Towards more scalability and flexibility for distributed storage systems

Guillaume Ruty

université PARIS-SACLAY

NNT : 2019SACLT006

Thèse de doctorat

TELECOM ParisTech

Towards more scalability and flexibility for distributed storage systems

Thèse de doctorat de l'Université Paris-Saclay
préparée à Télécom ParisTech

Ecole doctorale n°580 Ecole Doctorale Sciences et Technologies de l'Information et de la Communication (ED STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Paris, le 15 Février 2019, par

GUILLAUME RUTY

Composition du Jury :

André-Luc Beylot Professeur, ENSEEIHT (IRIT)	Rapporteur
Stefano Secci Professeur, CNAM	Rapporteur
Raouf Boutaba Professeur, University of Waterloo	Examinateur
Nadia Boukhatem Professeur, Telecom ParisTech (LTCI)	Président
Damien Saucez Chargé de Recherche, INRIA	Examinateur
Jean-Louis Rougier Professeur, Telecom ParisTech (LTCI)	Directeur de thèse
André Surcouf Distinguished Engineer, Cisco Systems (PIRL)	Co-encadrant de thèse
Mark Townsley Fellow, Cisco Systems (PIRL)	Invité

TELECOM PARISTECH

DOCTORAL THESIS

Towards more scalability and flexibility for distributed storage systems

Author:
Guillaume Ruty

Supervisor:
Dr. Jean-Louis Rougier

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Cisco Systems Paris Innovation and Research Lab (PIRL)
Laboratoire Traitement et Communication de l'Information (LTCI)

May 1, 2019

Declaration of Authorship

I, Guillaume Ruty, declare that this thesis titled, "Towards more scalability and flexibility for distributed storage systems" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 01/05/2019

"La maturité de l'homme est d'avoir retrouvé le sérieux qu'on avait au jeu quand on était enfant."

Alain Damasio, La Horde du Contrevent

TELECOM PARISTECH

Abstract

Laboratoire Traitement et Communication de l'Information (LTCI)

Doctor of Philosophy

Towards more scalability and flexibility for distributed storage systems

by Guillaume Ruty

The exponentially growing demand for storage puts a huge stress on traditionnal distributed storage systems. While storage devices' performance keep improving over time, current ditributed storage systems struggle to keep up with the rate of data growth, especially with the rise of cloud and big data applications. Furthermore, the performance balance between storage, network and compute devices has shifted and the assumptions that are the foundation for most distributed storage systems are not true anymore.

This dissertation explains how several aspects of such storage systems can be modified and rethought to make a more efficient use of the resource at their disposal. It presents 6Stor, an original architecture that uses a distributed layer of metadata to provide flexible and scalable object-level storage, then proposes a scheduling algorithm improving how a generic storage system handles concurrent requests. Finally, it describes how to improve legacy filesystem-level caching for erasure-code-based distributed storage systems, before presenting a few other contributions made in the context of short research projects.

Les besoins en terme de stockage, en augmentation exponentielle, sont difficilement satisfaits par les systèmes de stockage distribué traditionnels. Même si les performances des disques continuent à s'améliorer, les systèmes de stockage distribué actuels peinent à suivre le croissance du nombre de données requérant d'êtres stockées, notamment à cause de l'avènement des applications de big data. Par ailleurs, l'équilibre de performances entre disques, cartes réseau et processeurs a changé et les suppositions sur lesquelles se basent la plupart des systèmes de stockage distribué actuels ne sont plus vraies.

Cette dissertation explique de quelle manière certains aspects de tels systèmes de stockages peuvent être modifiés et repensés pour faire une utilisation plus efficace des ressources qui les composent. Elle présente 6Stor, une architecture de stockage nouvelle qui se base sur une couche de métadonnées distribuée afin de fournir du stockage d'objet de manière flexible tout en passant à l'échelle. Elle détaille ensuite un algorithme d'ordonnancement des requêtes permettant à un système de stockage générique de traiter les requêtes de clients en parallèle de manière plus équitable. Enfin, elle décrit comment améliorer le cache générique du système de fichier dans le contexte de systèmes de stockage distribué basés sur des codes correcteurs avant de présenter des contributions effectuées dans le cadre de courts projets de recherche.

Acknowledgements

The work presented here could not have been done without the help and support of many people.

I would first and foremost like to thank my advisors, Jean-Louis Rougier and André Surcouf for their continuous support and insight as well as for their good company. They made these 3 years feel like 1 and really focused my attention on the relevant topics when I started to feel lost in the diversity of subjects at hand.

I would also like to thank Aloys Augustin and Victor Nguyen, who joined the 6Stor project as developpers under a Cisco tech fund. Aloys really fleshed out the crude code base that I wrote as a first prototype of 6Stor. We also had lengthy discussions about certain design or implementation details of 6Stor during which his insight helped me elaborate the global architecture. He also implemented RS3 – our storage scheduler – in 6Stor's storage servers. Victor mainly worked on the 6Stor block device implementation and on the erasure-code based storage system replica cache.

This section could not go without a hearty mention to Cisco and to Mark Townsley, who founded and runs Cisco's PIRL (Paris Innovation and Research Lab), and recruited me first as a research intern then as a PhD student, and without whom this work would simply not exist. He has consistently been a driving force behind 6Stor, from the project's origins to the end of my PhD. The same mention goes to Jérome Tollet, who piqued my curiosity on numerous occasions and subjects during our car rides or coffee breaks, and participated to the elaboration of RS3 with Aloys and me, in addition of being a merry desk neighbour.

My final thanks go to my fellow PhD students and friends, namely Jacques Samain, Yoann Desmouceaux, Marcel Enguehard, Mohammed Hawari and Hassen Siad. Whether we gathered around the lunch table, the coffee machine or the babyfoot, they always kept the occasional dullness at bay and heavily contributed to making these 3 years truly special.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation and Durability
ACK	Acknowledgment
ACL	Access Control List
AFS	Andrew File System
AFP	Apple Filing Protocol
API	Application Programmable Interface
BASE	Basically Available, Soft state, Eventual consistency
BIER	Bit Indexed Explicit Replication
BUSE	Block device in User-space
CAP	Consistency , Availability, Partition-resiliency
CBQ	Class-Based Queuing
CDN	Content Delivery Network
CFQ	Completely Fair Queuing
COW	Copy On Write
CPU	Central Processing Unit
CRUSH	Controlled Replication Under Scalable Hashing
CSI	Container Storage Interface
DHT	Distributed Hash Table
DPDK	Data Plane Development Kit
DRAM	Dynamic Random Access Memory
EC	Erasure Codes/Erasure Coding
ECMP	Equal Cost Multi-Path
FCP	Fibre Channel Protocol
FIOS	Flash I/O Scheduler
GFS	Google File System or Global File System
HDD	Hard-Disk Drive
HDFS	Hadoop Distributed File System
HTTP	Hyper Text Transfer Protocol
ICN	Information Centric Networking
I/O	Input/Output
I/Ops	Input/Output per second
iSCSI	internet Small Computer System Interface
LAN	Local Area Network
LRU	Least Recently Used
MDS	Maximum Distance Separable
MG	Metadata Group
MN	Metadata Node
MRC	Monotonic Read Consistency
MWC	Monotonic Write Consistency
NAS	Network Attached Storage

NBD Network Block Device
NFS Network File System
NoSQL Non SQL or Not only SQL (see **SQL**)
NVMe Non-Volatile Memory express
N2OS Network-Native Object Store
ON Orchestrator Node
OPCISS Optimized Predictive Container Image Storage System
OSD Object Storage Dæmon
PACELC Partition: Availability or Consistency, Else: Latency or Consistency
PCIe Peripheral Component Interconnect express
PG Placement Group
P2P Peer-to-Peer
QoS Quality of Service
QUIC Quick UDP Internet Connection Protocol
RADOS Reliable Autonomic Distributed Object Store
RAID Redundant Array of Independant Disks
RAM Random Access Memory
RDMS Relational Database Management System
RGC Regenating Codes
RPM Revolutions Per Minute
RS3 Request Scheduler for Storage Servers
RTT Round Time Trip
RYWC Read Your Writes Consistency
SaaS Storage as a Service
SATA Serial AT Attachment
SDS Software Defined Storage
SFQ Start-time Fair Queuing
SFQ(D) Depth-based Start-time Fair Queuing
SLA Service Level Agreement
SMB Server Message Block
SN Storage Node
SPDK Storage Plane Development Kit
SQL Structured Query Language
SR Segment Routing
SRLB Segment Routing Load Balancing
SSD Solid-State Drive
TCP Transmission Control Protocol
UDP User Datagram Protocol
VFS Virtual File System
VIP Virtual IP
VM Virtual Machine
VPP Vector Packet Processing
WFQ Weighted Fair Queuing
WFRC Write Follows Read Consistency
YFQ Yet another Fair Queuing algorithm

Introduction

Motivations

More than two decades ago, the first distributed storage systems were born in an attempt to guarantee that important data would never be lost and would always be available, in a more convenient way than just regular backups.

The vast majority of distributed storage systems follow the same pattern: they are designed to run on numerous cheap, unreliable and generic devices, composed of multiple storage disks, a processor, a network interface, and sometimes some additional memory. The software of these systems organizes these devices in clusters, distributes data among servers, and generally makes sure that failures have the least possible impact on the cluster's performance. However, the landscape of storage has shifted in numerous ways in the last few years.

First, for more than a decade, storage hardware has lagged behind its CPU, memory and network counterpart after Hard Disk Drives (HDDs) hit their mechanical limitations at 15K Revolutions Per Minute (RPM). However, with flash memory becoming more and more available and performant –even outpacing Moore's Law [1]–, this physical bottleneck has been removed. Unlike HDDs that are slow and have a high latency ($\sim 5 - 15ms$) as well as a low Input/Output per second (I/Ops) ($\sim 50 - 200$ depending on the rotational speed and bus type [2]) because of the spinning mechanical parts, Solid State Drives (SSDs) provide a better throughput as well as a much lower access latency ($\sim 0.05 - 0.2ms$ [3]). This is even more true for Non-Volatile Memory express (NVMe) – SSDs that are accessible through PCI-express (PCIe) bus rather than Serial AT Attachment (SATA) – that can yield multiple hundred thousands I/Ops for a throughput of several Gigabits per second (Gbps) [4].

Furthermore, the ever-increasing flash memory quality and affordability means that the HDD and SSD price curves are expected to keep drawing closer or even to cross in the next few years as shown in figure 1. Combined to the DRAM throughput evolution compared in figure 2 to its storage and network counterpart, the natural consequence is that the traditional storage software – aimed at working around previous HDD bottlenecks – has to evolve to adapt to shifting bottlenecks. Notably, new generation NVMe makes traditional interrupt-driven I/O highly uneffective with regard to CPU consumption [6]. Additionally, an increase in remote storage (accessed by the

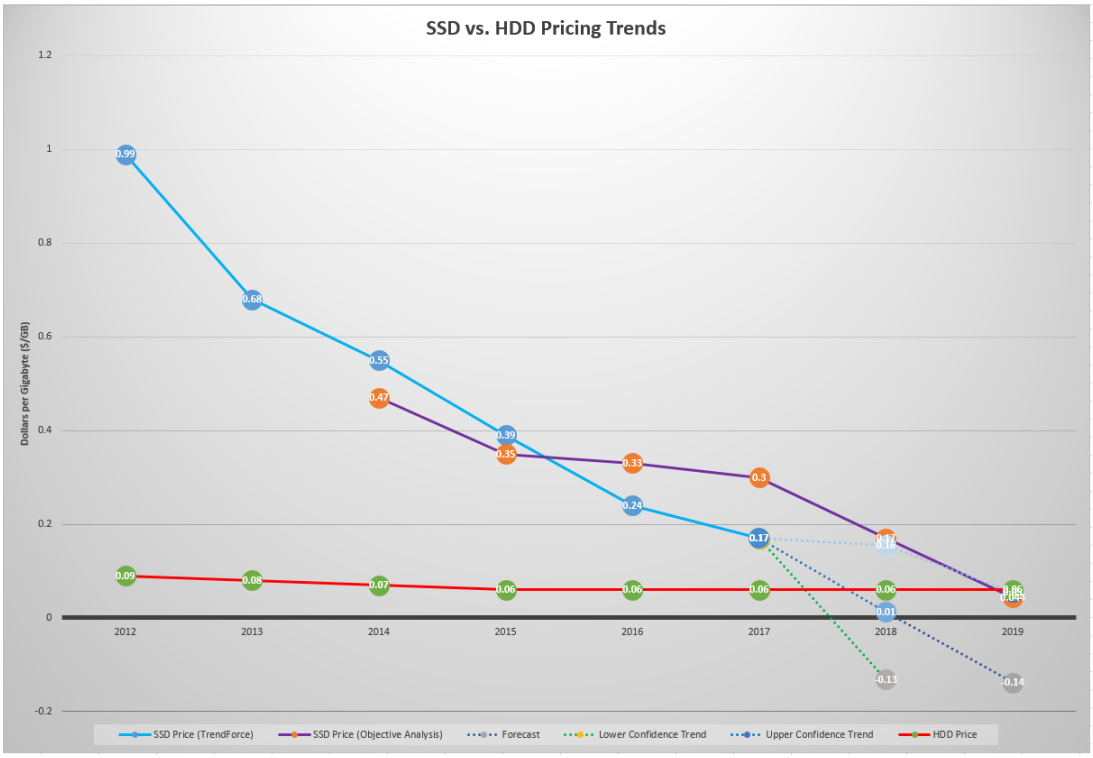


Figure 1: **SSD and HDD cost evolution prediction, as presented in [5].**

network) performance will also come at the cost of dedicating more networking hardware than what was traditionally deployed: where a 10GbE adapter was enough to saturate tens of HDDs on dedicated storage servers, 2 NVMe drives can more than saturate such an adapter [7].

Second, the way storage itself is organized and accessed has evolved in the two last decades. Where applications were deployed on individual servers and accessed local disks and filesystems, storage and compute are now separate entities. Most storage devices are assembled in large-scale clusters that run distributed databases, key-value stores or distributed filesystems, that in turn are shared between many services and applications to leverage economies of scale. These storage layers themselves have evolved and range from highly consistent databases working on very structured data to eventually consistent object stores storing data without any pattern.

Third, the amount of data to be stored grows exponentially, posing serious scalability issues: 90 % of the data stored by humanity has been generated in the last two years [8], and big companies have been storing and processing petabytes of data for years now (see table 1).

Furthermore, this exponential growth in quantity has been coupled with an ever increasing demand for fast services, becoming a real business issue for these companies: Amazon revealed that a 100ms increase in reponse time

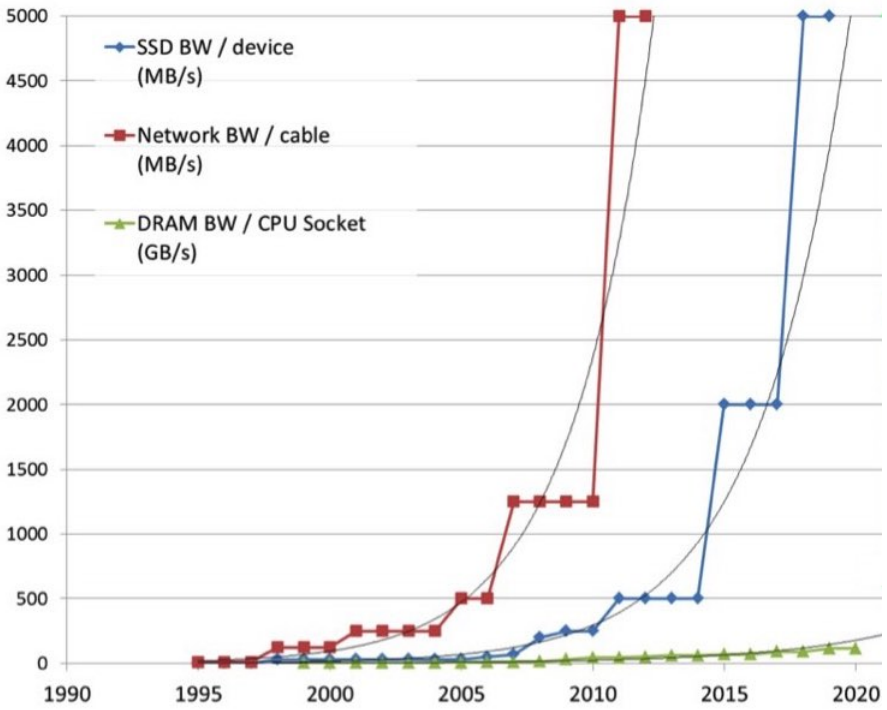


Figure 2: **Network, Storage and Memory hardware throughput evolution, as illustrated in [1].**

on their platform induces a 1% drop in associated sales [20]. Likewise, increasing the load time of a page with 10 different google search results containing 30 entries from 400 too 900ms decreased Google's advertising revenues by 20% [21]. The same has been shown to be true for every web-related business company [22]. This rapid change of scale drove big tech companies to develop their own storage backend during the mid-2000's: Amazon with DynamoDB [23], Google with GFS, BigTable and Spanner [24, 25, 26], Facebook with Cassandra [27] and others like Ceph and Apache HDFS [28, 29].

However, these new architectures that bloomed in the 2000's are showing their limitations now that the order of magnitude of storage and performance required has increased again [30, 31, 32]. There are two recurring issues in these architectures:

	Estimation of data stored	Estimation of data processed daily
Google [9, 10]	15 000 pb	100 pb
NSA [11, 12]	10 000 pb	29 pb
Baidu [13]	2 000 pb	10-100 pb
Facebook [14]	300 pb	600 Tb
eBay [15, 16]	90 pb	100 pb
Sanger (DNAsequencing) [17, 18]	45 pb	1,7 Tb
Spotify [19]	10 pb	64 Tb

Table 1: **Estimation of data stored and processed daily by big tech companies.**

- **Heavily layered software:** A high number of software layers leads to two undesirable consequences: the overhead in CPU utilization can actively impact performance on the storage server side and add latency to transactions, and reduce the efficiency of data transfer between the storage device and the network –for example because of unneeded memory copies. While this impact was almost negligible with HDDs, this is not the case anymore. This is why, for instance, Ceph has launched its project Crimson in 2018 to modernize its implementation and improve its performance by reducing the computing and memory overhead of different members of its architecture [33, 34].

- **Architectural limitations:** There are two – generally exclusive – types of architectures that are generally at fault. On one side, DHTs – initially used to avoid a central bottleneck and point of failure – are inherently not flexible and not well-suited to store varied types of data in various storage devices in the same cluster because they rely on pure flat data addressing. Furthermore, they react poorly to topology changes –which become common occurences when the number of participating storage nodes increases to follow the data growth. On another side, centralized architectures that rely on a central node and a master/slave architecture do not scale for obvious reasons. In these architectures, a single master has a full knowledge of the data placement, access control ... and is on the path of every storage request.

Thus we argue that storage systems must be rethought to overcome these techonological evolutions.

Contributions

The present dissertation has two aims: to propose a distributed storage architecture overcoming the technical and structural limitations found in present deployed software, and to optimize specific storage mechanisms in order to improve scalability, performance, or flexibility of storage systems.

To this end, I initially give a broad description of the state of the art for storage systems in chapter 1: how they evolved in the last decades, what challenges they faced, what solution they proposed to tackle their issues as well as what theoretical framework they put in place to describe their problematics.

In chapter 2, I describe in depth 6Stor, a distributed object store that we created from the ground up during my Thesis and which architecture overcomes traditionnal limitations of such distributed storage systems. The way a 6Stor cluster functions is described, including its reaction to failures, its reliability mechanisms as well as its consistency schemes. A set of benchmark is presented as well as an analysis of the performance impact of some design

decisions. The work presented in this chapter has been presented as a poster, been published as a workshop paper, and is the object of an accepted journal journal paper not yet published. Moreover, a Cisco techfund project was funded for a year to implement and improve 6Stor under the name Network-Native Object Store (N2OS). Two related patents are currently pending in the US patent office.

Chapter 3 presents two extensions made to the initial design of 6Stor during the tech fund. The first section describes how we implemented a block device on top of 6Stor's object store. The second section presents mechanisms leveraging 6Stor's IPv6 capabilities and segment routing to improve load balancing and latency. It is the object of a patent pending.

Chapter 4 describes a Request Scheduler for Storage Servers (RS3). RS3 adapts well-known network and compute scheduling algorithms to storage. It is designed to help services and applications share storage in two ways: it ensures that applications and services contending for storage are all allocated a fair share of the throughput, and it suppresses a usual problem in shared storage systems where small requests take a disproportionate amount of time to complete when they are processed in parallel with larger requests. RS3's algorithm is described in details, and the throughput and response time of several patterns of requests sent to two similar implementations of storage servers –one incorporating RS3 and the other not– are analyzed to verify that the aforementioned issues are tackled. A paper presenting RS3 is soon to be submitted, and a related defensive publication was issued by Cisco.

Chapter 5 explains how to leverage the caching capabilities of regular storage servers when they are used to deploy distributed storage systems using erasure codes as their reliability mechanisms. Simulations show that locally handling data fragment like traditionnal files on servers leads to unoptimized cache usage. A straightforward way to handle this problem is proposed and its impact evaluated. The work presented in this chapter is soon to be submitted as a workshop paper and is the object of a patent pending.

Finally, appendix A describes work towards incremental improvements with regard to container downloading and execution that was partially made in the context of a research internship and is the object of a patent pending, while appendix B describes how to build a storage server taking advantage of a user-space network stack to deliver high performance. That work was also the object of a research internship and is also the object of a patent pending.

Chapter 1

What you should know about distributed storage systems

The need for distributed storage systems has driven both industry and academia to innovate on this subject for the last 30 years. This chapter presents a brief overview of the different types of architectures that have been developped in this context, and their strenghts and limitations. It also presents the different issues and tradeoffs inherent to the distributed nature of those storage systems, what solutions exist to solve these. We also show how different systems make different choices in order to provide some guarantees for applications or filesystems running on top of them.

Section 1.1 describes the underlying architectures that serve as a backend for most distributed storage systems, and the impact their inherent structure has on their performance, reliability and flexibility. Section 1.2 explains how distributed storage systems guarantee that data is not lost and is available, even in case of failures, while section 1.3 explains how they deal with the possible inconsistency when multiple copies of same data objects do not match. Finally, section 1.4 presents a non-exhaustive list of widely used distributed storage system and their characteristics with regard to the previous sections.

1.1 The different types of distributed storage system architectures

Distributed storage architectures are numerous and span widely different scales, from a few servers to tens of thousands. Some storage systems are composed of loosely connected anonymous servers while others are centralized organizations with a single master node that has full knowledge and authority on the cluster. This section proposes a classification of storage clusters in 5 main families. This categorization is in no way the only one and some systems might even fit multiple families, but it has the merit of underlining the strength and limitations of each family. Because the storage systems adopting these architectures are varied (filesystems, SQL databases, key-value stores, object stores ...), we call *data object* the basic piece of data on which those systems operate (file, database entry, key-value pair, object ...) for the remainder of this chapter.

1.1.1 Network Attached Storage (NAS) and Storage Area Network (SAN)

A NAS is the most basic form of distribution with regard to storage. It is simply a storage device accessible through a Local Area Network (LAN), usually though an operating system on a host server. It is a commonly used way to share data among users or computers and as a storage backend for devices with small storage devices. The Network File System (NFS) [35], Server Message Block (SMB) [36] and Apple Filing Protocol (AFP) [37] are the most popular protocols used to interact with a NAS. Even though the first version of NFS was developped in 1984, these protocols are still widely in use today. NAS are often used by clients as remote filesystems with a local mount point. More intricate versions of multi-NAS systems include the Andrew File System (AFS) [38] and other filesystems inspired by it such as OpenAFS [39], the Coda File System [40], Intermezzo [41], etc.

