Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique

# On Reducing Latency in Geo-Distributed Systems through State Partitioning and Caching

par

**Raluca Halalai**

Thèse

présenté à la Faculté des Sciences
pour l'obtention du grade de Docteur ès Sciences

Acceptée sur proposition du jury:

**Prof. Pascal Felber**, directeur de thèse
Université de Neuchâtel, Suisse

**Prof. Philippe Cudré-Mauroux**,
Université de Fribourg, Suisse

**Prof. Fernando Pedone**,
Université de la Suisse italienne, Suisse

**Prof. Etienne Rivière**,
Université catholique de Louvain, Belgique

**Dr. Valerio Schiavoni**,
Université de Neuchâtel, Suisse

**Prof. François Taïani**,
Université de Rennes 1, France

Soutenue le 14 mai 2018

UNIVERSITÉ DE NEUCHÂTEL

**Faculté des Sciences**
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel – Suisse
Tél : + 41 (0)32 718 21 00
E-mail : secretariat.sciences@unine.ch

# IMPRIMATUR POUR THESE DE DOCTORAT

**La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par**

## Madame Raluca HALALAI

Titre:

# "On Reducing Latency in Geo-Distributed Systems through State Partitioning and Caching"

**sur le rapport des membres du jury composé comme suit:**

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse
- Prof. François Taïani, Université de Rennes 1, France
- Prof. Fernando Pedone, Université de la Suisse italienne, Lugano, Suisse
- Prof. Etienne Rivière, Université catholique de Louvain, Belgique
- Prof. Philippe Cudré-Mauroux, Université de Fribourg, Suisse

Neuchâtel, le 15 mai 2018

Le Doyen, Prof. R. Bshary

# Acknowledgements

This work would not have been possible without the people who have supported me throughout this journey. I am deeply grateful to my advisor, Pascal Felber, for his patience and wisdom, and for always encouraging me while also giving me the freedom to become independent in pursuing my goals. I feel very fortunate to having worked with him. I thank my outstanding thesis committee: Philippe Cudré-Mauroux, Fernando Pedone, Etienne Rivière, Valerio Schiavoni, and François Taïani. Their insightful comments and advice helped crystallize the vision of this thesis. All the work presented here is the result of collaboration with many incredibly bright people to whom I am thankful for the many insightful discussions that brought clarity to the most difficult problems. Last but not least I thank my family and friends for their patience, support, and encouragement throughout these years.

# Résumé

Les systèmes distribués modernes sont de plus en plus grands, et sont déployés dans plusieurs régions géographiques. L'objectif final de tels systèmes est de fournir des services à leurs utilisateurs ainsi que haute disponibilité et bonne performance. Cette thèse propose des techniques pour réduire le latence perçue par des utilisateurs.

Pour commencer, nous considérons les systèmes qui utilisent la technique de réplication de machines à états afin de garantir la cohérence des données. La technique de réplication de machines à états copie un service à plusieurs emplacements et coordonne les répliques afin de sérialiser toutes les commandes émis par des clients. La coordination à grande échelle a un impact significatif sur la performance du système. Nous étudions comment le partitionnement d'état peut aider à réduire les performances sans affecter la sémantique du système. Premièrement, nous formalisons les conditions dans lesquelles un service est partitionnable et proposons une approche de partitionnement d'état générique. Nous partitionnons un service de coordination géo-distribué et montrons qu'il surpasse son homologue non partitionné, tout en offrant les mêmes garanties. Nous augmentons notre système avec un partitionnement d'état dynamique, qui s'adapte à la charge de travail. Notre évaluation montre que le partitionnement d'état dynamique a un impact positif sur les performances du notre système de fichiers.

Finalement, nous étudions le compromis entre la latence et les coûts de stockage dans les systèmes de stockage qui utilisent des techniques de codage d'effacement. Afin d'améliorer les performances de lecture, les systèmes de stockage utilisent des caches qui sont proches des clients. Cependant, les stratégies de mise en cache traditionnelles ne sont pas conçu pour les particularités du codage d'effacement et ne sont pas bien adaptés à ce scénario. Nous avons proposé un algorithme pour mettre en cache des données codées et nous l'avons utilisé pour implémenter une système de mise en cache basée sur Memcached. Notre algorithme reconfigure le cache en fonction de la charge de travail et peut surpasser la performance des politiques de mise en cache traditionnelles comme Least Recently Used et Least Frequently Used.

**Mots clés** : systèmes géo-distribués, cohérence, partitionnement d'état, mise en cache, codage d'effacement

# Abstract

Modern distributed systems are increasingly large, spanning many datacenters from different geographic regions. The end goal of such systems is to provide services to their users with high availability and good performance. This thesis proposes approaches to reduce the access latency perceived by end users.

First, we focus on systems that rely on the state machine replication approach in order to guarantee consistency. State machine replication copies a service at multiple physical locations and coordinates replicas – possibly from distant regions, in order to serialize all requests issued by clients. Coordination at large scale has a significant impact on the performance of the system. We investigate how state partitioning can help reduce performance without breaking the semantics of the system. First, we formalize conditions under which a service is partitionable and proposed a generic state partitioning approach. We build a partitioned geo-distributed coordination service and show that it outperforms its non-partitioned counterpart, while providing the same guarantees. We further apply state partitioning in order to build a geo-distributed file system, which performs comparable to other de-facto industry implementations. We augment our system with dynamic state partitioning, which moves files among data centers in order to adapt to workload patterns. Our experiments show that performing state partitioning on the fly has a positive impact on the performance of the file system when the workload exhibits access locality.

Second, we investigate the tradeoff between latency and storage cost in storage systems that employ erasure coding techniques. In order to improve read performance, storage systems often use caches that are close to clients. However, traditional caching policies are not designed for the particularities of erasure coding and are not well-suited for this scenario. We proposed an algorithm for caching erasure-coded data and use it to implement a caching layer based on Memcached in front of the Amazon S3 storage system. Our caching algorithm reconfigures the cache based on workload patterns and is able to outperform traditional caching policies such as Least Recently Used and Least Frequently Used.

**Keywords**: geo-distributed systems, strong consistency, state partitioning, caching, erasure coding

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Distributed systems are becoming increasingly integrated in everyday life. This has been enabled by the widespread adoption of the Internet, which currently brings together over half of the world's population [1]. Nearly a quarter of a billion new users came online in 2017, much of this growth being driven by more affordable smartphones and mobile data plans [2]. The rapid growth of the Internet resulted in increasingly larger systems that run over the Internet, up to 1000x the size of anything previously built. Each 10x growth in scale typically requires new system-specific techniques [3], making it challenging for system designers to keep up.

Many large-scale systems are built on top of cloud computing platforms, such as Amazon Web Services (AWS) and Google's Cloud Platform, which provide basic building blocks running on clusters of commodity hardware. Many things can go wrong: hardware components fail, systems outgrow their allocated resources, entire datacenters go offline due to natural disasters, etc. To prevent data loss, distributed systems employ redundancy, either in the form of replication or by using more complex data redundancy mechanisms, such as erasure coding.

Distributed systems need to work around the limitation formulated by the CAP theorem [4]. No distributed system is safe from network failures, so *partition tolerance* is generally expected. When network partitions occur, systems need to choose between *availability* – i.e., servicing user requests, and *consistency* – i.e., guaranteeing that data remains in sync across datacenters. Many designs proposed in the literature [5],[6],[7],[8],[9] sacrifice consistency for availability. While this might work well for domain-specific applications, which can anticipate and provide resolution methods for conflicts, other general-purpose applications require strong consistency, as (i) it is both more intuitive for the users, and (ii) it does not require human intervention in case of conflicts.

In this thesis, we address questions related to the way one may design and build distributed systems that span many geographically distant regions. We focus on reducing the access latency perceived by the user in (i) systems that provide strong consistency guarantees at large scale (Chapter 3 and Chapter 4), and (ii) systems that use complex data redundancy mechanisms to ensure high availability (Chapter 5).

## 1.1 Problem definition

This thesis addresses the problem of reducing latency in geo-distributed systems. By "geo-distributed systems" we mean systems that span many datacenters from geographically distant regions. Many popular systems deployed over the Internet fall into this category: search engines (e.g., Google, Bing), online payment services (e.g., PayPal), social networks (e.g., Facebook, Twitter), video sharing (e.g., YouTube), and many others.

We focus on answering two classes of questions that lie at the core of building large systems that are reliable and serve user requests with low latency.

**How to enforce strong consistency at large scale without crippling performance?** State machine replication (SMR) is a general approach for building fault-tolerant and consistent distributed systems by replicating servers and coordinating client interactions with server replicas. Serializing client commands has a significant impact on performance and also impedes scalability.

**How to lower access latency in storage cost-aware geo-distributed systems?** Distributed storage systems represent the backbone of many Internet applications. They aim to serve content to end users with high availability and low latency. Storage systems guarantee high availability either by replicating data across multiple regions or by employing more complex mechanisms such as erasure coding. Storage cost-aware systems prefer the latter, as it achieves equivalent levels of data protection with less storage overhead. On the flip side, erasure coding incurs higher access latency. Caching is the go-to approach for reducing read latency, as the cache needs to determine not only which data items to store, but also how many blocks for each of them.

## 1.2 Proposed solution

We explore techniques to reduce latency in fault-tolerant geo-distributed systems. For systems built using the state machine replication approach, our goal is to reduce the amount of coordination between replicas located in distant regions. For systems that employ erasure coded data, we aim to speed up read operations by relying on specialized caches.

### 1.2.1 State partitioning in geo-distributed systems

Distributed services form the basic building blocks of many contemporary systems. A large number of clients access these services, and when a client performs a command on a service, it expects the service to be responsive and consistent. The state machine replication approach offers both guarantees: by replicating the service on multiple servers, the commands are wait-free despite failures, and by executing them in the same order at all replicas, they are sequentially consistent. However, since SMR serializes all commands, it does not leverage the intrinsic parallelism of the workload. This results in poor performance, especially at large scale.

A promising approach to tackle this problem is to partition the state of the service and distribute the partitions among replicas [10]. When the workload is fully parallel, the scale-out of the partitioning approach is optimal. Hence, we look further into this approach.

We devote Chapter 3 to a principled study of geo-distributed service partitioning. First, we look into conditions under which it is possible to partition a service, and then propose a general approach to build dependable and consistent partitioned services. We assess the practicability of our approach by building a partitioned coordination service at scale.

### 1.2.2 Workload-aware state partitioning

We next focus on answering the following question: "How can state partitioning dynamically adapt to workload patterns?", in the context of a geo-distributed file system. In this work, we extend our approach in Chapter 3 and apply it to geo-distributed file systems. To this end, we adapt state partitioning to the particularities of POSIX file system semantics. In order to adapt to the changing workloads of a general-purpose file system, we also propose a mechanism that can dynamically alter the partitioning scheme.

Most previous designs for distributed file systems [5, 6, 7] provide weak consistency guarantees. Weak consistency has two main drawbacks. First, it exposes execution anomalies to users. Second, it induces programming complexity, as system developers need to anticipate conflicts and provide appropriate resolution mechanisms. While weak consistency models might be suitable for applications without strict requirements and domain-specific applications, it is not appropriate for general-purpose applications such as a file system. Strong consistency suits file systems, as it is both more intuitive for the users and does not require human intervention in case of conflicts.

We use state partitioning to build a POSIX-compliant distributed file system that spans geographically diverse regions. Partitioning the file system tree allows us to exploit locality in access patterns without compromising consistency. We use this distributed file system to illustrate the practical implications of state partitioning, and to explain how partitioning can dynamically adapt to workload patterns. Workload-aware state partitioning introduces new challenges, which we address in Chapter 4.

### 1.2.3 Caching tailored to erasure-coded storage systems

In the third part of this thesis, we explore techniques to lower access latency in storage cost-aware geo-distributed systems. Distributed storage systems are an integral part of many Internet applications. They span geographically diverse regions in order to serve content to end users with high availability and low latency.

**High availability.** Distributed data stores provide high availability by relying on data redundancy, which consists in either (i) replicating data across multiple sites or (ii) using more

elaborate approaches such as erasure coding [11].

*Replication* implies storing full copies of the data in different physical locations. This approach is simple to implement and provides good read performance, as users can get data from the nearest data site. However, replication suffers from high storage overheads.

By contrast, *erasure coding* achieves equivalent levels of data protection with less storage overhead but is more complex to set up and operate. The key idea is to split an object into data blocks and store these chunks at different physical locations. A client only needs to retrieve a subset of data blocks to reconstruct the object. Erasure coding is thus efficient in terms of storage space and bandwidth, but incurs higher access latency, as users now need to contact more distant data sites to reconstruct the data.

**Low latency.** Many systems employ caching in order to lower access latency. Caching implies storing a subset of data in memory that is faster or closer to users than the original source. In the case of storage systems that span geographically diverse regions, caching decreases access latency by bringing data closer to end users. The application of caching to erasure-coded data is, however, not obvious, as the presence of blocks offers many caching configurations that traditional caching policies such as *Least Recently Used* (LRU) or *Least Frequently Used* (LFU) are unable to capture or exploit. More precisely, a caching mechanism for erasure-coded data should be able not only to decide *which* data items to cache, but also *how many blocks* to cache for each data item.

In Chapter 5, we investigate how a tailored caching mechanism can help minimize access latency in storage systems that span many geographical regions and use erasure coding to ensure data availability. The main challenge resides in designing the caching policy, namely deciding *which data to cache* and *how many blocks to cache for each data item*. We design and prototype a caching system that periodically reconfigures the cache based on workload patterns.

### 1.2.4 Summary of results

To sum up, this thesis makes the following contributions towards reducing access latency in geo-distributed systems:

- **Reducing access latency through state partitioning.** We formulate conditions under which a service can be partitioned and propose a general partitioning algorithm. To validate our partitioning approach, we build a geo-distributed coordination service based on Apache ZooKeeper [12]. Our evaluation shows that, when the workload exhibits access locality, partitioning the state of a service reduces latency by up to 75% compared to a non-partitioned deployment, while offering the same core guarantees.
- **Workload-aware state partitioning.** We build a geo-distributed file system using state partitioning and compare it against de-facto industry implementations. Experimental

results show that our partitioned file system performs on par with the others. We further augment the partitioned file system with support for on-the-fly state partitioning, which allows the system to adapt to workload patterns by moving files around. Our preliminary results show that being able to move files according to access patterns has a positive effect on the performance of our geo-distributed file system; moving file reduces read latency by around 88% compared to static partitioning.

- **Caching erasure-coded data.** We propose an approach and an algorithm to optimize the cache configuration for erasure-coded data. Our caching algorithm relies on a dynamic programming approach to optimize the cache configuration under a given workload. We implemented a prototype of a caching system using our algorithm and integrated it in a basic geo-distributed storage system. Our evaluation shows that our algorithm can achieve 16% to 41% lower latency than systems that use classical caching policies, such as LRU and LFU.

## 1.3 Dissertation plan

This thesis is organized as follows. In Chapter 2, we describe techniques employed by current state-of-the-art geo-distributed systems to ensure high availability and low latency. In addition, we survey consistency models and describe their performance trade-offs. Chapter 3 describes a general state partitioning approach to build strongly consistent systems at scale without crippling performance. We use this state partitioning approach in Chapter 4 to build a geo-distributed file system and show that adapting the partitioning to workload patterns benefits performance. Chapter 5 proposes a new algorithm for caching erasure coded data and evaluates the access latency for read-only workloads. In Chapter 6, we conclude.

# 2 Background and Related Work

This chapter provides some background for our work, as well as a brief overview of its context. We describe the challenges and trade-offs that lie behind building large-scale distributed systems and give an overview of the related work that addresses them. Our goal is to highlight how the algorithms and approaches proposed in this thesis complement and advance the state of the art.

## 2.1 CAP theorem

Eric Brewer's CAP theorem [4] formulates the idea that there is a fundamental trade-off between *consistency* and *availability* when *network partitions* inevitably occur in distributed systems. The CAP theorem states that any distributed system can simultaneously have at most two of three desirable properties:

- **C**onsistency, i.e., for every request, the system returns a response that is correct in accordance to the system specification;
- **A**vailability, i.e., each request eventually receives a response;
- **P**artition tolerance, i.e., the system is able to function as expected even when servers are split in separate groups that do not communicate.

Communication among servers is unreliable in practical deployments, so network partitions are bound to happen. Therefore, for any practical deployment, system designers need to choose between availability and consistency. System designers and researchers explore the various points along this trade-off between availability and consistency in a wide variety of novel distributed systems.

On the one hand, there are systems that choose strong consistency and settle for best effort availability. Strong consistency means that all clients of the system share the same view of the system state. This means that all system servers need to agree on the state and employ protocols to coordinate for any state change. However, since the servers can no longer coordinate with each other during network partitions, they will sacrifice availability.

On the other hand, there are systems that choose to provide high availability over strong consistency. Coordination is no longer done at every state change, but rather lazily, in a best effort manner. When network partitions occur, service is still provided to clients by local servers, but since servers cannot coordinate, they may expose different views to their respective clients. Furthermore, if the clients change state in their respective servers, this state may diverge and need reconciliation when the network partitioning ends. Weak consistency models have two main drawbacks: (i) users get to witness execution anomalies, and (ii) it leads to increased programming complexity, as system designers need to anticipate conflicts and provide specialized resolution mechanisms.

In this thesis we focus on strong consistency, as it has more desireable qualities for general-purpose distributed systems. Strong consistency is more easily understood by both developers of systems and clients. We focus on addressing the latency and storage overhead of strongly-consistent distributed systems.

## 2.2 Replication mechanisms for strong consistency

There are two fundamental approaches to maintain replicated services in a consistent state: *primary-backup replication* [13] – proposed by Alsberg and Day in 1976, and *state machine replication* [14], introduced by Lamport in 1978.

**Primary-backup replication.** The primary-backup replication model consists of one *primary replica* and multiple *secondary replicas* (or *backups*). The primary serves all client requests that modify state, thus totally ordering them. Periodically, the primary sends state updates to the secondaries. The secondary replicas, in turn, monitor the primary in order for one of them to take over if the primary crashes.

**State machine replication (SMR).** The SMR model allows clients to issue requests to different service replicas. Replicas coordinate in order to agree on a total execution order for client operations that modify state. The main challenge in implementing SMR is to ensure that the client-issued operations are executed in the same order at all replicas, despite concurrency and failures.

A large body of research focuses on implementing SMR protocols that are resilient to different types of failures. The most well-known SMR algorithm is based on the Paxos [15] consensus protocol, which was proposed by Lamport in the 1980s. Paxos is able to tolerate crash failures. In 1999, Castro and Liskov proposed PBFT [16], a consensus protocol that is able to tolerate *Byzantine failures*, i.e., arbitrary failures. PBFT sparked great interest in BFT protocols, but with little practical adoption to this day.

With the advent of Internet-scale services, there has been renewed interest in SMR [17], both from academia and industry. Several optimizations have been proposed to make traditional SMR protocols viable in wide-area deployments. Fast Paxos [18] allows commands to be

committed in fewer communication steps when there are no concurrent updates. Ring Paxos [19] exploits the characteristics of IP multicast networks for implementing efficient replica coordination (i.e., total order multicast) in this environment. Finally, Raft [20] is equivalent to Paxos in terms of fault tolerance and performance, but it is considered easier to understand.

The strong consistency guarantees provided by SMR have a performance cost: because it serializes all commands, SMR does not leverage the intrinsic parallelism of the workload. Several directions have been investigated to limit the impact SMR has on performance:

1. Operations that do not change the service state can be executed at a single replica. This approach implies dropping linearizability (i.e., ordering operations across all clients based on time) for sequential consistency (i.e., ordering operations such that it preserves the program order of each client), but such a limitation is unavoidable in a partially-asynchronous system [21].
2. SMR can leverage the commutativity of updates to improve response time. This strategy exhibits a performance improvement of at most 33% in comparison to the baseline [22].
3. One can partition the state of the service and distribute the partitions among servers [10].

In this thesis, we focus on the third approach, which has fewer limitations than the other two alternatives, and hence, we consider it to be the most promising.

## 2.3 State partitioning

Replication distributes copies of the system state across different geographical regions. Besides fault tolerance, this provides the advantage that read-only client requests, which do not modify state, can be served locally by each server, reducing the latency perceived by clients. However, requests that change state are slow, as they require coordination across the regions, which implies several rounds of message exchanges over high-latency connections.

State partitioning allows dynamic trade-offs between performance and availability of data. By reducing the degree of replication for some parts of the state, the amount of coordination required to keep that state in sync is also decreased and the performance of requests that modify it increases. This comes at the cost of fault tolerance: at the extreme, if state is partitioned such that a single server stores it, then the state will be lost if that server crashes.

With state partitioning, some state can be replicated globally, ensuring fault tolerance and fast reads across all geographic regions, at the cost of slow writes, while other state can be replicated locally in a subset of regions, ensuring fast reads and writes in those regions, at the cost of slower reads from other regions and a lower degree of fault tolerance. This maps well to workloads that exhibit high locality, i.e., if there is some state that is most often accessed in specific regions.

When a system employs state partitioning, a fully parallel workload translates into optimal scale-out. Hence, we consider state partitioning as the most promising direction towards improving the performance of geo-distributed systems that rely on SMR. We investigate state partitioning in Chapters 3 and 4.

Several scientific publications (e.g., [23, 24]) observed that it is not necessary to order commuting service operations. This observation was used recently by Kraska et al. [25] in order to build a distributed database system.

Some approaches [26, 27] leverage application semantics to compute dependencies among commands in order to parallelize SMR. Commands are assigned to groups such that commands within a group are unlikely to interfere with each other. Eve [27] allows replicas to execute commands from different groups in parallel, verifies if replicas can reach agreement on state and output, and rolls back replicas if necessary. Parallel State-Machine Replication (P-SMR) [26] proposes to parallelize both the execution and the delivery of commands by using several multicast groups that partially order commands across replicas. These two techniques aim at speeding-up the execution of commands at each replica by enabling parallel execution of commands on multi-core systems. The approach we describe in Chapter 3 is orthogonal to this body of work. It improves performance by enabling different replicated state machines to execute commands in parallel without agreement.

Marandi et al. [10] employ Multi-Ring Paxos [28] to implement consistent accesses to disjoint parts of a shared data structure. However, by construction, if an invariant is maintained between two or more partitions, the approach requires that a process receives all the messages addressed to the groups, sacrificing Disjoint Access Parallelism (DAP). (The DAP property captures the requirement that unrelated operations progress independently, without interference, even if they occur at the same time.) [29] and [30] construct a shared tree in a purely asynchronous system under strong eventual consistency. In both cases however, the tree structure is replicated at all replicas.

