressed in the edge. Nevertheless, the lessons learned from *Engage* were extremely valuable for the development of *Arboreal*, the third main contribution of this thesis.

Publications and Artifacts

The source code for *Engage* is open source and fully available at `https://github.com/pfouto/engage`. This work resulted in the publication of a research paper in the 2022 International Conference on Computer Communications and Networks (ICCCN) [12].

7 Conclusion

In this chapter we conclude the dissertation, summarizing our contributions and showing how they effectively contribute to the overall goals of the thesis. In Section 7.1 we consider how the main contributions relate to each other and how they can be integrated both with each other or with existing solutions. Finally, Section 7.2 presents some possible directions for future work.

The research conducted during this thesis was aimed at studying and developing distributed data storage solutions that can be used to build scalable and fault-tolerant systems spanning the entire spectrum of *replica distribution levels* (co-located, geo-replication and edge computing), while providing the strongest practical consistency guarantees. As the challenges and requirements of each level vary significantly, we approached the problem by addressing each level individually, focusing on different consistency models, replication scopes, and adapting the scalability and fault-tolerance requirements based on what we considered to be the most appropriate for each level.

In summary, the three main contributions of this thesis are, by chapter:

- Chapter 3 addresses the co-located (or datacenter) level. In this contribution, we focused on providing strong consistency (linearizability) in a full-replication setting. From this contribution resulted ChainPaxos, a state machine replication protocol based on a chain topology, minimizing the number of messages exchanged between replicas (both for read and write operations) which maximizes its performance when compared with state-of-the-art solutions. ChainPaxos fulfills our scalability requirements for the co-located level by minimizing the throughput degradation of write operations as the number of replicas increases, while scaling the throughput of read operations. For fault-tolerance, ChainPaxos includes an integrated reconfiguration mechanism that allows it to both recover from failures and dynamically change the set of replicas without relying on external coordination services.

- Chapter 4 addresses the geo-replication level. In this contribution, we focused on causal consistency solutions that support (static) partial replication, mainly on a family of protocols which use a sequencer-based approach to enforce causal order. We identified the main performance and fault-tolerance bottlenecks of these protocols, the sequencer itself, and demonstrated that it is possible to replicate the sequencer using state machine replication for fault-tolerance, while not compromising the performance of the protocol. For this, using a performant and scalable SMR solution with low overhead is crucial.

- Chapter 5 addresses the edge computing level. This contribution also focuses on causal consistency, but on a much larger scale and combined with a dynamic partial replication model. From this contribution resulted Arboreal, a novel causal consistency data management solution specifically designed for the edge. Based on a tree topology, Arboreal can scale to hundreds of edge locations, while supporting crucial edge computing requirements, such as mobile clients the aforementioned dynamic partial replication model. For fault-tolerance, the decentralized nature of Arboreal allows it to recover from node failures in a localized manner, without relying on the datacenter or any other centralized service, and without affecting the rest of the system.

We note that, while each of these contributions is focused on a specific *replica distribution level*, theoretically each one of them could be used in any of the other levels. However, in practice, applying them outside their intended scope would make them suboptimal or even impractical when compared with specialized solutions. The reason for this is that, as the reader may have noticed throughout this document, every data replication protocol has its own set of trade-offs, always sacrificing some properties in favor of others. As such, in the design of replication protocols, these trade-offs are balanced based on the expected deployment environment, sacrificing properties that are less important for that environment. For example, by enforcing linearizability, ChainPaxos favors data consistency over availability during network partitions, as these are expected to be rare and short-lived in datacenter environments. Deploying ChainPaxos in an edge environment with hundreds of replicas would be impractical, as propagating every single write operation to all replicas would result in very poor performance, and the failure of *any* edge replica would affect the availability of the entire system. Similarly, deploying Arboreal (or a geo-replicated causal solution) in a datacenter, while not affecting their performance, would render their main advantage, availability during network partitions, mostly useless, as these are expected to be rare in datacenters. In such cases, these solutions would sacrifice data consistency for no practical gain.

Overall, the take-away message from this discussion is that the numerous trade-offs in data replication protocols make it impossible to have a one-size-fits-all solution, justifying our decision to focus on individual specialized contributions for each *replica distribution level* in this work.