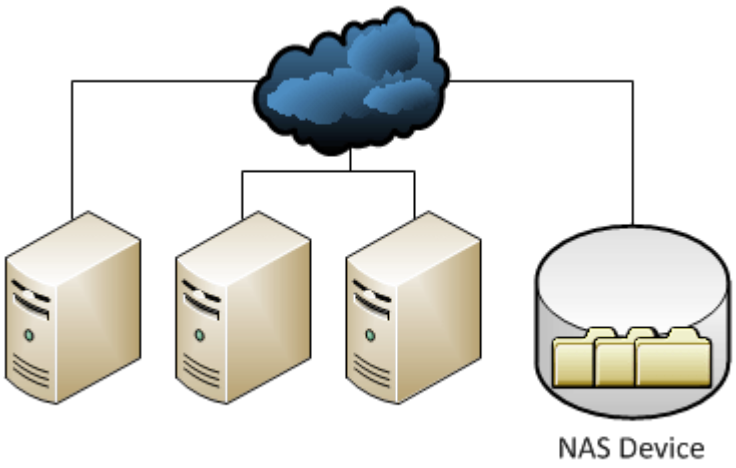
While most NAS are regularly backed up, thus offering some kind of reliability in the face of hardware failure, they are still usually composed of a single storage device or server, which is a throughput and capacity bottleneck. Furthermore, data transfers take place on the LAN and depending on the use-case, it can take a large part of the available throughput.

Therefore, the 1990's saw the emergence of Storage Area Network (SAN) technologies. A SAN is a network purely dedicated to storage. It is composed of several storage devices that are interconnected through dedicated links. These links are most often fibre channel or ethernet, with the servers running the Fibre Channel Protocol (FCP) or the internet Small Computer System Interface (iSCSI) protocol. While SANs are performant, they are also not flexible and require a dedicated infrastructure that is hard to scale. Unlike a NAS, SANs operate at the block level (rather than filesystem).

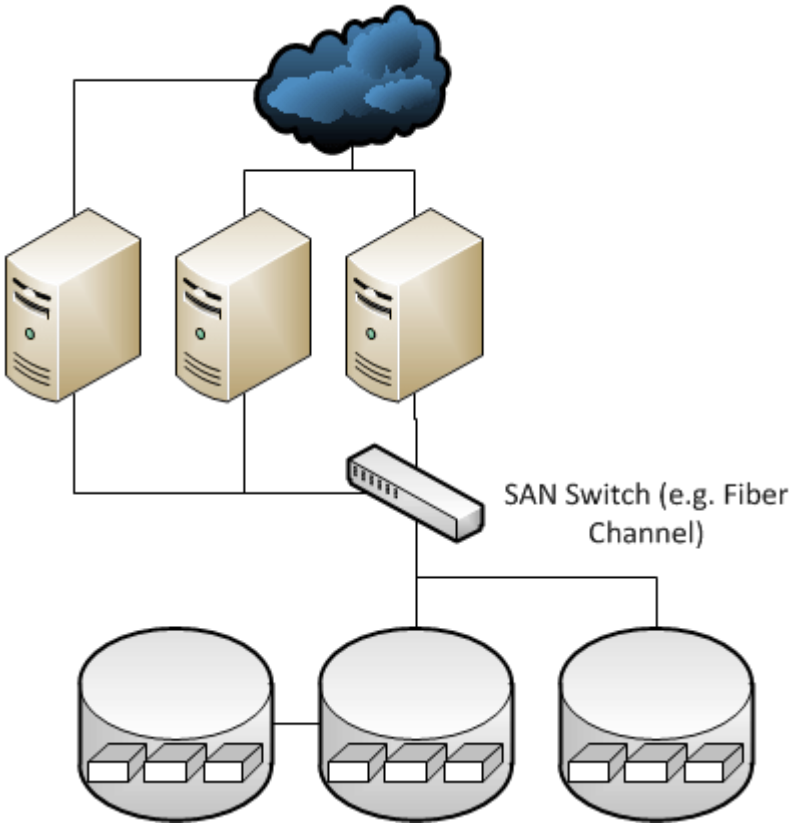
Both NAS and SAN architectures are illustrated in figure 1.1.

1.1.2 Peer-to-Peer (P2P) networks

On the opposite spectrum of NAS and SANs that are local architectures, P2P networks aim at regrouping a large number of anonymous nodes to store and distribute data on a large scale. The first generation of P2P networks that arose around 2000 –Freenet[42, 43], Gnutella [44], Kazaa [45] etc ... – is composed of unstructured networks formed by nodes that randomly form temporary connections to each other rather than follow a global structure, as illustrated in figure 1.2. These systems are designed to be robust to a high churn rate, notably by keeping data blocks on numerous users that in turn share these blocks with new arriving users. However, they are typically not performant as request for data are often flooded and are not sure to be met with success [46]. Furthermore, while popular data can usually be found



(a) NAS



(b) SAN

Figure 1.1: **NAS and SAN architectures.**

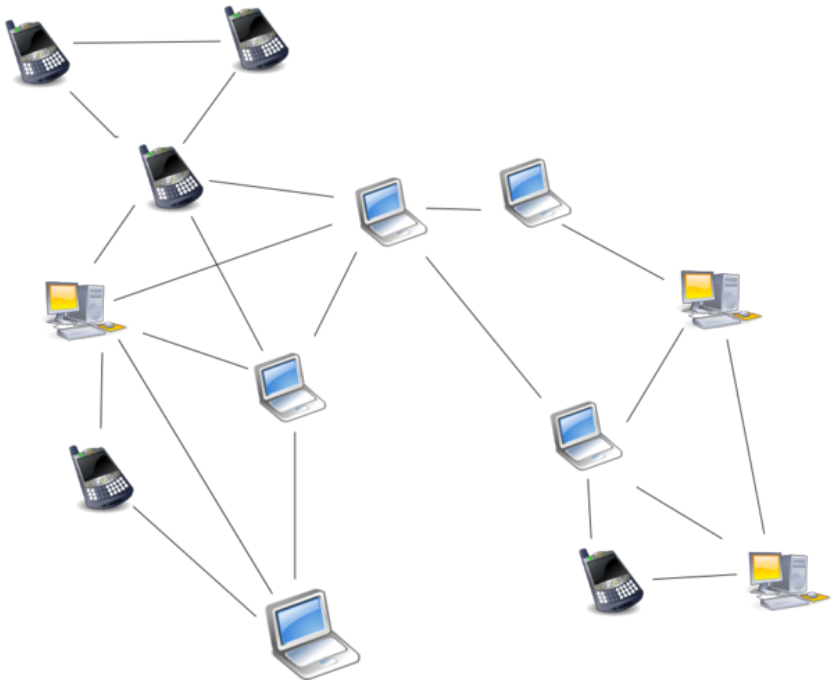


Figure 1.2: **Example of an unstructured P2P network with ad hoc connections between nodes.**

easily, cold data can often remain unreachable because the few nodes storing it are unavailable. It it thus not very well-suited for high reliability.

The second generation of P2P systems revolves around specific topologies that ensure that any node can efficiently route requests to files, even when the resource is rare. Chord [47], Kademlia [48] and Pastry [49] are examples of this approach, which use a Distributed Hash Table (see section 1.1.3) to construct the network overlay assigning resource ownership by consistent hashing (figure 1.3). The consequence of this architecture choice is that they are less robust when the churn rate is high, because that implies frequent rebalancings, as explained in the next section.

1.1.3 Distributed Hash Tables (DHTs)

Distributed Hash Tables (DHTs) are the core of many distributed systems. A DHT is a decentralized distributed system associating pairs of (*key*,*value*). Responsibility for mapping key to values and storing the values themselves is distributed amongst the nodes by assigning ranges of keys to specific nodes as illustrated in figure 1.4. DHTs are not necessarily storage systems, and should in a broader sense be viewed as a consistent way to map resource to servers holding them. As such, DHTs are at the foundation of multiple distributed storage systems.

In a DHT, keys belong to a keyspace (usually *n-bits* strings or an equivalent) that is partitioned to split the keys between the nodes. An overlay network connects the nodes and allows them to find where keys belong. Typically, each data object stored is associated with a name that is hashed

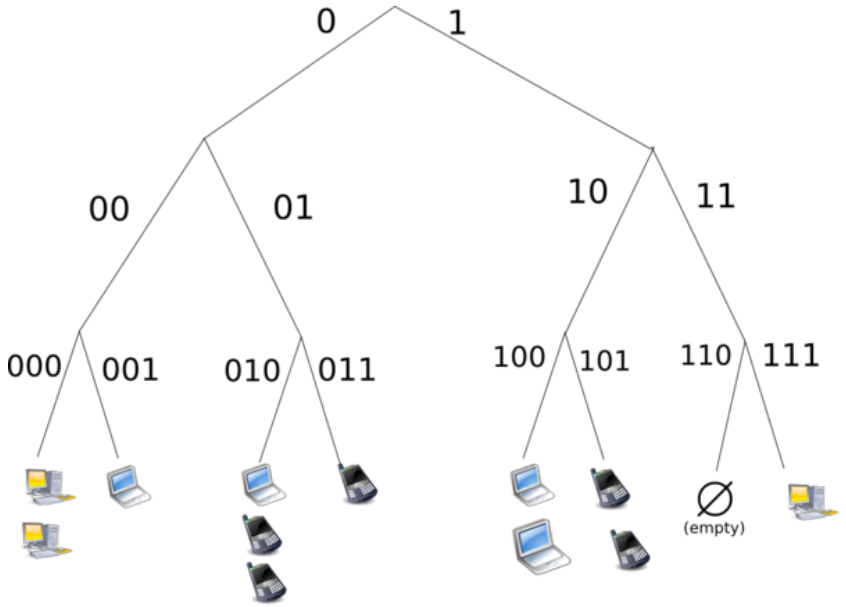


Figure 1.3: **Example of a structured P2P network using a DHT to identify nodes.**

(for example using MD5 or any SHA hash function) to provide the key associated with data itself.

This overlay network is not necessarily a full mesh and queries might have to go through several nodes before reaching the relevant one. The average number of connections to other nodes per node of a DHT is called its *degree*. A DHT composed of n nodes has a degree between 1 and n. A higher degree means that each node has more knowledge of the whole DHT and less redirections are needed for queries, but it also requires more synchronisation and has more memory footprint for DHT nodes. A common trade-off, used for example by Chord [47], is a degree of $\log(n)$ and an average number of redirection per request of $\log(n)$, for n nodes in the DHT. Table 1.1 summarizes the link between degree and average route length, as well as gives several examples of DHTs.

DHTs are designed to scale effectively with the number of participating nodes. However, the scalability of DHTs comes with negative consequences:

Unidimensionality: Because a standard DHT relies on a single-dimension keyspace, it reduces the characterization of a participating node to a single information: the portion of the keyspace allocated to the node. In general, most systems relying on DHTs try to correlate this portion to the storage capacity of the node, so that every node is responsible for a portion of the keyspace equal or close to the proportion of its own storage capacity compared to the sum of the storage capacities of all participating nodes.

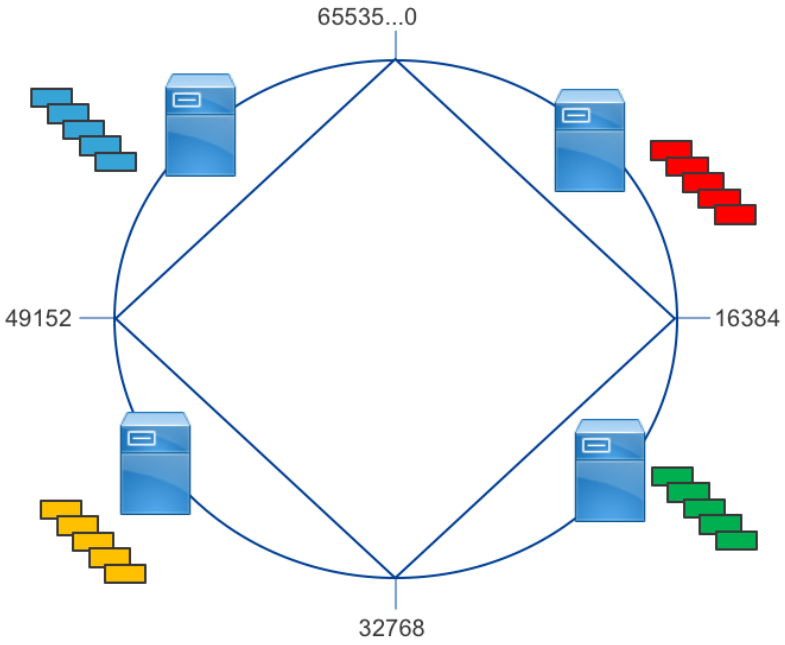


Figure 1.4: **Example of the assignment of the Keyspace of 16-bits strings for a DHT composed of 4 nodes storing 20 equally distributed data objects.**

Degree	Average Route Length	Used in	Comment
$O(1)$	$O(1)$		Longest lookup
$O(\log(n))$	$O(\log(n))$	Chord [47], Kademlia [48] Pastry [49], Tapestry [50]	Common but not optimal
$O(\log(n))$	$O(\frac{\log(n)}{\log(\log(n))})$	Koorde	Complex to implement and less flexibility for neighbours choice
$O(\sqrt{n})$	$O(1)$		Requires constant synchronisation and more memory footprint

Table 1.1: **Trade-off between degree and route length for DHTs.**

This approach doesn't allow for more complex storage systems incorporating storage tiering (storage nodes with different type of storage devices such as HDD,SSD or NVMe). To do that, the storage system has to use different DHTs for different storage tiers and keep track of which object is stored in which DHT. Furthermore, a simple DHT doesn't accommodate the potential heterogeneity in the performance of devices (outside of pure capacity) such as throughput and latency. Even though disparities between the same models of storage devices are not expected to be too high initially, they are expected to grow during the lifetime of a distributed storage system, when failed devices have to be replaced and other devices age differently. This issue has been raised for example in [51] and has led to developing fine-grained load balancing techniques when data is stored as multiple replicas in DHTs.

Finally, this uni-dimensionality also reduces the flexibility of storage systems using them. Namely, a replication or erasure-coded policy has to be system-wide: it is not possible to store different objects under different representations (for example some replicated and some erasure coded) in a DHT.

Re-balancing: In a DHT, servers are assigned key ranges. In general, key ranges are evenly distributed among servers according to their capacity. Consequently, when a new server joins a DHT, key ranges are redistributed so that the new server takes its fair share of the load as illustrated in figure 1.1.3 when adding a node to the DHT of figure 1.1.3. This poses two difficulties.

First, when key ranges change server assignment, all the data objects belonging to these ranges have to be moved. In the ideal use case of perfect balance and perfect reassignment (the only data moved is the data that the new node will serve), if we denote by c the capacity of the node, C the total capacity of the cluster with the new node and D the total amount of data stored in the DHT, an amount of $\frac{c*D}{S}$ data has to be transferred in average. Even worse, this is often not possible if the DHT has constraints on the keyspace assignment (for example contiguity requirements for DHT nodes): a key range reassignment can lead to some key ranges being passed between two servers that were already in the DHT before, like illustrated in figure 1.5. The trade-off is typically between aggregating the keyspace ranges assigned to nodes –leading to more data to rebalance– and optimizing the keyspace rebalancing –leading to keyspace disaggregation and more complexity.

Second, this re-balancing procedure has to be finished before the new node can serve requests and has to happen in parallel with the regular workload of the DHT, if one doesn't want to put it offline. This can lead to very long bootstrap time: in [23], Amazon states that the early iterations of their Dynamo key-value store had boostrap times as high as almost 24 hours when the rebalancing process had to be run in background during periods with intensive workloads. However, this can be prevented by a progressive key reassignment, at the cost of having the joining nodes only slowly getting

to their stable state. This approach has recently been pursued for example in Ceph since the Luminous release in 2017 with the upmap function, that allows to progressively assign bundle of objects to new servers when bootstraping [52].

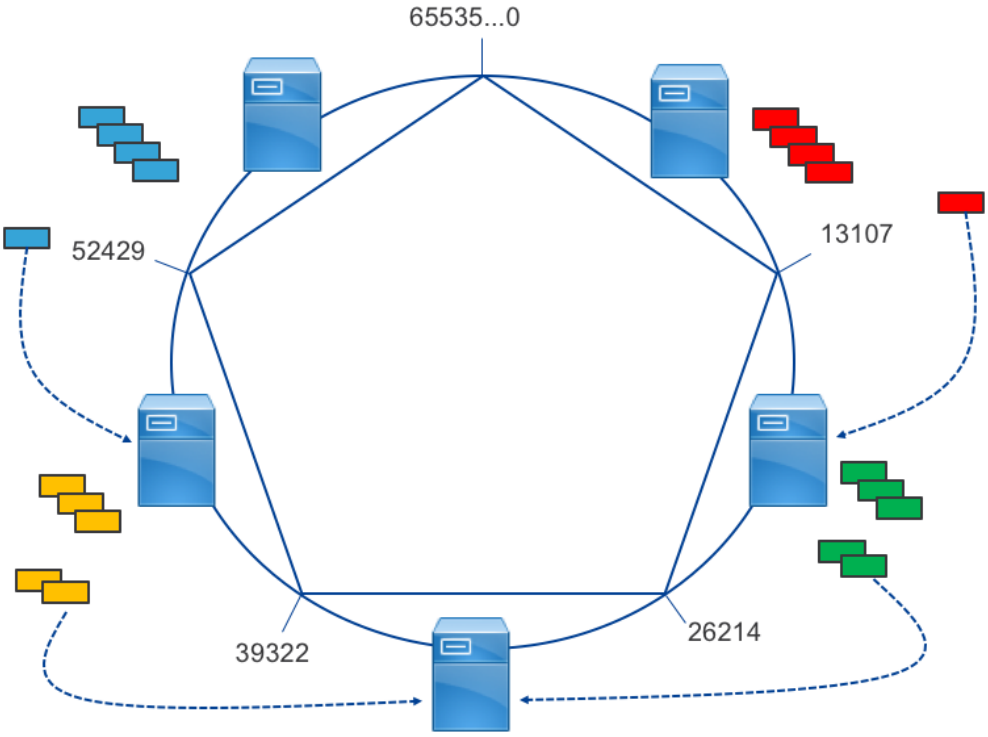
Both these issues have been identified for a long time and can be attenuated or circumvented by adopting more complex structures : separate DHTs for different storage tiers, buckets in DHT-like architectures to isolate different parts of the cluster and allow them to be unavailable for a short time for a fast bootstrap etc... However, this adds complexity to systems implementing these workarounds, that require complex management planes.

1.1.4 Master-Slaves architectures

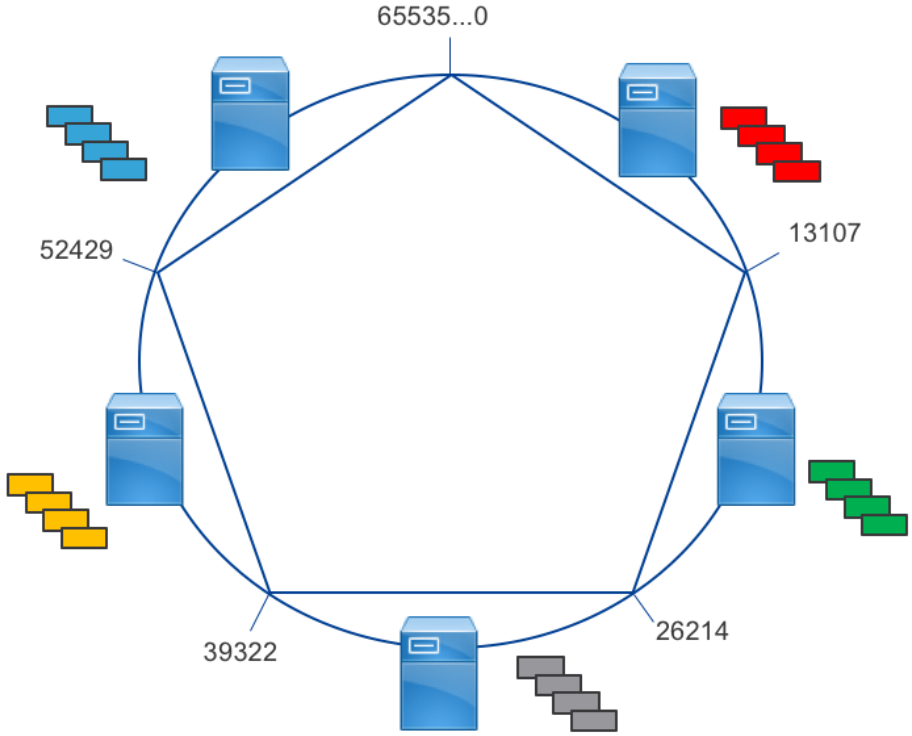
Where DHTs provide a distributed and consistent way to assign data objects to servers, master-slaves architectures put the burden of data placement and indexing on a master server. Requests to write or read data from the system must go through this master, either to decide where the data will be written or to know where to find it. Because data placement is not tied to consistent hash, it can be flexible and allow for more fine-grained policies than DHTs. Furthermore, having a single master reduces the complexity of concurrent operations, since the master can keep track of who is interacting with which data object. For this reason, distributed filesystems often rely on master-slaves architectures since operations on file can range from simple file creation to random writes in different places of the file. The Google File System (GFS) [24] and its open source counterpart, the Hadoop File System (HDFS) [29] adopt this architecture, as shown in figure 1.6.

However, these single master architectures have an obvious scalability limitation. While efforts can be made to increase the number of concurrent client or amount of data that a single master server can handle, like in GFS where numerous optimisations were made (such as very large filesystem blocks – several MB instead of the traditionnal 4KB – and aggressive prefetching), a point comes where the master can not handle too many clients. Moreover, the single master is a Single Point Of Failure (SPOF). For this reason the master is often replicated on inactive masters ready to serve as fallbacks, but the transition period when a master fails can lead to cluster inactivity. These limitations have been observed and discussed both for GFS and HDFS [30, 31], pushing Hadoop to develop a distributed master layer for HDFS [53] and Google to develop a new distributed filesystem named Colossus [54] – on which there is no public information. There are many other single-master filesystems such as QFS [55], GPFS [56], the Global File System [57]...

For these reasons, some filesystems have begun resorting to multi-master architectures. For instance, the Lustre file system [58] supports Distributed Namespace (DNE) since its 2.4 release in 2013. DNE allows subdirectory

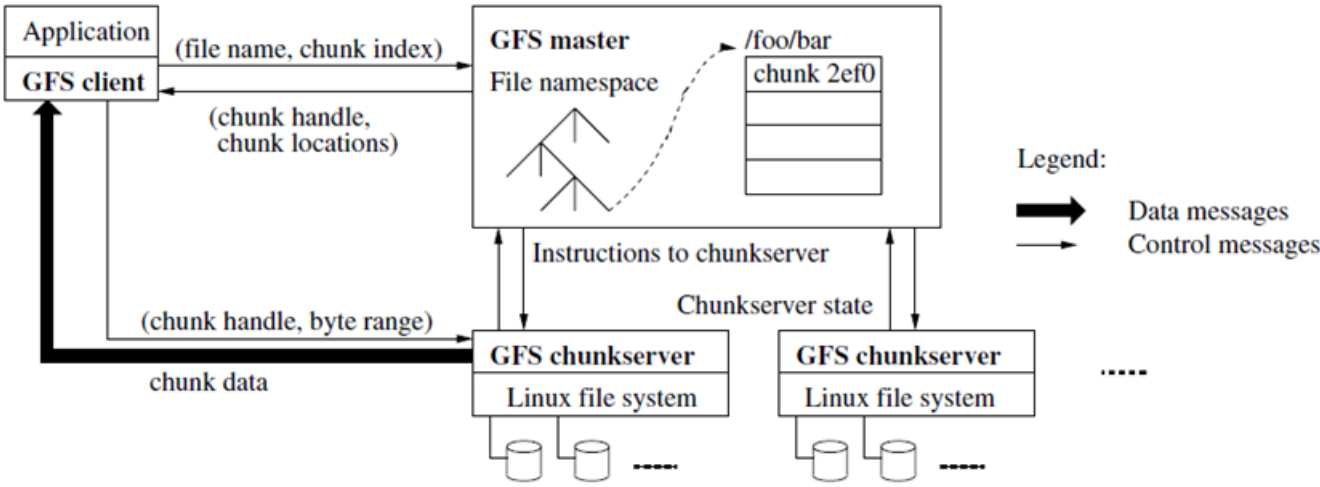


(a) Necessary rebalancing to match the new DHT Attribution.