Concurrently to our work described in Chapter 3, Bezerra et al. [31] described an approach to partition a shared service. To execute a command, the client multicasts it to the partitions in charge of the state variables read or updated by that command. Each partition executes its part of the command, waiting (if necessary) for the results of other partitions. In comparison to our approach in Chapter 3, this solution(i) does not take into account the application semantics implying in some cases an unnecessary convoy effect, and (ii) it requires to approximate *in advance* the range of partitions touched by the command. In the ZooKeeper use case (Volery), P-SMR stores the tree at all replicas and commands modifying the structure of the tree (`create` and `delete`) are sent to all replicas. In contrast, ZooFence (presented in Chapter 3) exploits application semantics to split the tree into overlapping sub-trees, one stored at each partition.

The transactional paradigm is a natural candidate to partition a concurrent tree (e.g., [32]). In ZooFence , there is no need for transactional semantics because the implementation of the tree is hierarchical. A transactional history is serializable when its serialization graph is acyclic

[33]. Theorem 3.2 (Chapter 3) can be viewed as the characterization of strictly serializable histories over abstract operations.

Ellen et al. [34] prove that no universal construction can be both DAP and wait-free in the case where the implemented shared object can grow arbitrarily. We pragmatically sidestep this impossibility result in our algorithms by bounding the size of the partition.

## 2.4 Adaptive state partitioning

State partitioning improves the scalability and performance of SMR-based systems by splitting state among replicas in a manner that reduces the amount of coordination required. Devising a good state partitioning scheme requires workload knowledge. In Chapter 3 and the first part of Chapter 4 we assume that workload patterns do not change and devise static partitioning schemes. However, when the workload patterns change, the partitioning scheme must adapt in order to preserve the benefits of state partitioning.

Dynamically adapting state partitioning to workload patterns must overcome two main challenges. First, the original guarantees of the SMR-based system should be preserved. Second, the reconfiguration should be transparent to the users and have minimal impact on the performance of the system.

**Dynamic state partitioning** Dynamic Scalable State Machine Replication (DS-SMR) [35] is a technique that allows a partitioned SMR-based system to reconfigure its data-to-partitions assignment on-the-fly. Similarly to the dynamic state partitioning approach we describe in Chapter 4, DS-SMR repartitions the state based on workload patterns. It uses a centralized oracle to organize partitioning, as well as a client-side cache to reduce communication with the oracle. The main goal of DS-SMR is to increase throughput by balancing operations across several partitions. However, unlike DS-SMR, our dynamic partitioning approach is tailored to POSIX-compliant geo-distributed file systems, like the one we build in Chapter 4. This choice requires several design decisions that differ from DS-SMR's. We need to build a distributed oracle, rather than centralized, in order to minimize latency in a geo-distributed system. Our partitioning function needs to move data close to the clients accessing it most often, instead of randomly for load balancing. In order to satisfy the file system semantics, there are several conditions that our partitioning must comply with (e.g., parent nodes must be informed of any new or deleted children, therefore the parent nodes must be shared across partitions).

**Elastic state partitioning** The elastic state partitioning technique proposed by Nogueira et al. [36] differs from our work presented in Chapter 4 in several ways. First, the elastic state partitioning protocol works for Byzantine fault tolerant SMR-based systems, while we only focus on crash tolerant systems. Second, unlike our dynamic state partitioning approach which targets redistributing state across a predefined set of partitions, elastic state partitioning

focuses on increasing or, respectively, decreasing the number of partitions at runtime. Third, we monitor workload patterns and take into account access locality when reconfiguring partitions, while the approach proposed by Nogueira et al. focuses on load, in terms of storage capacity and throughput.

## 2.5 Consistency in distributed file systems

Distributed file systems are integral part of many of today's workflows, making them the focus of a large body of research from both industry and academia. In Chapter 4, we apply state partitioning to build a strongly consistent geo-distributed file system. This section provides an overview of existing designs for distributed file systems, which we categorize based on the consistency guarantees they provide.

### 2.5.1 File systems with strong consistency

Strong consistency is a natural fit for distributed file systems. File systems are general-purpose applications, which makes it difficult for system designers to anticipate conflicts and provide appropriate resolution mechanisms. Unlike weak consistency models, strong consistency is both more intuitive for the users and does not require human intervention in case of conflicts.

In Chapter 4, we describe GlobalFS, a distributed file system that leverages atomic multicast to order operations that modify state. We partition state, i.e., the file system structure, by assigning each file or directory to a set of regions. Atomic multicast coordinates the operations corresponding to each file/directory, allowing for locality while still providing consistent operations over the whole file system.

CalvinFS [37] is a multi-site distributed file system built on top of Calvin [38], a transactional database. Metadata is stored in main memory across a shared-nothing cluster of machines. File operations that modify multiple metadata elements execute as distributed transactions. CalvinFS supports linearizable writes and reads using a global log to totally order transactions. While the global log can scale in terms of throughput, it does not allow for exploiting locality. On the other hand, GlobalFS provides strong consistency guarantees while allowing for files to be either locally or globally replicated and thus, providing the option for users to choose between availability (disaster tolerance) and performance (throughput and latency).

CephFS [39] is a file system implementation atop the distributed Ceph block storage [40]. It uses independent servers to manage metadata and to link files and directories to blocks stored in the block storage. CephFS is able to scale up and down the metadata servers set and to change the file system partition at runtime for load balancing through its CRUSH [41] extension. Nevertheless, geographical distribution over Amazon's EC2 is discouraged by the CephFS developers [42].

The Google File System (GoogleFS) [43] stores data on a swarm of *slave* servers. It maintains

metadata on a logically centralized master, replicated on several servers using state machine replication and total ordering of commands using Paxos [44]. It does not consider the case of a file system spread over multiple datacenters and the associated partitioning. MooseFS [45] is designed around a similar architecture and has the same limitations.

HDFS [46] is the distributed file system of the Hadoop framework. It is optimized for read-dominated workloads. Data is replicated and sharded across multiple data nodes. A name node is in charge of storing and handling metadata. As for GoogleFS, this node is replicated for availability. The HDFS interface is not POSIX-compliant and it only implements a subset of the specification via a FUSE interface. QuantcastFS [47] is a replacement for HDFS that adopts the same internal architecture. Instead of three-way replication, it exploits Reed-Solomon erasure coding to reduce space requirements while improving fault tolerance.

GeoFS [48] is a POSIX-compliant file system for WAN deployments. It exploits user-defined timeouts to invalidate cache entries. Clients pick the desired consistency for files and metadata, as with WheelFS's semantic cues [49].

Red Hat's GFS/GFS2 [50] and GlusterFS [51] support strong consistency by enforcing quorums for writes, which are fully synchronous. GlusterFS can be deployed across WAN links, but it scales poorly with the number of geographical locations, as it suffers from high-latency links for all write operations.

XtreemFS [52] is a POSIX-compliant system that offers per-object strong-consistency guarantees on top of a set of independent volumes. Metadata is replicated by metadata servers (MRCs) while files themselves are replicated by object storage devices (OSDs). Write and read operations are coordinated by the OSDs themselves, without involving metadata servers.

BlueSky [53] is a filesystem backed by storage from cloud providers (e.g., Amazon S3). Clients access the filesystem through a local, non-replicated, proxy. The proxy can cache data and use its local disk to improve the response time for local clients. Concurrent access from multiple proxies is left as future work.

SCFS [54] is another distributed filesystem backed by cloud storage. Is uses a distributed coordination service (e.g., Paxos) to handle metadata and provide strong consistency on top of eventually consistent storage. SCFS provides an optimization for non-shared files, where a client can lock a file to bypass the coordination service until the file is closed. Otherwise, all operations go through the single coordination service.

PVFS [55] can be adapted to support linearizability guarantees [56] by delegating the storage of the file system's metadata to Berkeley DB [57], which uses Paxos to totally order updates to its replicas.

### 2.5.2 File systems with weak consistency

There are several distributed file systems for high-performance computing clusters, such as PVFS, PVFS2/OrangeFS [58], Lustre [59], and FhGFS/BeeGFS [60]. These systems have specific (e.g., MPI-based) interfaces and target read-dominated workloads. GIGA+ [61] implements eventual consistency and focus on the maintenance of very large directories. It complements the OrangeFS cluster-based file system.

ObjectiveFS [62] relies on a backing object store (typically Amazon S3) to provide a POSIX-compliant file system with read-after-write consistency guarantees. If deployed on a WAN, ObjectiveFS suffers from long round-trip times for operations such as `fsync` that need to wait until data has been safely committed to S3.

Close-to-open consistency was introduced along with client-side caching mechanisms for the Andrew File System and available in its open-source counterpart OpenAFS [63]. This was a response to previous distributed file systems designs such as LOCUS [64], which offered strict POSIX semantics but with poor performance. Close-to-open semantics are also used by NFS [65], HDFS [46], and WheelFS [49].

OCFS [66] is a distributed file system optimized for the Oracle ecosystem. It provides a cache consistency guarantee by exploiting Linux's `O_DIRECT`. Its successor OCFS2 [66] supports POSIX while guaranteeing the same level of cache consistency.

## 2.6 Storage cost-aware systems

The last challenge we address in this thesis is minimizing the storage overhead in geo-distributed storage systems. Distributed data stores are core components of many cloud applications. It is, therefore, crucial to serve content to end users with *high availability* and *low latency*.

The first property, *high data availability*, is achieved through redundancy. To that end, distributed storage systems either replicate data across multiple sites or use more elaborate approaches such as erasure coding [11]. *Replication* implies storing full copies of the data at different sites. This approach is simple to implement and provides good performance, as users can get data from the nearest data site. However, replication suffers from high storage and bandwidth overheads.

By contrast, *erasure coding* achieves equivalent levels of data protection with less storage overhead but is more complex to set up and operate. The key idea is to split an object into $k$ data blocks (or "chunks"), compute $m$ redundant chunks using an encoding scheme, and store these chunks at different physical locations. A client only needs to retrieve any $k$ of the $k+m$ chunks to reconstruct the object. Erasure coding is thus efficient in terms of storage space and bandwidth, but unfortunately incurs higher access latency, as users now need to contact more distant data sites to reconstruct the data.

The second property, *low read latency*, is in turn attained in systems through *caching*. The principle consists in storing a subset of the available content in memory that is faster or closer to users than the original source. In the case of systems that span geographically diverse regions, caching decreases access latency by bringing data closer to end users.

We first provide background on *the use of erasure coding in distributed storage systems* and *caching*, the go-to approach for improving performance.

### 2.6.1 Erasure coding in storage systems

Erasure coding is a well-established data protection mechanism. It splits data in $k$ blocks (or *chunks*), and computes $m$ redundant blocks using an erasure code. This process allows clients of a storage system to reconstruct the original data with any $k$ of the resulting $k+m$ blocks. Erasure codes deliver high levels of redundancy (and hence reliability) without paying the full storage cost of plain replication. This conciseness has made them particularly attractive to implement fault-tolerant storage back-ends [11, 67].

Beyond the inherent costs of coding and decoding, however, erasure codes lead to higher latency, notably within geo-distributed systems. This is because they only partially replicate data, in contrast to a full replication strategy. Thus, they are more likely to force clients to access remote physical locations to obtain enough blocks to decode the original data. Because of this performance impact, some systems relegate erasure codes to the archiving of *cold*, rarely-accessed data and resort back to replication for *hot* data (e.g., Windows Azure Storage [68] uses this mechanism). Other systems, like HACFS [69], adopt different erasure codes for *hot* and *cold* data: a fast code with low recovery cost for hot data, and a compact code with low storage overhead for cold data. This dual-code strategy helps alleviate the computational complexity of decoding data, but HACFS still suffers from latency incurred by accessing data blocks from remote locations.

Throughout Chapter 5 we use Reed-Solomon [70] (RS) codes. Despite being some of the oldest error correcting codes, RS codes are still widely used in storage systems and digital communications. This is due to two main advantages. First, RS codes are capable of correcting both random and burst errors. Second, there exist efficient decoding algorithms for data encoded with RS.

### 2.6.2 Caching

The classical way to prioritize "hot" data over "cold" data is via a separate caching layer: caches store hot data and are optimized to serve that data quickly. Since cache memory is limited, caching works well for systems whose workloads exhibit a considerable degree of *locality*. Internet traffic typically follows a probability distribution that is highly skewed [71, 72, 73]. In particular, workloads from Facebook and Microsoft production clusters have shown that the top 5% of objects ca be seven times more popular than the bottom 75% [74]. This observation

implies that a small number of objects are more likely to be accessed and would benefit more from caching. The *Zipfian* distribution has been widely used for representing such workloads; for instance, Breslau et al. [71] modeled the popularity of Web content using a Zipfian distribution.

A caching policy represents a heuristic that is applied at a local caching node to pick an eviction candidate, when the cache is full. Jin et al. [75] identify three categories of caching policies, based on different properties of Internet-specific workloads:

1. temporal access locality,
2. access frequency,
3. a mix between the previous two.

We extend this classification with a fourth category, as suggested by DaSilva et al. [76], that takes into account the size of objects.

**Least Recently Used (LRU)** LRU is a policy that relies on temporal access locality, which assumes that an object that has been recently requested is likely to be requested again in the near future. Thus, when the cache is full, LRU chooses the least recently accessed object as eviction candidate. The main advantage of LRU is that it automatically adapts to changes in access patterns. For example, Mokhtarian et al. [77] propose a LRU-based solution for caching in planet-scale video CDNs. While LRU is simpler to implement, Agar (the system we propose in Chapter 5) can bring better latency improvements.

**Least Frequently Used (LFU)** LFU is a policy that relies on metadata that captures the object access history: objects that are most popular are kept in the cache, at the expense of the less popular objects. LFU works best when the access pattern does not change much over time. The main challenge for LFU is to minimize the amount of metadata needed and still take good decisions regarding what objects to cache. A recent example of LFU-based policy is TinyLFU [78], a frequency-based cache admission policy: rather than deciding which object to evict, it decides whether it is worth admitting an object in the cache at the expense of the eviction candidate. Like Agar, TinyLFU is dedicated to caches subjected to skewed access distributions. TinyLFU builds upon Bloom filter theory and maintains an approximation of statistics regarding the access frequency for objects that have been requested recently. Unlike TinyLFU, Agar is designed for erasure coded data and tries to optimize the entire cache configuration rather than taking decisions per object. However, we believe that Agar can benefit from some of the optimizations in TinyLFU (e.g., space saving, aging mechanism) to make it more scalable.

**Hybrid LRU/LFU** Some policies aim to combine the advantages of LRU and LFU by taking into account both the popularity and temporal locality of access in order to determine an optimal cache configuration. For example, WLFU [79] takes decisions based on statistics from the $W$ most recent requests, rather than keeping track of the entire object access history. WLFU uses LFU by default to choose which object to evict from the cache; if there are multiple objects with the same popularity score, WLFU uses LRU to break the tie and evict the least recently accessed object.

**Largest File First (LFF)** Since objects on the Web vary dramatically in size, some papers advocate for the idea of extending traditional caching policies to take into consideration the object size. GreedyDual-Size [80] combines recency of reference with object size and retrieval cost. An object is assigned an initial value based on its size and retrieval cost; this value is updated when the object is accessed again. Unlike Agar, GreedyDual-Size does not take the popularity of an object into account. GDSF [81] is a popularity-based extension to GreedyDual-Size. Like GreedyDual-Size, it computes the cost of objects based on information regarding the recency of access, and the size of an object, but also takes into account access frequency. In GDSF, larger objects have higher cost, and are, thus, more likely to be chosen as eviction candidates. LRU-SP [82] is a LRU extension that takes into account both the size and the popularity of an object. It is based on Size-Adjusted LRU [83]—a generalization of LRU that sorts cached objects in terms of the ratio between cost and size, and uses a greedy approach to evict those with the least cost-to-size ratio from the cache, and Segmented LRU [84]—a caching strategy designed to improve disk performance by assigning objects with different access frequency to different LRU queues. LRV [85] also takes into account the size, recency, and frequency of objects, but it is known for its large number of parameters and implementation overhead. While Agar draws inspiration from these approaches, none of these address the problems brought by keeping data coded. For example, we found that greedy algorithms are not suitable choices among Agar's caching options (see Chapter 5).

### 2.6.3 Caching erasure-coded data

CAROM [67] is a LRU-based caching scheme tailored for erasure-coded cloud file systems, whose workloads are known to exhibit temporal locality. CAROM considers both read and write operations and needs to provide strong consistency guarantees. CAROM totally orders writes by assigning each object to a primary data center, which becomes solely responsible for the object (encoding it and distributing the chunks during writes and storing the chunks during reads). CAROM mainly addresses the problem of supporting writes in erasure-coded systems, while Agar addresses the problem of optimizing cache configuration during reads.

Concurrent to our work, Aggarwal et al. [86] developed Sprout to address erasure coded chunks in caches. They develop an analytical model for the latencies of retrieving data and solve the integer optimization problem to find the cache parameters (cache contents) that minimize the latency. Agar differs in the approach to the problem, by mapping it to Knapsack. In the end,

both Sprout and Agar obtain approximate solutions to the problem, as solving the optimization accurately is computationally intensive and impractical in a large system. While Sprout was still at the simulation level (at the time of this work), we have deployed and evaluated the Agar prototype across a wide area network in Amazon Web Services. We are looking forward to further experimental validation of Sprout in order to draw conclusions on how Agar's strategy compares.

Rashmi et al. recently proposed EC-Cache [87], which applies online erasure-coding to objects stored in cluster caches. In contrast, Agar is a stand-alone caching system that augments multi-site erasure-coded storage systems with caches which it populates based on live information regarding data popularity and access latency to different storage sites.

# 3 State Partitioning in Geo-Distributed Systems

## 3.1 Introduction

This chapter focuses on improving the performance and scalability of large-scale distributed systems that rely on state machine replication (SMR). SMR guarantees that a service is (1) *responsive* – by replicating the service at multiple sites such that it remains available even in the presence of failures, and (2) *consistent* – by having service replicas execute client commands in the same order. In practice, SMR is used to store configuration and control information for large scale distributed systems. Read-only operations can be served locally with low latency. However, SMR incurs a performance penalty for write operations, as replicas need to agree on the order in which to execute commands. The cost of coordination increases with the scale of the service, as latency is larger when replicas are deployed at distant data centers.

We address this problem by partitioning the state of the service and distributing the resulting partitions among servers. Our approach allows system administrators to choose the degree of replication for different partitions. Some partitions could still be fully replicated, while others could be local to the data centers from where they are being accessed most frequently. The use of our partitioning approach is transparent to applications, since the partitioned service offers the exact same semantics and API as the original.

We observe empirically that partitioning can significantly improve the performance of a service. Figure 3.1 shows the benefits of partitioning in the context of a distributed producer–consumer service. The classic producer–consumer problem describes two client types: producers and consumers, who insert and, respectively retrieve, objects from a shared queue. We deployed a distributed producer–consumer service in our university's cluster, using four virtual machines with 2 GB RAM and 2 CPU cores each. We first replicated the service such that all clients access one global queue. Next, we split the service into partitions (initially two, then four) and deployed one queue per partition, in addition to the global one. We consider a workload in which accesses are 80% local (i.e., only affect the queue in the local partition).

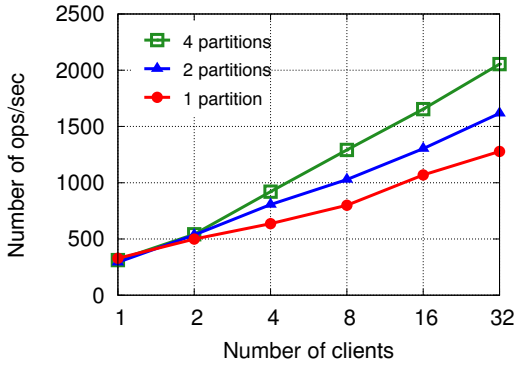



Figure 3.1: Partitioning a producer–consumer service leads to performance improvement.

The aggregated number of operations increases almost linearly with the number of clients. For 32 clients, the service split into four partitions outperforms the non-partitioned one by a factor of 1.6. This is because all servers need to coordinate when accessing the global queue. On the other hand, when using partitioning, 80% of the commands are solely executed against local queues; this requires little or no coordination among servers, depending on the scheme used to partition the service.

The remainder of this chapter begins by formally defining our system model and the notion of partition. Next, we formulate specific conditions under which it is possible to partition a service. We present a general algorithm to build a dependable and consistent partitioned service and assess its practicability using the ZooFence coordination service – an application of our principled approach to Apache ZooKeeper [12].

The material in this chapter is adapted from previously published work [88].

## 3.2 System model

A service is specified by some serial data type. The serial data type defines the possible states of the service, the operations (or *commands*) to access it, as well as the response values from these commands. Formally, a serial data type is an automaton $S = (States, s^0, Cmd, Values, \tau)$ where $States$ is the set of states of $S$, $s^0 \in States$ its initial state, $Cmd$ the commands of $S$, $Values$ the response values and $\tau : States \times Cmd \to States \times Values$ defines the transition relation. A command $c$ is *total* if $States \times \{c\}$ is in the domain of $\tau$. Command $c$ is *deterministic* if the restriction of $\tau$ to $States \times \{c\}$ is a function. Hereafter, we assume that all commands are total and deterministic. We use $.st$ and $.val$ selectors to respectively extract the state and the response value components of a command, i.e., given a state $s$ and a command $c$, $\tau(s,c) = (\tau(s,c).st, \tau(s,c).val)$. Function $\tau^+$ is defined by the repeated application of $\tau$, i.e.,

given a sequence of commands $\sigma = \langle c_1, \ldots, c_{n\geq 1} \rangle$ and a state $s$:

$$\tau^+(s,\sigma) \triangleq \begin{cases} \tau(s,c_1) & \text{if } n = 1, \\ \tau^+(\tau(s,c_1).st, \langle c_2, \ldots, c_n \rangle) & \text{otherwise.} \end{cases}$$

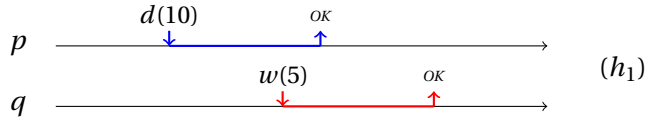
Two commands $c$ and $d$ *commute*, written $c \asymp d$, if in every state $s$ we have:

$$c \asymp d \triangleq \begin{cases} \tau^+(s,\langle c,d \rangle).st = \tau^+(s,\langle d,c \rangle).st \\ \tau^+(s,\langle d,c \rangle).val = \tau^+(s,c).val \\ \tau^+(s,\langle c,d \rangle).val = \tau^+(s,d).val \end{cases}$$

For any two commands $c$ and $d$, we write $c = d$ when in every state $s$, $\tau(s,c) = \tau(s,d)$. By extension, for some command $c$ and some sequence $\sigma = \langle c_1, \ldots, c_{n\geq 2} \rangle$, we write $c = \sigma$ when $\tau^+(s,\langle c_1, \ldots, c_n \rangle) = \tau(s,c)$.