Finally, considering the contributions of this thesis, an answer can be given to the main research question addressed by this thesis: *Is it possible to build scalable and fault-tolerant distributed data storage systems that extend from the cloud to the edge, while providing the strongest practical consistency guarantees at each level?*

The results presented in this thesis, and discussed above, addressed this question by focusing on the individual *replica distribution levels* of co-located, geo-replication, and edge computing. As discussed in their respective chapters, all approaches provide scalable and fault-tolerant solutions, improving one or both of these properties when compared to state-of-the-art solutions, while employing the strongest practical consistency guarantees for each level.

7.1 Combining Contributions

While during most of this work we focused on each *replica distribution level* individually, it is important to note that each of the contributions was designed to be used in combination with other solutions, to support distributed data management systems that span multiple *replica distribution levels*. In a way, the second contribution (Chapter 4) shows exactly this, by leveraging on ChainPaxos (the first contribution) to improve the fault-tolerance of geo-replicated causal consistency protocols.

While the three contributions of this thesis can be combined to span all levels, this may not be the most efficient combination for all application needs. Being isolated allows each contribution to be used independently, or combined with other existing solutions for different levels, allowing for more flexibility to achieve adequate solutions for each specific scenario. We now briefly discuss how the two novel protocols, ChainPaxos and Arboreal, can be combined with existing solutions:

ChainPaxos Being a state machine replication protocol, ChainPaxos is the most versatile contribution of this thesis, as it can be used to replicate components in any distributed system. As shown in Chapter 4, this includes replicating critical components of geo-replicated causal consistency protocols. Additionally, while in Arboreal we abstract each edge location as a single entity, it would also be possible to use ChainPaxos to replicate each Arboreal instance across multiple physical nodes in each edge location. Furthermore, ChainPaxos can be used not only to replicate the logical components of a system, but also to replicate application data. For instance, many geo-replicated causal consistency protocols assume that in each datacenter, data is replicated using some kind of linearizable protocol across multiple nodes.

Arboreal Arboreal is a more specialized solution, designed specifically to replicate data in edge computing environments. However, as discussed during its presentation (Chapter 5.3), it assumes that the application which wishes to use it is already deployed in a geo-replicated manner, leveraging a distributed (also geo-replicated) database which replicates data across the multiple datacenters and ensures its durability. In this sense, Arboreal can be seen as a complement to existing geo-replicated distributed databases. This means that any of the existing geo-replicated causal consistency solutions could be used in conjunction with Arboreal (with some engineering effort to forward and translate operations from/to Arboreal) to provide global causal consistency across the entire system. Nevertheless, Arboreal could also be used with solutions with different consistency guarantees. For instance, pairing it with an eventually consistent geo-replicated database would guarantee causal consistency inside each geographic region, relaxing the consistency guarantees across regions.

7.2 Future Work

The scope of this work allows for several research directions to be pursued in the future. Some of these are:

7.2.1 Transactional Causal Consistency

Arboreal provides causal+ consistency which provides intuitive semantics that avoid many data anomalies. However, causal consistency enforces a partial order across operations over individual data objects, which can be limiting for some applications. Extending Arboreal to support *transactional causal consistency* would allow for generic read-write transactions that span any number of data objects. This would allow for more complex applications to be built on top of Arboreal. While this has been achieved in the context of geo-replicated systems [107, 2, 74] with (static) partial and full replication, as far as we know, doing it with dynamic partial replication in a large-scale edge computing environment is an open problem. Adapting Arboreal to support transactional causal consistency would be a significant contribution, however, it would likely require a more complex communication protocol and additional metadata, which could affect its performance and scalability.

7.2.2 Byzantine Fault Tolerance

During this thesis, the fault model considered for all contributions was crash-stop failures and network partitions. While this is a common fault model for systems which are expected to run in controlled environments, such as datacenters or edge locations controlled by the same organization, adapting our solutions (both ChainPaxos and Arboreal) to tolerate Byzantine faults would be an interesting research direction.