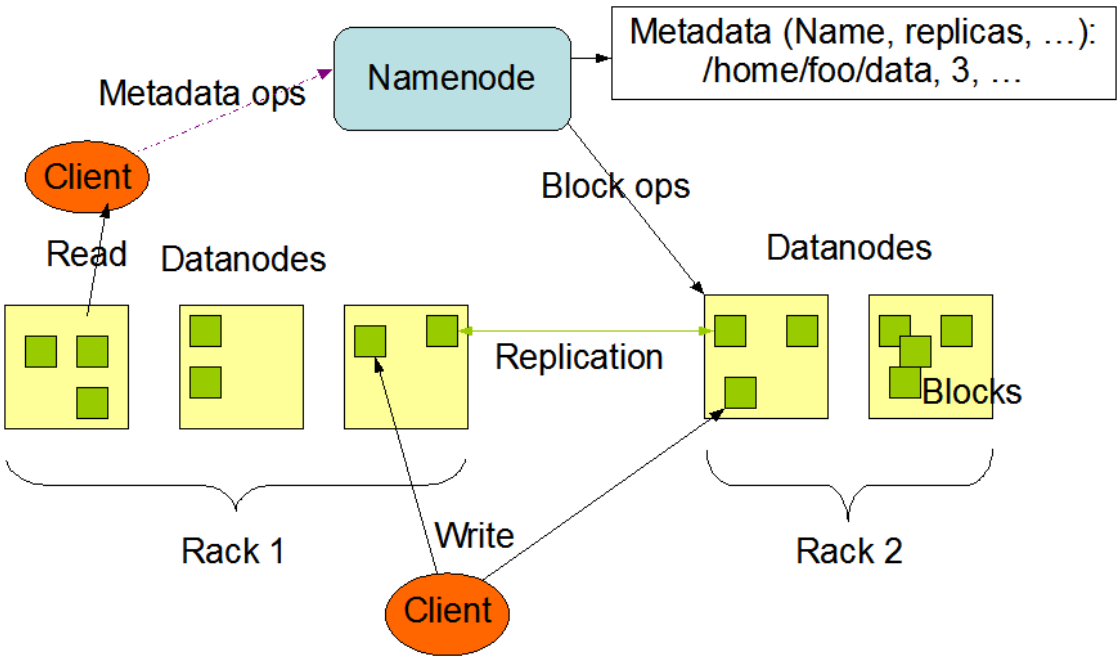


(b) Stable state when rebalancing is done.

Figure 1.5: **A rebalancing has to occur to ensure that data objects are stored on their key's newly attributed storage node.**



(a) GFS Architecture



(b) HDFS Architecture

Figure 1.6: **GFS and HDFS both have a single node dealing with all filesystem metadata operations: the GFS Master and the HDFS Namenode.**

inode trees to be located on separate servers. Other examples of such architectures include zFS [59], OrangeFS [60] or Farsite [61].

1.1.5 Summarize

Table 1.2 summarizes the characteristics of each type of architecture.

Architecture	Type of Application	Used in	Comment
NAS	Shared local storage Backup	NFS [35] AFS [38]	Easy to use/deploy
SAN	Enterprise storage Used for performance		Requires dedicated infrastructure
P2P	Large scale file sharing Decentralized storage	Chord [47] Gnutella [44] IPFS [62]	Weak guarantees Low performance Very large scale
DHT	Scalable enteprise storage Various consistency schemes	DynamoDB [23] Cassandra [27] Ceph [28, 63]	Low flexibility Requires rebalancing High scalability
Master-Slave	Mostly distributed filesystems Constrained environments Flexible	GFS [24] HDFS [29]	Not scalable Master bottleneck Can enforce strong constraints

Table 1.2: **Summarize of the different types of architectures and their characteristics**

1.2 Reliability in distributed storage systems

Most distributed storage systems are used for scalability but also for reliability: one advantage of deploying such systems on multiple servers is that one of these servers failing is not necessarily equivalent to data being unavailable or worse, lost. There are multiple ways to ensure data remains available when failures occur. This section presents the 3 main ones, that cover almost every distributed storage system. For the remainder of this section, we call *reliability* the capacity of a storage system to guarantee its data is available even in the face of failures, and *storage overhead* the ratio between the amount of data written on storage medium and the actual amount of data stored in the system, higher than 1 for reliable systems.

1.2.1 Mirroring

Mirroring is the most basic approach for reliability. It consists in maintaining copies of a full data set in several places. There are several approaches to mirroring but the most common one is backups. Often used in conjonction with NAS, regularly scheduled backups allow not to lose the bulk of data when a storage server fails. However, even incremental backups do not contain the data that has been generated/stored between the last backup and

the failure. Additionally, backups are used to restore systems, not for data to remain available after a server failure. As such, they offer neither strong reliability nor strong availability guarantees.

When stronger guarantees are required, which is usually the case in enterprise environments, the mirrors take part in every storage operation (data write, change or deletion) that changes the data set so that the mirrors are an exact copy of the original data set. This is the case, for example, of the enterprise Microsoft Structured Query Language (SQL) Server [64], a Relational Database Management System (RDBMS), when used in "hot standby" mode. In this mode, the mirror can – as its name implies – be hot swapped to continue operations when the principal server fails. However, these guarantees come at the cost of more latency for basic operations, since the mirrors have to acknowledge every operation before they are acknowledged to the clients rather than regularly lazily fetch the incremental changes.

Mirroring has the advantage of preserving the consistency inside a data set when it is required. It is a desirable property, notably for some SQL databases which have consistency rules between different data items that make them inter-dependant. However, mirroring does not allow for much flexibility since it is a straightforward one-to-one mapping between storage servers.

1.2.2 Replication

Another very common approach for reliability in storage systems is replication: every data object is replicated on multiple storage nodes. The number of times every object is replicated is called the *replication factor r* and is often configurable, with a replication factor of 3 being the industry standard. Replication differs from mirroring by its granularity: where mirroring duplicates an entire monolithic data set, replication duplicates copies of single data objects. It is usually used in larger storage systems where data sets are split on multiple storage nodes, whereas mirroring is generally used on databases spanning only a single node.

While mirroring is mostly used as a backup and hot swap technique, replication is sometimes used along with load balancing techniques to allow for smoother performance [65, 66]. However, with a better granularity than mirroring comes a more complex data placement management. Indeed, mirroring architectures only need to know the list of mirrors which is typically just a few servers, whereas replication-based architectures must be able to know the location of replicas for every object. This is why replication is often used in DHT structures where every data object is associated with several keys, generally through a consistent hash mechanism so that a data object name is enough to know the locations of all its replicas. Furthermore, this mapping from data object name to key sets is often constrained for reliability

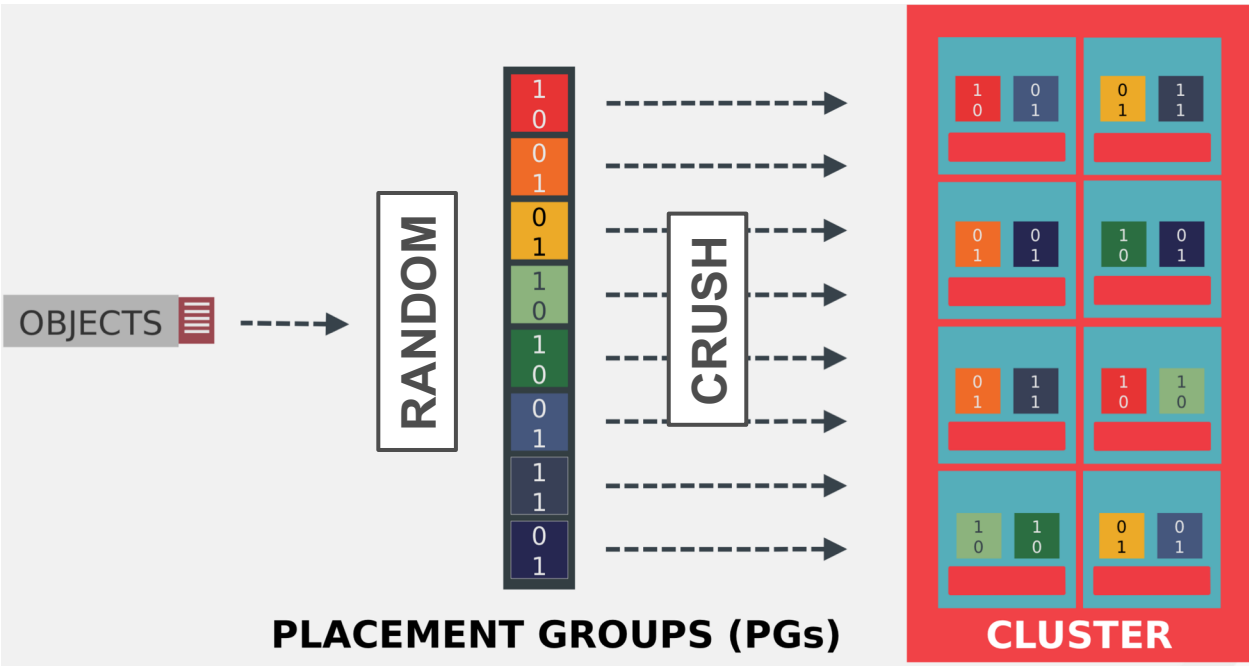


Figure 1.7: **Ceph object-to-server two-step mapping: objects are uniformly mapped to PGs, themselves mapped to OSDs following the rules of the CRUSH map**

reasons.

For example, Ceph [28] assigns objects to Placement Groups (PGs) according to object name hashes with the Reliable Autonomic Distributed Object Store (RADOS) algorithm [63]. The number p of PGs of a storage pool is configured at storage pool creation, and every PG is randomly is associated with an ordered set of Object Storage Daemons (OSDs) following a Controlled Replication Under Scalable Hashing (CRUSH) map [67], each OSD being in charge of a disk partition on a storage node. Moreover, policies can be defined in the CRUSH map to ensure that OSDs assigned to PGs are physically located in different servers, racks, datacenters ... The first OSD assigned to a PG is called primary and is the "master" OSD of this PG: it is responsible for replicating data and is the one accessed for data retrieval (there is no load balancing). This two-step object-to-server mapping is illustrated in figure 1.7.

Thus, the number of PGs is decorrelated from – and usually much lower than – the number of possible combinations of n OSDs. This limits the number of different sets of servers storing the same data. As a consequence, when r OSDs fail simultaneously, it is unlikely that any data is lost, since there is only a chance $\frac{p}{n(n-1)...(n-r)}$ that precisely those r OSDs are assigned to the same PG. However, more data is lost when this unlikely occurence happens. Augmenting p is thus a tradeoff between data loss probability in case of failures and amount of potential data loss. Most large-scale deployments configure $p << n(n-1)...(n-r)$ since it is generally considered worse to often lose a small amount of data than to rarely lose a lot of data.

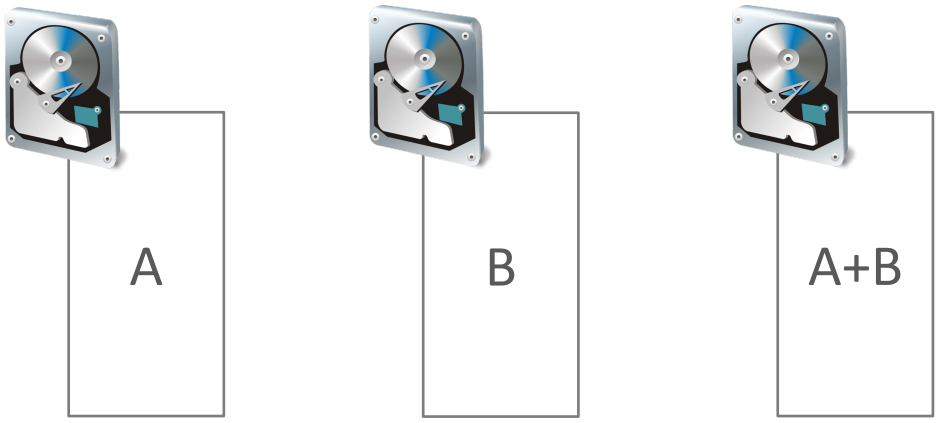


Figure 1.8: **Inter-object erasure code with storage overhead of** 1.5 **capable of withstanding** 1 **disk failure.**

1.2.3 Erasure Codes

Erasure codes (EC) find their inspiration in network codes, used in network transmissions in high loss rate or high latency environment to avoid packet retransmission, at the cost of a fixed overhead. They are the distributed storage system equivalent of local Redundant Array of Independant Disks (RAID) [68], and guarantee a higher reliability than replication techniques for a lower storage overhead. They are almost all derivations of the original Reed-Solomon codes used in many settings [69]. There are three different ways to erasure code data objects:

Inter-object erasure codes: In this approach, instead of purely replicating data objects, linear combinations of objects are stored in addition to objects themselves. When a disk fails and an object is lost, it is possible to rebuild it from the linear combination and the other object(s). The example shown in figure 1.8 shows an inter-object erasure code with a storage overhead of 1.5 able to withstand 1 disk failure.

There are three issues with this approach: objects in the linear combination have to be of the same size (or padding has to be added), objects have to be stored at the same time for the linear combination to be done (although it would be possible to initially store a pure replica of an object, then encode it with another object when one is stored – but it adds complexity to deal with the different possible "encoded states" of objects), and the linear combinations of objects have to be updated when either of the objects is modified or deleted from the system. Inter-object erasure codes are the equivalent of network codes [70, 71] that combine data packets of same size to make sure the recipients can reconstruct data even when some packets are lost in the way.

Intra-object erasure codes: In this approach, objects are split in k fragments of same size that are used to generate $k + r$ fragments of the same size as the original ones in a way that any k of the $k + r$ encoded fragments can be used

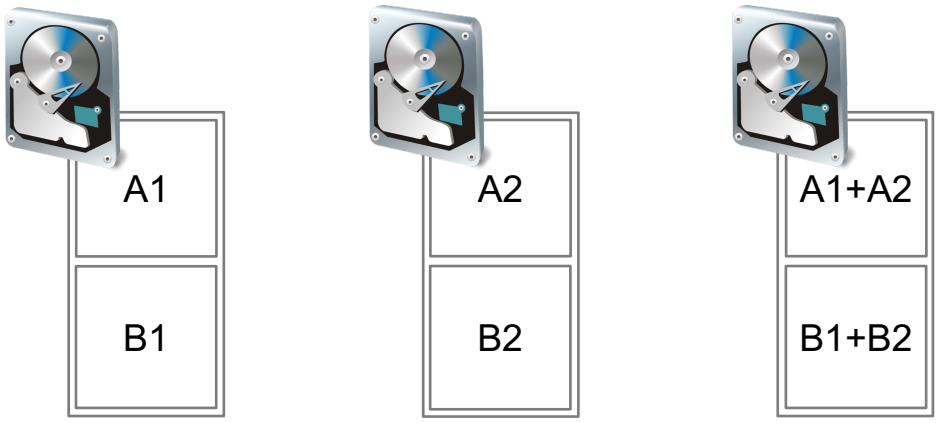


Figure 1.9: **Intra-object $(2,1)$ erasure code with storage overhead of 1.5 capable of withstanding 1 disk failure.**

to reconstruct the object. In most cases, the k original fragments, called the *systematic* fragments, are conserved in the $k+r$ generated fragments – this is however not always the case, and some encodings, such as the Mojette Transform [72, 73], notably used in the distributed file system RozoFS [74], do not store the systematic fragments directly. The r encoded fragments are called *parity fragments* and such a code is called a (k,r) erasure code. It follows that a (k,r) erasure code has a $\frac{k+r}{k}$ storage overhead and can withstand up to r failures. Figure 1.9 illustrates this approach with a $(2,1)$ erasure code. The majority of distributed storage system using erasure codes choose this per-object approach.

Hybrid erasure codes: There exist a variety of other codes that stripe different objects' fragments and combine them to reduce the amount of data required for reconstruction or for other purposes. For example, the Hitchhiker's code [75], implemented in Facebook's HDFS clusters, pairs different object "stripes" (corresponding to encoded fragments) and encodes them together as proposed in [76] to reduce disk and network I/O for reconstructions when compared to traditional Reed-Solomon codes. A simple example of such a hybrid code is illustrated in figure 1.10.

1.2.4 Erasure codes and replication: what is the trade-off

In previous section we saw why erasure codes have a lower storage overhead than pure replication. However, they come with some drawbacks:

- The main drawback of erasure codes is the repair cost. In replicated setups, when a b bytes object's replica is lost, another replica can be fetched to reconstruct it, effectively costing b bytes in both disk and network I/O. This is not the case for erasure codes, that require more bytes for reconstruction than the amount of data reconstructed. In the

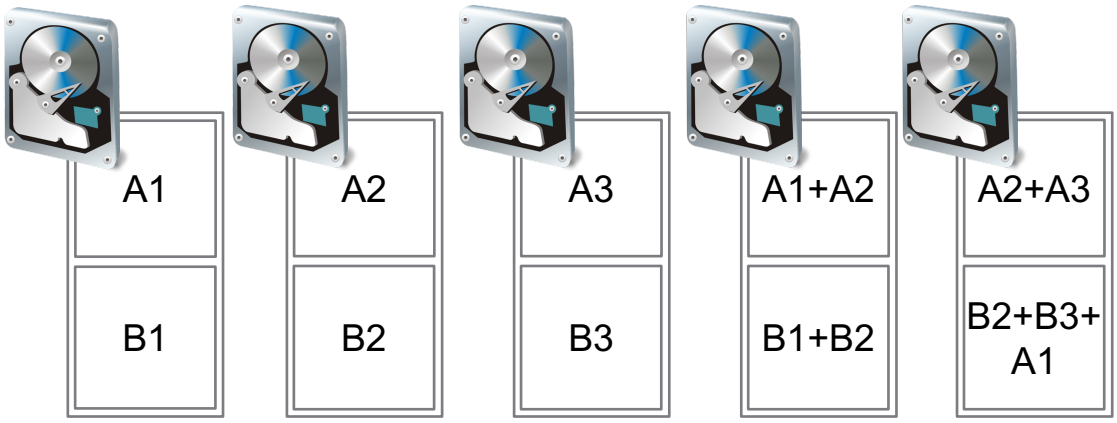


Figure 1.10: **Hybrid $(3,2)$ erasure code with storage overhead of $5/3$ capable of withstanding 2 disk failures.**

examples of figures 1.8 and 1.9, twice as much data as what is lost is required when a disk fails (B and $A+B$ if A is lost in the first case, $A2/B2$ and $A1+A2/B1+B2$ if the first server fails in the second case).

This is why, in addition to the "k out of $k+r$" repairability property, it is desirable for erasure codes to also have the Maximum Distance Separable (MDS) property, minimizing the amount of fragments required to reconstructed missing fragments. Most of the work in the field aims at designing codes that minimize the required disk I/O, network I/O or both at the same time required for fragment reconstruction such as Regenerating Codes (RGC) [77, 78, 79, 80], hierarchical codes [81], or fountain codes [82, 83, 84].

- Storing data as encoded fragments requires computation. Most erasure codes use simple XOR operations but this is not always the case [72]. Moreover, since all data is not located on a single server, k separate connections usually have to be opened to retrieve the data, which is very impactful on storage systems where the network stack usually amount for a large portion of the CPU time used – more than 50% in some cases [85].

- Furthermore, there is added complexity in handling object fragments rather than replicas. First, it requires keeping track of more separate data entities since there are more fragments than object replicas. Second, every data operation requires the participation and synchronization of more servers. Finally, it is impossible or inefficient to make relevant local decisions since fragments represent only a fraction of the data. This complicates, for example, the caching strategies of erasure coded systems, as elaborated on further in chapter 5.

For these reasons, some work has been done on hybrid erasure code/replication architectures [86] to obtain the best of both worlds. However, such approaches can inherently not have a storage overhead lower than 2.

1.3 Consistency and consensus

The previous section presented how distributed storage systems guarantee reliability in the face of failures. However, this comes with a significant drawback: when several copies of data are stored in a storage system, it is hard to make sure that all replicas are consistent with each other. For the remainder of this section, we will suppose that our storage system is reliable through replication. However, most concepts discussed here could be translated in erasure coded setups with additional encoding steps.

First, it is important to make a distinction between consensus and consistency:

- A *consensus* is reached when all members of a storage system have the same version for the object replicas they store. For mirroring-based architectures, it is when all members store the exact same data. For replicaton-based architectures, it is when all replicas of every object are identical.
- *Consistency* is the ability of a distributed storage system to converge – slowly or quickly – towards a consensus. A storage system with strong consistency will quickly converge to a consensus whereas weak consistency means that such a consensus might not even be reached.

Throughout the last decades, there have been multiple theorems, formulations, acronyms ... describing consensus or consistency characteristics, guarantees or algorithms. These definitions often come from different backgrounds and sometimes even use the same words to describe different concepts, which can be confusing. This section aims at describing the most well-known definitions and concepts, what they entail and to what context they apply.

1.3.1 Theoretical frameworks

Consistency and Availability: the CAP theorem

The CAP theorem – for Consistency Availability and Partition tolerance – is one of the most well-known theoretical result on distributed system. It formally formulates the intuitive fact that a distributed system that is split in two (or more) parts unable to communicate with each other can not provide availability for every operation on both sides of the separation and consistency in the request answers. For example, a data object which would be stored in a distributed storage system as multiple replicas either has all replicas on the same "side" in which case it is unavailable on the other "side" or has replicas on both sides in which case it is impossible to ensure that a modification triggered on on side is propagated to the other, as illustrated in figure 1.11.

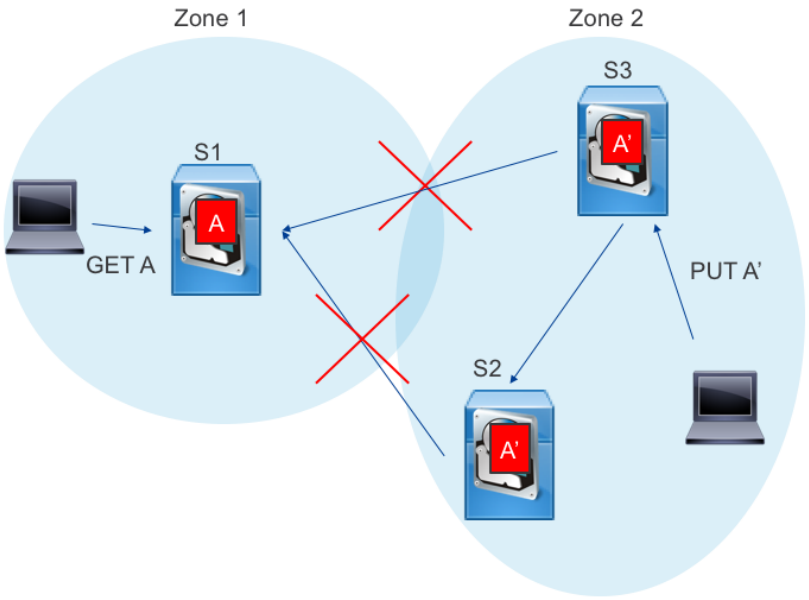


Figure 1.11: **Distributed storage system network partitioned in 2 zones. In this case, the CAP theorem states that it is impossible to guarantee that object A is available in both zones *and* that all replicas of A are identical.**

This intuition has first been formally described in [87] in 1999, presented as a conjecture in [88] in 2000, then proven, effectively becoming the CAP *theorem* [89] in 2002.

Formally, it states that a distributed data store can't simultaneously provide the three following guarantees:

- **Consistency:** Every data read receives either an error or the latest write.
- **Availability:** Every request receives a valid response (with no necessary guarantee of being the latest version).
- **Partition tolerance:** The system keeps operating when a partition happens in the system. A partition is defined as an arbitrary number of messages being dropped or delayed between nodes composing the cluster.

Note that the consistency of the CAP theorem amounts to a consensus on every data object as defined in this section's introduction. Because failures are bound to happen, this theorem states that a distributed store has to choose between consistency and availability **in the event of a partition**. As explained in [90], this theorem has often been misinterpreted as the fact that data stores can not guarantee consistency and availability simultaneously under normal conditions. This is however not the case since a partition is not "normal conditions" for such a storage system.

This theorem can be misunderstood to justify why some distributed storage systems provide only eventual consistency instead of strong consistency (the consistency of the CAP theorem). It is perfectly possible for a distributed system to provide strong consistency and availability at the same time as long as there is no network partition, which represent only a fraction of failures [91]. However, systems providing only eventual consistency, even in normal conditions, do so because of another intrinsic tradeoff of distributed systems between consistency and latency.

This tradeoff has been explored in [92] and has led to an extension of CAP called PACELC. This acronym states that in the case of a Partition, a distributed system has to choose between Availability or Consistency (this is the usual CAP formulation), but Else, it has to choose between Latency and Consistency. This trade-off, while intuitive, will be further explained in section 1.3.2.