To illustrate the above notations, let us consider a bank account equipped with the usual withdraw and deposit operations. We define $States$ as $\mathbb{N}$, with $s_0 = 0$. A deposit operation $d(10)$ updates $s$ to $s+10$ and returns *OK*. In case the bank prohibits overdrafts, a withdraw operation $w(x)$ returns *NOK* if $s < x$; otherwise it changes $s$ to $s - x$.

**History** We consider a global time model and some bounded set of client processes that may fail-stop by crashing. A history is a sequence of invocations and responses of commands by the clients on one or more services. When command $c$ precedes $d$ in history $h$, we write $c <_h d$. We use timelines to illustrate histories. For instance the timeline below depicts the interleaving of commands $d(10)$ and $w(5)$, executed by respectively clients $p$ and $q$ in some history $h_1$.



$(h_1)$

Following [89], histories have various properties according to the way invocations and responses interleave. For the sake of completeness, we recall these properties in what follows. A history $h$ is *complete* if every invocation has a matching response. A *sequential* history $h$ is a non-interleaved sequence of invocations and matching responses, possibly terminated by a non-returning invocation. When a history $h$ is not sequential, we say that it is *concurrent*. A history $h$ is *well-formed* if (i) $h|p$ is sequential for every client process $p$, (ii) for every command $c$, $c$ is invoked at most once in $h$, and (iii) for every response $res_i(c)\ v$ there exists an invocation $inv_i(c)$ that precedes it in $h$.[1] A well-formed history $h$ is *legal* if for every service $S$, $h|S$ is both complete and sequential, and denoting $\langle c_1, \ldots, c_{n\geq 1} \rangle$ the sequence of commands appearing in $h|S$, if for some command $c_k$ a response value appears in $h|S$, it equals $\tau^+(s^0, \langle c_1, \ldots c_k \rangle).val$.

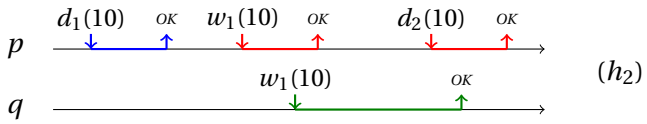
[1] For some service or client $x$, $h|x$ is the projection of history $h$ over $x$.

**Linearizability** Two histories $h$ and $h'$ are said *equivalent* if they contain the same set of events. Given a service $S$ and a history $h$ of $S$, $h$ is *linearizable* [89] if it can be extended (by appending zero or more responses) to some complete history $h'$ equivalent to a legal and sequential history $l$ of $S$ with $<_{h'} \subseteq <_l$. In such a case, history $l$ is named a *linearization* of $h$. For instance, the history $h_1$ above is linearizable since it is equivalent to a sequential one in which $d(10)$ occurs before $w(5)$. The histories of a service $S$ are all the linearizable histories that are constructible with the commands of $S$. A service $S$ *implements* a service $T$ when for every linearizable history $h$ of $S$, there exists a linearizable history $h'$ of $T$ such that $h'$ is a high-level view of $h$ [90].[2]

**Partition** Given a finite family of services $(S_k)_{1\leq k\leq n}$, the synchronized product of $(S_k)_k$ is the service defined by $(\prod_k States_k, (s_1^0,\ldots,s_n^0), \bigcup_k Cmd_k, \bigcup_k Values_k, \tau)$ where for every state $s=(s_1,\ldots,s_n)$ and every command $c$ in some $Cmd_k$, the transition function $\tau$ is given by $\tau(s,c) = ((s_1,\ldots,\tau_k(s_k,c).st,\ldots,s_n), \tau_k(s_k,c).val)$. Given a service $S$, the family $(S_k)_{1\leq k\leq n}$ is a *partition* of $S$ when its synchronized product satisfies (i) $States \subset \prod_k States_k$, (ii) $s^0 = (s_1^0,\ldots,s_n^0)$, and (iii) for every command $c$, there exists a unique sequence $\sigma$ in $\bigcup_k Cmd_k$, named the *sub-commands* of $c$, satisfying $\sigma = c$. The partition $(S_k)_k$ is said *consistent* when it implements $S$.

To illustrate the notion of partition, let us go back to our banking example. A simplistic bank service allows its clients to withdraw and deposit money on an account, and to transfer money between two accounts. We can partition this service into a set of branches, each holding one or more accounts. A transfer of an amount $x$ between accounts $i$ and $j$ is modeled as the sequence of sub-commands $\langle w_i(x).d_j(x)\rangle$, where $w_i(x)$ represents a withdrawal of the amount $x$ into account $i$, and $d_j(x)$ represents a deposit of the amount $x$ into account $j$. However, precautions must be taken when concurrent commands occur on the partitioned service.

For instance, the following history should be forbidden by the concurrency control mechanism to avoid money creation. If $p$ deposits 10 in account 1, the concurrent withdrawal attempts of $p$ and $q$ from account 1 cannot both succeed as the balance is not sufficient. Note that the third operation executed by $p$ is depositing 10 in another account (account 2).



$(h_2)$

In the section that follows, we characterize precisely when the partition of a service is correct.

[2] A high-level view is generally constructed via a refinement mapping from the states of $S$ to the states of $T$ [91].

## 3.3 Partitioning theorems

When there is no invariant across the partition and every command is a valid sub-command for one of its parts, the partition is *strict*. We first establish that a strict partition is always consistent.

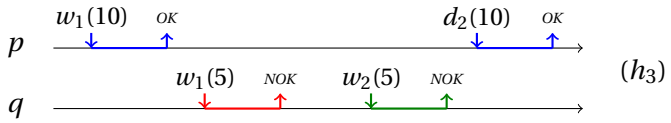
**Theorem 3.1** *Consider a service $S$ and a partition $(S_k)_k$ of $S$. If both $\prod_k States_k = States$ and $Cmd = \bigcup_k Cmd_k$ hold then $(S_k)_k$ is a consistent partition of $S$.*

The above theorem is named the locality property of linearizability [89]. It states that the product of linearizable implementations of a set of services is linearizable. From the perspective of service partitioning, this suggests that each part should be implemented as a replicated state machine. Such an idea forms the basic building block of our protocols.

When commands contain several sub-commands, Theorem 3.1 does not hold anymore. Nevertheless, it is possible to state a similar result when constraining the order in which sub-commands interleave. This is the observation made by [10], despite a small omission in the original paper. Below, we show where the error occurs and propose a corrected and extended formulation. To state our results, we first need to introduce the notion of conflict graph.

**Definition 3.1** *(Conflict Graph) Consider a history $h$ of a partition $(S_k)_k$ of some service $S$. The* conflict graph *of $S$ induced by $h$ is the graph $G_h = (V, E)$ such that $V$ contains the set of commands executed in $h$, and $E$ is the set of pairs of commands $(c, d)$ with $c \neq d$, for which there exist two sub-commands $c_i$ and $d_j$ executed on some $S_k$ such that $c_i <_{h|S_k} d_j$.*

In [10], the authors claim that the partition $(S_k)_k$ is consistent, provided that $h$ is linearizable for each part $S_k$ and $G_h$ is acyclic. Unfortunately, this characterization is incorrect because $G_h$ does not take into account the causality with which commands are executed in $h$. We argue this point with a counter-example. Let us consider again that our banking service is partitioned in a set of branches. Clients $p$ and $q$ execute the three commands $\langle w_1(10), d_2(10)\rangle$, $w_1(5)$ and $w_2(5)$ as in history $h_3$ where account 1 is initially provisioned with the amount 10.




Since both withdrawals of client $q$ fail, history $h_3$ is not linearizable. However, $G_{h_3}$ remains acyclic since it does not capture that process $q$ creates the order $w_1(5) <_{h_3} w_2(5)$.

In what follows, we prove an extended and corrected formulation of the partitioning result of [10]. Our characterization is based on the notion of semantic graph that we define next.

**Definition 3.2** *(Semantic Graph) Consider a history $h$ of a partition $(S_k)_k$ of some service $S$. The* semantic graph *of $S$ induced by $h$ is the graph $G_h = (V, E)$ such that $V$ contains the set of*

**Algorithm 1** Base construction – code at client $p$

```
1: invoke(c) :=
2:     let S_k such that c ∈ Cmd_k
3:     return M(S_k).invoke(c)
```

*commands that appear in $h$ and $E$ is the set of pairs $(c,d)$, with $c \neq d$, for which either(i) there exist two non-commuting sub-commands $c_i$ and $d_j$ in some part $S_k$ such that $c_i <_{h|S_k} d_j$, or (ii) $c <_h d$, where we note $c <_h d$ when all the sub-commands of $c$ precede all the sub-commands of $d$ in history $h$.*

In contrast to the notion of conflict graph, a semantic graph takes into account the commutativity of sub-commands. To understand why, assume that the banking service allows unlimited overdraft. In such a case, any interleaving of the sub-commands would produce a linearizable history. The partitioning theorem that follows generalizes this observation. It states that the acyclicity of non-commuting sub-commands in the semantic graph is a sufficient condition to attain consistency.

**Theorem 3.2** *A partition $(S_k)_k$ of a service $S$ is consistent if for every history $h$ of $(S_k)_k$, there exists some linearization $l$ of $h$ such that the semantic graph of $S$ induced by $l$ is acyclic.*

## 3.4 Protocols

Building upon our previous theorems, this section describes several constructions to partition a shared service. Our presentation follows a refinement process. We start with an initial construction requiring that strictly disjoint services form the partition, then we introduce a more general technique that can accommodate any type of partitioning. Our last construction improves parallelism at the cost of constraining how the partition is structured. To ease the presentation of our algorithms, we shall be assuming hereafter that sub-commands are idempotent and that no two sub-commands in the same command access the same part. Nevertheless, all of our algorithms can be easily adapted to handle the cases where such properties do not hold.

### 3.4.1 Initial construction

We depict in Algorithm 1 a first construction when the partition $(S_k)_k$ of service $S$ is strict. This algorithm makes use of a mapping $\mathcal{M}$ satisfying that for every $S_k$, $\mathcal{M}(S_k)$ is a replicated state machine implementing $S_k$. When a client $p$ executes a command $c$ on $S$, it uses $\mathcal{M}$ to retrieve the state machine implementing $S_k$, where $S_k$ is the service on which $c$ executes (line 2). Then, client $p$ invokes the command on $\mathcal{M}(S_k)$ and returns the result of this invocation (line 3).

Since $(S_k)_k$ is strict and $\mathcal{M}(S_k)$ is a linearizable implementation of $S_k$, Algorithm 1 implements a consistent partition of $S$ by Theorem 3.1. Besides, the implementation of $(S_k)_k$ obtained

through Algorithm 1 is wait-free [92]. This property is inherited from the underlying replicated state machines that support Algorithm 1. In addition, this base construction is optimal in terms of scalability since, when clients access uniformly the parts, the throughput of the partitioned service is $|(S_k)_k|$ times the throughput of $S$.

### 3.4.2 A queue-based construction

In what follows, we refine Algorithm 1 to handle the case where multiple sub-commands compose a command. A naive solution would consist in modifying Algorithm 1 so that when the client process $p$ executes a command $c = \langle c_1, \ldots, c_n \rangle$, it applies in order all the sub-commands $c_1, \ldots, c_n$ to the appropriate part. Such an approach however fails since (i) an invariant may link different parts of the partition, and (ii) if client $p$ crashes in the middle of its execution, not all the parts will reflect the effects of command $c$.

Algorithm 2 depicts a solution to deal with these two issues. This algorithm ensures that either all the sub-commands of a command execute, or none of them, and that the state of the partitioned service remains consistent. It is based on a shared FIFO queue abstraction (variable $Q$) and an eventual leader election service (variable $\Omega$). Clients use $Q$ to submit the commands they wish to execute. Submitted commands are then executed in the queue order by the leader elected by $\Omega$.

With more details, our algorithm works as follows. Upon invoking a command $c = \langle c_1, \ldots, c_n \rangle$, a client $p$ appends $c$ to the queue $Q$, then it starts participating in the leader election (line 8). In case $p$ is elected, it processes the commands in $Q$ (line 15). For each such command $d$, $p$ executes all the sub-commands of $d$ once every non-commuting command before $d$ has been executed (line 15). The result of the last sub-command of $d$ is stored as the response of $d$ in the queue $Q$ (lines 17 to 21). This pattern is repeated until the leader, which might not be $p$, executes command $c$.

The leader election service $\Omega$ allows a process to register (line 8) and to unregister (line 10). This service satisfies that eventually (i) only registered processes are elected, and (ii) at least one correct process considers itself as the leader. We note here that property (ii) was previously mentioned in [93], and that $\Omega$ is a form of restricted leader election [94]. This makes $\Omega$ strictly weaker than the leader oracle used in consensus [95].

### 3.4.3 Ensuring disjoint access parallelism

Both the protocol of [10] and Algorithm 2 order submitted commands through some global shared object: an instance of the Ring Paxos protocol in the case of [10], and a shared queue for Algorithm 2. As a consequence, the synchronization cost of executing a command is related to the number of concurrent commands. This defeats the primary goal of partitioning which is to scale-up the service by leveraging parallelism for commands that access different parts of the service. Such a property is named *disjoint-access parallelism* (DAP) in the literature

**Algorithm 2** Queue-based construction – code at client $p$

```
 1: Shared Variables:
 2:     Ω                                                    ▷ a leader election
 3:     Q                                                    ▷ an atomic queue
 4:
 5: invoke(c) :=
 6:     r ← ⊥
 7:     Q ← Q ∘ (c, r)
 8:     Ω.register()
 9:     wait until r ≠ ⊥
10:     Ω.unregister()
11:     Q ← Q \ (c, r)
12:     return r
13:
14: when p = Ω.leader()
15:     let (d, r') ∈ Q : ∀(e, r̂) <_Q (d, r') : r̂ ≠ ⊥ ∨ d ≍ e
16:     let d_1, ..., d_m : d = ⟨d_1, ..., d_m⟩
17:     for all j ∈ [[1, m]] do
18:         let S_k : d_j ∈ Cmd_k
19:         r'' ← M(S_k).invoke(d_j)
20:     end for
21:     r' ← r''
22: end when
```

on shared memory computing [96]. In what follows, we depict a refinement of our previous algorithm that ensures the following DAP property:

**Definition 3.3 (Disjoint-Access Parallelism)** *Consider an algorithm $\mathscr{A}$ implementing a partition $(S_k)_k$ of some service $S$. We say that $\mathscr{A}$ is* disjoint-access parallel *when in each of its histories $h$ there exists a linearization $l$ of $h$, such that if $p$ and $q$ concurrently executing commands $c$ and $d$ contend on some shared object in $\mathscr{A}$, there exists a non-directed path linking $c$ to $d$ in the conflict graph of $l$.*

Algorithm 3 depicts our construction of a DAP consistent partitioning. For each part $S_k$, we assign respectively a queue $Q[k]$ and a leader election $\Omega[k]$. When a client $p$ invokes a command $c = \langle c_1, \ldots, c_m \rangle$, it iteratively executes each sub-command $c_i$ on the appropriate replicated state machine. To that end, client $p$ adds $c_i$ to the queue $Q[k]$ and then joins leader election $\Omega[k]$ to execute all the sub-commands in $Q[k]$. The helping mechanism in Algorithm 3 is similar to the one we employed in Algorithm 2: when $p$ is the leader and a sub-command $d_j$ occurs before $c_i$ in the queue $Q[k]$, $p$ must first execute $d_j$ as well as the sub-commands following it, before it can execute $c_i$ (lines 18 to 24). This pattern ensures the correctness of our construction in the case where the following property holds:

(P1) There exists an ordering $\ll$ of $(S_k)_k$ such that for any two sub-commands $c_i$ and $c_j$ accessing respectively parts $S_k$ and $S_{k'}$, if $c_i$ precedes $c_j$ in $c$ then $S_k \ll S_{k'}$ holds.

Unfortunately, property P1 does not hold for every partition $(S_k)_k$ of a service $S$. For instance in

**Algorithm 3** DAP construction – code at client $p$

```
1:  Shared Variables:
2:      Ω                                              ▷ an array of leader election objects
3:      Q                                              ▷ an array of atomic queues
4:
5:  invoke(c) :=
6:      return invoke_sub(c)
7:
8:  invoke_sub(c_i) :=
9:      r ← ⊥
10:     let S_k : c_i ∈ Cmd_k
11:     Q[k] ← Q[k] ∘ (c_i, r)
12:     Ω[k].register()
13:     wait until r ≠ ⊥
14:     Ω[k].unregister()
15:     Q[k] ← Q[k] \ (c_i, r)
16:     return r
17:
18: when p = Ω[l].leader()                             ▷ for some l
19:     let (d_j, r') ∈ Q[l] : ∀(e_j', r̂) <_Q[l] (d_j, r') : r̂ ≠ ⊥ ∨ d_j ≃ e_j'
20:     r'' ← M(S_l).invoke(d_j)
21:     if d_j ≠ last(d) then
22:         r'' ← invoke_sub(d_j+1)
23:     end if
24:     r' ← r''
25: end when
```

our previous banking example, P1 only holds if money transfers between accounts in different branches occur in some canonical order: e.g., $\langle w_i(x), d_j(x)\rangle$ is allowed if and only if $i < j$ holds.

Our key observation is that we can nevertheless enforce the acyclicity of the semantic graph by implementing the partition in a hierarchical manner. We achieve this via two modifications to Algorithm 3. First, we replace P1 by the fact that:

(P2) Function $\mathcal{M}$ returns a set of replicated state machines for each $S_k$ such that for any two sub-commands $c_i$ and $c_j$ accessing respectively parts $S_k$ and $S_{k'}$, if $c_i$ precedes $c_j$ in $c$ then either $\mathcal{M}(S_k) \subseteq \mathcal{M}(S_{k'})$ or the converse holds.

Second, upon executing a sub-command $c_i$ (at line 20 in Algorithm 3), we apply $c_i$ in some canonical order to all the replicated state machines in $\mathcal{M}(S_k)$ before returning the value $r''$. These two modifications ensure that the premises of Theorem 3.2 hold for any partition of some shared service $S$.

Going back to the design of a partitioned banking service, applying P2 requires the addition of a special account $t$ replicated at all branches, such that when a money transfer occurs between two accounts in different branches, money goes through account $t$, i.e., $\langle w_i(x), d_t(x).w_t(x).d_j(x)\rangle$.

Algorithm 3 with property P2 is the general method we employ to partition a service. We im-

plemented it in ZooFence where we partition the shared tree interface exposed by the Apache ZooKeeper coordination service. By partitioning the tree, ZooFence reduces contention and leverages the locality of operations. Both effects contribute to improve latency of operations and increase overall throughput. We describe ZooFence in detail in the next section.

## 3.5 ZooFence

In this section, we present an application of our principled partitioning approach to the popular Apache ZooKeeper [12] coordination service. The resulting system, named ZooFence, orchestrates several independent instances of ZooKeeper. The use of ZooFence is transparent to applications: it offers the exact same semantics and API as a single instance of ZooKeeper. However, the design of ZooFence allows avoiding synchronization between parts when it is not necessary. This reduces the impact of convoy effects that synchronization causes [97].

The partitioning of the ZooKeeper service follows the approach introduced in Algorithm 3. ZooFence splits the tree structure between multiple ZooKeeper instances. Commands that access distinct parts of the tree run in parallel on distinct instances, while guaranteeing both strong consistency and wait-freedom. Commands that access a single part of the tree run on a single instance. This section presents the main components of ZooFence, discusses our design choices with regard to Algorithm 3, then details some specific aspects of its implementation.

### 3.5.1 Overview

Figure 3.2 depicts the general architecture of ZooFence. The system has four components:(i) A set of independent ZooKeeper instances that ZooFence orchestrates; (ii) A client-side library; (iii) A set of queues storing commands that need to be executed on multiple instances; and (iv) A set of executors that fetch commands from the queues, delegate them to the appropriate instances, and return the result to the calling clients.

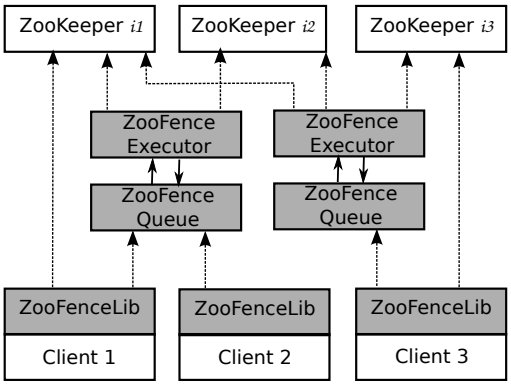



Figure 3.2: ZooFence design.

### 3.5.2 Client-side library

ZooFence clients execute commands through a client-side library implementing the ZooKeeper API. This interface consists of a set of commands accessing a concurrent tree data structure composed of *znodes*. A znode in the tree stores some data, and is accessible via a *path* as in UNIX filesystems. Znodes can be *persistent*, meaning they belong to the tree until a client explicitly deletes them, or *ephemeral*, in which case they are automatically removed once the client that created them disconnects or crashes. In addition, znodes can be *sequential*. For such znodes, the system automatically appends a monotonically increasing counter to their names at creation time. A client can manipulate a znode through read or write commands. Read commands, e.g., `exists`, `getChildren` or `getData` return a sub-state of the shared tree without modifying it. Write commands such as `create`, `delete` or `setData`, modify the state of the tree and return metadata information to the client.

Clients execute a command on one or more of the orchestrated ZooKeeper instances according to a *flattening function*. This function plays the role of function $\mathscr{M}$ in Algorithm 3 and maps paths to ZooKeeper instances. Multiple flattening functions can be used. The choice of a flattening function depends on how the application accesses the concurrent tree structure. When a client executes a command $c$ on a znode $n$, ZooFence determines, using the flattening function, the set of ZooKeeper instances $I = f(n)$ on which $c$ is executed. Command $c$ is *trivial* when $f(n)$ returns a single ZooKeeper instance. In such a case, the command is directly forwarded to that instance. If $c$ is *non-trivial*, it is inserted into the appropriate queue. The executor associated to the queue forwards the command to the corresponding ZooKeeper instances, then returns the result to the client.