ChainPaxos While Byzantine fault-tolerance for state machine replication is a well-studied problem [19, 13], most solutions are based on the common communication pattern of Multi-Paxos (or its variants) where the leader communicates directly with all replicas. Adapting the chain topology, and its integrated reconfiguration mechanism, to tolerate Byzantine faults could be a challenging research direction.

Arboreal On the other hand, Byzantine fault-tolerance for decentralized data replication protocols with causal consistency is a less explored (but not empty [66]) area. While we designed Arboreal assuming an edge infrastructure controlled by a single organization, if we consider a more adversarial scenario where some edge locations are controlled by different organizations, Byzantine adversaries could disrupt the system, by either trying to disconnect branches of the tree or by tampering with user operations. Adapting Arboreal to tolerate such attacks could make it a much more attractive solution for edge computing environments.

Bibliography

[1] N. Afonso, M. Bravo, and L. Rodrigues. "Combining high throughput and low migration latency for consistent data storage on the edge". In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2020, pp. 1–11 (cit. on pp. 1, 104, 105, 114).

[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)*. IEEE. 2016, pp. 405–414 (cit. on pp. 3, 17, 18, 20, 55, 78, 83, 84, 113, 138, 146).

[3] M. Alfatafta et al. "Toward a generic fault tolerance technique for partial network partitioning". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 351–368 (cit. on pp. 3, 28, 30, 50, 51).

[4] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 85–98 (cit. on pp. 17, 18, 24, 55, 56, 76, 83, 113, 138).

[5] H. Attiya, A. Bar-Noy, and D. Dolev. "Sharing memory robustly in message-passing systems". In: *Journal of the ACM (JACM)* 42.1 (1995), pp. 124–142 (cit. on p. 13).

[6] H. Attiya, F. Ellen, and A. Morrison. "Limitations of highly-available eventually-consistent data stores". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 2015, pp. 385–394 (cit. on pp. 2, 16, 55, 83).

[7] *AWS Global Infrastructure*. URL: `https://aws.amazon.com/about-aws/global-infrastructure/` (cit. on pp. 82, 84).

[8] *Azure Front Door Documentation*. `https://learn.microsoft.com/en-us/azure/frontdoor/front-door-overview` (cit. on p. 82).

[9] P. Bailis et al. "The potential dangers of causal consistency and an explicit solution". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–7 (cit. on p. 72).

[10] M. Balakrishnan et al. "Virtual consensus in delos". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 617–632 (cit. on pp. 3, 13).

[11] D. Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by I. I. Ivanov et al. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1 (cit. on p. 40).

[12] M. Belém et al. "ENGAGE: Session Guarantees for the Edge". In: *2022 International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2022, pp. 1–10 (cit. on pp. 1, 83, 104, 114, 142).

[13] A. Bessani, J. Sousa, and E. E. Alchieri. "State machine replication for the masses with BFT-SMART". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 355–362 (cit. on pp. 3, 13, 146).

[14] R. Bhardwaj et al. "Ekya: Continuous learning of video analytics models on edge compute servers". In: *19th USENIX Symposium on Networked Systems Design and Implementation*. 2022 (cit. on p. 20).

[15] W. J. Bolosky et al. "Paxos replicated state machines as the basis of a {high-performance} data store". In: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. 2011 (cit. on p. 52).

[16] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A distributed metadata service for causal consistency". In: *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, pp. 111–126 (cit. on pp. 1–4, 17, 18, 20, 22, 24, 55, 56, 64, 76, 91, 113, 135, 136, 139).

[17] E. Brewer. "CAP twelve years later: How the" rules" have changed". In: *Computer* 45.2 (2012), pp. 23–29 (cit. on pp. 2, 11).

[18] M. Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 335–350 (cit. on pp. 3, 13, 14, 19, 27, 53, 65).

[19] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, 173–186. ISBN: 1880446391 (cit. on pp. 38, 146).

[20] T. D. Chandra, R. Griesemer, and J. Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM. 2007, pp. 398–407 (cit. on p. 50).

[21] F. Chang et al. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26 (cit. on p. 54).

[22] X. Chen et al. "Achieving low tail-latency and high scalability for serializable transactions in edge computing". In: *Sixteenth European Conference on Computer Systems*. 2021 (cit. on p. 114).