Database characteristics: ACID and BASE

ACID (for Atomicity, Consistency, Isolation and Durability) is a set of properties initially described in [93] (in 1981, without Durability) then in [94] in 1983 that, when they are verified by a sequence of database operations, define a database *transaction*. A transaction can be perceived as a single logical operation on data, a typical example being a fund transfer between banks. These properties guarantee the following:

- **Atomicity** requires that the transaction either succeeds or fails. In other termes, operations in the transaction either all succeed or all fail, even in the case of errors, power failures or crashes.
- **Consistency** guarantees that a transaction changes the database from a valid state to another. This means that any constraint (such as the unicity of keys in a database) applying to the database can't be violated by a transaction.
- **Isolation** ensures that concurrent transactions have the same effect on the database as if they were executed sequentially.
- **Durability** states that a committed transaction stays committed in the event of power loss, crash or error. This is mostly obtained by recording transactions on non-volatile memory before acknowledging the commit.

Even though ACID properties were not created with distributed systems in mind – ACID's consistency derives from constraints between different objects, not from differences between a single object replicas –, they can still apply to distributed databases. Moreover, I felt they were a worthwile inclusion in the list both to avoid confusion on the different meanings of *consistency* and because another set of properties – called BASE and more relevant

to distributed systems – was defined in opposition to ACID.

Typically, ACID's properties are exhibited by relational databases. Because of the Consistency and Isolation properties, ACID operations can't complete when a database is partitioned (when some components of the database can't interact due to network failures on any other likewise event). Therefore, distributed databases guaranteeing ACID transactions choose *C* over *A* in the CAP theorem.

It follows that the drawback of a distributed database following these principles is a reduced availability in the presence of failures or delays, as well as generally low performance and high latency per operation because inter-servers locking mechanisms are required. Some systems can accommodate less strict consistency, and for those, a set of semantics called BASE (for Basically Available, Soft state, Eventual consistency) have been devised:

- **Basically available** means that any request to the system gets an answer. However, that answer may be a failure to obtain the data or the data may be inconsistent between requests.
- **Soft state** means that the state of the system can change over time, even without input. The reason for this is that an input can be partially processed and acknowledged to the emitter before the whole system updates the corresponding state.
- **Eventual consistency** guarantees that if the system doesn't receive inputs, it *eventually* becomes consistent when every soft state has been propagated to the whole system.

Systems that follow the BASE semantics are generally more complex to build upon since applications have to be designed knowing the potential data inconsistency. However, the lack of strong consistency allows those systems to generally provide better performances with regard to latency and availability. Non-relational databases often (but not always) exhibit the BASE properties to guarantee a high availability.

Client-centric and data-centric consistency models

We saw that the CAP theorem only describes the reaction of storage systems to partitions, while ACID and BASE are mostly used in the context of databases. Therefore, two perspectives on consistency for generic storage systems have emerged over the years [95]. Client-centric consistency models view the storage system as a blackbox and describe the guarantees offered to an external client, while data-centric consistency models refer to the internal state of the system and the synchronization processes at stake. This section describes the most used models of both approaches.

Client-centric consistency:

- **Monotonic Read Consistency (MRC)** guarantees that a client that has read a version n of an object can only read ulterior versions of this object afterwards (including the n^{th}). This is useful for applications having chronology consistency requirements. This guarantees, for instance, that the following occurence can not happen: person A wires money to person B who sees his account credited, B later tries to wire money to C but can not because the account version he sees dates from before A's transfer was credited.

- **Read Your Writes Consistency (RYWC)** guarantees that a client that has written a version n of an object can only read ulterior versions of this object afterwards (including the n^{th}). This is particularly useful for applications in which a user could issue the same requests multiple times because he is under the impression that the request has not been registered, causing either superfluous load or sever inconsistencies: A wires money to B. Later, A's account does not reflect the transfer, so A assumes the transaction has been lost and transfers the same amount again.

- **Monotonic Write Consistency (MWC)** guarantees that two writes from the same client will always happen in the same order that the client submitted them. This is useful when working on different data objects that are related. For example, the following can not happen with MWC: A corrects B's banking information, then wires him money. But the transfer took the old B's banking information as a reference and was sent to the wrong person.

- **Write Follows Read Consistency (WFRC)** guarantees that a new version of an object which version n was previously read will only execute on ulterior replicas. This extends MWC to objects written from other clients when their write has been read at least once.

Data-centric consistency:

- **Weak consistency** does not provide any guarantee. It is used to describe systems that might update from time to time but with no guarantee. The best of example of such a system is a browser cache.

- **Eventual Consistency** guarantees the same consistency as BASE. It is the most adopted consistency scheme for distributed object stores because it allows for great scalability and performance while providing consistency in the vast majority of cases.

- **Causal Consistency** is the strongest possible consistency model for systems aiming for availability rather than consistency with regard to the CAP theorem. It ensures that requests that have a causal relationship with one another are executed in the same order on all replicas. However, this requires a complex dependancy tree and thus adds a computing and complexity overhead.

- **Sequential Consistency** can not be achieved in always available systems. It extends causal consistency to all requests. All requests must be executed in the same order on all replicas. Additionally requests from the same client must be executed in the order they are received by the storage system. Sequential consistency is typically a level of consistency reached using the Paxos algorithm described next section.
- **Linearizability** extends sequential consistency to multiple clients: all requests mut be executed in the order they arrived to the storage system. With sequential consistency, it is possible for two clients requesting a read at the exact same time to have different data, for instance if one of the clients overwrote the object in a precedent request but the write was not acknowledged yet, in which case this client's read must wait for the write to complete whereas the other client has no such blocking pending request. Linearizability is impossible to reach in distributed systems but can be approached by precise clock synchronization.

Client-centric and data-centric consistency models do not have a perfect mapping from one to each other, but some work has been done to analyze the impact data-centric models have on client-centric ones [96]. Furthermore, multiple other consistency models have been proposed over the years [97, 98].

Finally, extensive work has been done to evaluate, measure and verify the consistency of distributed storage systems [99, 100, 101, 102, 103, 104, 105].

1.3.2 Consensus and consistency: how to reach it

The two previous subsections explained how consensus and consistency properties can be described and what they mean for distributed storage systems. This section describes two algorithmic tools used to either reach consensus or guarantee custom levels of consistency.

Consensus algorithms: Paxos and Raft

Consensus protocols are a fundamental aspect of state machine replication in distributed computing. They ensure a group of members of a distributed system can reach a consensus on a value, even in the presence of failures and delays. Paxos, firstly published in 1989 [106] and later republished as a journal article in 1998 [107], is the most well-known algorithm guaranteeing the perfect consistency of states in an asynchronous network as long as certain conditions are maintained (typically as long as a majority quorum of members can be obtained to elect a leader). It has been proven that under the following assumptions, consensus is guaranteed:

- Processors in the nodes operate at arbitrary speed, can fail, may re-join the cluster after failures with their last version of the states (if they have persistent storage), but do not actively lie or try to subvert the protocol.

- Processors can send message to any other processor, but messages are sent asynchronously and may take an arbitrary time to reach their destination. They can also be lost, reordered or duplicated, but are delivered without corruption.

Under these conditions, Paxos guarantees the three following safety properties for the distributed state machine:

- Only values proposed by a member of the quorum can be learned.
- Two different members can't learn two different values.
- If a value is proposed, every member eventually learns it (if it joins the cluster and enough members remain non-faulty).

However, the way Paxos is proven and described in the initial papers [106, 107] makes it complex and difficult to understand and implement it, as shown by several papers dedicated to either simplifying the explanations [108, 109] or detailing specific implementations of variants of Paxos [110, 111]. To this end, other consensus algorithms such as Raft [112] have been invented and designed with ease of implementation in mind. But even with proven algorithms such as Paxos and Raft, it can be tricky for actual implementations to provide the guarantees provided by the theoretical algorithms [113].

All these algorithms behave mostly in similar ways: they periodically elect a leader that is in charge of *proposing* new values and making sure that a majority of the participants agree on the value before commiting it. When the leader changes, a replicated log ensures that what was previously committed stays committed. Raft is the base for the widely-used ETCD [114] distributed key-value store.

It is important to point out that most of these algorithms guarantee nothing in case of byzantine-type failures – which is to say when nodes can voluntarily lie and messages can be corrupted.

Latency and Consistency, the (N,W,R) quorum model

Section 1.3.1 detailed the CAP theorem and its extension, PACELC. The (N, W, R) quorum model is used in replication-based distributed storage systems to determine when a read or write operation should be acknowledged as successful to the client, depending on the number of individual replicas for which the operation has been acknowledged.

The model goes as follows:

- Every data object is stored as N replicas;
- A write returns a success to the client when $W \leq N$ replicas have acknowledged the write;

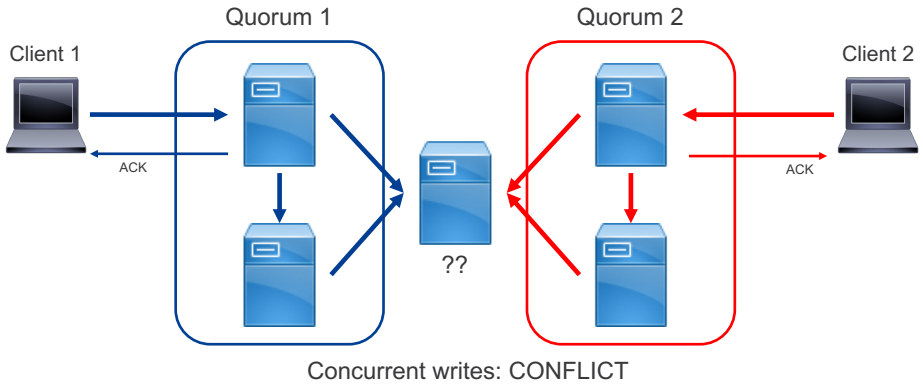
- A read returns the data to the client when $R \leq N$ identical replicas have been fetched.

It has the following properties:

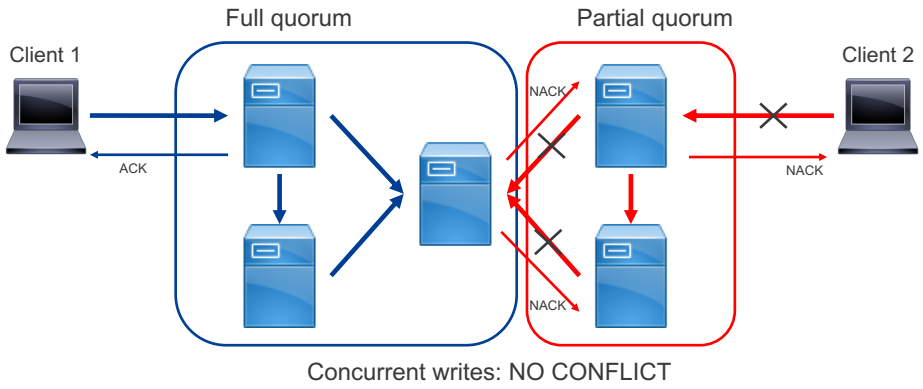
- Increasing W and R respectively increases the write and read operations latency, since more answers are required before the operations return.
- If $W + R > N$, the system is fully read-after-write consistent: a read sent after a write has returned will never return a stale version of a data object. In that case, there is at least one replica overlap between a write and read operation, which means the storage system can determine that there is a version conflict. Techniques exist to resolve such version conflicts such as timestamps or vector clocks [115, 116].
- If $W < \lceil \frac{N}{2} \rceil$, an additional mechanism is necessary to resolve concurrent writings conflicts. Indeed, if a quorum can be formed with only half or less of the nodes, two distinct quorums can emerge for different versions of the same object, as illustrated in figure 1.12. Again, vector clocks and timestamps can help resolve such problems. On the contrary, when $W > \lfloor \frac{N}{2} \rfloor$, it is impossible for multiple concurrent writes to be acknowledged in the system.

  However, such a system still requires a mechanism to resolve deadlock situations, such as 3 concurrent writes, none of them obtaining a majority like in figure 1.13. This kind of deadlock can be solved by random timeouts triggering a failure return, as is done in Raft for the leader election phase [112].
- Increasing N increases the reliability of the system but also its storage overhead. At R and W fixed, it also increases its availability but reduces its consistency.
- Increasing W increases the reliability of the system since it reduces the probability that all storage nodes that have acknowledged a write simultaneously fail before the write is propagated to other replicas.
- Because R and W are separate parameters, the system can be available for an operation but not for another. Typically, many storage systems adopt a $(N, N, 1)$ configuration. In that case, the system is always available for reads (if the client can communicate with at least one member storing the desired object) but never for writes if there is a partition that isolates at least one storage server storing one of the object's replicas.

Most of the points made here are true when the replicas are indiscriminate. This is however not always the case. For example, Ceph's writes return when the N replicas have been written but reads only target the *primary* replica (the one stored on the first OSD of the object's PG, as explained in



(a) N=5,W=2: Conflict during concurrent writes: one replica in undetermined state and two different writes have returned.



(b) N=5,W=3: No conflict during concurrent writes: a write that succeeds guarantees that no other simultaneous write does so.

Figure 1.12: $W > \lfloor \frac{N}{2} \rfloor$ **guarantees the impossibility of concurrent and distinct writes to be simultaneously successfull.**

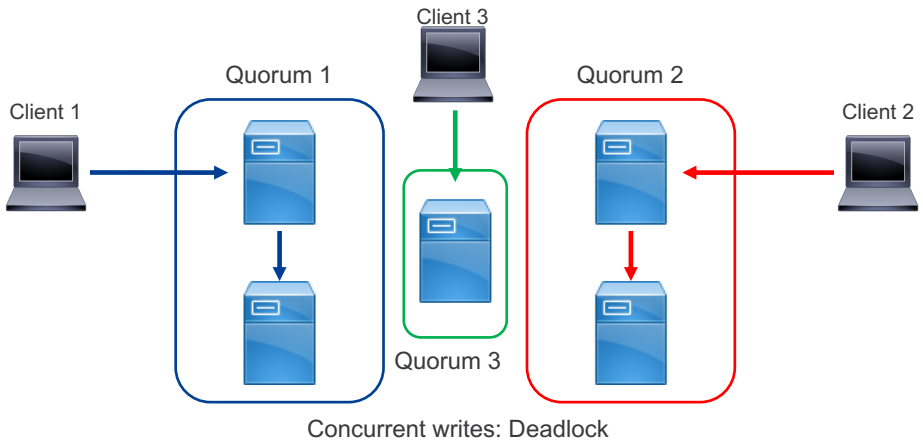


Figure 1.13: **N=5,W=3: three concurrent write requests, none obtained a majority quorum.**

section 1.2.2.

The N,W and R values are often parameters of storage systems, configured to fit relevant use-case. For instance, Amazon states that they tune Dynamo [23] differently depending on the use-case: $(3,3,1)$ for many-reads-few-writes applications, $(3,1,R)$ for applications that need a high write-availability (such as the service hosting the shopping carts of clients), and $(3,2,2)$ for their average deployment. They also explain that they use timestamp-based reconciliation when there are divergent object versions.

1.4 Examples of distributed storage systems

This section gives a non-exhaustive overview of different distributed storage systems and their properties in table 1.3. They are sorted by type of storage application. This classification is partly subjective – for instance, key-value stores are a specific type of database that store non-relationnal data – and is just supposed to give a hint into the use-cases for which they are used.

When explicitely described in available resource, reliability and consistency mechanisms and guarantees are given. The nomenclature depends on the use-cases and some systems fit different classification (such as BASE and (N,W,R) quorumn since BASE describes consistency guarantees and the quorum model is a tool to provide such guarantees). Most systems using a (N,W,R) quorum model offer the possibility to configure N,W, and/or R. When this is not the case, the fixed value for these parameters is given.

Type of storage system	Storage System	Underlying architecture	Reliability mehanism	Type of consistency mechanism	Notes
Filesystem	GFS [24]	Master-Slaves	Replication Mirroring for metadata	(N,N,1)	Relaxed consistency model: tolerates duplicate writes
	HDFS [29]	Master-Slaves Masters-Slaves [53]	Replication Mirroring for metadata	(N,N,1)	Distributed metadata layer ongoing work, inspired from GFS
	GlusterFS [117]	DHT	Mirroring Replication EC	(N,N,1)	
	LustreFS [58]	Master-Slaves Masters-Slaves (since 2.4)	Mirroring Replication	(N,1,1) (N,N,1)	
	IPFS [62]	P2P DHT	NA	NA	Possible to locally store some objects to ensure reliability
Database	Aerospike [118]	DHT	Replication	ACID	
	Cassandra [27]	DHT	Replication	BASE (N,W,R)	
Key-value store	Dynamo [23]	DHT	Replication	BASE (N,W,R)	
	Riak [119]	DHT	Replication	BASE (N,W,R)	Open-source, inspired from Dynamo
	Ceph [28]	DHT	Replication EC	BASE (N,N,1)	Reads issued only to first replica
	ETCD [114]	Mirrors	Mirroring	Raft	Used to propagate configuration information
	ZooKeeper [120]	Mirrors	Mirroring	ZAB (Raft equivalent)	Used to propagate configuration information

***Table 1.3:* Overview of different distributed storage systems and their characteristics.**

Chapter 2

6Stor, an IPv6-based fully distributed object store: architecture and evaluation

As explained in the introduction, distributed storage systems are mostly deployed on numerous generic servers that are organized in clusters to provide reliability, availabality, and sometimes some performance or consistency guarantees. These clusters are generally organized in three layers.

The metadata layer has the role of distributing data objects amongst the various storage devices and of keeping track of object location, and the data layer is in charge of storing the objects themselves, of sending them to clients when requested. Finally, the control plane organizes the cluster, configures the servers, deploys the appropriate software, and in some cases detects server failures and trigger appropriate repairs or rebalancing.

We saw in section 1.1 that every traditionnal storage architecture has strengths and shortcomings. For this reason, we created 6Stor, a distributed object store that adresses most of these shortcomings. 6Stor uses IPv6 addresses as a means to improve its scalability and flexibility, and is composed of as few software layers as possible.

Through this approach, our goal is twofold: first, we identify key traditional design points in distributed storage systems that affect their flexibility, performance and scalability. Secondly, from our previous considerations, we build from scratch an architecture that circumvents these limitations and evaluate the feasibility of this architecture.

The rest of this chapter is organized as follows. Section 2.1 explains in more details the issues we are trying to solve with 6Stor. Then, section 2.2 thoroughly describes 6Stor's architecture and how basic operations are processed. Section 2.3 presents how 6Stor scales up and down without triggering heavy rebalancing and thus long bootstraps. Reliability and reaction to failures is explored in section 2.4 before general considerations on 6Stor's structure and architecture are pointed out in section 2.5. A set of benchmarking results from various workloads is presented in section 2.6, showing how a prototype of 6Stor fares when compared to a similar object store. Lastly, the

performance impact of the HTTP protocol (and its absence in 6Stor's data plane) is evaluated before conclusions are drawn.

2.1 Why we built 6Stor from scratch

There are two main issues we try to tackle with 6Stor: software layering and architectural limitations.

2.1.1 Software layering

Because the main bottleneck for storage systems has been disk I/Ops for a long time, most distributed storage system were built assuming that compute was not a problem and network a secondary problem. But as explained in the introduction, the evolutions of storage technologies seen in figures 1 and 2 mean that this assumption does not hold true anymore [6, 7]. Therefore, we wanted to build a storage system from scratch rather than iterate on an already existing open-source architecture such as Ceph [28], which newly-founded foundation [121] has started to look into in order to reduce the software computing overhead [34, 33].

But layering can have more pernicious consequences and pose real issues with system configuration. For example, Ceph OSDs that back their data on a regular filesystem generally have very poor write performance because of data journaling. In some extreme cases, data can be written on disk as much as 13 times per object [32, 85] because of redundant journaling mechanisms – from Ceph itself, the underlying filesystems etc...

2.1.2 Architectural reasons

Almost all distributed storage systems correspond either to DHT or master-slaves architectures as depicted in section 1.1. These architectures come with drawbacks that we wanted to avoid. In order to better understand both architectures, we have chosen to describe in more details one widely deployed and documented storage system of each category – Ceph [28], based on a DHT and GFS [24], built as a master-slave system.

2.1.3 Ceph

Ceph [28] is a distributed object store, that can also provide a block device and a distributed filesystem on top of its object store. Ceph is fault-tolerant and self-repairing via replication and automated repairs. A Ceph cluster uses three or four types of agents (the last one is only used whenever a filesystem is needed):

- Cluster monitors, that maintain a representation of the topology of the storage system (cluster map), detect node failures and start repairs when required. It contains information regarding the IP addresses of storage

nodes, their weight (used to distribute data according to each node capacity) and other similar information about these nodes.

- Object Storage Devices (OSDs), that store objects in Placement Groups (PGs) on a local filesystem (usually XFS or btrfs) and communicate between each other for replication, repairing or rebalancing purposes.
- Proxy Gateways that are used to connect to the storage cluster. The cluster monitors make sure that the gateways always have an up-to-date cluster map in order to reach the cluster.
- A metadata agent, that stores filesystem inodes (metadata about files and directories) when Ceph is used as a filesystem. [1]

Ceph does not need object metadata thanks to the way data is balanced across the cluster using the RADOS [63] and CRUSH [67] algorithms. Rados spreads data evenly among PGs, and CRUSH assigns each PG to several OSDs, following numerous constraints such as OSD weights, failure domains, replication rules etc... The weight is usually set based on the storage capacity, so that a storage node with more capacity is assigned more PGs. The cluster map, maintained by monitors, organizes storage devices in different buckets that usually correspond to some physical topology: buckets can correspond to racks, rows, rooms ... and is used by CRUSH to map PGs to OSDs. Different buckets are used to store replicas of objects to ensure that not all replicas are unavailable at the same time in case of disaster or maintenance.

To place an object, the gateway deterministically hashes the object name to determine which PG will store the replicas by applying a placement policy to the cluster map. This provides a deterministic and independent way to place and retrieve replicas and ensure that two replicas are not on the same failure domain (i.e. buckets). The hash is designed such that every storage node holds an average fraction equal to the node's relative weight $\frac{w}{W}$ of the data, with w the node's weight and W the total cluster's weight. Ceph's placement algorithm was illustrated earlier in figure 1.7.

This deterministic placement has the benefit of allowing the system to run in a fully decentralized manner. However, this comes with two drawbacks. First, when the storage topology changes, for instance when nodes are added to the cluster, or nodes are permanently removed from it, data has to be moved to fit the new distribution of the hash function on the new cluster map, as illustrated earlier in figure 1.5. If we suppose that the relative weight $\frac{w}{W}$ of a device is equal to the relative storage capacity of the device $\frac{c}{C}$ (c the storage device capacity and C the total capacity of the cluster), an OSD has to store an average of $\frac{cS}{C}$ of data, where $S < C$ is the amount of

[1] Agents can be colocated on the same server. For example, monitors can also act as gateways but don't function properly when located with OSDs. There are typically multiple OSDs per server corresponding to different storage devices.

data stored in the cluster. This means that, for a cluster that is about half full ($\frac{S}{C} = 1/2$), adding a new node of capacity c forces it to download half its capacity of data. Furthermore, additional data transfers between other nodes may be required to fit the new distribution, depending on the hashing algorithm chosen as illustrated in figure 1.5 with a naive redistribution algorithm. Even worse, the deeper the bucket hierarchy is, the more data potentially needs to be transfered [67].

The fact that storage capacity increases much faster than network bandwidth increases the overhead of this redistribution of data — raising scalability issues. These transfers can saturate the cluster (or at least the bucket involved) which makes the storage system temporarily unavailable or operating with degraded performance during the process. Workaround exists, consisting of gradually increasing the weight of the new storage node (starting from zero) to reduce the volume of data to be transferred and thus the performance degradations. However this means that the re-distribution process lasts hours or even days for large storage nodes, making the bootstrap process last longer [23].

The other main drawback is that it is not possible to define a placement policy other than by changing the cluster map. One must keep in mind that the more complex the map is, the more difficult it is to balance the data. This means that heterogeneity in the devices is possible but difficult to accommodate. Hot spots can also arise if, by mischance, multiple heavily accessed objects have replicas stored on the same OSD [51]: a dynamic load-balancing policy is not possible with this hash-based approach.

2.1.4 GFS

The Google File System [24] (GFS) is a distributed file system that provides fault-tolerance while running on commodity hardware. It has been used by Google for over a decade and is designed to accommodate their usual workloads. As such, each cluster is expected to store a few millions of big files (typical size of 100MB or even GB), mostly serve large streaming reads and small random reads, and mostly receive large sequential write-appends to the end of files (but still supports small random writes). It is also expected to serve multiple clients reading and writing at the same time. A priority is also given to high bandwidth rather than low latency.