The flattening function $f$ satisfies property P2. This means that if a znode $n$ with parent $p$ is mapped to a set of instances $I = f(n)$ then $I$ is also a subset of $f(p)$. To understand why, consider that znode `/a` is mapped to instances $\{i_1, i_2, i_3\}$, and `/a/b` to $\{i_1, i_2\}$. In case a client executes `create(/a/b)`, because both instances $i_1$ and $i_2$ hold a copy of `/a`, the creation of `/a/b` succeeds if and only if `/a` was created previously. This ensures that a znode $n$ with parent $p$ is in the tree if and only if $p$ also exists in the tree. The next section provides additional details on the internals of ZooFence, explaining how we maintain this key invariant.

### 3.5.3 Executor

The core component of ZooFence is the executor. Each executor implements the logic of the sub-commands execution mechanism we depicted at lines 18 to 24 in Algorithm 3. There is one executor for each set of ZooKeeper instances replicating a common path. For instance, in the above example, there is one executor for `/a`, replicated at $\{i_1, i_2, i_3\}$ and one for `/a/b`, replicated at $\{i_1, i_2\}$. As explained previously, each executor is associated with a FIFO queue. When a client executes a non-trivial command, it adds that command to the corresponding queue according to the flattening function. The executor scans the queue in order to retrieve the next command $c$ it has to execute. Then, it forwards $c$ to all the associated ZooKeeper

instances, merges their results, and sends the final result back to the client before deleting $c$ from the queue.

**Queue synchronization.** Using multiple executors improves the performance of ZooFence, but requires additional synchronization. To illustrate this point, let us consider again that `/a` is replicated at $\{i_1, i_2, i_3\}$ and `/a/b` at $\{i_1, i_2\}$. The set of instances associated with znode `/a/b` has a smaller cardinality than the set of instances associated with its parent, `/a`; we call `/a/b` a *fringe* znode. Assume that a client attempts to delete `/a`, while another client concurrently attempts to delete `/a/b`. Due to the tree invariant, the deletion of `/a` succeeds only if it does not have any children. In the scenario above, if the deletion of znode `/a/b` finishes on $i_1$, but not on $i_2$, before the deletion of znode `/a` is executed, then `/a` would be deleted from $i_1$, but not from $i_2$, leaving replicas in an inconsistent state. We solve the problems related to fringe znodes by synchronizing queues following the approach depicted at lines 21 to 22 in Algorithm 3. Upon creation of such a znode, the executor adds the command to the parent queue. Upon deletion, the executor first executes the command then, in case of success, it adds the command to the parent queue. When adding a command to the parent queue, the executor waits for a result before returning.

**Failure recovery mechanism.** The executor is a dependable component of ZooFence. To ensure this guarantee, we replicate each executor and employ the same leader election mechanism as in Algorithm 3. When the previously elected executor is unresponsive or has crashed, ZooFence nominates a new one and resumes the execution of commands. ZooFence prevents inconsistencies that might result, as follows: (i) we ensure idempotency at the client side; and (ii) we use the command semantics to resume incomplete commands on the associated ZooKeeper instances.

**Queue monitoring.** Executors retrieve commands from their respective queues and keep them in a local cache. Instead of actively polling the queue for new commands, executors use the ZooKeeper event notification mechanism, watches, which are triggered when the queues are modified. Executors check their local caches every time the watch set on their queue is triggered; if the cache is empty, the executor performs a `getChildren` command to repopulate its cache.

**Asynchronous commands.** Each executor retrieves commands from its local cache and forwards them to its set of ZooKeeper instances. We use asynchronous commands between an executor and its ZooKeeper instances, regardless of the type of command issued by the ZooFence client. The only difference between synchronous and asynchronous client commands is local, in the ZooFence client library; for synchronous commands, the library blocks and waits for the reply to arrive. Since ZooKeeper guarantees FIFO order even for asynchronous commands, this optimization which is transparent to the clients, increases the throughput of the executor without changing the semantics of synchronous

**Colocation.** An executor is a stand-alone process that runs on a different machine than clients; it executes at the same site as the ZooKeeper instance that hosts its corresponding queue.

Our prototype implementation of ZooFence is written in Java and contains around 2,500 SLOC. It is built upon the out-of-the-box ZooKeeper 3.4.5 distribution. Clients transparently instantiate a ZooFence deployment when the connection string contains multiple ZooKeeper instances separated by a "|" character, e.g., "127.0.0.1:2181 | 127.0.0.1:2182". As discussed in Section 3.5.2, ZooFence clients forward non-trivial commands to executors via queues. Client commands are serialized and stored in *persistent sequential znodes*; we use the *sequential* attribute offered by ZooKeeper to assign a sequence number to each command in the queue. Queues are stored in dedicated administrative ZooKeepers. Executors are identified by *ephemeral znodes*, also stored in administrative Zookeeper instances. The choice of the administrative Zookeeper instances among the existing instances is currently not automated and left as a future work. Clients open TCP connections to executors before putting commands in queues. When commands complete, executors send the results over the respective connections.

## 3.6 Evaluation

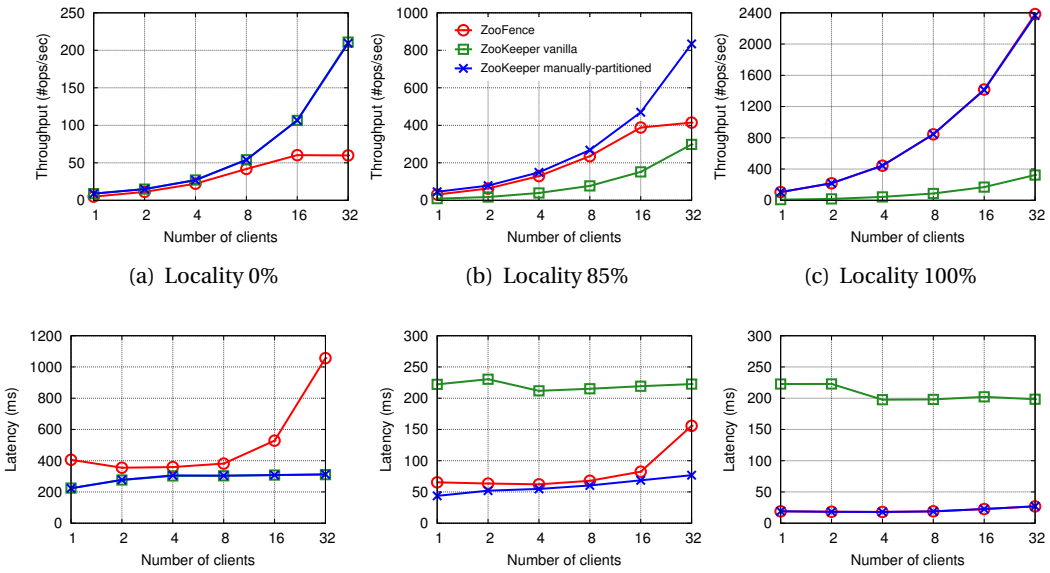
We evaluate our ZooFence prototype in two scenarios: a synthetic concurrent queue service deployed at multiple geographically-distributed sites, and the BookKeeper distributed logging service [98]. We compare ZooFence against vanilla ZooKeeper deployments in terms of throughput, latency and scalability.

### 3.6.1 Concurrent queues service

In this experiment, we show that ZooFence can leverage access patterns locality to improve both throughput and responsiveness of distributed applications. To that end, we emulate a geographically-distributed deployment. Each site consists of dual-core machines with 2 GB RAM, and hosts one ZooKeeper instance and several clients. Inside a site, machines communicate using a (native) gigabit network. Between sites, we set up the round-trip delay to approximately 50 ms. Our experiment uses a single executor, collocated with the administrative ZooKeeper on an eight-core machine at a different site than all clients.

In this environment, we deploy a service consisting of five concurrent queues. Four of the queues exhibit strong locality, and are used exclusively by clients from the same site (i.e., *queue1* is used only by clients from *site1*, *queue2* from *site2*, etc.). We refer to these queues as *site queues*. The fifth queue is used by clients from all sites. We refer to it as the *geo-distributed queue*. We vary the number of clients within each site from one to eight. Each client runs two producer processes. Producers mostly create znodes in their site queue, but occasionally write to the geo-distributed queue as well. We note that this is a write-heavy workload, which represents a worst-case scenario for both ZooFence and ZooKeeper.

Our experiments compare in terms of performance three different deployments:




(a) Locality 0% (b) Locality 85% (c) Locality 100%

(d) Locality 0% (e) Locality 85% (f) Locality 100%

Figure 3.3: Comparison of ZooFence against ZooKeeper.

(1) ZooFence with a flattening function that assigns site queues to local ZooKeeper instances, and the geo-distributed queue to all instances.

(2) A vanilla ZooKeeper, which involves all the available ZooKeeper instances: a leader and three followers, with synchronization bound set to 175 ms. All queues are stored by this ZooKeeper instance. This is the baseline for our experiments – how ZooKeeper would be used in the present.

(3) A manually-partitioned ZooKeeper. Each machine runs two ZooKeeper instances: one instance stores the site queue, and the other one stores the geo-distributed queue; the latter is a ZooKeeper instance covering all sites. This deployment is the optimum an experienced ZooKeeper administrator can achieve: writes accessing the geo-distributed queue are broadcast to all sites, otherwise they are served locally.

Figure 3.3 presents the latency and the throughput of the system for each of the three deployments when we vary the number of clients and the percentage of operations on the geo-distributed queue.

As shown by figures 3.3(b) and 3.3(e), with a locality of 85%, ZooFence is close to the manually-partitioned ZooKeeper. This happens because in both deployments most operations occur inside a site, avoiding the cross-site communication in most cases. The vanilla ZooKeeper deployment exhibits lower throughput and higher latency because all queues are replicated across all sites. Since the leader ZooKeeper propagates updates to its followers, the inter-site communication penalty cannot be avoided.

When locality is maximum, ZooFence is identical to the manually-partitioned ZooKeeper

(figures 3.3(c) and 3.3(f)). The vanilla ZooKeeper deployment performs significantly worse because it fully replicates all znodes.

Finally, when all operations are performed on the geo-distributed queue (Figures 3.3(a) and 3.3(d)), the performance of ZooFence becomes worse than that of the other two deployments, both in terms of throughput and latency. This is due to the overhead incurred by our execution mechanism for operations on replicated znodes: the executor fetches commands from the execution queue, delegates them to the responsible ZooKeeper instances based on the flattening function and forwards the result to the client.

The executor itself can become saturated and degrade performance, in terms of both throughput and latency. In this experiment, we used a single executor, deployed on the same VM as the one hosting the administrative ZooKeeper. The executor becomes saturated at around 60 operations per second, as shown in Figure 3.3(a) and Figure 3.3(b) (only 15% of operations go through the executor, and 15% * 400 = 60). This is mainly due to our design choice of making ZooFence modular, on top of ZooKeeper. This choice implies that our system does not benefit from optimizations such as batching, which require tighter integration. Since the executor is in a separate site, it pays the latency penalty two times for global operations by communicating with the ZooKeeper instances and the clients, which limits performance.

We have performed a similar experiment for read workloads, where conclusions are similar. ZooFence allows local reads to exhibit low latency, and not interfere with the performance of remote partitions.

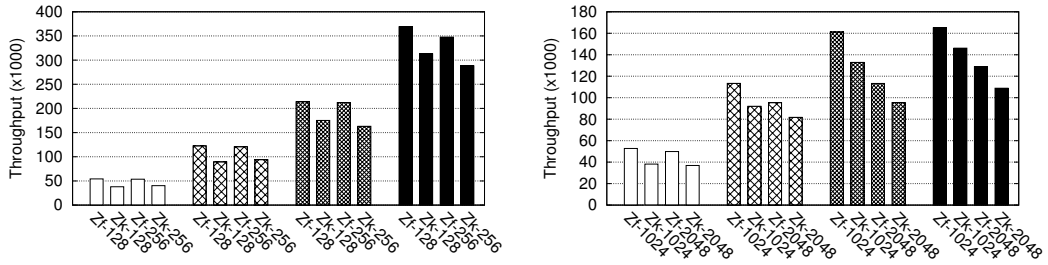
Overall, our experiments show that ZooFence performs close to an optimal deployment when access patterns exhibit strong locality. ZooFence can enable even inexperienced administrators to obtain good performance without the burden of partitioning the state manually.

### 3.6.2 BookKeeper

This section presents experimental results that assess the performance gains of ZooFence over ZooKeeper. To that end, we compare the two systems in a cluster deployment when supporting the BookKeeper logging service [98]. Below, we first give a brief description of BookKeeper, then detail our experimental protocol and comment on our results.

BookKeeper is a distributed system that reliably stores streams of records. Each stream is stored in a write ahead log, or *ledger*, that contains multiple *entries*. A ledger is replicated at one or more dedicated storage servers, named *bookies*. BookKeeper uses the ZooKeeper tree structure to stores the metadata of the ledgers, as well as the set of available bookies. The tree also ensures that a single client writes to a ledger at a time. BookKeeper supports close-to-open semantics: when writing to a ledger, a client appends new entries at a quorum of its replicas. Changes to an open ledger are visible to a concurrent client only when the ledger is closed.

Our experiments consist in evaluating the maximal throughput of BookKeeper when clients concurrently open ledgers and start appending entries to them. In all our experiments, we employ 18 bookies with a replication factor of 3. Such a setting ensures that the system can sustain the failure of one bookie without interruption. We vary both the length of an entry and the frequency at which clients create a new ledger. This last parameter is controlled by the number of entries each client writes to a ledger before closing it.




(a) Small entries

(b) Large entries

Figure 3.4: BookKeeper performance (from left to right, the amount of written entries is 100, 250, 500 and 1000).

Figures 3.4(a) and 3.4(b) present a detailed performance comparison of BookKeeper when using ZooKeeper and ZooFence. We vary the length of an entry from 128 to 2048 bits, and the number of written entries from 100 to 1000. In both figures, Zk stands for ZooKeeper, while Zf means ZooFence; the throughput is measured as the total amount of operations per second. When clients write 1000 entries (or more, not shown on the plots), the two systems achieve close performance. In such a case, the throughput is limited by either the bookies or the network. During our experiments, the MTU is set to 1500 bytes. This explains the performance gap between large and small entries. On the other hand, when clients open concurrently more ledgers, fast operations on the metadata storage matter. In such a case, because ZooFence provides parallel accesses to the shared tree, it outperforms ZooKeeper. The difference increases as clients access new ledgers more frequently. In our experiments, ZooFence improves the throughput of BookKeeper by up to 45%.

## 3.7 Summary

In this chapter, we proposed a partitioning approach to improve the performance and scalability of distributed services. We validated our approach by building ZooFence, a system that automatically partitions ZooKeeper. ZooFence reduces contention and increases the overall throughput of applications using ZooKeeper, especially in geo-distributed scenarios. We assess the practicability of a prototype implementation of ZooFence on two benchmarks: a concurrent queue service and the BookKeeper distributed logging engine. Our experiments show that when locality is optimum, ZooFence improves ZooKeeper performance by almost one order of magnitude.

# 4 Workload-Aware State Partitioning

## 4.1 Introduction

In this chapter, we apply the state partitioning technique described in Chapter 3 to design GlobalFS, a POSIX-compliant distributed file system that spans geographically-diverse regions. We use GlobalFS to illustrate the practical implications of state partitioning, and to explain how partitioning can dynamically adapt to workload patterns.

**Challenges of building a strongly consistent distributed system.** Like other distributed systems that rely on state machine replication, distributed file systems need to choose between availability and consistency in order to work around the limitations formulated by the CAP theorem [4], when network partitions occur. State-of-the-art designs, like Oceanstore [5] and Ivy [6], sacrifice consistency in favor of availability; such systems must be able to anticipate conflicts and provide suitable conflict-resolution mechanisms in order to reconcile diverging replicas. However, distributed file systems are general-purpose applications – i.e., they support a variety of task flows, and thus, strong consistency is better suited, as it is both more intuitive for the users and does not require human intervention in case of conflicts.

Enforcing strong consistency at large scale has a significant impact on the latency of the file system, as replicas from different regions need to coordinate and agree on the order in which to execute commands. As a consequence, performance decreases as the number of regions spanned by the file system increases.

**How state partitioning can help.** Partitioning the state of the file system allows us to exploit workload locality without compromising consistency. We propose a partitioning model in which files are placed according to access patterns: e.g., in the same region as their most frequent users, as well as four execution modes corresponding to the operations that can be performed in the file system: (1) single-partition operations, (2) multi-partition uncoordinated operations, (3) multi-partition coordinated operations, and (4) read-only operations. While single-partition and read-only operations can be implemented efficiently by accessing a single region, the other two operations require interactions across multiple regions.

As the workload patterns may change, we further explore the idea of performing the state

partitioning on the fly. Workload-aware state partitioning introduces new challenges, which we address in this work:

- How to apply partitioning to the semantics of the POSIX file APIs?
- How to build and integrate within GlobalFS a dynamic partitioning component that moves data around based on client access patterns?
- How to build a low-latency data store?

The rest of this chapter begins by defining the system model. We first present GlobalFS, a distributed file system using a novel design that supports partitioning; we explain how the file system tree can be partitioned and replicated across regions. Next, we describe how workload-aware state partitioning works in the context of GlobalFS. In the evaluation, we show that GlobalFS compares favorably to de-facto industry implementations of distributed file systems, and that adding support for workload-aware partitioning in GlobalFS has positive effects on performance when the workload exhibits access locality.

This chapter expands on previously published work [99], with a focus on the dynamic partitioning functionality and the low-latency data store, which are my main contributions to this project.

## 4.2 System model and definitions

**System model.** We assume a distributed system composed of interconnected processes that communicate by message passing. There is an unbounded set of *client processes* and a bounded set of *server processes*. Processes may fail by crashing, but do not experience arbitrary behavior (i.e., no Byzantine failures).

Client and server processes are deployed at *datacenters* distributed over different geographic *regions*. Processes located in the same region communicate with low latency, while processes residing at different regions experience larger latency. Links are seemingly reliable: if both the sending and the receiving processes are non-faulty, then every message sent is eventually received.

The system is *partially synchronous* [100]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *global stabilization time* (GST) and is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and operations to be executed. After GST, such bounds exist but are unknown. In practice, "forever" means long enough for the atomic multicast protocol to make progress (i.e., deliver messages).

**Definitions.** We define two consistency models that are central to this work:

*Sequential consistency.* A system is *sequentially consistent* if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands as defined in their sequential specification, and (ii) respects the ordering of commands as defined by each client [101].

*Causal consistency.* A system is *causally consistent* if the result of read operations respect the causal ordering of events as defined by the "happens-before" relation [14].

## 4.3 GlobalFS: a partitioned file system

This section presents GlobalFS, a distributed file system that uses state partitioning to achieve geographical scalability while providing strong consistency guarantees. We designed GlobalFS to be POSIX-compliant, i.e., it supports atomic operations without enforcing close-to-open semantics. Specifically, GlobalFS guarantees *sequential consistency* for update operations and *causal consistency* for read operations.

Designing strongly consistent distributed systems that provide good performance requires careful trade-offs. In more details, ensuring strong consistency at scale is slow since it orders commands across replicas, potentially residing at geographically distant regions. State partitioning can help in this scenario by limiting the amount of coordination to involve replicas within the same region in most cases. The original approach we explore in this work is to trade the performance of global operations, spanning multiple regions, for the scalability of intra-region operations.

Next, we describe the overall architecture of GlobalFS and how the file system can be partitioned and replicated across datacenters. We then explain GlobalFS execution modes, as well as failure handling.

### 4.3.1 Architecture

GlobalFS consists of five components, depicted in Figure 4.1: client interface, partitioning oracle, data store, metadata management, and atomic multicast.

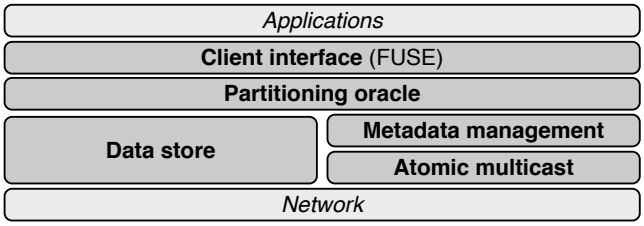



Figure 4.1: Overall architecture of GlobalFS.

**Client interface.** The client interface provides a file system API supporting a subset of POSIX 1-2001 [102]. GlobalFS implements file system operations sufficient to manipulate files and directories. It supports file-specific operations: `mknod`, `unlink`, `open`, `read`, `write`, `truncate`, `symlink`, `readlink`; directory-specific operations: `mkdir`, `rmdir`, `opendir`, `readdir`; and general-purpose operations: `stat`, `chmod`, `chown`, `rename`, and `utime`. It supports symbolic links, but not hard links.

**Partitioning oracle.** The partitioning oracle governs how files are assigned to partitions: files are either *local* – i.e., assigned to the partition that the oracle resides in, or *global* – i.e., shared among all partitions. File system clients contact their closest oracle (known in advance or discovered dynamically) to query it regarding the partitions where files of interest reside. The oracle takes into account client access patterns and a partitioning policy predefined by the system administrator to decide when to move files between partitions.

**Data store.** Clients divide the content of files in data blocks (`dblocks`) and store them in the data store. The data store provides a linearizable key-value store with primitives to read (`get`) and create (`put`) data items. It is implemented as a collection of distributed hash tables (DHTs), with one instance of the data store per datacenter. Maintenance of the data in the DHT is simple and efficient given the assumption that data blocks are immutable and block ids are unique: there is no need to enforce write consistency between replicas. Horizontal scalability is achieved by mapping keys to different sets of replicas inside the DHT [8, 7, 103, 104].

**Metadata management.** Like most contemporary distributed file systems (e.g., [43, 46, 55, 59]), GlobalFS decouples metadata from data storage. Metadata is handled by the metadata management layer. Each file has an associated *inode* block (`iblock`) containing the file's metadata (e.g., its size, owner, and access rights) and pointers to its data blocks; `iblocks` are mutable and maintained by the metadata servers.

**Atomic multicast.** In GlobalFS, every update operation is ordered by atomic multicast [28]. Atomic multicast is a one-to-many communication abstraction that implements the notion of *groups*. Servers subscribe to one or more groups and every message multicast to a group $g$ will be delivered by processes that subscribe to $g$. Let relation $<$ be defined such that $m < m'$ iff there is a process that delivers message $m$ before message $m'$. Atomic multicast ensures that (i) if a process delivers $m$, then all non-faulty processes that subscribe to the same group deliver $m$ *(agreement)*; and (ii) relation $<$ is acyclic *(order)*. The (partial) order property implies that if processes $p$ and $q$ deliver messages $m$ and $m'$, then they deliver them in the same order.