[23] D. Chou et al. "Taiji: managing global user traffic for large-scale internet services at the edge". In: *27th ACM Symposium on Operating Systems Principles*. 2019 (cit. on pp. 20, 82).

[24] *CloudFront Documentation*. URL: https://aws.amazon.com/cloudfront/features (cit. on pp. 4, 20, 82).

[25] *CloudPing*. https://www.cloudping.co/grid. (Accessed Oct 2019.) (cit. on p. 47).

[26] B. F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154 (cit. on pp. 39, 40, 65, 68, 103).

[27] J. C. Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22 (cit. on p. 82).

[28] P. A. Costa, P. Fouto, and J. Leitao. "Overlay networks for edge management". In: *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2020, pp. 1–10 (cit. on pp. 102, 134).

[29] A. V. Dastjerdi and R. Buyya. "Fog computing: Helping the Internet of Things realize its potential". In: *Computer* 49.8 (2016), pp. 112–116 (cit. on pp. 1, 20).

[30] G. DeCandia et al. "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220 (cit. on pp. 1–3, 16–18, 20, 54, 81–83, 85).

[31] J. Du et al. "Gentlerain: Cheap and scalable causal consistency with physical clocks". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–13 (cit. on pp. 1, 3, 17, 18, 20, 24, 55, 78, 84, 113).

[32] J. Du et al. "Orbe: Scalable causal consistency using dependency matrices and physical clocks". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–14 (cit. on pp. 1, 17, 20, 24, 55, 77, 113, 140).

[33] V. Enes et al. "Efficient replication via timestamp stability". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 178–193 (cit. on p. 51).

[34] V. Enes et al. "State-machine replication for planet-scale systems". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15 (cit. on pp. 15, 39, 51).

[35] C. J. Fidge. "Timestamps in message-passing systems that preserve the partial ordering". In: (1987) (cit. on p. 56).

[36] P. Fouto, J. Leitão, and N. Preguiça. "Practical and fast causal consistent partial geo-replication". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2018, pp. 1–10 (cit. on pp. 3, 4, 17, 18, 24, 55–57, 64, 65, 76, 83, 138).

[37] P. Fouto, N. Preguiça, and J. Leitão. "High Throughput Replication with Integrated Membership Management". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 575–592 (cit. on p. 53).

[38] P. Fouto et al. "Babel: A framework for developing performant and dependable distributed protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2022, pp. 146–155 (cit. on pp. 39, 129).

[39] 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 (cit. on pp. 2, 11).

[40] *Google Peering Documentation*. URL: `https://peering.google.com/#/infrastructure` (cit. on pp. 4, 20, 82, 84).

[41] R. Guerraoui et al. "A high throughput atomic storage algorithm". In: *27th International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE. 2007, pp. 19–19 (cit. on p. 52).

[42] H. Gupta and U. Ramachandran. "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access". In: *12th ACM International Conf. on Distributed and Event-based Systems*. 2018 (cit. on p. 114).

[43] H. Gupta, Z. Xu, and U. Ramachandran. "DataFog: Towards a Holistic Data Management Platform for the IoT Age at the Network Edge". In: *USENIX Workshop on Hot Topics in Edge Computing*. 2018 (cit. on pp. 104, 114).

[44] M. P. Herlihy and J. M. Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492 (cit. on pp. 3, 12, 136).

[45] H. Howard, D. Malkhi, and A. Spiegelman. "Flexible paxos: Quorum intersection revisited". In: *arXiv preprint arXiv:1608.06696* (2016) (cit. on pp. 13, 19, 26, 50).

[46] P. Hunt et al. "{ZooKeeper}: Wait-free coordination for internet-scale systems". In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010 (cit. on pp. 3, 13, 14, 19, 27, 39, 45, 50, 65, 66, 79, 102).

[47] M. Jelasity, A. Montresor, and O. Babaoglu. "T-man: Gossip-based fast overlay topology construction". In: *Computer networks* 53.13 (2009), pp. 2321–2339 (cit. on p. 132).