Like in every filesystem, files are divided in fixed-sized chunks. A GFS cluster consists of a single active Master, some inactive fallback masters, and multiple Chunkservers.

- The Master is responsible for the file system metadata (namespace, access control information, location of files' chunks, filesystem inodes and directories etc...), garbage collection, chunk migration, maintenance of the cluster's health, and file chunk lease management.

- ChunkServers store the different chunks as Linux files on their local filesystem.

The introduction of a master node allows GFS to have sophisticated placement and replication decisions. It is however an important bottleneck that has gone through a fair share of optimizations with regard to scalability. Indeed, any client accessing a file for a read or a write first has to interact with the master to get the relevant chunk information, although the reads and writes themselves don't pass through the master.

To reduce the single master bottleneck, some design decisions have been taken. First, the typical chunk size of GFS is 64 MB (compared to the typical 4 or 8KB chunk size of usual filesystems), which effectively reduces the number of chunks. This design choice has several advantages: it reduces the number of chunk locations the master has to keep up with — potentially making it possible to store all the metadata in RAM. It also reduces the number of connections the clients have to establish with ChunkServers.

However, this comes at the cost of storing small files very inefficiently (each file takes at least one chunk) — which is not an issue for GFS because small files are expected to be very rare. Secondly, clients don't receive the file metadata one by one. They send requests in batch and the master answers a request for a chunk with the information of the location of this chunk but also those for the next chunks, that the client is likely to need in the near future. However, these optimizations are not always possible for filesystems built with more generic use-cases in mind.

Moreover, even with these optimizations, having a single master puts a serious strain to the scalability of GFS [30], which explains why Google has been developing a new distributed storage system called Colossus [54]. However, to the best of our knowledge, there is no public information available about this architecture at the time this chapter was written.

2.1.5 Scaling the metadata layer and embracing the heterogeneity

In the past few years, efforts have been made to tackle the issues previously identified regarding the metadata layer. For example, since their 2.4 release in May 2013, the Lustre file system (LustreFS) [58, 122] allows for horizontal metadata scaling by using multiple Metadata Target Devices at the same time. Similar work has been conducted in 2017 to allow HDFS to store metadata in distributed NewSQL Databases [53].

In parallel, some work has been done to understand how heterogeneity affects cluster performance, and how to accommodate it while maintaining near-optimal performance. Such heterogeneity can be created by hotspots (objects heavily accessed) [51] or just by natural device heterogeneity [123],

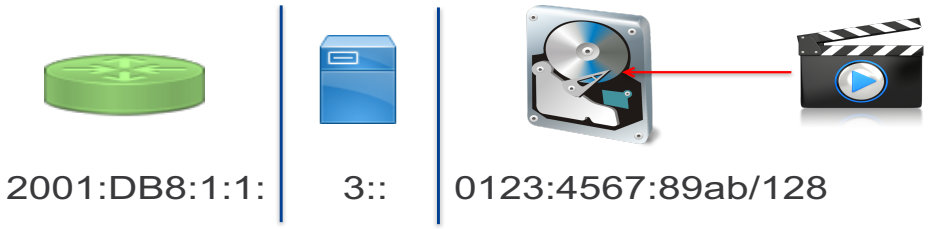


Figure 2.1: **Example of a routable object replica IPv6 address decomposition.**

which increases over time in a cluster when new devices replace failed ones and drives age differently.

Some new architectures that naturally distribute the metadata layer and use fine-grained load-balancing such as OpenIO [124] have also appeared, but it is hard to find precise documentation on these commercial solutions.

2.2 6Stor architecture

6Stor is a hybrid IPv6-centric distributed object storage system that borrows ideas from the approaches described in 2.1.3 and 2.1.4 to alleviate their shortcomings. To interact with the cluster, clients first interact with a fully distributed metadata layer –addressed via a consistent hash– and are then redirected towards the data layer.

The main characteristic of 6Stor is that it is "IPv6-centric", as we use routable IPv6 addresses to identify and reach objects replicas and their metadata replicas, and routable IPv6 prefixes to identify the servers of the storage cluster. Specifically, an object in a 6Stor cluster is represented by two sets of IPv6 addresses: its metadata replicas addresses and its object replicas addresses. The metadata addresses are obtained by a consistent hash that uses the object name as an input and the replica addresses are assigned by storage nodes storing the data, as illustrated in figure 2.1.

2.2.1 Architecture Description

We consider objects that are immutable but can be overwritten. For the sake of clarity, we will also consider that fault-tolerance is obtained through simple replication[2] for both the objects and their metadata. A 6Stor cluster is composed of 3 types of nodes:

[2]Erasure Coding schemes are being investigated to greatly reduce the storage overhead but are not yet fully integrated to our 6Stor prototype. However, most of the architecture would work exactly the same way with encoded fragments instead of replicas.

- **Storage Node (SN):** The SNs store the objects. Every SN is assigned an IPv6 prefix from which it chooses IPv6 addresses to identify the objects it stores. The prefix is assigned by an orchestrator and must be large enough for the node to name all the objects it stores uniquely. Clients interact directly with SNs when reading or writing objects.

- **Metadata Node (MN):** The MNs store objects' metadata. The metadata of an object contains **at least** the object name and the two sets of associated IPv6 addresses (for both the metadata replicas and the object replicas). It can also contain all other kind of object-related information that may be useful like Access Control Lists (ACLs), video metadata, date of creation, version number...

  Every metadata node is also assigned an IPv6 prefix by the Orchestrator. MNs create and store metadata identified by IPv6 addresses belonging to their prefix. When a MN creates a metadata for an object, it selects the SNs where replicas will be uploaded by the client.

- **Orchestrator Node (ON):** The ON is in charge of organizing the cluster, and in particular of assigning IPv6 prefixes to SNs and MNs and of making these prefixes routable. For instance, when a new SN joins the system, the ON assigns a storage IPv6 prefix to it and advertises it to MNs along with relevant informations (capacity, type of storage device ...). When a new MN joins the system, the ON reassigns others MNs IPv6 prefixes, advertises them and triggers the routing changes.

2.2.2 Attributing IPv6 prefixes to MNs

A 6Stor cluster is identified by a single fixed IPv6 prefix. This prefix is split in 2^n equal-length subprefixes, each subprefix defining a *Metadata Group (MG)*. At all point, each MG is assigned to a single MN with two requirements in mind: the mapping has to be **balanced** and, to a latter extent, **aggregated** as much as possible.

The trade-off incurred by this structure is between **load balance** and **route disaggregation**. Increasing n provides a better load distribution by assigning a larger number of smaller prefixes to nodes : 8 MGs assigned to 6 servers mean that 2 servers have twice the load other servers have, as illustrated in figure 2.2, whereas with 1024 MGs, this difference is reduced to $\simeq$ 0.6% but this comes at the price of an increase of the number of different routable IPv6 prefixes that need to be injected in the cluster. Note that the potential negative effects of disaggregation is limited to the storage cluster because of subnetting: the cluster is seen as a single prefix from outside, and MG prefixes can be aggregated at the MN level.

For example, let us imagine the 6Stor cluster depicted in Fig. 2.3, composed of 4 SNs, 2 MNs and a single ON. This cluster is configured to use the metadata IPv6 prefix `2001:DB8:1::/64` with 8 MGs. The orchestrator respectively assigns prefixes `2001:DB8:1:1:0::/80`,

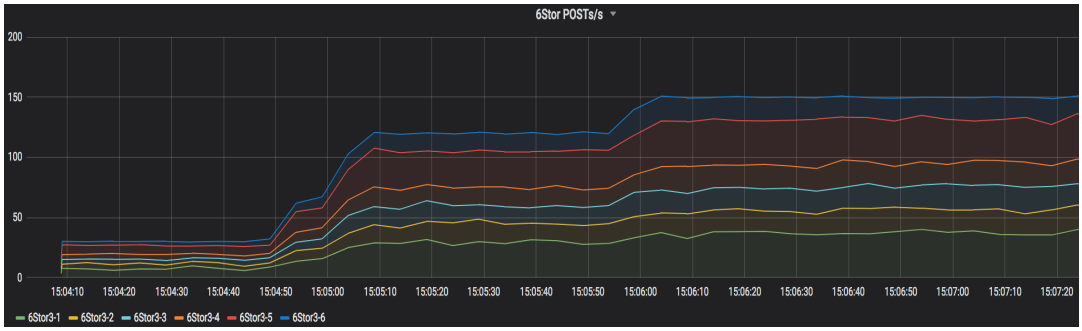


***Figure 2.2:* A 6 MN/6 SN 6Stor cluster with 8 MGs storing up to 5 video streams frame by frame –at 30 FPS per stream– in parallel. The MNs 6Stor3-1 and 6Stor3-5 are assigned 2 MGs and thus receive in average twice as many Post requests.**

`2001:DB8:1:1:1::/80`, `2001:DB8:1:1:2::/80` and `2001:DB8:1:1:3::/80` to the storage nodes. The metadata nodes are respectively assigned the IPv6 prefixes `2001:DB8:1::/65` and `2001:DB8:1:0:8000::/65` (by aggregating 4 MGs in each MN), the concatenation of which is the fixed metadata IPv6 prefix, as illustrated in figure 2.3.

2.2.3 6Stor: An IPv6-centric architecture

Why IPv6 addresses? The IPv6 prefixes assigned to SNs and MNs are routable towards these nodes, and these nodes listen on the whole prefix [3]. This means that any request sent to an IPv6 address within a prefix assigned to a SN or MN is routed to and answered by this node. When clients want to post or get an object, they send their requests directly to the corresponding IPv6 address.

Using specific IPv6 addresses for objects allows us to shifts a portion of the cluster's complexity to the network layer, providing efficient cluster elasticity and repairability, as discussed in section 2.3 and 2.4. Furthermore, being able to integrate object-related information in specifically-assigned IPv6 addresses has some potential for optimizations of the network-storage stacks. Efforts are already being made in user-space applications by using the Intel DPDK and SPDK libraries [125, 126], and we envision that we could go further by embedding storage- and network-stack related information (inode number, size, video quality, necessary throughput ...) in the IPv6 addresses, for instance to help on-path routers take better decisions based on the type of data they are forwarding but also to help the gathering of statistics.

[3] In practice, the traditionnal linux network stack does not allow to listen on an arbitrary prefix. So implementation-wise, we use the workaround of binding the socket to ANYADDR and creating a local route redirecting the prefix to the loopback interface. This way, the packet is sent to the loopback interface and matches the ANYADDR criteria. There are no MAC resolution issues as the servers are seen as next-hop routers in the routing tables.

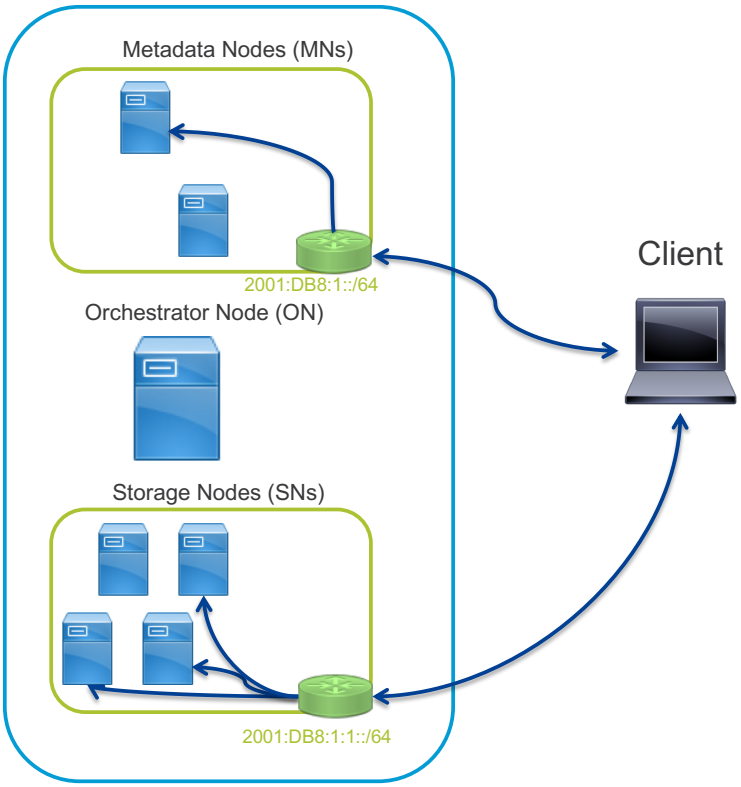


Figure 2.3: **6Stor architecture example**

The idea of overloading IPv6 addresses with application-level meaning is not new and has already been explored in work such as [127] for multiplexing or locator-identifier routing, [128] for video delivery or [129] for content distribution. The ICN (Information Centric Networking) [130] approach to change the networking paradigm from host-centric to content-centric also leverages such overload techniques in the hICN (hybrid ICN) project [131, 132].

Furthermore, providing a well-defined quality of service (QoS) at the network level for specific flows or data has already been identified as a key point [133, 134] to optimize the utilization of resource in datacenters.

Consequences of this design choice regarding scalabiliy, orchestration and analytics are elaborated on further in section 2.5.

Determining IPv6 addresses There are two types of IPv6 addresses related to an object: the metadata addresses identifying and locating the different copies of the metadata, placed on different metadata servers, and the addresses of the replicas of the object, identifying and locating the object's fragments on the different storage nodes. These two sets of addresses are determined by different means.

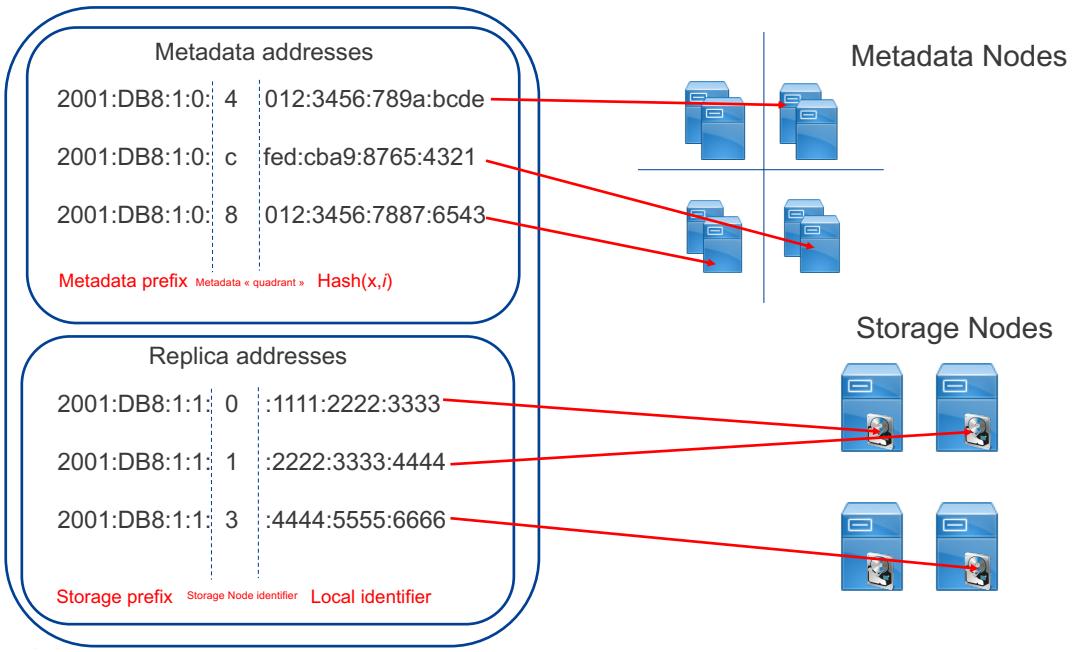


Figure 2.4: **Object metadata example with $N_m = N_s = 3$. In this case, $N_q = 4$.**

For the replicas, the SNs themselves give a unique IPv6 address to the replicas they store that belong to their assigned prefix and contains a locally unique identifier to be able to retrieve the replica (which could for example contain a 32-bit inode number of the file containing the replica on the local filesystem).

For the metadata, addresses are obtained by a consistent and deterministic mechanism. To obtain the metadata IPv6 addresses for an object named x with N_m metadata replicas in a cluster with a given prefix, this prefix is firstly split in N_q same-size subprefixes, identifying *metadata quadrants*, N_q being the lowest power of 2 superior or equal to N_m, equal to $2^{\lceil \log_2 N_m \rceil}$. Secondly, N_m distinct metadata quadrants are deterministically chosen among the N_q and ordered, based on the object name through the use of a hash. Lastly, N_m IPv6 addresses matching the prefixes are generated by concatenating the prefixes and hashes of the object name x seeded by the index of the metadata address. An example of an object's metadata with a metadata replication factor N_m and a data replication factor N_d set to 3 is given in figure 2.4.

This mechanism ensures that the N_m metadata addresses for an object are: (i) consistent no matter the time and cluster layout, (ii) belong to different subprefixes to ensure they match with different MGs, for reliability purposes.

2.2.4 Description of basic operations

In the rest of this section, the different basic operations of 6Stor are described. The corresponding sequence diagrams can be found in figure 2.5. We make

use of a custom set of signaling messages that identify an operation (Post, Get, Delete, Rename) and an interaction type (external, when the request comes from a client, and internal, when the request comes from another node inside the cluster) to obtain the desired behavior.

The general idea is to use the first metadata and replica for every object as "masters" that are in charge of ensuring the operations return the expected result and avoid basic consistency issues.

Post: To post an object with N_m metadata replicas and N_d object replicas, the operation goes as follows:

1. The metadata addresses for the object are obtained through the algorithm described in paragraph 2.2.3. A UDP [4] packet containing a Post request identifier, the object name and its size is sent to the first metadata IPv6 address;

2. When the MN matching this address receives the packet, it creates a metadata by selecting the appropriate SNs and generating the additional metadata addresses for the object. Temporarily, the metadata contains generic addresses identifying the SNs assigned to store the replicas (for example, the first address of their storage IPv6 prefix);

3. The same MN sends a UDP packet containing an internal Post request identifier and the created metadata (containing the object name, its size and the two sets of IPv6 addresses identifying the metadata and the SNs) to the other metadata IPv6 addresses and to the SN chosen to store the first replica;

4. The SN assigns a unique IPv6 address to the replica and sends it to the MN;

5. When more than half of MNs (including the original one) have acknowledged that they have stored the metadata, the IPv6 address for the first replica is sent to the client;

6. The client opens a TCP connection with the first replica address, sends an external Post request identifier and starts transmitting the corresponding data;

7. The primary SN of the list accepts the TCP connection and use the metadata that the MN previously sent to open TCP connections (or use already opened ones) towards the alternate SNs of the list and forwards them the metadata along with an internal Post request identifier. It then receives the data, locally writes it and in parallel forwards it to the other SNs;

[4] UDP is used for metadata exchanges because they fit in one packet. It provides better latency and less computing overhead but requires a timeout to retransmit when losses happen.

8. When a SN has finished writing the object, it acknowledges it by sending the unique IPv6 address identifying the object replica (within its assigned prefix) to the object metadata addresses;

9. Once a SN receives $W_m \leq N_m$ number of acknowledgements from MNs (including the first of the list) that testify that the write has been registered by enough metadata nodes for reliability purposes, it sends an acknowledgement to the first SN of the list;

10. At the same time, once a MN receives the same parametrable W_m number of unique replica addresses, it locally commits the metadata (and overwrites it if it previously existed). In addition, if the first MN has overwritten previous metadata, it means that the operation is an overwrite of an existing object, and the MN thus triggers the deletion of the previous version of the object on its corresponding SNs;

11. Once the first SN receives $W_d \leq N_d$ number of acknowledgements from concerned SNs (including itself), it sends an acknowledgement to the client and terminates the connection[5].

Get: To get an object the client follows those steps:

1. The metadata addresses for the object are obtained through the hash, described in 2.2.3. A UDP packet containing an external Get request identifier and the object name is sent to the first metadata IPv6 address;

2. The MN matching this address receives the packet and sends the metadata back in a UDP packet to the client;

3. The client opens a TCP connection to one of the object's replica addresses;

4. The SN matching this address accepts the TCP connection, uses the unique identifier contained in the address to identify the local file storing the replica, and sends the data to the client. When all data has been sent over TCP and acknowledged, it terminates the connection.

Delete: To delete an object:

1. The client sends a UDP packet containing an external Delete request and the object name to its first metadata address;

2. The matching MN sends a UDP packet containing an internal Delete request and the object name to the metadata and replica IPv6 addresses;

[5]the lower the parameters W_m and W_d, the lesser delay before the client receives an acknowledgement that the object is written to the cluster but the lower reliability in case of simultaneous failures during the operation, as explained earlier in section 1.3.

3. When every MN has acknowledged to the first MN the deletion of the metadata, the first MN acknowledges the deletion to the client. In parallel, the object is being deleted on SNs but the client doesn't need confirmation since the object is not reachable anymore without its corresponding metadata.

Rename: Due to the fact that only metadata addresses depend on the object name, a 6Stor cluster offers the possibility to easily rename an object without moving the data itself. This operation is not possible for purely DHT-based architectures. To perform this operation:

1. The client sends a UDP packet containing an external Rename request identifier, the current name of the object, and the new name to the first metadata address;

2. The matching MN generates the new N_m addresses corresponding to the new object name and sends a UDP packet containing an internal Rename request identifier containing the modified metadata (with the new object name and new metadata IPv6 addresses) to all those addresses;

3. When more than $max(W_m, \lceil N_m/2 \rceil)$ of the new MNs have acknowledged the new metadata creation, the first MN sends an internal Delete request to the other old metadata addresses;

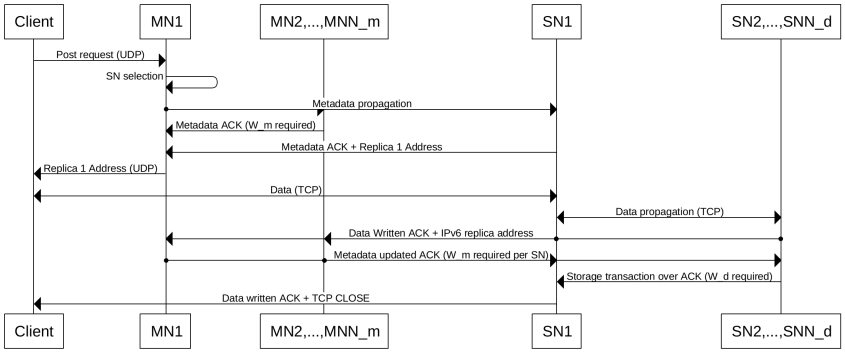
4. When every other old MN has acknowledged the metadata entry deletion, the first MN deletes its own old metadata entry and acknowledges the deletion to the client.

2.2.5 Consistency

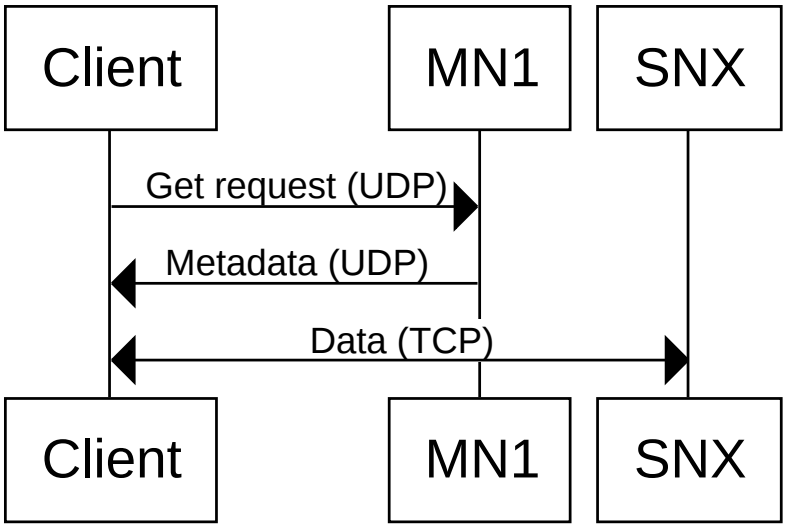
As for any distributed system, consistency is an issue when an object has replicas and metadata scattered across multiple nodes. As such, it is important to clearly define the consistency that the client should expect for basic operations.