It is important to understand the difference between atomic broadcast, as implemented by Paxos [44] and its variants (e.g., [19, 105, 106]), and atomic multicast. With atomic broadcast, for every pair of delivered messages $m$ and $m'$, either $m < m'$ or $m' < m$. With atomic multicast,

it is possible that neither $m < m'$ nor $m' < m$. This is the case, for example, if $m$ and $m'$ are multicast to groups $g$ and $g'$, respectively, and no process subscribes to both groups. Partially ordering messages, as defined by atomic multicast, is a fundamental requirement of scalable distributed systems.

### 4.3.2 Partitioning and replication

GlobalFS partitions the file system tree among regions according to client access patterns. The partitioning oracle assigns files that exhibit locality in their access patterns to local partitions, replicated at datacenters inside the region that those respective files are being accessed from the most. Files that are read-mostly or frequently accessed from multiple partitions are assigned to a global partition that is replicated across regions.

In this setup, a file in the global partition can be read from any region, resulting in high throughput and low latency for read operations. Updating a file in the global partition, however, involves all regions. Local partitions, on the other hand, can provide high throughput and low latency for both reads and updates, as long as the client is close to its location. Both local and global partitions can tolerate the failure of an entire datacenter. Moreover, the global partition can tolerate the failure of all datacenters in a region (i.e., a disaster). Table 4.1 summarizes the two partition types in GlobalFS.

| **Partition** | **Replication** | **Performance** | **Fault tolerance** |
|---|---|---|---|
| Global | across regions | best for reads | disaster |
| Local | within region | best for reads & writes | datacenter crash |

Table 4.1: Types of partitions in GlobalFS.

When access patterns change, the file system needs to adapt by dynamically performing state partitioning. Specifically, the partitioning oracle needs to (i) identify changes in access patterns and (ii) move files around without breaking the consistency guarantees of GlobalFS. We address dynamic state partitioning in Section 4.4.

### 4.3.3 Example deployment

Figure 4.2 illustrates how state partitioning applies to GlobalFS. We consider a deployment involving three regions: $R_1$, $R_2$, and $R_3$, each with three datacenters. GlobalFS is split in four partitions, $P_0, ..., P_3$, such that $P_0$ is replicated in datacenters across all regions (i.e., global partition), while each partition $P_i$, $1 \leq i \leq 3$ is replicated in datacenters from region $R_i$ (i.e., local partitions).

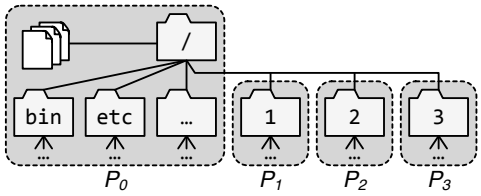



Figure 4.2: Partitioning in GlobalFS: $P_0$ is replicated in all regions, while $P_1$, $P_2$, $P_3$ are each replicated in one region.

In this scenario, clients and servers (metadata and data store) are distributed across the regions. In addition to the metadata associated with the region's partition, each datacenter also hosts an instance of the data store. More precisely, the metadata for the directory `/1` and all its contents (recursively) are stored only in $P_1$. In the same manner, `/2` and `/3` are respectively mapped to $P_2$ and $P_3$. Files not contained in any of these directories (e.g., `/`, `/bin`, `/etc`) are in partition $P_0$.

Each region has its own partitioning oracle, responsible for providing hints regarding the location of files to clients from that region. For instance, if a client from the region corresponding to $P_1$ wants to access $/bin$, it queries the oracle from the same region about the partitions that $/bin$ resides at. In response, the oracle indicates that $/bin$ is shared with other partitions, and thus, resides at $P_0$ – which is replicated across regions. Hence, the client accesses $/bin$ locally.

### 4.3.4 Execution modes

GlobalFS divides file system operations into four classes and defines an execution mode for each. These execution modes provide the basis for the implementation of each file system operation. They define how the client interacts with the different partitions of GlobalFS in order to successfully execute an operation. We next detail each GlobalFS execution mode without accounting for the case in which the partitioning oracle performs dynamic state partitioning; adapting to workload patterns is discussed in the Section 4.4.

a) *Single-partition operations:* A single-partition operation modifies metadata stored in a single partition. As a consequence, operations in this class are multicast to the group associated with the concerned partition and, when delivered, executed locally by the replicas. The execution of a single-partition operation follows state-machine replication [107]: each replica delivers a command and executes it deterministically.

   The following operations are single-partition in GlobalFS, where the terms *child* and *parent* are used to refer to a node and the directory that contains it.

   - `chmod`, `chown`, `truncate`, `open`, and `write`;
   - `mknod`, `unlink`, `symlink`, and `mkdir` when the parent and child are in the same partition; and

- `rename`, when the origin, origin's parent, destination, and destination's parent are in the same partition.

Note that while a single-partition operation in a local partition involves only servers in one region, a single-partition operation in the global partition (multicast to group $g_{all}$) involves servers in all regions of the system.

b) *Uncoordinated multi-partition operations:* An uncoordinated multi-partition operation accesses metadata in more than one partition, but the operation's execution at each partition can complete without any input from the other partitions involved. The partial ordering of atomic multicast is sufficient to guarantee consistency: partitions will independently reach the same decision in regards to success or failure. This is similar to the notions of *independent transactions* in Granola [108] or *one-shot transactions* in H-Store [109].

To execute an operation that concerns multiple partitions $P_1, P_2, ..., P_n$, the operation is atomically multicast to all replicas of all involved partitions. Upon delivery, each replica $P_i$ executes the operation. To reach replicas in multiple partitions, the operation is multicast to group $g_{all}$; if a replica delivers an operation it is not concerned about, the replica just discards the operation.

The following file system commands are implemented as uncoordinated multi-partition operations:

- `mknod`, `unlink`, `symlink`, `mkdir`, `rmdir` when the parent and child are in different partitions.

c) *Coordinated multi-partition operations:* Some operations require partitions to exchange information. In GlobalFS, this may happen in the case of a `rename` (i.e., moving the location of a file or directory). In this case, file metadata has to be moved from the origin's partition to the destination's partition. As a result, a rename may involve up to four partitions, given by the placement of the origin, origin's parent, destination, and destination's parent. Consequently, a `rename` operation might fail in one of the partitions (e.g., origin does not exist) but not in the other.

To execute a coordinated multi-partition operation, the client multicasts the operation to all concerned partitions (i.e., multicast group $g_{all}$). Upon delivery of the operation, the involved partitions exchange information about the command and whether it can or cannot be locally executed. This mechanism is similar to the signals exchanged in S-SMR [110]. In the case of a `rename`, the file's attributes and list of block identifiers need to be sent to the destination partition. As in classic two-phase commit protocols, the command is only executed if all involved partitions agree that it can be executed.

d) *Read-only operations:* Read-only operations are executed by a single metadata replica and data store server.[1] For read-only operations, GlobalFS provides causal consistency.

[1] GlobalFS does not implement *atime* (i.e., time of last access), as recording the time of the last access would essentially turn every read into a write operation to update the file's access time.

This is not straightforward to ensure since a client may submit a write operation against a server and later issue a read operation against a different server or even read from two separate servers. When the second server is contacted, it may not have applied the required updates yet.

We use an approach inspired by vector clocks [111, 112] where clients and replicas keep a vector of counters, with one counter per system partition. In the example shown in Figure 4.2, clients and replicas keep a vector with four entries, associated with partitions $P_0, ..., P_4$. Every request sent by a client contains $v_c$, the client's current vector, and each reply from a replica includes the replica's vector, $v_r$. A read is executed by a replica only when $v[i]_r \geq v[i]_c$, $i$ being the object's partition. The idea is that the replica knows whether it is running late, in which case it must wait to catch up before executing the request.

When a replica receives an update operation from a client, the client's vector $v_c$ is atomically multicast together with the operation. Upon delivery of the command by a replica of $P_i$, entry $v[i]_r$ is incremented. Every other entry $j$ in the replica's vector is updated according to the delivered $v_c$, whenever $v[j]_c > v[j]_r$. Clients update their vector on every reply, updating $v[i]_c$ if $v[i]_r > v[i]_c$, for each entry $i$.

The following file system commands are implemented as read-only operations: `read`, `getdir`, `readlink`, `open` (read-only), and `stat`. Table 4.2 summarizes the performance characteristics of GlobalFS operations.

| **Operation** | **Partitions** | **Multicast** | **Performance** |
|---|---|---|---|
| Read-only | one | no multicast | 1st (best) |
| Single-partition | one | $g_{all}$ or $g_i$ | 2nd |
| Uncoordinated multi-partition | two or more | $g_{all}$ | 3rd |
| Coordinated multi-partition | two or more | $g_{all}$ | 4th (worst) |

Table 4.2: Operations in GlobalFS.

### 4.3.5 Failure handling

In this section, we discuss the fault tolerance guarantees of GlobalFS for each .

**Client.** Client failures during a `write` or a file delete operation can leave "dangling" data blocks inside the data store. Data blocks that have no metadata pointing to them are unreachable and can be removed from the data store via garbage collection.

**Metadata layer.** GlobalFS uses state machine replication to build a reliable metadata layer within each partition. Metadata replicas use atomic multicast to totally order client commands; if a replica executes a command, other replicas in the same group will also execute the command. GlobalFS uses Multi-Ring Paxos as its atomic multicast (described in more detail in Section 4.5.2); as long as one replica and a quorum of acceptors are available in each of the groups, the whole file system is available for writing and reading.

The recovery of a metadata replica is handled by installing a replica checkpoint and replaying missing commands [113]. Coordinated multi-partition commands require one extra step. For coordinated multi-partition commands, replicas in the involved partitions need to exchange information before deciding whether the command can execute or not. A recovering replica, upon replaying a coordinated multi-partition command, requests this information from replicas in the other partitions. To allow for this, whenever a replica sends information out regarding a coordinated command, it also stores this information locally.

**Data store.** Each key-value pair stored by the backend is replicated in $f+1$ storage nodes. This implies that up to $f$ storage nodes can fail concurrently without affecting data block availability. To tolerate disasters (e.g, failure of entire datacenters – potentially all datacenters from one region), GlobalFS can replicate data (and its corresponding metadata) in different regions.

**Partitioning oracle.** Similar to the metadata layer, the partitioning oracle relies on state machine replication within each partition. The recovery of a partitioning oracle is handled by using checkpointing and write-ahead logs. Write-ahead logs protect against deadlocks, which may occur if an oracle crashes during a move operation. As previously explained, the oracle locks the path corresponding to a file before attempting to move it. If the oracle crashes during a move, a deadlock occurs.

## 4.4 Workload-aware state partitioning in GlobalFS

In this section, we examine the *partitioning oracle*, a component that enables GlobalFS to perform state partitioning dynamically. The partitioning oracle has three main functions:

1. assigns files to partitions according to a predefined policy,
2. answers client queries regarding the location of files,
3. moves files between partitions in order to adapt to workload patterns.

The challenge in supporting dynamic state partitioning is maintaining the strong consistency guarantees provided by GlobalFS. We next explain how the partitioning oracle works within GlobalFS and how we address the aforementioned challenge.

### 4.4.1 Overview

Figure 4.3 shows how the partitioning oracle integrates within the overall architecture of GlobalFS. The partitioning oracle consists of multiple components: open files cache, requests monitor, partitioning policy, partitioning logic, and path-to-partitions map. These components interact with each other in order to partition and replicate the file system tree in accordance with client access patterns.

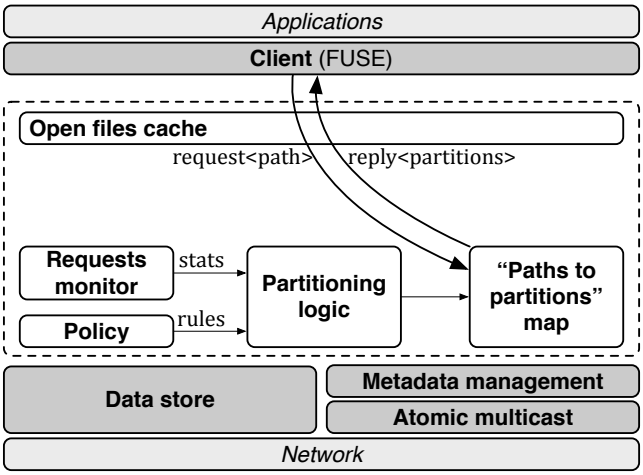



Figure 4.3: Integration of the partitioning oracle within GlobalFS.

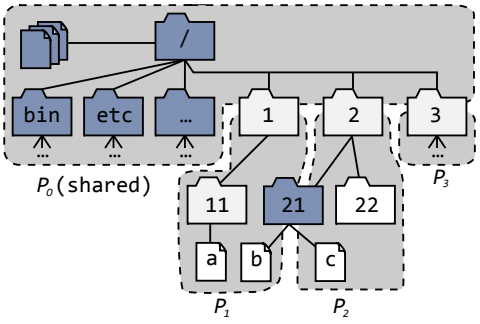
In order to access a file, a client needs to know the partitions responsible for storing it. The client contacts the partitioning oracle from the closest region and requests this information. The contacted oracle examines the path of the file and replies to the client with a *hint*, which includes: (1) the set of partitions responsible for the file, and (2) the set of partitions involved in updating the file. The latter is necessary because updating (or removing) a file also impacts the metadata of is parent directory, which might reside at different partitions. If the file of interest was moved, the oracle should be aware of the new partitions and, thus, communicates this information to the client.

The partitioning oracle decides when to move files between partitions based on client access patterns and a policy defined by the system administrator. Client access patterns within a region are inferred by a *requests monitor* component based on information received from clients regarding the number of `reads` and `writes` performed against each accessed file. The partitioning oracle's decision-making component (*partitioning logic* in Figure 4.3) takes into account the *policy* defined by the system administrator to determine where files should reside: e.g., if the `writes` performed on a file from a remote partition exceed a policy-predefined threshold, the oracle will initiate a move operation in order to bring that file in its partition.

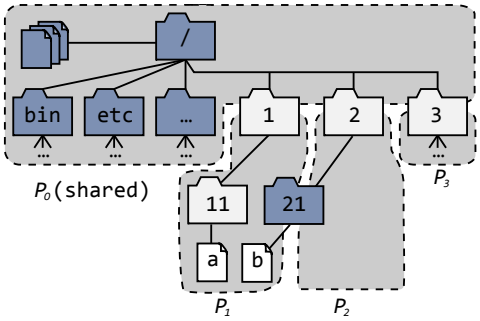
Each client maintains an *open files cache* in order to reduce the latency associated with querying a remote partitioning oracle. The open files cache keeps track of the oracle hints regarding the files that are currently open. Hence, a client only contacts the oracle (i) when opening a file or (ii) when an opened file has moved. In addition, each client remembers the *most recently accessed partition* and tries to access a file by contacting this partition first. Since most file system workloads exhibit access locality [114], this assumption saves bandwidth and decreases access latency.

### 4.4.2 Mapping files to partitions

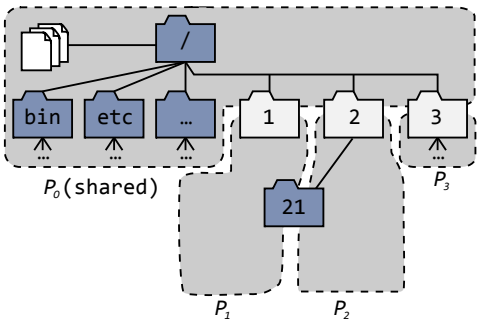
Figure 4.4 provides insight into how the partitioning oracle assigns files to partitions. We consider the deployment from Section 4.3.3, where GlobalFS spans three regions and is split in four partitions: $P_0$ is replicated in all regions, while each $P_i$ replicated in one region $R_i$.




(a) Cumulative view of partitioning oracles.




(b) Perspective of oracle from $P_1$.




(c) Perspective of oracle from $P_3$.

Figure 4.4: Assigning files to partitions in GlobalFS.

Figure 4.4(a) shows how the GlobalFS state (i.e., the file system tree) is divided among partitions. Shaded files are replicated across all regions (i.e., belong to $P_0$) and clear files are replicated within a single partition $P_i$, $1 \leq i \leq 3$. Examples of shared files include: (i) the root directory (/), (ii) system directories such as $/bin$ and $/etc$, which are typically read from all regions but rarely written, and (iii) directory /2/21 which has a child $/2/21/b$ in $P_1$ and another child $/2/21/c$ in $P_2$.

The partitioning oracle from a region assigns files to partitions by examining their paths. The main challenge is to differentiate between whether the file is: (i) new to the file system overall, or (ii) new to the partition that the oracle resides in. The partitioning oracle addresses this challenge by looking up the partitions corresponding to each of the file's *parent* directories. For example, in Figure 4.4(b), the oracle local to $P_1$ assigns file $/1/11/a$ to partition $P_1$, as its parent directory /1/11 is known to belong to $P_1$. On the other hand, the same oracle – local to $P_1$, marks file /2/21 as *shared*, since it does not know its parent directory /2, and only

knows that the root directory / is shared. The shared directory 2/21 is also visible to the oracle deployed at $P_3$, as shown in Figure 4.4(c).

For performance reasons, the partitioning oracle in each region only keeps track of a subset of the entire file system tree, focusing on the files assigned to its own partition. If an oracle receives a request that does not map to its own region, the oracle forwards the request to the appropriate region. In order to avoid broadcasting the request and putting strain on the entire system for each such “miss", we use the invariant that any updates to “fringe nodes", i.e., to nodes where the set of partitions changes from that of the parent’s, are submitted to inter-regions atomic multicast. This means that whenever a fringe node is created or deleted, this information reaches all regions and all oracles. The oracles store these files in their view of the file system. For example, Figure 4.4(c) shows that partition $P_3$ is aware of /2/21, because it represents a fringe node that has children both in partitions $P_1$ and $P_2$.

The way oracle assigns files to partitions determines the performance of `update` operations. If a file belongs exclusively to a partition (e.g., file /1/11/$a$ belongs to $P_1$), `updates` are fast since they rely on the instance of atomic multicast running in partition $P_1$. A file that is shared among partitions leads to slow `updates`, as it requires using an atomic multicast instance spanning all regions. For example, creating or removing files from directory /2/21 is slow, as it modifies the metadata of a shared directory.

### 4.4.3 Moving files

The partitioning oracle decides when to move a file into its local partition based on (i) the partitioning policy predefined by the system administrator, and (ii) workload patterns observed by its requests monitor. First, the partitioning policy provides thresholds for when files should be moved. Second, the region manager uses file access information from clients in its region to infer workload patterns; file access information includes the number of `reads` and `updates` that clients have performed and against which files.

Before opening a file, a client contacts the oracle in order to request the partition it needs to access. At this point, the oracle decides whether to move that file to its local partition or not. If it decides to move the file, the oracle will first perform the move, then reply to the client with the new partition.

The decision of whether a file should be moved is local; the oracle relies exclusively on the partitioning policy and on information from the requests monitor, and does not exchange information with oracles residing at different partitions. Moving a file to the local partition consists in four steps, during which the oracle takes the role of a specialized client from the point of view of GlobalFS and uses the global atomic multicast instance to update metadata replicas across regions:

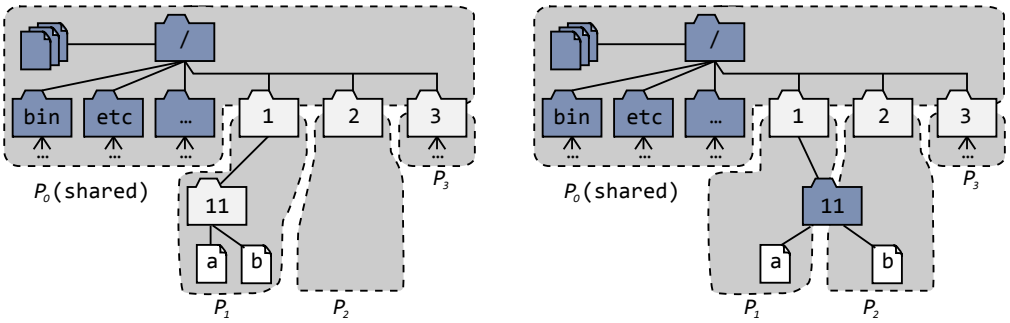
1. The oracle locks the path corresponding to the file of interest on all partitions; it does so

with the help of the global atomic multicast instance that runs across regions. The lock prevents destructive operations on the node or any of its direct descendants.

2. The oracle contacts the closest partition that owns the file in order to retrieve its metadata, as well as the metadata corresponding to its parent directory. Retrieving this information enables metadata replicas in the local region to update their own view of the file system hierarchy to reflect the move. For instance, in Figure 4.5(a), to move file $/1/11/b$ from $P_1$ to $P_2$, the oracle in $P_2$ needs to retrieve metadata corresponding to $/1/11/b$ and its parent directory $/1/11$ from partition $P_1$; hence, directory $1/11$ is shared – as shown in Figure 4.5(b). In addition, the oracle retrieves data blocks corresponding to $/1/11/b$ from the backend data store of $P_1$.

3. The oracle updates the metadata replicas across all involved regions to reflect the move operation. In our example, metadata replicas from $P_1$ write the retrieved metadata for $/1/11$ and $/1/11/b$, while metadata replicas from $P_2$ remove the metadata corresponding to the moved file $/1/11/b$ and mark directory $1/11$ as shared. Finally, the oracle stores the data blocks corresponding to the moved file $/1/11/b$ in the data store from $P_1$.

4. The oracle unlocks the path corresponding to the moved file, meaning that the file can now be modified or moved to a new partition if necessary.




(a) File system tree before move.

(b) File system tree after move.

Figure 4.5: Moving file $/1/11/b$ from $P_1$ to $P_2$.

Once a file has been moved from a source partition (e.g., $P_2$) to another destination partition (e.g., $P_1$), the metadata replicas in the source partition need to remember that the file no longer belongs to their partition. This information is necessary in order to properly reply to clients trying to access the node in the future (due to the clients having stale partitioning information, either due to concurrency or client-side caching of partitioning information). The oracle needs to be able to distinguish between the case where the file is not in the node's metadata because it is inexistent and that in which the file existed on the partition in the past but has since been moved.

For example, a client mistakenly trying to access the moved file /1/11/$b$ by contacting partition $P_1$ should not receive a "File not found" response from the oracle, but rather get redirected to $P_2$. We prevent this situation by having the metadata replicas keep track of the files that they have "lost" due to move operations. This enables any replica contacted by a client regarding file /1/11/$b$ to return an appropriate exception, indicating that the file has been moved to a different partition.