[48] F. P. Junqueira, B. C. Reed, and M. Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 245–256 (cit. on pp. 39, 45).

[49] R. Klophaus. “Riak core: Building distributed applications without shared state”. In: *ACM SIGPLAN Commercial Users of Functional Programming*. 2010, pp. 1–1 (cit. on p. 16).

[50] J. Kreps, N. Narkhede, J. Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. Vol. 11. 2011. Athens, Greece. 2011, pp. 1–7 (cit. on pp. 3, 13, 14).

[51] S. S. Kulkarni et al. “Logical physical clocks”. In: *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d’Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. Springer. 2014, pp. 17–32 (cit. on pp. 78, 94).

[52] A. Lakshman and P. Malik. “Cassandra: a decentralized structured storage system”. In: *ACM SIGOPS operating systems review* 44.2 (2010), pp. 35–40 (cit. on pp. 1–3, 14, 16, 17, 19, 20, 22, 24, 27, 54, 58, 65, 81–83, 85, 104, 139).

[53] *Lambda@Edge Documentation*. URL: `https://aws.amazon.com/lambda/edge/` (cit. on pp. 4, 20, 82).

[54] Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs”. In: *IEEE transactions on computers* 100.9 (1979), pp. 690–691 (cit. on p. 12).

[55] L. Lamport. “Fast paxos”. In: *Distributed Computing* 19 (2006), pp. 79–103 (cit. on pp. 13, 14, 26, 50).

[56] L. Lamport. “Generalized consensus and Paxos”. In: (2005) (cit. on pp. 13, 14, 23, 26, 50).

[57] L. Lamport. “Paxos Made Simple”. In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58. URL: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/` (cit. on pp. 14, 27, 29, 40).

[58] L. Lamport. “The part-time parliament”. In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–170 (cit. on pp. 3, 13, 19, 26, 29, 118).

[59] L. Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196 (cit. on pp. 15, 18, 83).

[60] L. Lamport, D. Malkhi, and L. Zhou. “Reconfiguring a state machine.” In: *SIGACT News* 41.1 (2010), pp. 63–73 (cit. on p. 26).

[61] L. Lamport and M. Massa. “Cheap paxos”. In: *International Conference on Dependable Systems and Networks, 2004*. IEEE. 2004, pp. 307–314 (cit. on pp. 19, 50).

[62] J. Leitao et al. “X-BOT: A Protocol for Resilient Optimization of Unstructured Overlay Networks”. In: *IEEE TPDS* 23.11 (2012-11) (cit. on p. 132).

[63] J. Leitao, J. Pereira, and L. Rodrigues. “Epidemic broadcast trees”. In: *Proc. of SRDS’07*. IEEE. 2007 (cit. on p. 132).

[64] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast". In: *37th International Conference on Dependable Systems and Networks*. 2007 (cit. on pp. 102, 118, 125, 130, 132).

[65] J. Leitao et al. "Towards enabling novel edge-enabled applications". In: *arXiv preprint arXiv:1805.06989* (2018) (cit. on pp. 1, 20, 82, 89).

[66] A. van der Linde, J. Leitão, and N. Preguiça. "Practical client-side replication: Weak consistency semantics for insecure settings". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2590–2605 (cit. on p. 147).

[67] 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*. 2020 (cit. on pp. 91, 94).

[68] S. Liu et al. "Edge computing for autonomous driving: Opportunities and challenges". In: *Proceedings of the IEEE* 107.8 (2019) (cit. on p. 20).

[69] 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*. 2011, pp. 401–416 (cit. on pp. 1, 3, 16, 17, 20, 24, 55, 77, 83, 113).

[70] W. Lloyd et al. "Stronger Semantics for {Low-Latency}{Geo-Replicated} Storage". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 313–328 (cit. on pp. 3, 17, 55, 77, 113).

[71] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: `https://github.com/joaomlourenco/novathesis/raw/main/template.pdf` (cit. on p. i).

[72] N. A. Lynch and A. A. Shvartsman. "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts". In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE. 1997, pp. 272–281 (cit. on p. 13).

[73] P. Mahajan, L. Alvisi, M. Dahlin, et al. "Consistency, availability, and convergence". In: *University of Texas at Austin Tech Report* 11 (2011), p. 158 (cit. on p. 2).