6Stor uses a variation of the (N, W, R) model presented in 1.3. It is a $(N, W, 1)$ version, where consistency for metadata operations is granted by the fact that every operation goes through the first MN every time (but still requiring W_m acknowledgements for reliability reasons). Consistency on data for reads is guaranteed by the fact that the first MN of an object knows which SNs have a correct version and which SNs do not. This design choice has been made to guarantee low latency while maintaining good consistency levels.

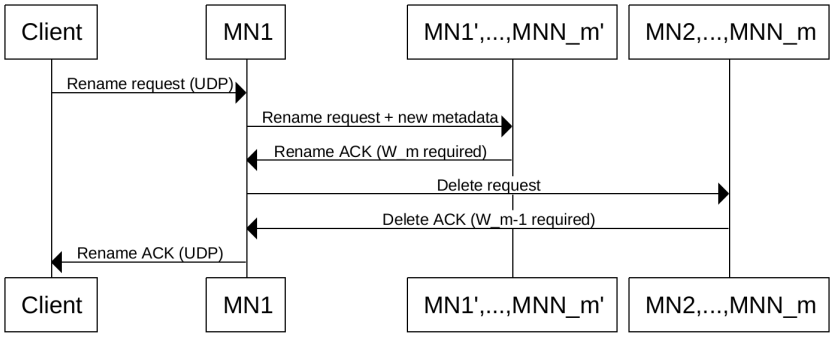
Let us enumerate typical conflict cases and describe what would happen in a 6Stor cluster:



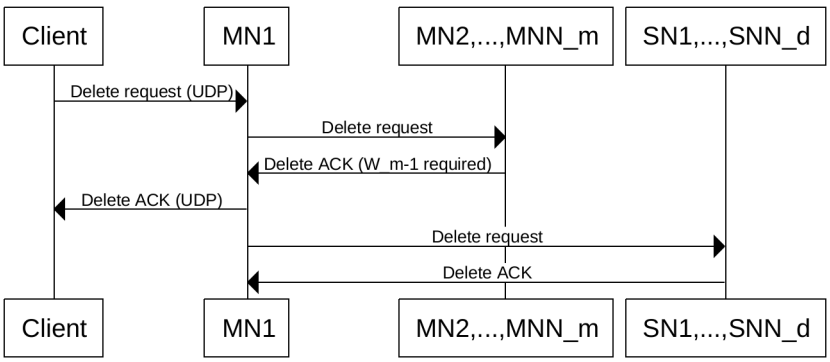
(a) POST



(b) GET



(c) RENAME



(d) DELETE

***Figure 2.5:* Sequence Diagrams of the 4 basic 6Stor operations.**

- Simultaneous read/write: A client tries to get an object while another one is overwriting it with an object with the same name. If the client writing has received an ACK before the client getting the object sends its request, the first MN has by definition updated its metadata so the client reading will get the new version. However, it is possible for the reading client to get the new version of the object before the writing client receives an ACK (because the ACK is sent by the SN when enough MNs have ACKed the write. This means that at one point the first MN has overwritten the old metadata but the first SN has not yet sent the ACK to the writing client).

- Several clients are trying to post objects with the same name: because Post operations pass through the first MN, a Post request arriving for an object which is already currently being posted will immediately fail for the second client;

- A client is trying to get an object that another client is deleting: the delete ACK is sent when every MN containing the metadata has deleted it. This means that a client will not be able to get the object after a delete ACK has been received by the deleting client. However, it is possible for a Get request to fail before the deleting client receives the ACK (it can happen while the ACK is on the wire, and it can also happen that SNs have deleted their local replica of the object before all the MNs have deleted their local metadata).

These three cases encompass most of the usual potential consistency issues in distributed storage systems. The consistency issues they can raise are kept minimal and correspond to what is expected by most applications using distributed storage backends.

2.3 Expanding or shrinking the cluster without impacting the cluster's performance

2.3.1 Storage Nodes

When a new SN is included in the cluster, it is assigned a storage IPv6 prefix. This prefix is advertised to some MNs that will have it in their storage pool along with relevant information (storage type, capacity ...). Once this prefix has been advertised by the ON, the MNs can immediately assign incoming post requests to the SN [6], making the bootstrap process almost instantaneous.

When a SN is scheduled to be excluded from the cluster, MNs holding metadata for objects which have a replica on the excluded SN are notified and place a new replica for each affected object on other SNs. Once every

[6] In the background, it is also possible to rebalance data from existing SNs to the new one. However, this is absolutely not mandatory and can be done object by object, without making any node or object replica unavailable as it is the case in DHT-based storage systems.

affected object has a new replica posted on another SN, the excluded SN can safely be removed from the cluster. The case where a SN unexpectedly fails is treated later in section 2.4.2.

2.3.2 Metadata Nodes

When a new MN is included in the cluster, it will take responsibility of a number of consecutive MGs (defined in section 2.2.2). These MGs are however currently assigned to other MNs so a transition is required. Let us denote by S the set of MGs assigned to the new MN by the Orchestrator.

Before making any change to the routing and to the other MNs MGs affectation, the MNs that are presently assigned prefixes within the set S are notified by the orchestrator. Once advertised, these MNs begin transferring all the metadata corresponding to these MGs to the new MN. The routing is also changed so that new requests matching the new MN's MGs are routed to the new MN. As a consequence, Post requests are dealt with by this new MN as expected. Get requests for object which metadata have not yet been transferred fail, and the clients retry with the next metadata address (remember that there is a redundancy of metadata), unless a Post has already been committed on the the new MN for the object.

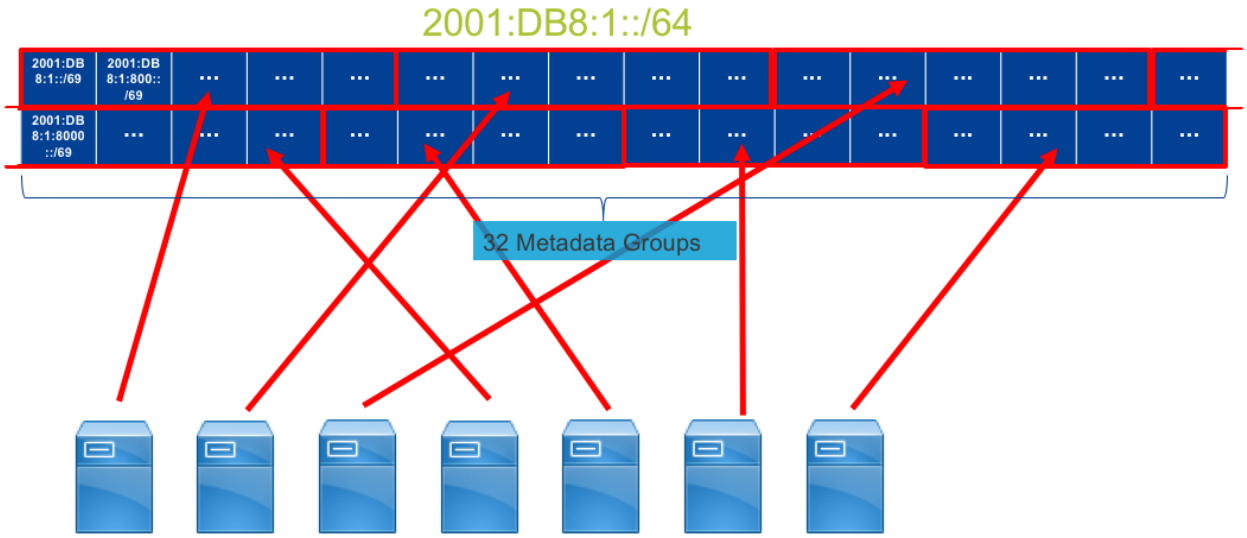
Note that during this transition period, consistency issues are avoided. A Get request will be treated by another MN only in the case where metadata has not yet been transferred *and* a post request has not yet occurred for this object (i.e. the object has not changed meanwhile). The MG redistribution is illustrated in figure 2.6 with 32 MGs and an expansion from 7 to 8 MNs.

When a MN leaves the cluster, the new MGs assignment is made and it works exactly the same way. The operation finishes when the leaving MN has transferred all its metadata to the MNs taking his MGs. Again, we do not treat here the case of failure which is specifically covered later in section 2.4.2.

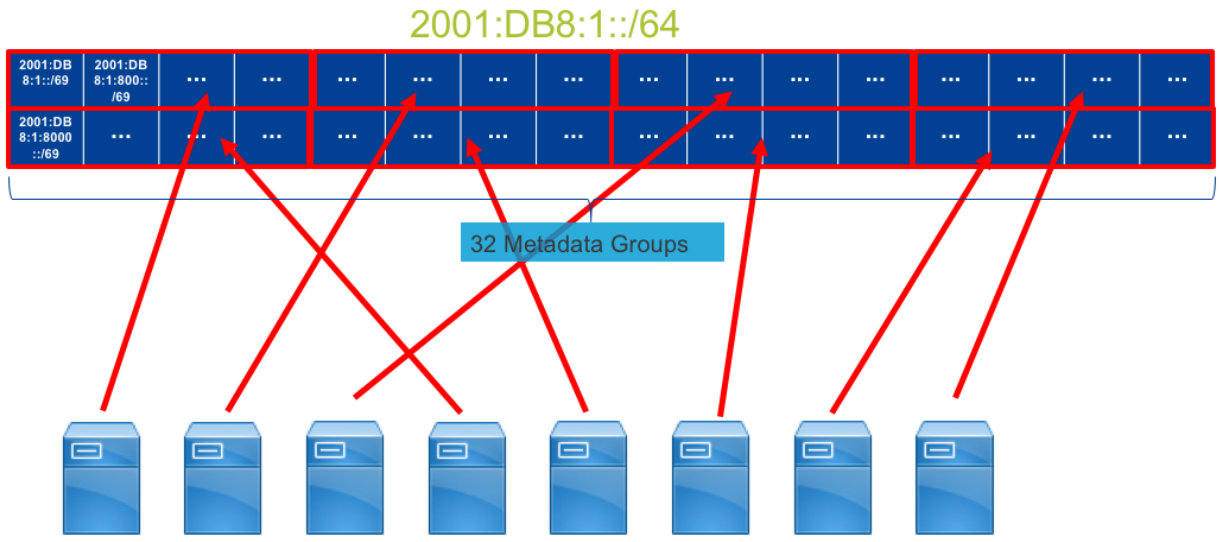
2.3.3 Availability and data transfer

An important feature of 6Stor is that none of the expansion or reduction operations makes any part of the cluster unavailable. While the object and metadata transfers happen, the cluster keeps working the same way even if slightly lower performances may be observed since links are shared between actual requests and data as well as metadata transfer.

In addition, when a MN is added to or removed from the cluster, a mandatory data transfer comparable to the Ceph case occurs. However, our approach has two advantages: as previously stated, there is no indeterminate state for the objects and metadata since they are reached through natural routing, so that no cluster part is unavailable even during the rebalancing.



(a) 32 MGs, 7 MNs



(b) 32 MGs, 8 MNs

Figure 2.6: **MG redistribution when including a new MN in the cluster.**

Furthermore, the amount of data that has to be transferred is very small compared to the amount of data stored in the cluster, since it only affects metadata, which are expected to be of very small size (typically less than 1kB for an object).

2.4 Coping with failures: reliability and repair model

2.4.1 Reliability

Like most distributed storage systems, we solve the problem of reliability via replication, both at the metadata and object level. Note that erasure codes [135] could also be used at the data level to reduce the storage overhead, but this subject is not covered in this chapter. When a MN selects SNs to store an object, it specifically chooses different SNs to store the different replicas. Furthermore, exclusive failure domains containing different SNs can be defined so that different replicas of an object are stored in different failure domains.

For metadata, the algorithm described in section 2.2.3 ensures that metadata replicas are stored on different MNs, as long as there are enough MNs.

2.4.2 Reacting to failures

Recent work [136] shows that failures come in different shapes and sizes, and that it is not always optimal to immediately trigger a repair when a failure occurs. It is an impactful decision to trigger a repair, as it degrades performance and mobilizes a consequent amount of I/O and bandwidth resources. Consequently, it is sometimes a better solution to wait for a temporary failure to settle rather than to trigger a full data reconstruction.

In the following, we shall make a distinction between two types of failures for our MNs and SNs: short and definitive failures. We differentiate them by using a simple timer after a failure occurs: the default reaction after a failure is a short failure reaction, and if a node does not rejoin the cluster when the timeout occurs, the system falls back in definitive failure mode. We distinguish these modes to avoid costly repairs and prefix reassignment for nodes that are only temporarily down or in maintenance.

Short failure

For SNs, once a failure is detected, a SN is put in a failure state in the SN maps held by MNs. MNs don't select a SN in a failure state to store objects' replicas. The on-going operations which were involving the failing SN will of course not be successful. However, after a timeout, retries can be made with the other SNs.

For MNs, the associated MGs are not immediately re-affected in the case of a short failure. The MGs assigned to the failed MN are temporarily assigned to another server (which can be a backup server or another MN) and the routing tables are updated. This temporary MN will obviously not be able to answer all get requests and will as such redirect clients to the next metadata address. However, it will accept post requests and store the corresponding generated metadata. If the failing MN rejoins the cluster before the timeout occurs, the prefix will be routed back to it and the temporary MN will transmit all newly created metadata to it.

Definitive failure

After the timeout occurs, the node is considered *permanently* disappeared. For a SN, two repair modes exist: either a new server is available and is assigned the IPv6 prefix of the failed node or the prefix is not reassigned. In the case where the prefix is re-assigned to a new SN, the other SNs storing the replicas of the objects that were previously stored on this prefix transmit their replicas [7] so that the effective replication factor (the real number of replicas stored) for these objects can be increased back to N_d (the configured expected replication factor N_d). This has the advantage of not requiring the MNs to re-locate the lost replicas on other storage nodes, by keeping the same IPv6 addresses.

If the prefix is not reassigned, the MNs containing the primary metadata replica for every object that had a replica on the failed SN have to assign the repaired replica to a new SN, trigger the replication, and update the other MNs.

In the case of MN failures, the reparation is triggered at the MG level. Every MN maintains, for each MG they are assigned and each other MG in the cluster, a list of the metadata that have a replica in both the MGs. When a repair is triggered, each MN uses the relevant lists to send the metadata requiring repair to the MN assigned the MG requiring repair.

Voluntary shutdown and maintenance

In some occasions, one can knowingly and voluntarily shutdown or restart a machine that hosts a server in the storage cluster. When this occurs, one knows in advance if the node is supposed to rejoin the cluster or not, and when. As such, a voluntary shutdown operation allows the user to define the timeout for the specific operation or to immediately trigger a modified definitive failure, where the node (MN or SN) is in charge of its own repair either on a new node or on the rest of the cluster before it shuts down.

[7] A quick handshake is being made before each replica transmission to avoid multiple retransmissions for the same object.

Maintaining reliability

Because of our model of delayed repair, it is possible for an object to simultaneously lose several replicas. For this reason, it is necessary to detect objects which replication factor falls below a configured threshold and repair them, even if the failed node's repair is not triggered, so that a healthy reliability is maintained. To that end, MNs keep track of the effective object and metadata replication factors of the objects they hold when they are made aware of failures.

2.5 Considerations on the Architecture

2.5.1 Client and Cluster Configuration

Most of the cluster configuration is hidden from the client: the only information it requires is the general IPv6 prefix for metadata. Due to this, a 6Stor cluster is dynamically reconfigurable and most configuration changes or maintenance operations can be conducted online while the cluster keeps working. This is not the case for most distributed storage systems, where clients either need access to a single master or a precise, up to date map of the cluster to operate.

2.5.2 Layer of Indirection

Having a layer of indirection is generally a trade-off with the following properties.
Cons :

- Layer overhead: Most operations require an interaction with the intermediate layer, which incurs a performance penalty (network Round Time Trip (RTT), compute, increase of the number of potential failure points ...);
- System complexity: Having an intermediate layer means that there are different types of nodes in the cluster. Each type of node has a different type of configuration and of interaction, and potentially different hardware requirements. This is more complex to deploy and maintain optimally.

Pros :

- Data plane flexibility: As we explained in section 2.3, having a layer of indirection displaces the need for a redistribution when the cluster changes from the data plane to the metadata plane. Because data repartition is decided by the intermediate layer, it is not subject to redistribution for causes external to explicit decisions from this intermediate layer (unlike systems relying on consistent hashing like Ceph [28] or GlusterFS [117]);

- Time-inconsistent decision making: Because a layer of indirection keeps track of where data is stored, there is no need for a consistent way to place data. This allows the cluster to accommodate to current circumstances such as a temporarily heavily accessed nodes, unreachable (but not flagged out of the cluster) servers ...

- Heterogeneity accommodation: When a cluster increases in size and age, it is more and more probable to have nodes comprised of different hardware and providing uneven performance. Managing this heterogeneity is nontrivial and efforts have been made to devise architectures that take it into account [123, 137, 138], both for storage systems and databases. It is impossible to incorporate this heterogeneity in consistent data placement schemes. As such, a layer of indirection allows to take this into account and adapt data distribution and access to the different devices present in the cluster.

As such, 6Stor aims at optimizing the interactions with the intermediate layer to reduce the overhead, but also at benefiting the maximum from what it allows through the use of placement policies. Placement policies are arbitrary policies that can be system-wide or user-defined and determine how data objects are placed in the cluster. For example, a placement policy in a heterogeneous cluster could be to place one replica for each object on a SN with a SSD and the other replicas on SNs with HDDs. Another placement policy could be to store some objects as replicas and others erasure coded to save some storage space, based on application-defined parameters. Placement policies are enforced by MNs and can also take into account specific SN metrics – such as average SN I/O load, CPU load, bandwidth ...– to balance the load.

Efforts have been made towards both static [139, 140] and dynamic [51] data placement strategies in a datacenter environment, and this is still an ongoing work in the context of 6Stor.

2.5.3 Scalability

The key contribution of the IPv6 addressing scheme is the scalability that it naturally provides. Indeed, the most common issue with distributed storage system with a layer of indirection is the non-distributability of this layer: a unique master node deals with all metadata operations [24, 29, 58].

By design, a 6Stor cluster naturally distributes this metadata layer by exposing a single IPv6 prefix that it internally splits and distributes between different physical nodes. The consistency is dealt with at the object level depending on its hash and objects' metadata are uniformly distributed amongst physical nodes by the means of a hashing function assigning objects to IPv6 addresses (and thus to the corresponding nodes).

A traditionnal property of DHTs is that they have to maintain an overlay network for each node to be able to know where to look for objects that

they don't store themselves. The immediate tradeoff is between the number of hops required to reach data and the amount of information about the cluster that each node must maintain – linearly correlated with the nodes' degrees, as defined in section 1.1 and illustrated in table 1.1. The more every node knows about which node is where and stores what, the less hops are required to reach data from any node. In resource management and performance, this translates as a tradeoff between the memory footprint of storage nodes and their ability to remain always reachable on one side versus the latency of operations. The most common choice is the one made by Chord [47] which has an average degree growing in $O(\log(n))$, with n the number of peers in the DHT. For Ceph [28] and its DHT-like structure, cluster overlay metadata size grows in $O(n)$ while the hop count stays in $O(1)$.

On the contrary, in a 6Stor cluster, the knowledge of the clusters' nodes locations is contained in the routing tables and aggregated: MNs only need to know how many MGs there are and SNs only need to know their configuration to function properly. This means that there is no structural limitation (other than orchestration itself) to the number of nodes that can participate in a 6Stor cluster [8]. Effectively, 6Stor maintains a hop count in $O(1)$ and a cluster overlay metadata in $O(1)$. However, MNs have to maintain a list of available SNs to place object replicas, but this list doesn't have to include all SNs. Additionally, this comes at the cost of increasing the size of routing tables, but current routers support multi-million-entries routing tables and, as explained in section 2.2.2, this only affects routers close to the cluster.

Consequently, a 6Stor cluster scales linearly with the numbers of servers both in the metadata and the data layers, and independently between these two layers.

2.5.4 Metrology and Analytics

Because every object is identified by specifics IPv6 addresses, gathering analytics on objects is as simple as analyzing the network traffic. A simple and common tool such as IPFIX or NetFlow [141, 142] systems allows a cluster administrator to estimate the popularity of objects.

Furthermore, it is possible to use network policies to enforce QoS towards certain objects. Actually, if the traffic characteristics of popular content are known, the required mean and peak rates or usefull QoS characteristics to be used for a new flow can be directly induced from the IP address of the object requested, without any need for signaling.

[8] This is not totally true, as there need to be at most as many MNs as there are different addresses in the cluster metadata prefix, but in practice this number can be as high as 10^64. It is hard to imagine a scale at which this limit would matter, since current single-master architectures can already handle hundreds and in some cases thousands of SNs.

2.5.5 Limitations

One of the main limitation of 6Stor is structural: to deploy a 6Stor cluster, one must both be able to use IPv6 and to adequately route the traffic. These are strong requirements that not every cloud yet provides and can make it impossible (or at least inefficient) to deploy a 6Stor cluster in a public cloud. However, 6stor has been designed for data centers as a first priority, where IPv6 is becoming the norm.

Another current limitation is that a client must open a TCP connection per object request (Post or Get). For clusters of moderate size where clients are expected to connect multiple times to each server, this can be sub-optimimal as the usual optimization is to keep a single connection open (as traditional HTTP servers do). However, workarounds are actively being investigated such as having the client keep partial knowledge of the architecture by caching information about the prefixes of the servers it interacts with, and thus keeping its connections towards these servers opened, or extending the TCP fast open mechanism to use it towards a whole prefix instead of a single address [143]. Furthermore, this limitation is less relevant when the number of clients and of servers scale up, as single clients have less and less overall importance and are less and less likely to connect multiple times to the same server.

2.6 Experimental Evaluation

2.6.1 Rationale

Our assumption is that the complex software layering that currently exists in distributed storage solutions limits their performance. We have developed a prototype of 6Stor in order to validate this assumption. We subjected this prototype to a simple benchmark. We also subjected Ceph, the most resembling and available distributed object storage solution, to the same tests.

This benchmark is by no means a definite and fair comparison, since our prototype does not yet implement some critical functionalities from the system, such as failure and error detection, repairs, authentication ... We however believe that it gives indicative results of where our prototype stands today and, thus, the potential gains than can be expected in the future.

2.6.2 Setup and Protocol

To make this experiment as relevant as possible, we disable every functionality that we can from our Ceph cluster, such as CRC checking and authentication — functionalities that are not yet available in our 6Stor prototype. We deploy in parallel a Ceph and a 6Stor cluster on the same physical servers:

- 8 servers on a rack, each one of them equipped with a HDD (Toshiba 600 GB 2.5 inches 6G SAS 15K RPM HDD) and a SSD (Intel 480GB 2.5

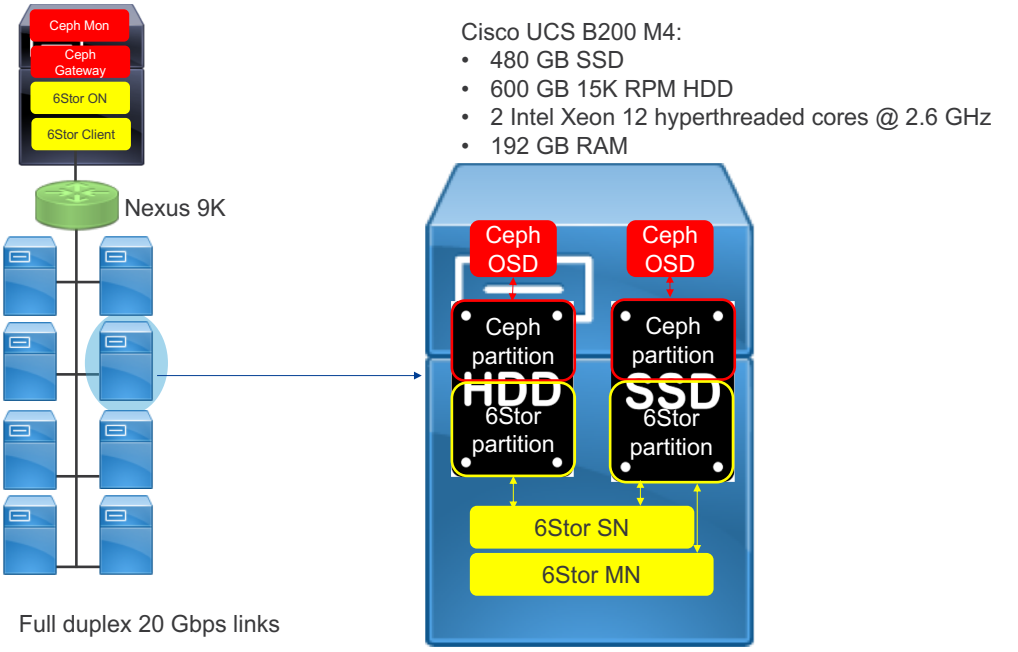


Figure 2.7: **Experimental setup**

inches Enterprise 6G SATA SSD) and hosting 2 Ceph OSDs (one per disk), one 6Stor SN (managing both disks) and one 6Stor MN;

- 1 server in another rack, hosting a Ceph Monitor and Gateway (acting as a client),a 6Stor ON and a 6Stor client;
- Both racks are connected by 20 Gbps links.