## 4.5 Implementation

In this section, we discuss the implementation details of the main components of GlobalFS. Figure 4.6 depicts a GlobalFS deployment that spans three regions, each hosting a partitioning oracle, metadata replicas, and data storage servers. The *local* and *global rings* represent multicast groups used to replicated metadata within and across regions.

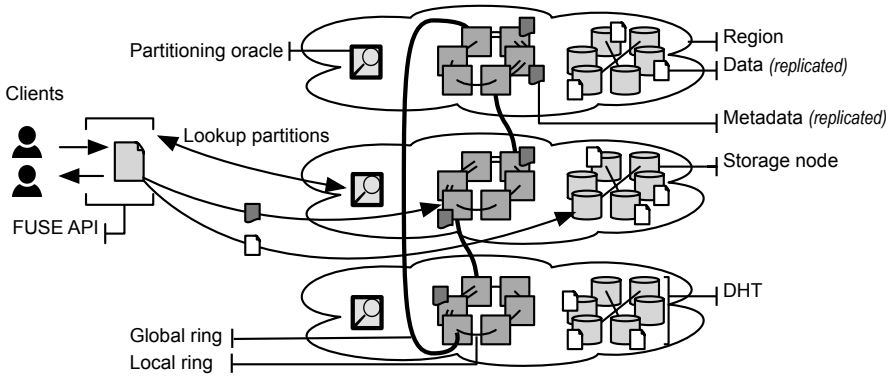



Figure 4.6: GlobalFS deployment spanning three regions.

### 4.5.1 Client

Clients access files through a *file system in user space* (FUSE) implementation [115]. FUSE is a loadable kernel module that provides a file system API to user space programs, allowing non-privileged users to create and mount a file system without writing kernel code. According to [116], FUSE is a viable option in terms of performance for implementing distributed file systems.

Clients locate local partitioning oracle in their closest partition and metadata replicas using ZooKeeper [12].

When using FUSE, every system call directed at the file system is translated to one or more callbacks to the client implementation. In GlobalFS, most FUSE callbacks have an equivalent RPC (remote procedure call) available in the metadata servers. A client discovers to which metadata replica or data store it needs to direct a given operation by (1) using its open files local cache, (2) optimistically contacting the most recently accessed partition, or (3) contacting

the closest partitioning oracle. When a client has the option of sending a command to more than one destination (metadata replica or data store), it chooses the closest one (with the lowest latency).

In most non-synthetic workloads, the most frequently used FUSE callback is `getattr`; this is due to the semantics imposed by the POSIX standard [102]. To reduce the execution time of `getattr`, each client remembers *the most recently accessed partition*. When accessing a file that is not in the open files cache, the client first optimistically contacts a metadata replica from the most recently accessed partition. If the workload exhibits locality, this approach saves a communication round-trip with the partitioning oracle.

### 4.5.2 Atomic multicast

We use URingPaxos [117], a unicast implementation of Multi-Ring Paxos [28], which implements atomic multicast by composing multiple instances of Paxos to provide scalable performance. Each multicast group is mapped to one Paxos instance. A message is multicast to one group only. Processes that subscribe to multiple groups use a deterministic merge procedure to define the delivery order of the messages such that processes deliver common messages in the same relative order. Processes in each Paxos instance are disposed in a logical directed ring in order to achieve high throughput.

In our setup, we keep a global ring that includes all metadata replicas in the system, as illustrated in Figure 4.6. This ring implements the $g_{all}$ group discussed in Section 4.3.4. Each other group is implemented by a ring that includes replicas in the same region.

### 4.5.3 Metadata replicas

Metadata in GlobalFS is kept by replicated servers, using state machine replication [107]. Replicas can be part of multiple multicast groups; in our prototype, each replica is a Multi-Ring Paxos learner. When a replica delivers a command, the replica checks whether it should execute the command by using the partitioning information, which piggybacks on the command. The file system metadata is kept in-memory by the replica and the sequence of commands is stored by Multi-Ring Paxos acceptors. Replicas can be configured to keep their state in memory or on disk, with asynchronous or synchronous disk writes.

The file system is represented as a tree of nodes. There are three node types: directory, file, and symbolic link. A directory node stores the directory properties (e.g., owner, permissions, times) and a hash table of its children nodes, stored by name. A file node keeps the file properties and a list of blocks representing its contents. Symbolic link nodes only need to store the node properties and the target path of the link.

The metadata replicas are implemented in Java and expose a remote interface to the clients via Thrift [118].

### 4.5.4 Data store

GlobalFS is designed to support any back-end data store that exposes a typical key-value store API and provides linearizability. We have implemented several versions of the data store.

**Splay-based data store.** We implemented the data store using Splay [119], a framework that facilitates the design, deployment and testing of large-scale distributed applications in Lua 5.1. In Splay, applications are run by a set of daemons distributed on one or several testbeds; daemons execute in a sandboxed environment that shields the host system. A controller manages applications, offering multi-criterion resource selection, deployment control, and churn management by reproducing the system's dynamics from traces or synthetic descriptions.

The data store is organized as a ring-based distributed hash table (DHT) and uses consistent hashing for data placement. We implemented two DHT versions, which differ in terms of the distributed lookup protocol used to locate the node that stores a particular data item: (i) Chord [120]-based DHT – each storage server maintains a partial membership of other servers on the ring, leading to multi-hop lookups, and (ii) one-hop DHT – each storage server maintains a full membership of other servers on the ring, allowing one-hop lookups. The one-hop design, similar to Cassandra [8] or Dynamo [7], provides good horizontal scalability and stable performance. Depending on the application requirements and fault model, data may be stored persistently on disk or maintained in memory.

Each block is assigned to the first server whose logical identifier follows the block identifier on the ring. A block is replicated as $r$ copies, by copying it onto the $r-1$ successors (i.e., servers that immediately follow the first server on the ring). This ensures data availability with up to $r-1$ simultaneous failures. Servers periodically check for the availability of copies of their blocks onto their successors and create additional copies when necessary. Similarly, servers periodically check for their predecessor availability and take over the responsibility for their ranges upon failure, also creating additional copies. We note that the blocks stored in the DHT are only written once. Clients contact the DHT via any of its proxy servers. The proxy creates the r copies of the block using the slower link from the client to send the block only once.

We added several performance optimizations to our Splay-based DHT:

- *Separate data and control.* We use separate channels for data (i.e., file content) and control messages (i.e., command, key, data size). Control messages are sent via Splay RPCs, while data is sent via raw TCP sockets.
- *In-kernel data transfers.* Our data store uses in-kernel data transfers via the *splice* system call, which we implemented as C modules. We support multiple types of in-kernel data transfers: (1) file to socket transfers – when the proxy gets a data block from the client, (2) socket to socket transfers – when the proxy passes a data block on to a storage node, and (3) socket to file transfers – when a storage node writes the data block in a persistent

manner. In-kernel data transfers are problematic, as Lua 5.1 does not support yielding the thread from C and thus, transfers cannot be interleaved.

- *Reusing TCP connections.* Having a pool of reusable TCP connections among storage nodes avoids the cost of establishing a TCP connection for each transfer.

**LevelDB-based data store.** A second version of our data store – described in [99], is implemented in Go and uses LevelDB [121] as its storage backend. Since unlike Lua 5.1's, the Go runtime environment supports true multithreading, we used this version in our experimental evaluation.

### 4.5.5 Partitioning oracle

Each region hosts a partitioning oracle that keeps track of (i) files assigned to that region, (ii) files shared with other regions, and (iii) files "lost" due to being moved to a different region. The partitioning oracle exposes an API that allows the client to request the partitions corresponding to a file path and periodically send information on its access patterns. The communication between client and oracle is reliable – via buffered TCP connections. Since the client is multi-threaded, in maintains a pool of connections to its closest partitioning oracle. In order to contact the oracle, the client requests a connection from the connection pool; after receiving a reply from the oracle, the client releases the connection back into the pool.

## 4.6 Evaluation

In the initial version of this work (Pacheco et al. [99]), we evaluated GlobalFS using a static partitioning function (hardcoded in the clients and metadata replicas). Static partitioning requires knowing access patterns in advance; otherwise, the function cannot provide a good partitioning of the file system tree, leading to high access latency. With an appropriate partitioning function, GlobalFS delivers good performance and scales well for all local operations.

In this thesis, we conduct an experimental evaluation with real-world applications. First, we show how GlobalFS compares against three widely used distributed file systems: NFS [65], GlusterFS [51], and CephFS [39]. Next, we evaluate the impact of adding support for workload-aware partitioning on the performance of GlobalFS.

### 4.6.1 Setup

We evaluate GlobalFS using Amazon's EC2 platform. We deploy virtual machines (VMs) in multiple EC2 regions. For each region, we distribute servers and clients in three separate availability zones to tolerate datacenter failures. We used `r3.large` and `c3.large` instance types, with 2 virtual CPUs, 32 GB SSD storage, and respectively 15.25 and 3.75 GiB memory [122]. We

use `r3.large` instances for servers and `c3.large` instances for clients.

When not using workload-aware partitioning, GlobalFS uses a hardcoded partitioning function which assumes that access patterns are known in advance and do not change. Clients uses the partitioning function to compute to which metadata replica or data store to direct a given operation. In turn, when a replica delivers a command, it checks whether it should execute the command by using the partitioning function.

We configure the atomic multicast layer based on Multi-Ring Paxos to use in-memory storage. In this setup, fault tolerance relies on having a majority of replicas that are always alive (i.e., never crashes). We use the data store implementation described in [99], as it has been proven to sustain enough throughput not to constitute a bottleneck in our GlobalFS evaluation.

### 4.6.2 Comparison against de-facto industry implementations

We consider a GlobalFS deployment that spans nine Amazon EC2 regions and evaluate the performance using real-world workloads when executed on global and local partitions. We compare the results against three widely-used distributed file systems: NFS (v4.1), GlusterFS (v3.7), and CephFS (v0.94). Our objective is to assess that, while providing stronger guarantees, GlobalFS compares favorably to de-facto industry implementations.

We configure NFS with one single shared directory mounted remotely by the same clients. The NFS server runs in the `us-west-2` region. We disable all caching features on GlobalFS, so the NFS clients mount the remote directory using the `lookupcache=none,noac,sync` option.

We use FUSE-based bindings for GlobalFS, GlusterFS, and CephFS. We chose the compilation of two well-known open-source projects as workload: the `bc` numeric processing language (v1.06) and the Apache `httpd` web-server (v2.4.12). These two projects differ in size of the compressed archives (278 kB and 6 MB), number of shipped files (94 and 2,452), and lines of ansi-C code to compile (8,510 and 157,575). They expose different workloads to the underlying file system and are often used as benchmarks [123].

Table 4.3 embeds the operations breakdown of the system calls issued by the different commands (decompress, configure, and compile) used for these experiments. We evaluate GlobalFS either within a global or a local partition and compute the average over 3 distinct executions. All file systems are mounted by 9 clients spread equally across 3 regions, but the workload is executed on a single client. We use equivalent settings for GlusterFS and CephFS. (Note that GlusterFS experiments over the global partition are executed only once due to AWS budget constraints.) For NFS, all clients mount a shared directory, and a client co-located with the service executes the commands. For GlusterFS we evaluate two different deployments, local (one region) and global (three regions). Each deployment consists of a distributed/replicated volume on top of regular storage bricks, one on each of the availability zones for the given EC2 regions. We deployed CephFS only at a single region (3 storage daemons, 1 metadata server, and 3 clients) because a deployment across regions would require forfeiting strong

consistency [42]. We set the replication factor of GlusterFS, and CephFS to 3.




| Command | Operations breakdown | NFS | GlobalFS *global* | GlobalFS *local* | GlusterFS *global** | GlusterFS *local* | CephFS *local* |
|---|---|---|---|---|---|---|---|
| `tar xzvf bc-1.06.tgz` | | 1.94 s | 47.09 × | 1.36 × | 149.05 × | 1.63 × | 0.17 × |
| `configure` | | 5.32 s | 44.66 × | 2.02 × | 45.67 × | 0.96 × | 0.56 × |
| `make -j 10` | | 5.9 s | 29.90 × | 2.38 × | 49.34 × | 1.17 × | 0.63 × |
| `make` | *(same as above)* | 13.14 s | 20.73 × | 1.16 × | 55.20 × | 0.92 × | 0.30 × |
| `gzip -d httpd-2.4.12.tgz` | | 3.87 s | 117.12 × | 2.47 × | 284.75 × | 0.37 × | 0.11 × |
| `tar xvf httpd-2.4.12.tar` | | 60.01 s | 41.46 × | 1.08 × | 99.17 × | 0.12 × | 0.14 × |
| `configure -prefix=/tmp` | | 29.32 s | 49.35 × | 2.04 × | 56.53 × | 1.34 × | 0.33 × |
| `make -j 10` | | 714.37 s | 2.74 × | 0.52 × | 139.68 × | 0.87 × | 0.48 × |
| `make` | *(same as above)* | 3432.72 s | 1.82 × | 0.36 × | 83.72 × | 0.50 × | 0.64 × |

Table 4.3: Execution times for several real-world benchmarks on GlobalFS with operations executed over global and local partitions. Execution times are given in seconds for NFS, and as relative times w.r.t. NFS for GlobalFS , GlusterFS and CephFS. *Note that GlusterFS does not support deployments with both global and local partitions; thus, we report results from two separate deployments.

We observe that GlobalFS performs consistently better than GlusterFS when operating across regions. GlobalFS performs competitively against the other file systems across the whole suite of benchmarks. Indeed, GlobalFS is up to `50.9x` faster that GlusterFS in compiling Apache `httpd` over the global partition. Note that for the same benchmark on a local partition, GlobalFS is actually faster than NFS. When evaluating GlusterFS and CephFS we use their default, out-of-the-box configuration. Both are heavily optimized systems and some optimizations are on by default (e.g., clients in CephFS use write-back caching, which improves write performance by batching small writes). As expected, the performance penalty for accessing the global partition is higher for write-dominated workloads (extracting an archive, configuring the software package). For read-dominated or compute-intensive (make) operations, this overhead decreases because read operations can be completed locally. For comparison purposes, we also tested HDFS (v2.6) with FUSE bindings on a local partition with some of the benchmarks and observed performance in the order as GlobalFS and GlusterFS (e.g., `2.12x` slower for the first command as compared to `1.36x` and `1.63x`, respectively).

Our real-world benchmarks demonstrate that GlobalFS performs on par with widely adopted distributed file systems, it ensures a stronger consistency model, it supports replication, and allows users to benefit from locality thanks to its partitioning model.

### 4.6.3 Benefit of workload-aware partitioning

In this experiment, we deploy GlobalFS across three Amazon EC2 regions, each corresponding to a partition (`us-west-2`, `us-east-1`, and `eu-west-1`). Our workload consists of recursively searching through all Apache `httpd` source files using the `grep` utility. We mount the file system in clients in all three partitions, then copy the source code in the `us-west-2` partition. After this, all files will be local in that partition. Next, we run `grep` from a different partition and

measure total execution time with and without our workload-aware partitioning algorithm. We configure our oracle to use an *"eager" partitioning policy*: when it learns that a client reads a file residing at a remote partition, the oracle attempts right away to move the file to its local partition. Our goal is to show that adding support for workload-aware partitioning in GlobalFS has positive effects on performance.

Table 4.4 shows the time (in seconds) to execute the search operation from different regions, with workload-aware state partitioning enabled. First, we use a client from `us-west-2`; the search operation is local and, thus, fast (19s). Second, we use a client from `us-east-1`; this results in moving all files to the partition corresponding to `us-east-1` and an execution time 200x larger than in the previous case. If we repeat the second case, we notice that the execution time is much smaller (211s versus 3853s), since files have been already moved to `us-east-1`. However, the execution time is still 10x larger than in the first case because our partitioning oracle does not also move directories.

| Client location | Data location | Time | Move performed |
|---|---|---|---|
| `us-west-2` | `us-west-2` | 19s | No |
| `us-east-1` | `us-west-2` –> `us-east-1` | 1h 4m 13s | Yes |
| `us-east-1` | `us-east-1` | 3m 31s | No |

Table 4.4: Execution of <`grep -r "protocol" .`> with workload-aware state partitioning enabled.

We next disable workload-aware state partitioning and repeat the experiment. Table 4.5 shows that performing the search from a remote region (`us-east-1`) without moving files results in around 50 % lower latency. However, every subsequent operation from the remote region (`us-east-1`) is equally slow. This suggests that workload-aware state partitioning can be beneficial for the performance of GlobalFS for workloads that exhibit locality of access. Files that are frequently accessed from multiple partitions are better shared.

| Client location | Data location | Time | Move performed |
|---|---|---|---|
| `us-west-2` | `us-west-2` | 19s | No |
| `us-east-1` | `us-west-2` | 29m 4s | No |

Table 4.5: Execution of <`grep -r "protocol" .`> with workload-aware state partitioning disabled.

| Client location | Data location | Duration (seconds) | Move performed |
|---|---|---|---|
| `us-west-2` | `us-west-2` –> `us-east-1` | 7h 13m 20s | Yes |
| `us-east-1` | `us-east-1` | 2h 7m 9s | No |

Table 4.6: Building the Apache Web Server using <`make -j4`> with workload-aware state partitioning enabled.

| Client location | Data location | Duration (seconds) | Move performed |
|---|---|---|---|
| us-west-2 | us-west-2 | 22m 15s | No |
| us-east-1 | us-west-2 | 7h 48m 29s | No |

Table 4.7: Building the Apache Web Server using <make -j4> with workload-aware state partitioning enabled.

## 4.7 Summary

This chapter focused on workload-aware state partitioning in the context of GlobalFS, a geographically distributed file system that accommodates locality of access, scalable performance, and resiliency to failures without sacrificing strong consistency. GlobalFS builds on two abstractions: single-site linearizable data stores and atomic multicast; this modular design was crucial to handle the complexity of the development, testing, and assessment of GlobalFS. Our evaluation reveals that (1) GlobalFS outperforms other geographically distributed file systems that offer comparable guarantees, and (2) workload-aware state partitioning can be beneficial for the performance of GlobalFS for workloads that exhibit locality of access.

# 5 Cache Replacement Policy for Erasure-Coded Data

## 5.1 Introduction

This chapter focuses on large-scale storage systems that ensure high availability using erasure-coding instead of replication. Whereas replication implies storing full copies of data in several data centers, erasure-coding splits an object into $k$ data blocks (or "chunks"), computes $m$ redundant chunks using an encoding scheme, and stores these chunks at different physical locations. A client needs to retrieve any $k$ of the $k+m$ chunks in order to reconstruct the object. Erasure coding represents a middle ground between storing full copies of data and having no redundancy in the system. It requires less storage space, but also incurs access latency, as some of the $k$ necessary chunks may need to be retrieved from remote regions. This is a typical use case for caching.

The application of caching to erasure-coded data is not straightforward, as having objects split into blocks offers many caching configurations that traditional caching policies such as *Least Recently Used* (LRU) or *Least Frequently Used* (LFU) are unable to capture or exploit. More precisely, a caching mechanism for erasure-coded data should be able to decide:

1. *Which* objects to cache?
2. *How many blocks* to cache for each of the objects?

We propose Agar, a caching mechanism developed specifically for erasure-coded data, and investigate how it can help minimize read latency in storage systems that span many geographical regions and use erasure coding to increase data availability. The name Agar is a reference to an online game[1] in which each player controls a cell that aims to gain as much mass as possible at the expense of other cells. Similar to the game, in our system, objects deemed valuable are able to "expand" in terms of the number of data chunks cached, causing less valuable objects to "shrink" or even disappear from the cache. Agar explores the trade-off

[1] http://agar.io/

between the storage cost of locally caching chunks of an object and the expected latency improvement. We adapt a dynamic programming approach designed for the "Knapsack" problem and use it to optimize the cache configuration.

The remainder of this chapter begins by discussing the rationale of designing a dynamic caching system dedicated to erasure-coded data. Next, we describe the design of the Agar caching system and detail its cache configuration algorithm. We compare Agar against alternative caching systems using the classic LRU and LFU cache replacement policies.

The material in this chapter is adapted from previously published work [124].

## 5.2 A case for caches tailored to erasure-coded data

We use a practical example to highlight the interplay between erasure coding and caching policies. We notice that caching erasure-coded data is akin to the Knapsack problem and explain why a solution more complex than a greedy algorithm is needed.

### 5.2.1 Interplay between erasure coding and caching

Figure 5.1 shows an erasure-coded object store deployed on top of Amazon Web Services (AWS). Each region hosts an S3 `bucket` as persistent backend and a `memcached` server running on a large EC2 instance (2 vCPUs, 8 GiB RAM) as caching layer. We populate the backend with 300 × 1 MB objects, encoded using a Reed-Solomon scheme with $k = 9$ data chunks and $m = 3$ redundant chunks (the total storage size, including redundancy, is thus 400 MB). For each object, the resulting twelve chunks are distributed among regions in a round-robin manner, with each S3 bucket storing two. The cache in each region is large enough (500 MB) to accommodate our complete working set, in practice emulating an infinite cache. This means that all requests for a given object are cache hits, except for the first one.

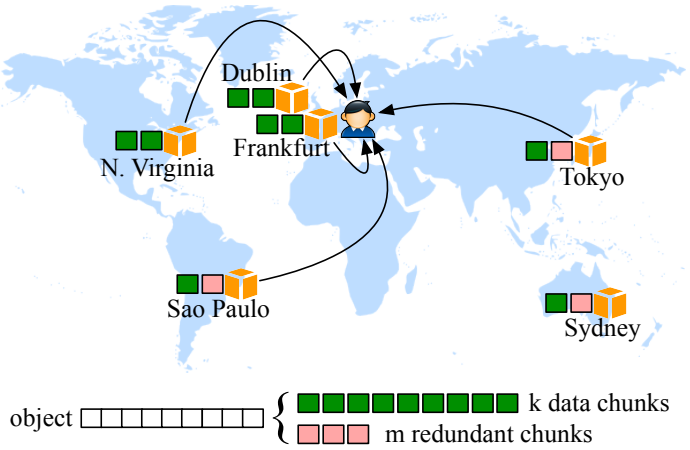



Figure 5.1: An erasure-coded storage system spanning six AWS regions.

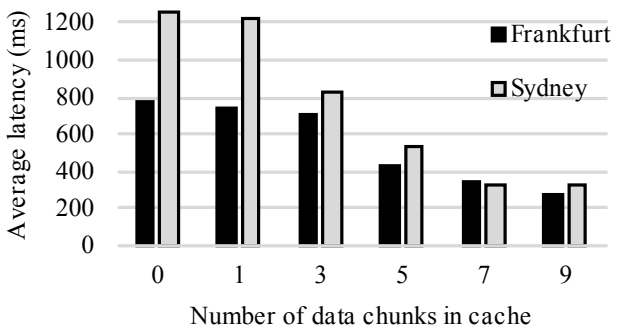



Figure 5.2: Average read latency when caching a variable number of chunks. The relationship between the number of chunks cached and the latency improvement obtained is non-linear.