[74] T. Mahmood et al. "Karma: cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency". In: *IEEE Transactions on Cloud Computing* 9.1 (2018), pp. 197–211 (cit. on pp. 4, 17, 55, 77, 146).

[75] Y. Mao, F. P. Junqueira, and K. Marzullo. "Mencius: building efficient replicated state machines for WANs". In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. 2008 (cit. on pp. 13, 15, 26, 51).

[76] P. J. Marandi. *U-Ring Paxos code*. https://github.com/sambenz/URingPaxos. (Accessed Oct 2019.) (cit. on p. 40).

[77] P. J. Marandi, M. Primi, and F. Pedone. "Multi-ring paxos". In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE. 2012, pp. 1–12 (cit. on pp. 50, 51).

[78] P. J. Marandi et al. "Ring Paxos: A high-throughput atomic broadcast protocol". In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2010, pp. 527–536 (cit. on pp. 13, 14, 19, 23, 26, 40, 50).

[79] P. J. Marandi et al. "Ring Paxos: High-Throughput Atomic Broadcast". In: *The Computer Journal* 60.6 (2017), pp. 866–882 (cit. on pp. 14, 40, 50).

[80] S. A. Mehdi et al. "I {Can't} Believe {It's} Not Causal! Scalable Causal Consistency with No Slowdown Cascades". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 453–468 (cit. on pp. 16, 55, 79, 83).

[81] *MongoDB Causal Consistency Documentation*. URL: `https://www.mongodb.com/docs/manual/core/causal-consistency-read-write-concerns/` (cit. on p. 17).

[82] I. Moraru, D. G. Andersen, and M. Kaminsky. *EPaxos code*. https://github.com/efficient/epaxos. (Accessed Mar-2019.) (cit. on p. 40).

[83] I. Moraru, D. G. Andersen, and M. Kaminsky. "There is more consensus in egalitarian parliaments". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 358–372 (cit. on pp. 13, 15, 26, 39, 40, 50, 51).

[84] S. H. Mortazavi et al. "Cloudpath: A multi-tier cloud computing framework". In: *Second ACM/IEEE Symposium on Edge Computing*. 2017 (cit. on p. 113).

[85] S. H. Mortazavi et al. "Feather: Hierarchical querying for the edge". In: *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2020, pp. 271–284 (cit. on p. 113).

[86] S. H. Mortazavi et al. "Sessionstore: A session-aware datastore for the edge". In: *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*. IEEE. 2020, pp. 59–68 (cit. on p. 113).

[87] S. H. Mortazavi et al. "Toward session consistency for the edge". In: *USENIX Workshop on Hot Topics in Edge Computing*. 2018 (cit. on p. 113).

[88] *MySQL Blog Archive: The king is dead, long live the king: Our Paxos-based consensus*. URL: `https://dev.mysql.com/blog-archive/the-king-is-dead-long-live-the-king-our-homegrown-paxos-based-consensus/` (visited on 2024-01-20) (cit. on p. 19).

[89] *Netty Framework*. `https://netty.io/` (cit. on pp. 39, 102, 120, 127).

[90] D. Ongaro and J. Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 305–319 (cit. on pp. 19, 40, 50).

[91] K. Petersen et al. “Flexible update propagation for weakly consistent replication”. In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*. 1997, pp. 288–301 (cit. on pp. 2, 15).

[92] D. Porto et al. “Visigoth fault tolerance”. In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–14 (cit. on p. 35).

[93] N. Preguiça et al. “Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine”. In: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*. IEEE. 2014, pp. 30–33 (cit. on pp. 3, 18, 22, 55, 56, 76).

[94] *Protocol Buffers*. `https://developers.google.com/protocol-buffers` (cit. on p. 127).

[95] *Redis Causal Consistency Documentation*. URL: `https://docs.redis.com/latest/rs/databases/active-active/causal-consistency/` (cit. on pp. 17, 55).

[96] M. Roohitavaf, M. Demirbas, and S. Kulkarni. “Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks”. In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2017, pp. 184–193 (cit. on pp. 17, 78).