The setup is summarized in figure 2.7.

We evaluate Post and Get performance, on HDD and SSD respectively, for different factors of replication and for varying number of parallel clients threads. The tests are conducted on 3 different datasets: (i) 56662 small text files obtained from a wikipedia dump, of average size 2,6KB; (ii) 21631 video fragments of average quality and medium duration, of average size 306 KB from the open movie project *Big Buck Bunny* [144]; (iii) 1164 video fragments of high quality and high duration, of average size 2,3 MB from the open movie project *Valkaama* [145].

This data, that reside in memory of the client server beforehand, is first posted to the clusters then retrieved. For Post experiments, the dataset is split in n subsets of roughly the same size, each given as input to 1 of the n client threads. Thus, each thread actually writes $1/n^{th}$ of the dataset in average. For 6Stor, we mirror the metadata and the data replication factors. Timestamps are collected by the client threads and analyzed after the Posts are completed to compute the average latency.

For Get tests, a complete list of the objects to be retrieved is computed then pseudo-randomly shuffled n times, n being the number of parallel client threads, so that all threads do not fetch objects in the same order and at the

same time. Because the shuffling is pseudo-random, the x^{th} list, $x \leq n$, is always the same for the same dataset across all tests. The n client threads are then launched, each one fetching the listed objects in the order given by their respective shuffled list and writing timestamps for every operation on separate .csv files. It should be noticed that each data set is fetched n times with this set-up.

During these tests, the average CPU consumption is also measured on servers to account for the CPU consumption of 6Stor Metadata and Storage Nodes and Ceph OSDs. The results are presented and analyzed in section 2.6.3.

2.6.3 Results

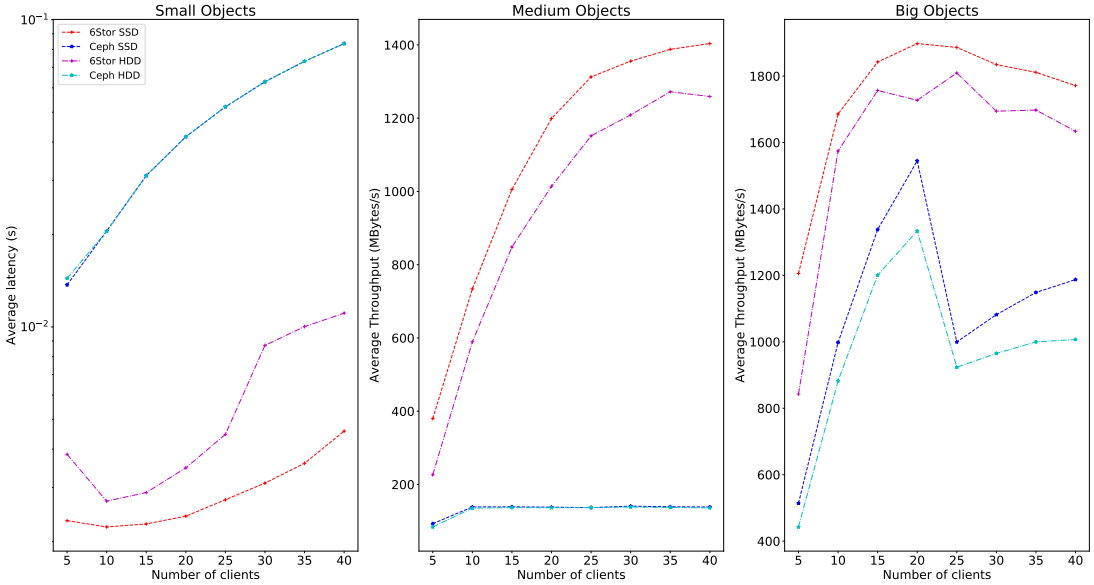
As we stated before, the results presented below cannot be used for the sake of comparaison. It does not represent a fair comparison as we compare a prototype and a fully fledged and used-in-production solution, but allows us to validate if our assumptions actually show promise and to analyse the behavior of our prototype.

The results are shown in figure 2.8 and split in three graphics: the first one for Get operations on both SSD and HDD clusters, the second graphic for Post operations on SSD clusters, and finally for Posts on HDD clusters. For each part, results are differentiated based on object types (small, medium, big). For small objects, the average operation latency is displayed, as it represents the most relevant performance indicator. For medium and big objects, the average throughput is provided. In the three cases, the test are repeated for different replication number and client thread numbers. Each of the data points shown here is the average of the repetition of the same experiment repeated 10 times for Posts and 6 times for Gets.

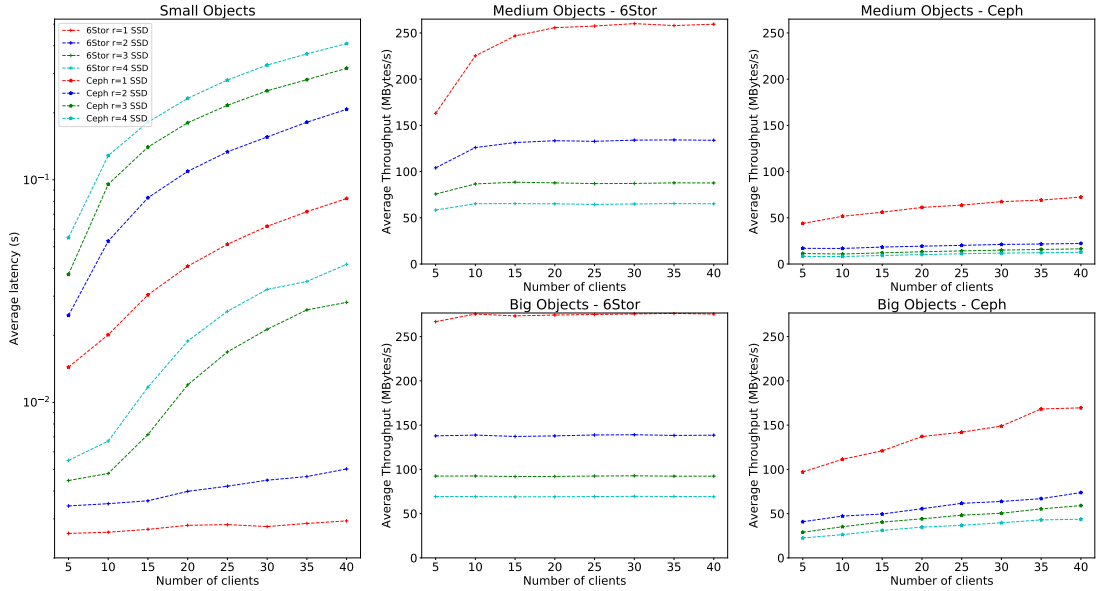
2.6.4 Get Tests

For small objects, the latency is fortunately lower for 6Stor. 6Stor performs better with SSD than with HDD, as expected. The difference is likely due to the lighter implementation of 6Stor, and the fact that 6Stor doesn't use HTTP but directly TCP, which requires no computation for additional headers. This also likely explains the absence of difference between SSD and HDD performance for Ceph, because the disk latency has a less relative impact on the total operation latency.

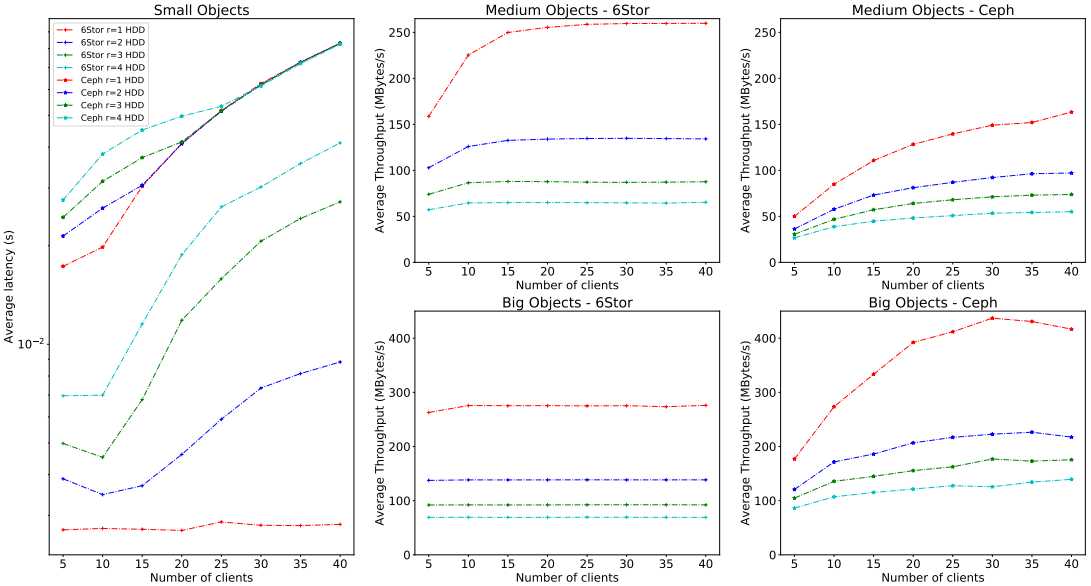
When increasing the number of client threads, one would expect the average latency to increase for HDDs because they have to serve more reads in parallel. However, this is surprisingly not the case as the average latency is consistently lower for 10 to 25 clients threads than for 5 clients threads. The difference comes from two mechanisms: first, the SNs' filesystems' I/O scheduler provides a better economy of scale when reordering parallel disk



(a) Gets



(b) Posts on SSD



(c) Posts on HDD

***Figure 2.8:* Test Results**

reads, resulting in much less time spent on disk seeks (during which the hard drive head moves between physical sectors and thus doesn't read anything) on average. Secondly, the small objects remain in local filesystems caches which means that the consecutive local reads after the first are much faster than the first one, diminishing the average disk read overhead per client thread. For more than 10 clients threads however, the increasing amount of parallel requests counterbalances these economies of scale and increases the average latency per operation.

For medium objects, we notice that Ceph peaks around 200 MB/s and presents no difference between HDD and SSD. 6Stor performs only slightly better on SSD than on HDD.

For big objects, we notice that both Ceph and 6Stor perform better on SSDs as expected. After around 20 client threads, 6Stor reaches a point where it is actually less efficient to have more clients in parallel. This can be explained by the fact that individual read operations are largely sequential for big objects, but the I/O scheduler sometimes force a read in another part of the disk for another request so as not to starve another concurrent Get request. At the peak of 20 clients for big objects, the maximum network throughput of 20 Gbps is almost reached.

A surprising and interesting result is the consequent drop in Ceph throughput between 20 and 25 clients for both HDD and SSD and slow increase of the performance after this drop. We did not investigate this result further as it was not the focus of our work.

2.6.5 Post Tests

Surprisingly, 6Stor has virtually the same results on the HDD and on the SSD clusters, except for small objects and a high number of client threads. This is related to the current implementation lack of optimization and is currently being worked on.

Even more surprisingly, Ceph actually performs better on the HDD cluster than on the SSD cluster in our experiments. This is in contradiction to the previous results (for Get operations), for which the SSD cluster performs better or the same as the HDD cluster.

In both cases, multi-threading on the client side benefits more to Ceph than to 6Stor. Furthermore, for big objects, Ceph performs better on HDD cluster than 6Stor does (both on SSD and HDD cluster). This is likely due to a crude implementation that lacks an optimized asynchronous threading architecture on the server side for 6Stor, and that consequently scales less well.

	CPU efficiency	
Object size	6Stor	Ceph
Small	0.86	0.27
Medium	48.59	18.76
Big	91.14	43.97

Table 2.1: **CPU utilization efficiency average for Get requests**

	CPU efficiency			
	6Stor		Ceph	
Object size	SSD	HDD	SSD	HDD
Small	0.28	0.28	0.17	0.21
Medium	6.87	7.28	9.28	10.95
Big	12.57	12.86	16.82	16.28

Table 2.2: **CPU utilization efficiency average for Post requests**

2.6.6 CPU consumption analysis

During these tests, we measured the CPU time for each process involved on the servers in the storage cluster and averaged it to deduce their average CPU consumption (in %). For every test, we look at the CPU % time but also at the *CPU utilization efficiency* that we define by dividing the average cluster throughput by the average CPU % time (a higher value indicates a better CPU utilization).

To account for the replication, we consider the effective cluster throughput for each test, equal to the application-perceived throughput multiplied by the replication factor. The results do not significantly vary with the number of client threads for both Gets and Posts, with the replication factor in the case of Posts, and with the drive type in the case of Gets. Consequently, the results are presented as average per object size class for Gets and average per object size class and disk type for Posts in tables 2.1 and 2.2.

For Gets, 6Stor is around 2 times more efficient than Ceph for big objects, between 2 and 6 times more efficient for medium objects (on average around 2.7 times). For small objects 6Stor is between 3 and 4 times more efficient than Ceph. We suppose that this difference is at least partially due to the HTTP overhead of Ceph. This assumption is explored below in section 2.6.7.

For Posts, Ceph performs overall slighly more efficiently (between 20% and 25%) for medium and big objects, even when 6Stor actually has more throughput. We expect the crudeness of our implementation to be the cause for this slight performance. However, for small objects, 6Stor is more efficient. Again, we suppose this is due to the HTTP overhead, that has more relative weight for small objects.

Overall, these results show that 6Stor has comparable performance with regard to Ceph for medium and big objects – with slightly better and slightly lesser cases, but performs way better for every metric for small objects, as we expected since we did our best to reduce the fixed overhead per operation partially due to intermediate layers such as HTTP.

2.6.7 Performance impact of HTTP

In this subsection, we try to measure the performance impact of the HTTP protocol when compared to the simple protocol over TCP used by 6Stor to transfer files. To measure this difference, we setup two systems to serve the same set of test files, which contain random data.

Protocol

On one hand, the files are served by a nginx web server [146], and on the other hand by a 6Stor SN. Special care is taken to make sure that both servers perform the file transfer as similarly as possible, so that any difference in performance between are related solely to the protocol differences, and not to implementation differences. In particular, the same socket options are set on the server sockets used by nginx and the 6Stor SN, and the same system calls are used to transfer the file to the client. As the 6Stor protocol forces us to open a new connection to the server for each request, we configured nginx to close client connections after each request. Apart from this, the default nginx configuration is used. Both servers were using one thread.

To generate load towards the HTTP server, we use the wrk HTTP benchmarking tool [147]. To generate load towards the 6Stor SN, we write a tool that reproduces the load generation model used by wrk with the 6Stor protocol (i.e. a certain number of threads, each maintaining a certain number of concurrent requests open, all requesting the same object). In order to minimize the impact of network events on the measurements, the benchmarking clients are run on the same machine as the servers.

Given the high TCP connection opening rate, the following sysctls are set to prevent the exhaustion of local ports by connections in TIME_WAIT state:

- net.ipv4.tcp_fin_timeout=5
- net.ipv4.tcp_tw_reuse=1

We measure the successful request rate attained by nginx and 6Stor for varying object size. The results shown in figure 2.9 are obtained for 16 client threads, 1 connection per thread, and each test run for 10 seconds. We found out that changing the number of connections per thread has no measurable effect, and increasing the number of threads beyond 16 does not significantly change the results.

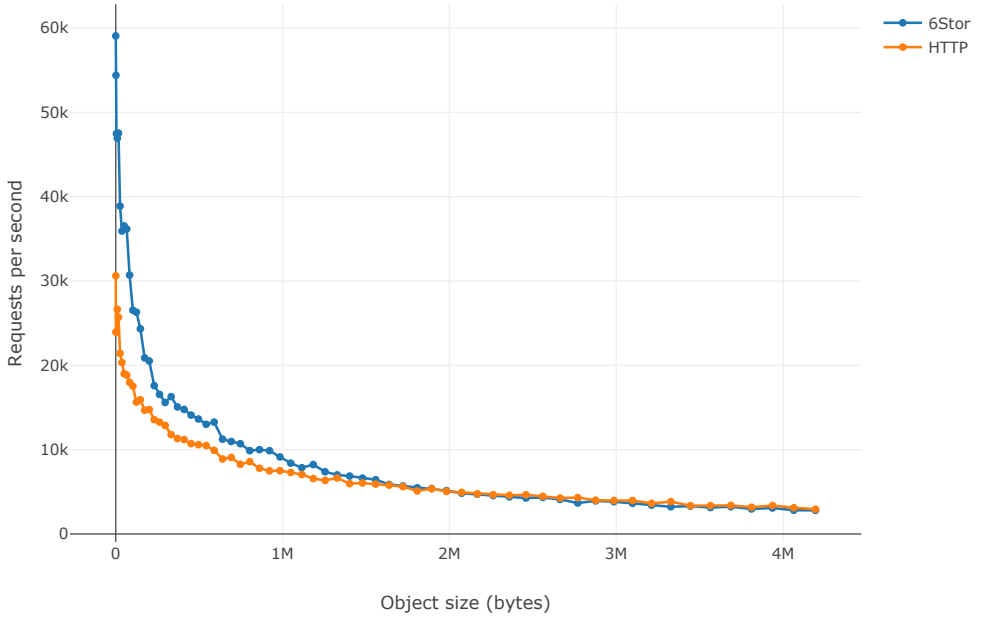


Figure 2.9: **Request per second per object size obtained with nginx and 6Stor**

Results

As expected, the difference is noticeable only for small objects, where the extra processing required to compute and transmit the headers is significant. For objects bigger than 2MB, the server spends the majority of its time sending the object, and the overhead of the HTTP protocol is negligible. For zero-size objects, 6Stor is able to process twice as many requests as nginx.

To conclude, it seems HTTP does indeed imply a consequent overhead, that is more noticeable for small objects. That validates our assumption but does not explain the performance difference showed for medium and big objects in our benchmark.

2.7 Conclusion

This chapter concentrated on distributed storage systems. We argued that with the rapid deployment of SSDs and their expected evolutions with regard to price and performance, existing storage systems must be reconsidered. Actually, CPU and network capacities are expected to become bottlenecks in the future, which poses a significant pressure on existing, heavily layered, systems.

We have described the characteristics of the two most common distributed storage systems, illustrated by Ceph and GFS. We have then introduced a new distributed storage architecture: 6Stor. This architecture uses a metadata layer accessed by consistent hashing and a flexible data layer to address common bottlenecks and loss of efficiencies in those previous systems, notably by shifting some of their complexity to the routing and network layers: every metadata and object is routable and accessible through specific IPv6 addresses at the object/metadata granularity.

After having described in depth how this architecture works and achieves resiliency, consistency and repairability, we presented an initial evaluation of 6Stor's performance in a few sets of benchmarks and, for indicative measures, how another similar distributed object storage system -Ceph- behaved for the same set of benchmark.

This benchmark consisted of bandwidth and latency measurements for post and get requests for datasets comprised of objects of different sizes, different replication factors, and different storage devices. The results point at the fact that our simple implementation, as expected due to the current lack of some fundamental functionnalities as well as to some design decisions, yields performance around at least as good as that of Ceph, and even better overall for Gets and for small objects for both Gets and Posts.

To evaluate the impact of one of these design decisions –the use of specific IPv6 addresses for metadata and data– we decided to compare the performance of a traditionnal HTTP server and a 6Stor Storage Node when distributing objects of specific size. As expected, using TCP rather than HTTP provides performance as far as twice better for zero-size objects, and about as good for objects of a size superior to 2 MB.

Through this work, we showed that it is possible to create an architecture that leverages the network layer to provide the same guarantees as traditional distributed storage systems while remaining fully flexible and scalable. It remains to be seen as the implementation progresses and if we can hold to the performance while providing all the required functionalities from a distributed object store. Furthermore, the flexibility of our architecture will have to be explored further to see how we can leverage it to improve its performance and bring additional features.

To this end, we plan to develop a more rigorous consistency model to adjust to circumstances (for example by not necessarily going through the first MN for an object every time, to allow for load balancing). The flexibility offered by the architecture could also be further exploited, for instance with placement and load balancing policies that integrate device heterogeneity, erasure coding, adaptation to popularity, QoS, geographic placement...

Another promising area is the integration of the storage system with the

networking stack. One direction is to optimize further the main operations (Post and Get), for instance by utilizing network stack tools such as VPP [148] to increase the overall resource utilization efficiency. We also investigate the use of segment routing [149, 150] to reduce the number of RTTs for basic operations and improve the load balance, by redirecting requests directly towards the SNs instead of having back and forth communication with the client. This particular point is addressed in chapter 3.

Likewise, we are investigating the use of Bit Indexed Explicit Replication (BIER) [151] for the replication operations to reduce the induced network overhead in order to diminish the network footprint of distributed storage systems by deduplicating replicated flows.

Contributions

This chapter contains work that has been disclosed in the form of a poster, papers, and patents.

- 6Stor: A Scalable and IPv6-Centric Distributed Object Storage System, poster in: 15th USENIX Conference on File and Storage Technologies (FAST 17) [152]
- Collapsing the layers: 6Stor, a scalable and IPv6-centric distributed storage system, workshop paper in: 2017 Fourth International Conference on Software Defined Systems (SDS) [153]
- An Initial Evaluation of 6Stor, a Dynamically Scalable IPv6-Centric Distributed Object Storage System, journal paper published in Springer's Cluster Computing Special Issue: "Software Defined Technologies for Computng Systems and Networking" [154]
- Distributed object storage, in US Patent App. 15/408,129 [155]
- Delivering content over a network, patent, not yet issued [156]

The 6Stor architecture was also the object of a Cisco Tech Fund, allowing us to hire two developpers for a period of one year to develop a working prototype beyond what I was able to develop myself. The code base, including all deployment and benchmarking tools as well as several APIs are currently in the process of being open-sourced.

Chapter 3

Extending 6Stor

3.1 Building a block device on 6Stor

A lot of distributed object stores offer a block device API in addition to their object API, because most applications rely on block or filesystem semantics rather than object semantics. Since filesystems are usually deployed on block devices, providing a block device is enough to satisfy the requirements of both block- and filesystem-semantics applications.

Moreover, distributed storage systems are often used to store Virtual Machine (VM) disks that themselves are accessed by a hypervisor as a block device. A similar use-case with containers is getting traction: individual containers' data is stored in storage systems accessed through a Container Storage Interface (CSI) that implements a way to create and interact with a logical block device in container platforms such as Google Kubernetes [157]. Consequently, the capacity to provide block storage is commonly expected from distributed storage systems.

For this reason, we decided to implement a block device on top of 6Stor.

3.1.1 Different implementations

Our block device works by using BUSE[1]. BUSE runs a Linux Network Block Device (NBD) client and server on the same machine and enables custom code to run on the server side, thus providing a custom block device in user-space. Our implementation is a translation layer in this custom user-space NDB server between block and object semantics as illustrated in figure 3.1.