Our experiment involves two clients: one in Frankfurt and one in Sydney. We use a customized version of the YCSB client [125] modified to support erasure-coding [126]. Each client performs 1,000 read operations on a pool of 300 × 1 MB objects. The read operations are generated from a Zipfian distribution with a skew exponent of 1.1. We measure the average latency experienced by both clients in 6 scenarios, in which we vary the number of data chunks $c$ retained in local cache instances for each object, from $c = 0$ to $c = k = 9$. Our goal is to better understand how the partial replication allowed by erasure coding impacts access latency.

The first scenario ($c = 0$) is an extreme baseline case that does not use the caching layer: clients read data chunks directly from the backend and decode them. The remaining scenarios store $c$ chunks of each retrieved object in the `memcached` instance of the region the request originates from, for $c \in \{1,3,5,7,9\}$. The most distant chunks are cached first for each object, in effect progressively decreasing in each experiment the number of AWS regions clients must access in order to reconstruct an object.

Figure 5.2 shows that the gains in latency are not proportional to the number of chunks retained in the cache:

- If only a few chunks are cached (e.g., up to 3 for Frankfurt), benefits remain minimal. This is because in this case, the overall latency is dominated by the last and slowest chunk that the client still needs to retrieve from a remote location.
- Conversely, once a critical mass of chunks has been cached (e.g. 7 chunks in the case of both clients), caching more chunks brings only minimal returns. This is because the latency incurred by the slowest block(s) is masked by the delay required to retrieve the closest block(s).

Where these turning points lay further depends on the position of the client, and the latency and bandwidth experienced between regions: Sydney can already greatly benefit from 3 locally

cached chunks, while this level of caching makes very little difference to Frankfurt.

Our findings carry important lessons on caching applied to erasure-coded storage, thus opening the way for advanced caching policies, such as the one proposed in this chapter (section 5.4). To sum up:

- The improvement in latency is not a linear function of the number of cached data blocks; caching more blocks is not necessarily going to make the system faster.
- Finding the optimal number of blocks to cache for each object is difficult to achieve beforehand, as it depends on many external factors (requests distribution, network state, object popularity) that change over time.

### 5.2.2 The Knapsack Problem

The problem of caching erasure-coded data can be interpreted as a variant of the century-old Knapsack problem [127], which seeks to maximize the value of a set of elements packed into a container of bounded capacity. In our scenario, the container is a local cache, elements are blocks, and the value of individual blocks corresponds to the overall latency improvement that local clients will perceive over the entirety of their requests if this block is cached, i.e., how much faster it will be for clients to retrieve the data item from the cache instead of the backend. The *weight* of a block is the space it occupies in the cache.

In the absence of erasure coding, choosing which data items to cache in a storage system is an instance of the 0/1 Knapsack Problem, where each element is unique (i.e., you cannot put more than one of each elements in the knapsack). Greedy algorithms do not generally work well for 0/1 Knapsack and can err by as much as 50% from the optimal value [128].

Nonetheless, by splitting data items into chunks, erasure coding greatly increases the complexity of the corresponding Knapsack problem. The problem might at first appear related to Fractional Knapsack [129], where it is possible to put a fraction of an element into the knapsack (and for which greedy algorithms yield optimal solutions). However, (i) *the non-linear dependency between fractions of value* (latency improvement) *and weight* (cached blocks) (as shown in §5.2.1), and (ii) *the finite choices of fractions* (i.e., options dictated by the choice of erasure coding parameters) make the problem closer to 0/1 Knapsack than to Fractional Knapsack, and introduce the need for a tailored algorithm to select which blocks to cache. In the following, we therefore propose to adapt a dynamic algorithm solution used for 0/1 Knapsack to the problem of caching erasure-coded blocks.

## 5.3 Design

This section introduces Agar, a caching system we specifically developed for erasure-coded data. We designed Agar around the use-case described in §5.2.1: an erasure-coded object

store deployed across several regions. Agar maintains independent caches in each region, along with components that implement a dynamic caching algorithm.

Unlike a caching eviction policy that decides which object to remove from the cache, Agar estimates the *popularity of individual objects*, as well as *potential latency gains* in order to *pre-compute* a static cache configuration that will be used during a fixed period; this period is a system parameter and depends on how rapidly access patterns are expected to change. Agar's design exploits three core assumptions:

- Access patterns vary across regions, so caches from different regions require different configurations, and do not require coordination.
- Access patterns vary over time, so we need to periodically recompute the configuration of each individual cache.
- Individual objects do not have to be read entirely from the backend or entirely from the cache. Thus, Agar supports partial caching, and can benefit from partial cache hits.

The goal of Agar is to find a good trade-off between the *number of chunks to cache* and *the overall latency improvement* for each object. Latency improvement depends on the position of the client relative to the servers that store the content of interest and the access trend in the nearest region. Similar to the LFU cache eviction policy, Agar requires statistics regarding object popularity and an estimation of the latency cost incurred when reading data chunks from individual regions.

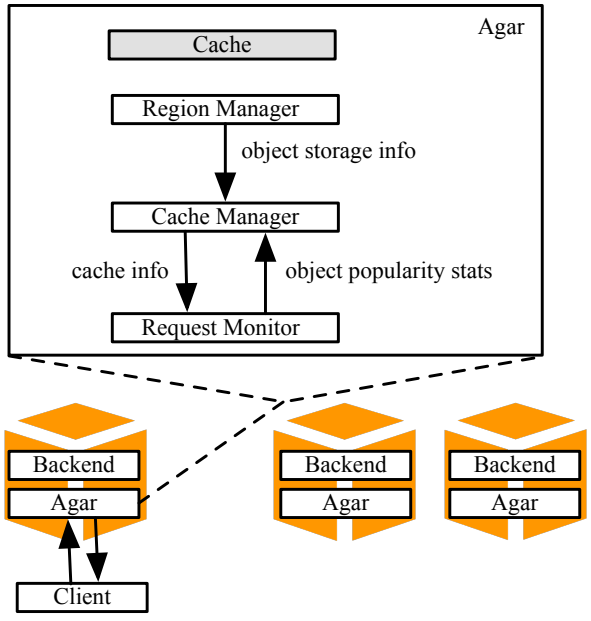



Figure 5.3: Design of Agar. We show how Agar integrates with a typical erasure-coded storage system and zoom in on the components of an Agar region-level deployment.

Figure 5.3 provides an overview of the components of Agar and shows how they fit within a typical erasure-coded storage system. We envision that Agar has nodes deployed within most of the regions that the storage system spans (represented as orange cubes in Figure 5.3). In our current design, Agar nodes from different regions do not collaborate with each other. On these nodes, Agar sits as a layer between clients and the region's backend instance, and contains four key components (zoomed-in frame in Figure 5.3):

1. *Region Manager* maintains a high-level overview of the storage system's topology, i.e., the regions that it spans and the policy it uses to distribute data chunks among regions. (We assume a round-robin distribution policy.) The region manager periodically measures how much it takes to read a data chunk from each region and uses this information to estimate the latency improvement that clients would get if blocks from that region were cached locally, thus removing the need to access that region.

2. *Request Monitor* listens for client requests and computes statistics regarding the popularity of each object over a predefined time interval (currently fixed). Agar uses an exponentially weighted moving average to keep track of popularity over time (more details in §5.4). Before a client reads an object, it contacts the request monitor asking for hints regarding where to get the data chunks necessary to reconstruct that object. The request monitor forwards such requests to the cache manager and updates the statistics on object popularity.

   In our current design, the request monitor is involved in each operation that clients perform. This did not increase latency in our experiments because Agar resides geographically close to the clients. For large deployments, we believe that techniques like TinyLFU's [78] approximate access statistics can avoid the request monitor becoming a bottleneck, while maintaining similar effectiveness.

3. *Cache Manager* periodically computes the ideal cache configuration (i.e., what objects to cache and how many chunks for each) based on object popularity statistics from the requests monitor and information about the system's backend deployment from the region manager. It provides hints regarding what data chunks should reside in the cache to the requests monitor, which forwards them to the client. The cache manager runs our dynamic algorithm to compute the ideal cache configuration (described in §5.4).

4. *Cache* provides volatile bounded storage where clients store chunks of erasure-coded data, according to the hints received from the requests monitor.

## 5.4 Algorithm

This section describes the algorithm that Agar's cache manager runs periodically in order to propose a static cache configuration. In more details, the cache manager chooses *which data objects to cache* and *how many chunks to store for each of them.*

As explained in §5.2.2, we map the problem of choosing which data chunks to cache to a Knapsack optimization problem. We compute the *weight* of a cached item as the amount of space that it would occupy in the cache (i.e., the number of its chunks that are cached), while the *value* is the overall latency improvement that caching the respective blocks would bring to the system.

Our caching algorithm works in two steps:

1. it first *generates caching options*, and then
2. it chooses a subset of caching options to *define the cache contents*.

### 5.4.1 Generating caching options

A *caching option* is a hypothetical configuration that captures the implications of caching a specific set of chunks for an object. Each caching option contains:

- a key identifying the object it corresponds to;
- a set of data chunks to cache for that object;
- a weight, given by the number of data chunks to cache;
- a value, computed as the overall latency improvement that caching this set of blocks would bring to the system.

For each object known to the request monitor, we iteratively generate caching options. For the sake of simplicity, we have not included the pseudo-code for this step. Each caching option includes a weight, which varies between 1 and $k$ data chunks; recall that a client needs exactly $k$ chunks to reconstruct an object, so it does not make sense to cache more. The algorithm needs to choose data chunks to put in each of the configurations. The cache stores the blocks that would be retrieved in the common case by the client, i.e., a client would not attempt to retrieve the furthest $m$ blocks unless there are failures. Thus, the algorithm first discards the $m$ blocks that are furthest away from the cache in terms of latency, because in the common case (without failures) those would not need to be accessed by clients. Caching items implies downloading them a priori; therefore, we optimize the latency penalty incurred by a cache miss by not caching the furthest blocks. The configurations are then filled with data blocks, from the most distant remaining data sites, until reaching the associated weight.

After choosing the set of blocks in each caching option, the algorithm needs to compute the associated values. To that end, we compute an estimation of the overall latency improvement that adopting a certain caching option would bring. This is computed as $popularity \times latency\_improvement$.

We compute the popularity of individual objects using an exponentially weighted moving average:

$$popularity^{i}_{key} = \alpha \cdot freq^{i}_{key} + (1-\alpha) \cdot popularity^{i-1}_{key},$$

where $key$ identifies the object to which the caching option corresponds, $i$ indicates the time period when this computation is done, $freq^{i}_{key}$ is the access frequency for that object in the current time period, and $\alpha$ is the weighting coefficient (0.8 in our experiments).

To compute the *latency improvement*, the algorithm needs to know the latency to each backend region. The region manager computes this by retrieving several data blocks from each region in a warm-up phase. Using this data, the algorithm computes the *latency improvement* as the difference between the latencies to the most distant region that is contacted when the set of blocks in a caching option are cached versus when they are not. This assumes that the client requests blocks in parallel and that the requests do not interfere with each other.

Table 5.1: Read latency from the point of view of Frankfurt.

| Frankfurt | Dublin | N. Virginia | Sao Paulo | Tokyo | Sydney |
|---|---|---|---|---|---|
| 80 ms | 200 ms | 600 ms | 1,400 ms | 3,400 ms | 4,600 ms |

**Example** We consider the deployment in Figure 5.1 and assume our algorithm is running on an Agar node hosted in Frankfurt. The algorithm first estimates the latencies of getting blocks from each backend region, shown in Table 5.1. There are five different caching options possible for an object identified by $key_1$, storing 1, 3, 5, 7, and 9 blocks, respectively. For the option with weight 1, the algorithm chooses to cache the block from Tokyo. Since the client only needs $k$ blocks to reconstruct the object, our algorithm discards the $m = 3$ blocks that are furthest away: two from Sydney and one from Tokyo. To compute the value corresponding to this caching option, it first takes the popularity of $key_1$. Suppose for simplicity that this is the first iteration of the algorithm, so the previous popularity is 0, and the current frequency of $key_1$ is 100. Thus, the popularity is 80 ($0.8 \times 100 + 0.2 \times 0$). The estimated latency improvement is 2,000 ms, computed as the latency difference between the furthest region contacted when the block is not cached (Tokyo) and the furthest one contacted when the block is cached (Sao Paulo). Then, the value is $80 \times 2{,}000 = 160{,}000$. Similar, for option 2 the algorithm chooses to cache blocks from Sao Paulo and Tokyo, and the value is $80 \times (1{,}400 - 600) = 64{,}000$.

### 5.4.2 Choosing the cache contents

Our algorithm uses the generated caching options to compute a cache configuration. Figure 4 shows the pseudocode for choosing the contents of the cache. The algorithm uses a dynamic programming approach: it computes intermediate configurations for subsets of objects and then iteratively improves these intermediate solutions as it considers new objects.

In Figure 4, *MaxV* is an associative array storing intermediate cache configurations. In partic-

**Algorithm 4** Algorithm that computes a static configuration for the cache, based on the weight and value of each caching option.

```
 1: function POPULATE(Keys, AllOptions, CacheSize)
 2:     ▷ AllOptions — set of caching options for all keys
 3:     ▷ Keys — set of keys sorted in decreasing value order
 4:     ▷ CacheSize — available cache size
 5:     ▷ MaxV — associative array [Size] → Config

 6:     MaxV[0] ← EMPTYCONFIG()                                  ▷ Initial state

 7:     ▷ Iterate through keys in decreasing value order
 8:     for Option ∈ ORDERBY(AllOptions, Keys) do
 9:         ▷ Improve config but keep the same weight
10:         for Config ∈ MaxV do
11:             RELAX(Config, Option, AllOptions)
12:         end for

13:         ▷ Improve config by adding option at the end
14:         for Config ∈ MaxV do
15:             Let W = Config.Weight + Option.Weight
16:             Let V = Config.Value + Option.Value
17:             Let C = MaxV[W]                                  ▷ Add new if missing
18:             if C.Value < V then
19:                 ADDTOCONFIG(C, Option)
20:             end if
21:         end for
22:     end for
23:
24:     return MaxV[CacheSize]
25: end function
```

ular, $MaxV[size]$ holds the best configuration discovered so far for a cache of size *size*. New configurations are implicitly created upon first access to a key, i.e., if no configuration has yet been stored under $MaxV[size]$, an empty configuration is added on-the-fly on line 17. There are two methods through which intermediate configurations are improved:

1. *Relaxation* (Figure 4, lines 10–12). The concept of relaxation is similar to the one in graph theory (e.g., Dijkstra's algorithm). RELAX checks if a new caching option *Option* can replace an existing one in an intermediate configuration, yielding a better overall value. The replacement can be *total*: an object already in the configuration is completely evicted in favor of the new option; or *partial*: the old option is only partially evicted, having fewer blocks in the configuration. Figure 5 shows the pseudocode for the relaxation method.

2. *Addition* (Figure 4, lines 14–21). The ADDTOCONFIG method (Figure 4, line 19) adds a new option at the end of an existing configuration, increasing its weight. If there is already another configuration with this new weight, it is replaced only if it has a lower value.

**Algorithm 5** Relaxation function that improves a configuration's value without increasing its total weight.

```
1: function RELAX(Config, Option, AllOptions)
2:     BestConfig ← Config
3:     for OldOption ∈ Config.Options do
4:         ▷ Replace OldOption with alternative option O for the same key, but with a lower weight W, making room for Option
5:         Let W = OldOption.Weight − Option.Weight
6:         Let O = SEARCHOPTION(AllOptions, W, OldOption.Key)
7:         Let V = Config.Value − OldOption.Value + O.Value + Option.Value
8:         if BestConfig.Value < V then
9:             BestConfig ← REPLACEANDADD(Config, OldOption, Option)
10:        end if
11:    end for
12:    Config ← BestConfig
13: end function
```

## 5.5 Evaluation

We built a prototype of Agar and used it to perform a thorough quantitative evaluation. We compare Agar against caching systems that use the classical Least Recently Used (LRU) and Least Frequently Used (LFU) cache replacement policies, while varying the amount of data kept in the cache from one chunk to a whole replica (i.e., $k$ chunks).

We first describe our experimental setup in §5.5.1. Then we answer the following questions:

- How does Agar compare to other caching policies (§5.5.2)?
- How do the cache size and workload influence the performance of Agar (§5.5.3)?
- What do the cache contents look like in Agar (§5.5.4)?

### 5.5.1 Setup

We implemented our Agar prototype in Java and integrated it in the deployment shown in Figure 5.1. For our experiments, we modified the YCSB client [125]:

- First, we added support for erasure coding via the Longhair library [126]. On a write operation, the client encodes the object and writes the resulted chunks to S3 buckets concurrently. On a read operation, the client requests data chunks in parallel and, after it has received $k$ chunks, it decodes them. The read latency measured by our modified YCSB client accounts for reading a full object, and not just a chunk.
- Second, we added support for Agar. The YCSB client communicates with Agar in order to know what regions to contact. The client is also responsible for writing data to caches. This operation does not impact the latency measurements, as it is done in a separate

thread pool, and not concurrently with reads. We deploy YCSB clients on large EC2 instances in the same regions as Agar.

We use several customized versions of the YCSB client, which differ in terms of reading strategy:

- *Agar*—reads content via our Agar caching system.
- *Backend*—reads content directly from the S3 buckets.
- *LRU*—reads content via a cache that stores a predefined number of erasure-coded chunks for each data record and supports the *Least Recently Used* policy. For our experiments, we rely on `memcached`'s LRU policy.
- *LFU*—reads content via a cache that stores a predefined number of erasure-coded chunks and supports the *Least Frequently Used* cache replacement policy. This client includes an additional proxy component that tracks request frequency for each object.

Unless stated otherwise, we use a read-only workload that follows a Zipfian distribution with skew factor 1.1, a cache size of 10 MB—which fits ten full objects (9 chunks each) and set the cache reconfiguration period to 30 seconds for Agar and LFU. All results represent averages of 5 runs. Each run contains 1,000 reads. Each YCSB instance is configured to run 2 clients; each client uses a thread pool to make requests for chunks in parallel.

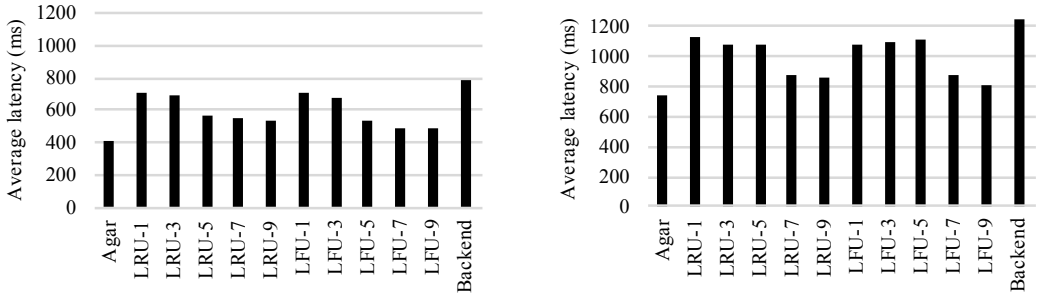
### 5.5.2 Agar compared to other caching policies

In this experiment, we compare the latency and cache hit ratio of Agar to those of the LFU and LRU policies with fixed number of chunks. We show that Agar can use the cache more efficiently, obtaining better performance than classical policies.

As shown in §5.2.1, the trade-offs regarding the number of chunks to store are different, depending on the region where the client runs. Therefore, we run this experiment using clients in two regions:

1. In the first scenario, we deploy our clients at Frankfurt, which has a relatively central position in our deployment, and is rather close to another region: Dublin.
2. In the second scenario, we deploy our clients at Sydney, which represents the opposite of Frankfurt, being far away from all other regions.

Figure 5.4 shows how Agar compares to LRU, LFU, and the backend, in terms of average read latency. Agar adapts to the particularities of each site and consistently outperforms the classical policies.



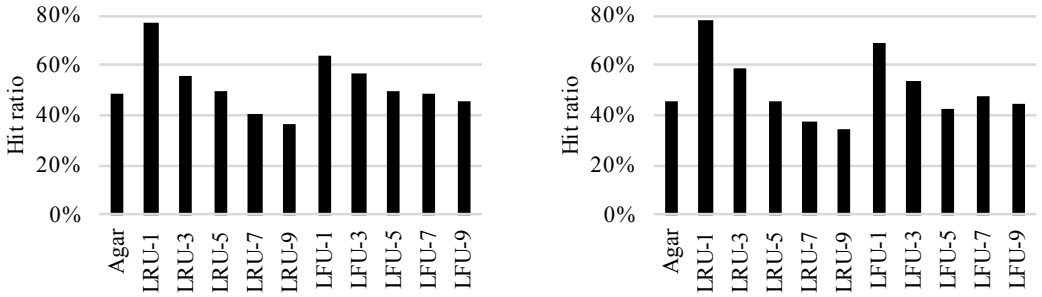

(a) Frankfurt (b) Sydney

Figure 5.4: Average read latency when using Agar vs. LRU- and LFU-based caching vs. Backend.

In Frankfurt, Agar obtains 15% lower latency than LFU-7, which is the next best policy (Least Frequently Used, caching 7 chunks for each object). Agar has an average latency of 416 ms versus LFU-7's 489 ms. When compared to the worst-performing setup, LRU-1, Agar yields 41% lower latency.

In Sydney, Agar obtains 8.5% lower latency than LFU-9, which is the next best policy. Agar obtains a latency of 736 ms, versus LFU-9's 803 ms.

We also examined the hit rates that the different policies obtained in this experiment. Figure 5.5 shows the hit ratio, computed as the number of cache hits – *total hits* (all blocks were read from the cache) or *partial hits* (only a subset of blocks were read from the cache) – divided by the number of requests issued.




(a) Frankfurt (b) Sydney

Figure 5.5: Hit ratio when using Agar vs. LRU- and LFU-based caching.