[97] F. Ryabinin, A. Gotsman, and P. Sutra. “SwiftPaxos: Fast Geo-Replicated State Machines”. In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 345–369 (cit. on p. 14).

[98] B. Schlinker et al. “Engineering egress with edge fabric: Steering oceans of content to the world”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017 (cit. on pp. 4, 20, 82).

[99] F. B. Schneider. “Implementing fault-tolerant services using the state machine approach: A tutorial”. In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319 (cit. on pp. 13, 26, 29, 118).

[100] M. Shapiro et al. “Conflict-free replicated data types”. In: *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer. 2011, pp. 386–400 (cit. on p. 17).

[101] W. Shi et al. “Edge computing: Vision and challenges”. In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on pp. 1, 20, 82, 135).

[102] K. Shvachko et al. “The hadoop distributed file system”. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10 (cit. on p. 14).

[103] L. Silva and J. Lima. “An evaluation of Cassandra NoSQL database on a low-power cluster”. In: *Symposium on Computer Architecture and High Performance Computing Workshops*. 2021 (cit. on p. 104).

[104] Y. Siriwardhana et al. “A survey on mobile augmented reality with 5G mobile edge computing”. In: *IEEE Communications Surveys & Tutorials* 23.2 (2021) (cit. on pp. 20, 100).

[105] K. Sonbol et al. “EdgeKV: Decentralized, scalable, and consistent storage for the edge”. In: *Journal of Parallel and Distributed Computing* 144 (2020) (cit. on p. 114).

[106] K. Spirovska, D. Didona, and W. Zwaenepoel. “Optimistic causal consistency for geo-replicated key-value stores”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2020), pp. 527–542 (cit. on pp. 55, 79).

[107] K. Spirovska, D. Didona, and W. Zwaenepoel. “Paris: Causally consistent transactions with non-blocking reads and partial replication”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 304–316 (cit. on pp. 4, 17, 24, 56, 78, 146).

[108] *TaRDIS project*. URL: `https://www.project-tardis.eu/` (cit. on p. 129).

[109] J. Terrace and M. J. Freedman. “Object storage on CRAQ: High-throughput chain replication for read-mostly workloads”. In: *USENIX Annual Technical Conference*. 2009 (cit. on p. 52).

[110] D. Terry et al. “Session guarantees for weakly consistent replicated data”. In: *PDIS*. Austin (TX), USA, 1994 (cit. on pp. 135, 138, 139).

[111] I. Toumlilt, P. Sutra, and M. Shapiro. “Highly-available and consistent group collaboration at the edge with colony”. In: *Proceedings of the 22nd International Middleware Conference*. 2021, pp. 336–351 (cit. on pp. 1, 104, 105, 114).

[112] M. Tyulenev et al. “Implementation of cluster-wide logical clock and causal consistency in mongodb”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 636–650 (cit. on pp. 16, 55).

[113] R. Van Renesse and F. B. Schneider. “Chain Replication for Supporting High Throughput and Availability.” In: *OSDI*. Vol. 4. 91–104. 2004 (cit. on pp. 14, 19, 23, 30, 40, 50–52).

[114] S. Voulgaris, D. Gavidia, and M. Van Steen. “Cyclon: Inexpensive membership management for unstructured p2p overlays”. In: *Journal of Network and systems Management* 13.2 (2005), pp. 197–217 (cit. on p. 132).

[115] A. Wool. “Quorum systems in replicated databases: Science or fiction?” In: *IEEE Data Eng. Bull.* 21.4 (1998), pp. 3–11 (cit. on p. 50).

[116] Z. Xiang and N. H. Vaidya. “Global stabilization for causally consistent partial replication”. In: *Proceedings of the 21st International Conference on Distributed Computing and Networking*. 2020, pp. 1–10 (cit. on pp. 24, 56, 78).

[117] K.-K. Yap et al. “Taking the edge off with espresso: Scale, reliability and programmability for global internet peering”. In: *Conference of the ACM Special Interest Group on Data Communication*. 2017 (cit. on pp. 4, 20, 82).

2024 Scalable Consistency for Data Replication Pedro Fouto