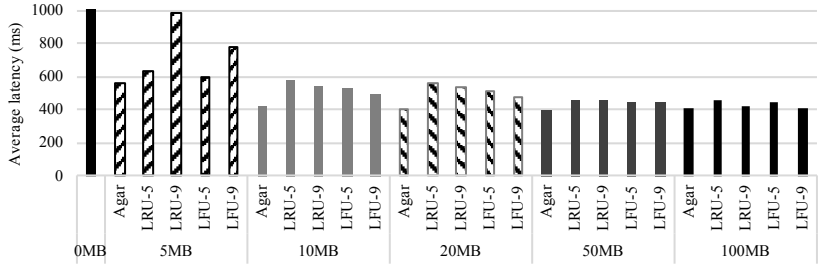
As expected, storing fewer chunks per object leads to higher hit rates, as high as 76%. However, storing fewer chunks leads to low overall latency improvement. Agar finds a good trade-off between storage cost and latency improvement for each object, storing more chunks for popular items, but then reducing the number of chunks corresponding to less popular items.

Overall, Agar's hit ratio is higher than the hit ratio of the LRU and LFU policies storing 7 or 9 chunks for each object (§5.5.4 has some more insight on how Agar manages its cache).

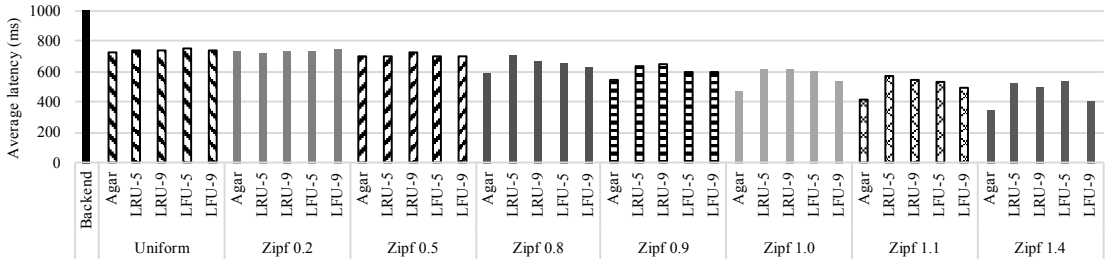
In this experiment, we showed that Agar outperforms classical policies like LRU or LFU. Moreover, unlike static policies, Agar can adapt to the particularities of the workload. Agar obtains this performance gain over LRU and LFU by carefully managing the trade-off between the size each object occupies in the cache and the overall latency improvement.

### 5.5.3 Influence of cache size and workload

In this experiment, we study the impact of external factors on the performance of Agar and its competitors—LRU and LFU. In the previous experiment, we kept the cache size and workload pattern fixed, while in this experiment we vary them to evaluate how the different policies react. We run this experiment using clients deployed at Frankfurt.




(a) Varying cache size.




(b) Varying workload.

Figure 5.6: Agar vs. different caching systems and the backend.

We vary the cache size between 5 MB (fits 5 full objects) and 100 MB (fits 100 full objects), while keeping the workload fixed (Zipfian, with skew 1.1). Figure 5.6(a) shows the average read latency.

When the cache is very small, there is little room for optimization, but Agar can still outperform all alternatives by more than 6.5%. As cache size increases, so does the advantage of Agar: it obtains 15% lower latency than alternatives for 10 MB cache size, and 16% for 20 MB. When the size increases beyond that, the cache becomes large enough to fit all popular data and

Agar's lead starts to decrease: 12% for 50 MB and 1% for 100 MB. Overall, Agar outperformed LFU and LRU over a wide range of deployment scenarios, with the cache size ranging from 1% to 25% of the backend (5 to 100 MB).

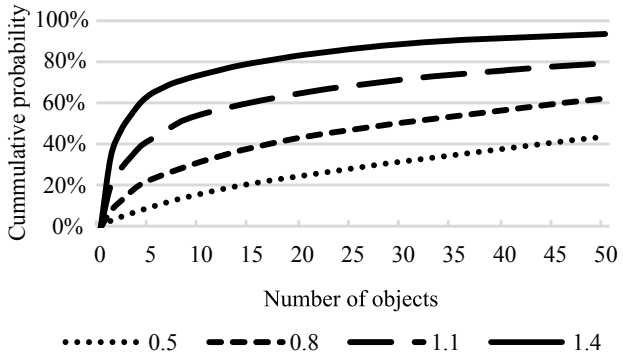



Figure 5.7: Cumulative distribution of the object popularity using Zipfian workloads with different skews. The $y$ axis shows the cumulative percentage of requests in the workload that refer to objects on the $x$ axis (e.g., $x = 5$, $y = 40\%$ means that the most popular 5 objects account for 40% of requests).

Next, we keep the cache size fixed at 10 MB and vary the workload. First, we experiment with a workload that follows a uniform request distribution. Next, we experiment with Zipfian workloads with different skews; the skew is the coefficient that determines the number of popular data elements: higher skew means that fewer and fewer items become increasingly popular. We show in Figure 5.7 how the skew influences the popularity of the objects in the workload. Since the object size is 1 MB, it is also easy to see how much of the total data (300 MB) can fit in a cache of a given size (the horizontal axis can be interpreted as cache size as well).

Figure 5.6(b) shows the average read latency of Agar and its alternatives for different workloads. When the workload follows a uniform distribution, all clients perform similarly. The cache hit rates are very small because all data items are equally popular; therefore, the choice of caching policy makes no significant difference. When the skew of the Zipfian distribution is low, the workload pattern is similar to a uniform distribution and thus, the same effect is observed.

As the skew of the Zipfian distribution increases, however, some elements become more popular and caching them has a higher impact on the overall latency. Agar and LFU are the quickest to benefit from this type of workloads and can lead to lower overall latencies, with Agar taking a 5.8% lead for skew 0.8, 7.2% for 0.9, 13% for 1.0, and peaking at 15% for 1.1. As the skew becomes larger, only a small subset of objects account for most of the reads in the workload, and LFU-9 can catch up to Agar, since all of the highly popular items can fit in the 10 MB cache. At skew 1.4, the lead of Agar starts to decrease, dropping to 14%.

In this experiment, we showed how Agar compares to LRU and LFU when the workload distribution and the cache size vary. Agar consistently outperforms static policies when *system*

*designers are cost-conscious* and *cache size is limited*, as it can better adapt to the environment where it runs and optimize the cache contents accordingly.

### 5.5.4 Cache contents

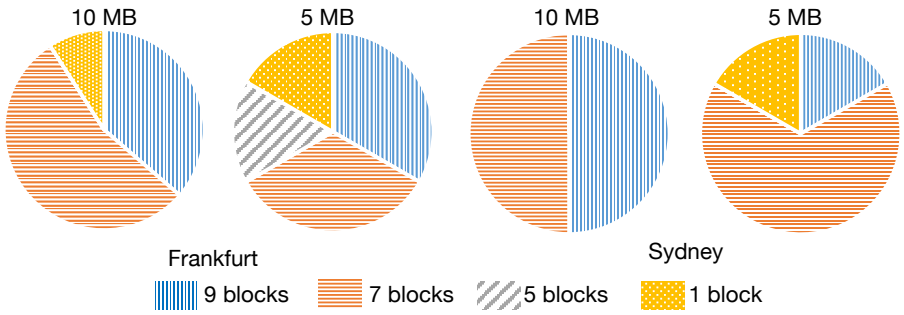



Figure 5.8: Cache contents in different scenarios

In this experiment, we take an inside look into how Agar manages the cache contents. We take snapshots of the data that Agar chooses to cache for clients running in Frankfurt and Sydney, and for cache sizes of 5 MB and 10 MB.

Figure 5.8 shows the distribution of object sizes in Agar's cache. There are several interesting aspects to note. First, Agar diversifies the contents of the cache, rather than having the majority of the cache filled by a certain object size. Second, for each scenario Agar chooses to manage its cache differently. This again argues for a dynamic policy, such as Agar.

Finally, despite the diminishing returns of storing entire replicas (9 blocks in cache), Agar chooses to allocate a significant fraction to this. This is explained by the high skew, which means that a few objects are so popular that the difference between disk and memory latency becomes important.

## 5.6 Summary

In this chapter, we argued for the need for a caching system specifically developed for erasure-coded data, providing high availability with low latency and without the need to store full object replicas. We designed and implemented Agar, a caching system tailored for erasure-coded data and explained how it integrates with a typical storage system. Agar uses a dynamic programming approach inspired by the Knapsack problem to optimize the cache configuration under a given workload. We compared our prototype with the LFU and LRU strategies and showed that Agar consistently outperforms them, obtaining 16%–41% lower latency.

# 6 Conclusion

Geo-distributed systems span many datacenters and are accessed by users from different geographic regions. These systems often rely on replication to place service replicas in all regions and on data redundancy to bring data close to the users that access it the most. This thesis proposed techniques to reduce the access latency perceived by end users.

First, we proposed an approach that reduces access latency through state partitioning. We defined the conditions under which a service can be partitioned and introduced a generic algorithm for partitioning the state of a service. Our approach allows system administrators to choose the degree of replication for different partitions. We build a geo-distributed coordination service and partitioned its state. For workloads which exhibit access locality, our partitioning technique reduced the latency perceived by users by up to 75% compared to a typical, non-partitioned deployment.

We also built a geo-distributed file system using state partitioning. Our evaluation showed that our file system performs on par with de-facto industry implementations. We further explained how to add support for performing state partitioning dynamically in order to adapt to workload patterns. Our preliminary results showed that being able to move files according to access patterns has a positive effect on the performance of our geo-distributed file system; read latency is reduced by up to 88%.

Finally, we proposed an algorithm that optimizes the cache configuration for erasure-coded data, which is useful in the context of geo-distributed storage systems that aim to reduce storage cost. We used a dynamic programming approach to optimize the cache configuration for a given workload. We integrated a caching system prototype that uses our algorithm into a geo-distributed storage system and showed that our algorithm can achieve 16% to 41% lower latency than systems that using the LRU and LFU caching policies.

# List of Publications

1. **R. Halalai**, P. Felber, A.-M. Kermarrec, F. Taiani. *Agar: A Caching System for Erasure-Coded Data*, 37th IEEE International Conference on Distributed Computing Systems (ICDCS), 2017. (Rank A, Acceptance rate: 16.9%)

2. L. Pacheco, **R. Halalai**, V. Schiavoni, F. Pedone, E. Rivière, P. Felber. *GlobalFS: A Strongly Consistent Multi-Site File System*, 35th Symposium on Reliable Distributed Systems (SRDS), 2016. (Rank A, Acceptance rate: 32.5%)

3. **R. Halalai**, P. Sutra, E. Rivière, P. Felber. *ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service*. 33rd International Symposium on Reliable Distributed Systems, 2014. (Rank A)

4. L. Charles, P. Felber, **R. Halalai**, E. Rivière, P. Felber, V. Schiavoni, J. Valerio. *An Overview of New Features in the SPLAY Framework for Simple Distributed Systems Evaluation*. University of Neuchâtel, 2012. (Technical report)

# Bibliography


[1] "Internet World Stats Usage and Population Statistics." https://www.internetworldstats.com/emarketing.htm. Accessed: 2018-04.

[2] "Digital in 2018." https://wearesocial.com/blog/2018/01/global-digital-report-2018. Accessed: 2018-04.

[3] J. Ousterhout, "Cs 142 large-Scale Web Applications." https://web.stanford.edu/~ouster/cgi-bin/cs142-fall10/lecture.php?topic=scale. Accessed: 2018-04.

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[5] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: a read/write peer-to-peer file system," in *OSDI*, 2002.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007.

[8] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, Apr. 2010.

[9] MongoDB. https://www.mongodb.org.

[10] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, DSN, 2011.

[11] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *IPTPS*, Springer-Verlag, 2002.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX Annual Techical Conference, USENIX Association, 2010.

[13] P. Alsberg and J. Day, "A principle for resilient sharing of distributed resources," in *2nd Int. Conf. on Software Engineering*, ICSE, 1976.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] L. Lamport, "Paxos made simple," *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, vol. 32, 2001.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pp. 173–186, USENIX Association, 1999.

[17] T. 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 (PODC)*, pp. 398–407, 2007.

[18] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[19] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A High-Throughput Atomic Broadcast Protocol," in *DSN*, 2010.

[20] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pp. 305–320, USENIX Association, 2014.

[21] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, pp. 91–122, 1994.

[22] P. Sutra and M. Shapiro, "Fast Genuine Generalized Consensus," in *Proceedings of the 30th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pp. 255–264, 2011.

[23] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing*, vol. 15, 2002.

[24] L. Lamport, "Generalized consensus and paxos," Tech. Rep. MSR-TR-2005-33, Microsoft, March 2005.

[25] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data center consistency," in *8th ACM European Conference on Computer Systems*, EuroSys, 2013.

[26] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *34th International Conference on Distributed Computing Systems*, ICDCS, July 2014.

[27] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *10th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2012.

[28] P. J. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *DSN*, 2012.

[29] G. Oster, P. Urso, P. Molli, and A. Imine, "Data Consistency for P2P Collaborative Editing," in *ACM Conference on Computer-Supported Cooperative Work*, CSCW, Nov. 2006.

[30] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *29th International Conference on Distributed Computing Systems*, ICDCS, pp. 395–403, 2009.

[31] C. E. Bezerra, F. Pedone, and R. van Renesse, "Scalable state-machine replication," in *45th International Conference on Dependable Systems and Networks*, DSN, June 2014.

[32] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proc. VLDB Endow.*, vol. 1, pp. 598–609, Aug. 2008.

[33] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[34] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers, "Universal constructions that ensure disjoint-access parallelism and wait-freedom," in *ACM Symposium on Principles of Distributed Computing*, PODC, 2012.

[35] L. H. Le, C. E. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[36] A. Nogueira, A. Casimiro, and A. Bessani, "Elastic state machine replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, 2017.

[37] A. Thomson and D. J. Abadi, "CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems," in *FAST*, 2015.

[38] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *ACM SIGMOD*, 2012.

[39] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI*, 2006.

[40] Ceph block storage. http://ceph.com/ceph-storage/block-storage/.

[41] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *ACM/IEEE SC*, 2006.

[42] Email exchange on CephFS mailing list. https://www.mail-archive.com/ceph-users@lists.ceph.com/msg23788.html.

[43] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP*, 2003.

[44] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, pp. 133–169, May 1998.

[45] MooseFS. https://www.moosefs.org.

[46] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE MSST*, 2010.

[47] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," in *VLDB*, 2013.

[48] G. Liu, L. Ma, P. Yan, S. Zhang, and L. Liu, "Design and Implementation of GeoFS: A Wide-Area File System," IEEE NAS, 2014.

[49] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, "Flexible, Wide-Area Storage for Distributed Systems with WheelFS.," in *NSDI*, 2009.

[50] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe, "A 64-bit, shared disk file system for linux," in *IEEE MSST*, 1999.

[51] A. Davies and A. Orsaria, "Scale out with GlusterFS," *Linux Journal*, vol. 2013, Nov. 2013.

[52] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The XtreemFS architecture—a case for object-based file systems in grids,"

*Concurrency and Computation: Practice and Experience*, vol. 20, no. 17, pp. 2049–2060, 2008.

[53] M. Vrable, S. Savage, and G. M. Voelker, "Bluesky: a cloud-backed file system for the enterprise," in *FAST*, 2012.

[54] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo, "SCFS: a shared cloud-backed file system," in *USENIX ATC*, 2014.

[55] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *4th Annual Linux Showcase and Conference*, 2000.

[56] D. Stamatakis *et al.*, "Scalability of replicated metadata services in distributed file systems," in *DAIS*, 2012.

[57] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *USENIX ATC*, 1999.

[58] PVFS2. http://www.pvfs.org.

[59] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Linux Symposium*, 2003.

[60] BeeGFS. http://www.beegfs.com.

[61] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *FAST*, 2011.

[62] ObjectiveFS. http://objectivefs.com.

[63] M. Satyanarayanan, "Scalable, secure, and highly available distributed file access," *Computer*, vol. 23, no. 5, pp. 9–18, 1990.

[64] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS distributed operating system," in *SOSP*, 1983.

[65] Sun Microsystems, Inc., "NFS: Network file system protocol specification," RFC 1094, Mar. 1989.

[66] "Oracle Cluster File System (OCFS)," in *Pro Oracle Database 10g RAC on Linux*, pp. 171–200, Apress, 2006.

[67] Y. Ma, T. Nandagopal, K. P. N. Puttaswamy, and S. Banerjee, "An ensemble of replication and erasure codes for cloud file systems," in *INFOCOM*, IEEE, 2013.

[68] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *ATC*, USENIX, 2012.

[69] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A Tale of Two Erasure Codes in hdfs," in *FAST*, USENIX Association, 2015.

[70] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, 1960.

[71] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *INFOCOM*, IEEE, 1999.

[72] F. Figueiredo, F. Benevenuto, and J. M. Almeida, "The tube over time: Characterizing popularity growth of YouTube videos," in *WSDM*, ACM, 2011.

[73] G. Szabo and B. A. Huberman, "Predicting the popularity of online content," *Communications of the ACM*, vol. 53, no. 8, 2010.

[74] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *NSDI 12*, USENIX, 2012.

[75] S. Jin and A. Bestavros, "GreedyDual*: Web caching algorithms exploiting the two sources of temporal locality in web request streams," in *WCW*, 2000.

[76] G. D. S. Silvestre, *Designing Adaptive Replication Schemes for Efficient Content Delivery in Edge Networks*. PhD thesis, Universite Pierre et Marie Curie, 2013.

[77] K. Mokhtarian and H.-A. Jacobsen, "Caching in video CDNs: Building strong lines of defense," in *EuroSys*, ACM, 2014.

[78] G. Einziger and R. Friedman, "TinyLFU: A highly efficient cache admission policy," in *PDP*, IEEE, 2014.

[79] G. Karakostas and D. Serpanos, "Exploitation of different types of locality for Web caches," in *ISCC*, IEEE, 2002.

[80] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *USITS*, USENIX, 1997.

[81] L. Cherkasova, "Improving WWW proxies performance with Greedy-Dual-Size-Frequency caching policy," tech. rep., HP Technical Report, 1998.

[82] K. Cheng and Y. Kambayashi, "LRU-SP: A size-adjusted and popularity-aware LRU replacement algorithm for Web caching," in *COMPSAC*, 2000.

[83] C. Aggarwal, J. L. Wolf, and P. S. Yu, "Caching on the World Wide Web," *Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, 1999.

[84] R. Karedla, S. J. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, 1994.

[85] L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," *Journal IEEE/ACM TON*, vol. 8, no. 2, 2000.

[86] V. Aggarwal, Y.-F. R. Chen, T. Lan, and Y. Xiang, "Sprout: A functional caching approach to minimize service latency in erasure-coded storage," in *Poster session at the ICDCS*, IEEE, 2016.

[87] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding," in *OSDI*, USENIX Association, 2016.

[88] R. Halalai, P. Sutra, E. Riviere, and P. Felber, "Zoofence: Principled service partitioning and application to the zookeeper coordination service," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pp. 67–78, 2014.

[89] M. Herlihy and J. Wing, "Linearizability: a correcteness condition for concurrent objects," *ACM Trans. on Prog. Lang.*, vol. 12, July 1990.

[90] L. Lamport, "On interprocess communication. part i: Basic formalism," *Distributed Computing*, vol. 1, no. 2, 1986.

[91] M. Abadi and L. Lamport, "The existence of refinement mappings," in *3rd Annual IEEE Symposium on Logic in Computer Science*, LICS, July 1988.

[92] M. Herlihy, "Wait-free synchronization," *ACM Trans. on Prog. Lang. and Systems*, vol. 11, Jan. 1991.

[93] P. Sutra and M. Shapiro, "Fault-tolerant partial replication in large-scale database systems," in *European Conf. on Parallel Computing*, Euro-Par, Springer, 2008.

[94] M. Raynal and J. Stainer, "From a store-collect object and Ω to efficient asynchronous consensus," in *European Conf. on Parallel Computing*, Euro-Par, Springer, 2012.

[95] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[96] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *13th Annual ACM Symposium on Principles of Distributed Computing*, PODC, 1994.

[97] K. Birman, G. Chockler, and R. van Renesse, "Toward a Cloud Computing research agenda," *ACM SIGACT News*, vol. 40, pp. 68–80, June 2009.

[98] Apache Software Foundation, "Apache Bookkeeper." https://bookkeeper.apache.org/. Accessed: 2018-04.

[99] L. Pacheco, R. Halalai, V. Schiavoni, F. Pedone, E. Riviere, and P. Felber, "Globalfs: A strongly consistent multi-site file system," in *2016 IEEE 35th Symposium on Reliable Distributed Systems(SRDS)*, vol. 00, pp. 147–156, Sept. 2016.

[100] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[101] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[102] IEEE, "IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6," 2001.

[103] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *SOSP*, 2001.

[104] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP*, 2001.

[105] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[106] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, 2014.

[107] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[108] J. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," in *USENIX ATC*, 2012.

[109] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a high-performance, distributed main memory transaction processing system," in *VLDB*, 2008.

[110] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *DSN*, pp. 331–342, IEEE, 2014.

[111] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in *11th Australian Computer Science Conference*, pp. 55–66, 1988.

[112] M. Raynal, A. Schiper, and S. Toueg, "The causal ordering abstraction and a simple way to implement it," *Inf. Process. Lett.*, vol. 39, pp. 343–350, 1991.

[113] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with atomic multicast," in *IFIP/ACM Middleware*, 2014.

[114] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX Association, 2000.

[115] File System in User Space (FUSE). http://fuse.sourceforge.net/.

[116] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra incognita: On the practicality of user-space file systems," in *USENIX HotStorage*, 2015.

[117] URingPaxos. https://github.com/sambenz/URingPaxos.

[118] Apache Thrift. https://thrift.apache.org.

[119] L. Leonini, E. Rivière, and P. Felber, "SPLAY: Distributed systems evaluation made simple," in *NSDI*, 2009.

[120] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, pp. 17–32, Feb. 2003.

[121] LevelDB. https://github.com/google/leveldb.

[122] AWS EC2 instances types. http://aws.amazon.com/ec2/instance-types/.

[123] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, "Benchmarking file system benchmarking: It *is* rocket science," in *USENIX HotOS*, 2011.

[124] R. Halalai, P. Felber, A. Kermarrec, and F. Taïani, "Agar: A caching system for erasure-coded data," in *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pp. 23–33, 2017.

[125] "Yahoo! Cloud Serving Benchmark." https://github.com/brianfrankcooper/YCSB. Accessed: 2016-12-01.

[126] "Fast Cauchy Reed-Solomon Erasure Codes in C." https://github.com/catid/longhair. Accessed: 2016-12-01.

[127] T. Dantzig, *Number, the Language of Science.* New York: Free Press, 1954.

[128] M. Kedia, "Lecture on Knapsack." https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf. Accessed: 2016-12-01.

[129] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, Inc., 1990.