In the first version, the block device itself is implemented as an array of blocks of equal size (by default 4KB) stored as objects in 6Stor. The name of the corresponding objects is in the form `\block_device_name:\block_index`. For I/O requests larger than one block, the driver sends parallel non-blocking requests for the corresponding objects and returns when they have all returned. When a read is unaligned, the whole object is retrieved and only the relevant part is returned. This naive implementation performs badly since

[1] https://github.com/acozzette/BUSE

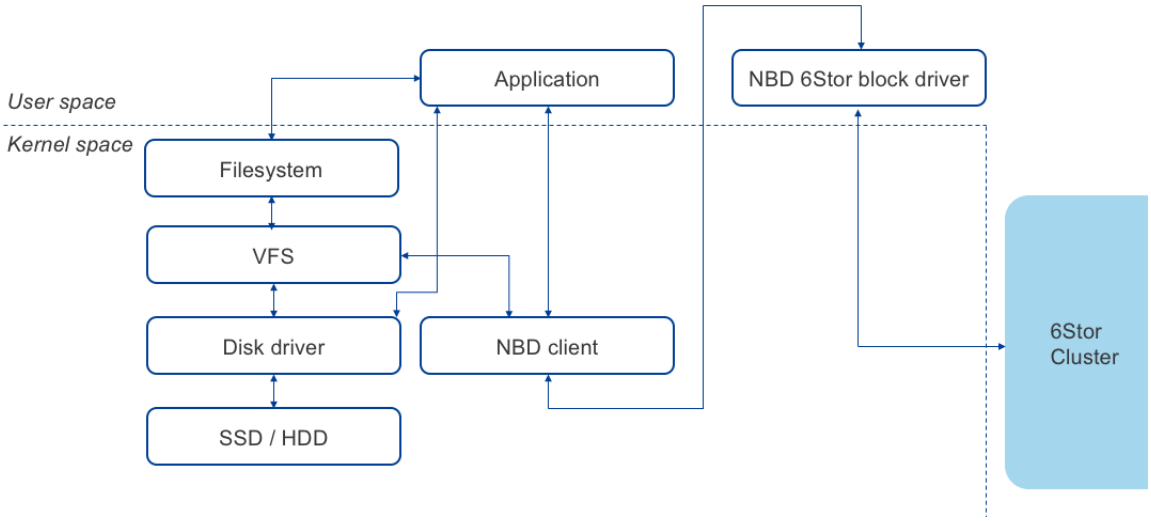


Figure 3.1: **BUSE: using NDB as a kernel virtual driver sending storage requests to a user-space custom 6Stor block driver.**

handling a 6Stor request per block is very inefficient.

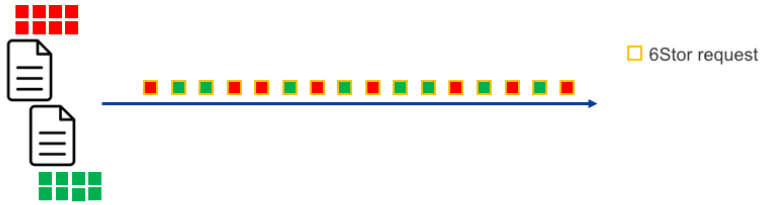
In the second version, we make two improvements:

- We use larger objects that contain multiple blocks at the same time;
- To avoid sequential I/O requests triggering multiple 6Stor requests for the same object, we implement a cache.

The cache size is defined when the block device is created and consists of a structure of block buffers that acts as a Least Recently Used (LRU) cache for the block device. I/O requests are handled sequentially, split in blocks and are only converted in 6Stor requests if the cache can not satisfy them. When a dirty block is removed from the LRU, it is written in 6Stor. The same thing occurs for every dirty block when the block device is closed. With this approach, large I/O requests are handled much more efficiently. However, obviously, the bigger the object size is, the lower the performance for small random I/O requests since the whole object has to be retrieved.

The latest version of 6Stor's block device allows parallel processing of I/O requests to take advantage from the inherent parallel processing capabilities from 6Stor. Indeed, while parallel requests can already benefit single hard drives through smart I/O scheduling, the benefit is even greater for a distributed architecture like 6Stor, since the parallel requests are dispatched on different disks and servers that can work independantly from one another.

The difference between the three implementations is illustrated in figure 3.2.



(a) First version: each block request is a 6Stor request.

(b) Second version: 1 object = 4 blocks, for each file, only 1/4ĥ of the block requests are sent as 6Stor requests.

(c) Third version: the I/O requests for the two files are processed in parallel.

***Figure 3.2:* Illustration of the three 6Stor block device implementations when reading two files in parallel.**

3.1.2 A note on caching and consistency

Caching in distributed storage systems often comes at the price of inconsistency: when a client caches data it just read, a posterior read fetches this data directly from the cache even if it has changed in the storage backend, unless the system integrates some update mechanisms like in AFS [38]. Likewise, when a client caches a write, other clients issuing reads before the cache is flushed see an outdated version of the data.

However, in the case of a block device, the expectation is in the vast majority of cases to be a single client: for VM disks, it is a single hypervisor running the VM, for filesystems, it is the Virtual File System (VFS) layer of the filesystem, for containers, it is a container interacting with the block device through its own filesystem. In these cases, there is a single cache through which the single client goes for every I/O operation, and thus there is no apparent inconsistency.

In the minority of other use-cases, either the applications running on top of the block device must tolerate the potential inconsistencies, or they must disable the cache and thus lose performance.

3.1.3 Performance benchmark

To test our early implementation, we benchmark it using Linux flexible I/O tester [2], also called fio. fio is a tool that spawns a number of threads sending read or write requests to a block device of a specific size and with a specific pattern (random or sequential). With this benchmark, one of our goals is to understand how the object size impacts the performance depending on the I/O pattern.

fio is called with the following parameters:

- Read, write
- Random access pattern
- I/O request of size [4, 16, 64, 256, 1024, 4096, 16384]KB

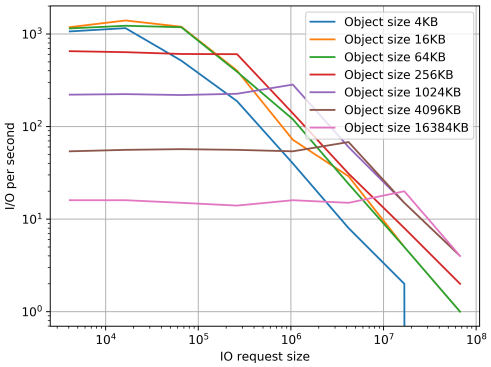
From our storage system perspective, a random I/O request of large size is strictly equivalent to several sequential requests of smaller size since the large request is split in smaller 6Stor requests anyway. fio sends requests following the chosen pattern continuously for 60 seconds, for each parameter set.

6Stor is configured in the following way:

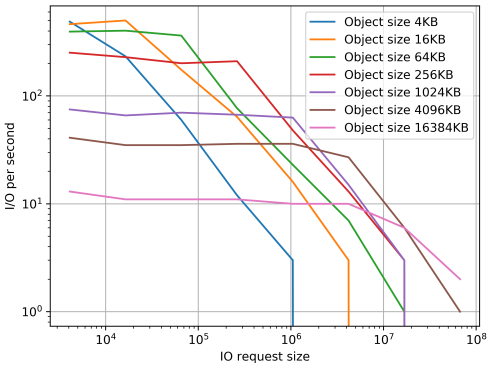
- $N_m = N_d = W_m = W_d = 3$
- The cluster is composed of [3, 4, ..., 16] servers
- The 6Stor block device API sends as many as 10 6Stor requests in parallel
- The 6Stor object containing the blocks are of size [4, 16, 64, 256, 1024, 4096, 16384]KB

For the rest of this section, we call *6Stor object size* the size of individual 6Stor objects containing the blocks (a 4MB 6Stor Object contains 1000 blocks) and *I/O request size* the size of individual I/O requests that are sent by fio in random patterns.

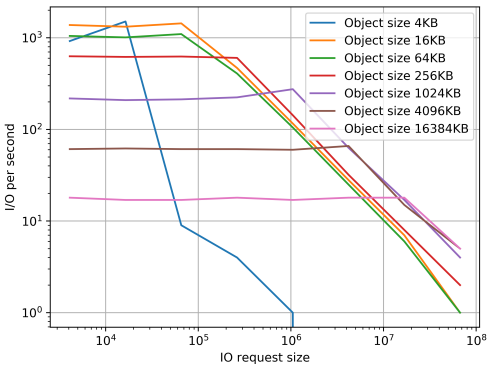
The cluster is the same as the one used in section 2.6 except that all drives are SSDs. The client is on a different rack than all the MNs and SNs, and is configured to have a cache of 64MB – similar to what most drives have as internal caches. For each global parameter set, the throughput and I/Ops obtained from the fio are registered. The results of this benchmark are presented in figures 3.3 and 3.4.



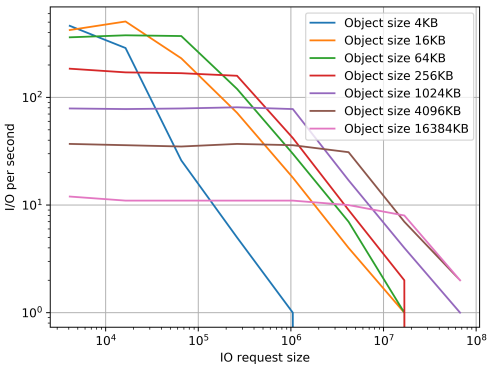
(a) I/O per second, 3 servers, read



(b) I/O per second, 3 servers, write



(c) I/O per second, 16 servers, read



(d) I/O per second, 16 servers, write

***Figure 3.3:* I/O per second benchmark results for 6Stor's block device**

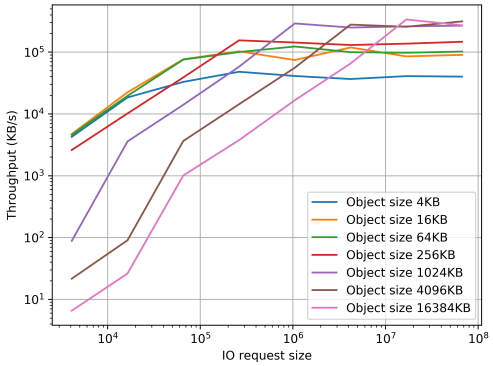
I/O per second results Figure 3.3 shows the I/O per second obtained by fio for 3 and 16 servers, for the read and write operations. Each individual subfigure show the I/O per second obtained by fio I/O request size for different 6Stor object size. As expected, the graph is flat for each object size until the I/O request size exceeds the object size: since the I/O request are random, each I/O request corresponds to at least one 6Stor object request: a 4KB I/O request correspond to a 4KB 6Stor object request for an object size of 4KB but to a 16MB 6Stor object request for an object size of 16MB.

However, for I/O requests larger than the object size, large objects become more efficient since they require less 6Stor object requests, each request corresponding to larger sequential reads on the physical drives. Furthermore, less 6Stor object requests generate less metadata interactions and thus less dRTTs and overhead. These graphs validate the intuitive fact that the optimal object size is around the order of magnitude of the expected I/O requests.

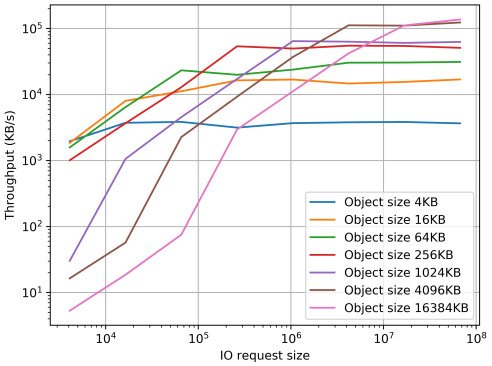
Finally, as expected, there are more I/Ops for reads than for writes, since writes must be replicated whereas reads just read from a single 6Stor object

[2] https://linux.die.net/man/1/fio

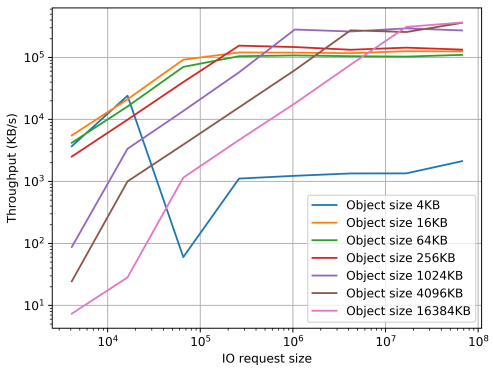
replica.



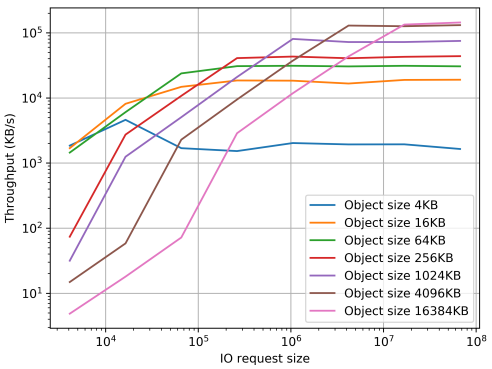
(a) Throughput, 3 servers, read



(b) Throughput, 3 servers, write



(c) Throughput, 16 servers, read



(d) Throughput, 16 servers, write

Figure 3.4: **Throughput benchmark results for 6Stor's block device**

Throughput results The throughput result shown in figure 3.4 show the same story as figure 3.3 through a different lens: the throughput for large 6Stor objects increases with I/O request size because the I/O per second are constant but the I/O request themselves increase in size. The throughput for an object size reaches a plateau soon after the I/O request size order of magnitude increases compared to the object size. Again, the throughput for writes are between 2 and 3 times lower than for reads, as expected because of the replication factor of 3.

Additional considerations There is however an unexpected result. Because of the parallelisation described in section 3.1.1, we expected the results for 16 servers to be widely better than the results for 3 servers. Surprisingly, our results are substantially the same for every number of servers in the cluster, which is why we showed them only for 3 and 16 servers in the cluster.

We are currently investigating this result and think ther explanation might lie in the number of concurrent 6Stor operations being too low: 10 concurrent 6Stor operations might not be enough to take advantage of the increased capacity for parallelism when the cluster size increases.

We put a single SSD to the same fio benchmarks on a local server (obviously only varying the access patterns and I/O request size since 6Stor is not involved) to compare our results to what a local disk could provide. The local SSD has on average between 3 and 10 more I/O per second and throughput than our block device. This can be partly explained by the network RTTs triggered by metadata operations, taking several microseconds that highly matter at this level of performance, but also by our first crude implementation for the block device client, that is probably not yet optimized to its full potential. The order of magnitude we reached for both I/O per second and throughput is comparable to what state of the art distributed storage systems acting as block devices such as Ceph [28] or Cloudian [158] provide.

While we implemented this block device on 6Stor, almost none of what we did is specific to 6Stor. One could go through the same steps to create such a block device on top of any distributed storage systems providing a generic object store API.

3.2 Decentralized dynamic load-balancing for distributed storage systems using Segment Routing

Distributed storage systems use replication or erasure coding to provide reliability for the data they store. Both these techniques consist of storing the data on several storage nodes to ensure that even when a certain number of storage nodes fail, data is not permanently lost. This means that when data is uploaded to or retrieved from a storage system, there are several candidate storage nodes between which to choose, and thus an opportunity to balance the load of the different storage nodes in the cluster.

3.2.1 Load balancing in distributed storage systems

There are two different stages when distributed storage systems can balance the load between storage servers. The first one is during data uploading, when the multiple data replicas or fragments are written to storage nodes, and the second one is when choosing which storage nodes to retrieve the data from. The latter is mostly relevant in storage systems using replication rather than erasure coding, where load balancing is more straightfoward since it is only a matter of choosing which identical replica to fetch, whereas most erasure-code-based systems rely on retrieving explicitely the k systematic

fragments as described in section 1.2.3, leaving no room for load-balancing in read operations unless some additional replication is done[3].

Writing data On most distributed storage systems, data placement is organized by a DHT-like mechanism. This is the case for Ceph, Cassandra and Dynamo [28, 27, 23]. In those systems, the load is statically balanced – usually with additional reliability constraints forcing replicas or fragments to be on different storage nodes/racks/datacenters – between nodes according to their storage capacity. However, as explained in chapter 1, DHTs do not allow for dynamic load balancing. Because object popularity can often not be predicted, this can lead to hot spots reducing the performance on some storage nodes [161, 51].

Other distributed storage systems –mostly distributed file systems– use a metadata layer that allows them not to have consistent data placement. However, they mostly take into account storage capacity and reliability constraints, not the dynamic load of servers. This is expected since this load can vary very quickly and it is not realistic to expect the nodes taking data placement decisions to know the exact load of every storage node at every instant. As such, it is possible for storage requests to be directed to storage nodes under a high I/O, network or CPU load even though some other storage nodes would be better candidates at that time.

Finally, some distributed systems move and replicate data constantly to adapt to popularity and load variations [162, 51]. While these solutions provide an effective load balancing, they require heavy data transfers and can not adapt to sudden changes of popularity since they require a constant synchronization and analysis of the data access patterns.

Retrieving data We focus on the case of replicated distributed storage.

Because there are several replicas to choose from, it is possible to load balance between them when retrieving data. When combined with placement policies, it is also possible to pick the best replica according to, for example, geographical position. For instance, icks the replica closest to the reader node, on the same rack/datacenter if possible.

Other systems such as Dynamo from Amazon send the get requests to all nodes storing the replica and acknowledge the response when enough nodes have sent back the same version of an object ("enough" being the R parameter in (W, R, N) quorum systems described in chapter 1).

In other cases, requests are all sent to the same replica. This is the case for Ceph, where all requests are sent to the replica corresponding to the first placement group obtained from RADOS [63], as well as for GFS, where each

[3]However, some work explore the possibility of also fetching $r' \leq r$ parity fragments and choosing the first k out of $k + r'$ fragments to reconstruct the object, which is a form of load balancing [159, 160]

set of chunk replicas periodically elects a master replica that is the one clients interact with.

None of these approaches takes into account the real-time load of storage servers, either for writing or reading data. As explained before, this means that requests can not always be steered out from hot spots when they occur [161, 51]. Work has also been done to regularly redistribute data according to popularity and application requirements, but this requires a constant reshuffling of the data, and does not react to real-time load changes in storage servers [162].

3.2.2 Segment-routing load-balancing

6LB [163] is a distributed, application load-aware, network-level load-balancer that leverages Segment Routing (SR) [149] to allow applications to take load balancing decision locally, based on their real-time, application-defined load. It incurs a minimal network overhead (due to the usage of segment routing) and requires no out-of-band signaling. Furthermore, decisions are made purely locally and do not depend on a centralized monitoring system – that could limit scalability and not react quickly enough to load changes.

Segment routing is an IPv6 service that permits directing IP packets (regardless of the protocol) through an ordered set of intermediaries, and instructing these intermediaries to perform specific function. In 6LB, this function is to process the query contained in the data packet if not too busy, to forward it to the next intermediary otherwise. SR information is defined in [164] and is expressed as an IPv6 Extension Header that comprises a list of segments under the form of IPv6 adresses and a counter `SegmentsLeft` that indicates the number of remaining segments to be processed.

6LB's architecture works as follows: edge routers use Equal Cost Multi-Path (ECMP) to assign incoming flows towards an application identified by a single Virtual IP address (VIP) a to load-balancers. For each flow, its assigned load-balancer chooses an ordered list of candidate instances running a, to which it forwards the flow's first packet (usually a TCP syn) using SR. Each intermediary in the list of candidates either accepts or forwards the flow, based on its own real-time state information about itself. If the flow reaches the last segment of the list, the candidate has to accept the flow. Once the flow is accepted by an instance, it is assigned to it by the load-balancer so that subsequent packets form the same flow are forwarded to the correct instance. Additionally, packets from the instance to the client do not go through the load-balancer. This architecture is illustrated in figure 3.5.

In this figure, a load balancer LB intercepts a TCP syn from a client c to a VIP a identifying an application, selects 2 out of 3 running instances of application a on servers $S1$ and $S2$, inserts a SR header containing $S1$'s and $S2$'s addresses, and forwards the syn to $S1$. In this scenario, $S1$ is under a heavy

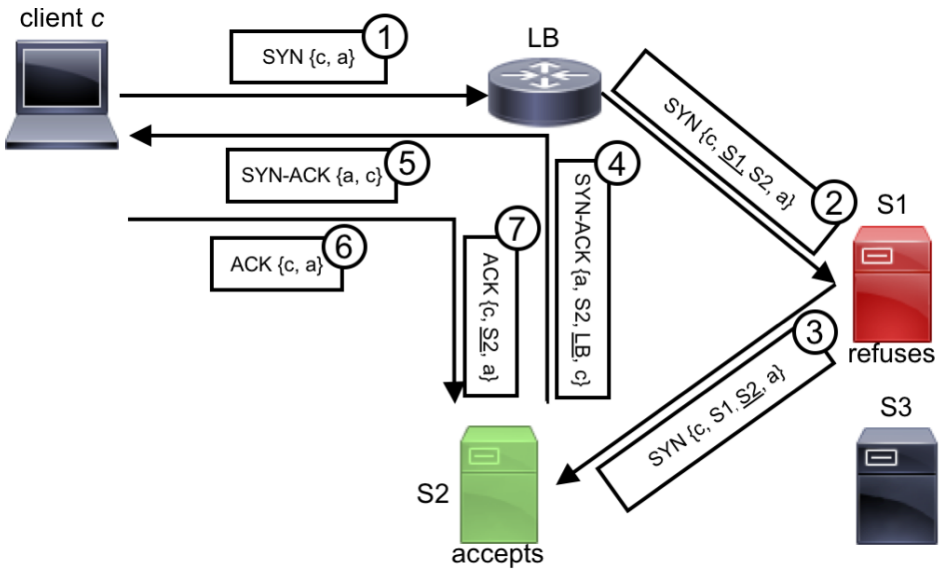


Figure 3.5: **6LB hunting example as found in [163].**

load and thus chooses to refuse the connection. $S2$ accepts it and sends the syn+ack back to c through LB, allowing LB to create an accurate flow-state. For further traffic, packets coming from c go through LB while packets coming from $S2$ are directly routed to c. This approach works very well when few data is exchanged or when data flows mostly from a to c, because LB is on the data-path only on the way from c to a.

3.2.3 Adapting 6LB to 6Stor

6LB has been designed for applications running several independent instances. By essence, servers of a distributed storage system are not independent: when data has been written on servers, it can only be retrieved from these servers. Besides, 6Stor does not use a single VIP but rather a set of adresses per object identifying data and metadata, as described in section 2.2. Therefore, we adapt 6LB the following way to fit with 6Stor's architecture.

Metadata Nodes As described in section 2.2, clients send post requests to the first MN of the list obtained by hashing. This MN is then in charge of creating the metadata itself and replicating it to the other adresses. This task is more CPU-intensive than the task of just receiving the metadata. Therefore, an SR-enabled client can, instead of sending its initial packet to the first MN, send it to an SR-list containing all the corresponding metadata adresses as segments. Then, each MN on the list can accept or refuse the task depending on its instant load, forwarding it to the next segment when refusing. The consequences on consisteny are explored in section 3.2.4.

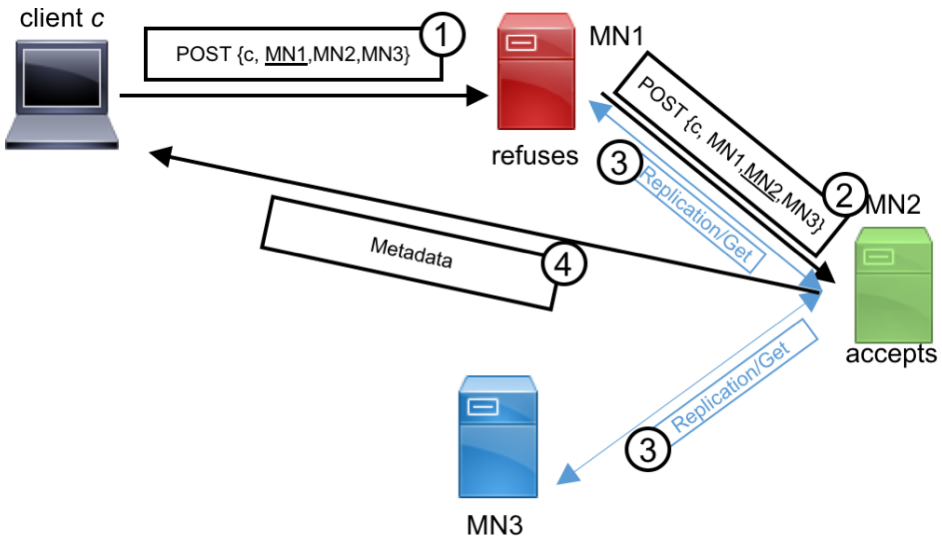


Figure 3.6: **6StorLB metadata hunting. In this example, the first MN refuses to create/get the metadata. The second MN accepts. Before returning the metadata to client c, it either waits for R_m^a metadata to be read or W_m^a metadata to be replicated.**

The same can be done for a Get request. However, the benefit is lower since retrieving metadata is less computing-heavy than creating it. The mechanism is illustrated in figure 3.6.

Storage Nodes Instead of selecting N_d SNs when creating the metadata, the MN in charge of the object post selects N_d *lists* of SNs and sends it back to the client. The client sends a TCP SYN packet to the primary list. The first SN in this list to accept the connection similarly sends TCP SYN packets to the secondary lists. Every SN at every step (except the last of each list) is free to refuse or accept the connection based on its real-time I/O, network and/or CPU load. Thus, even temporary hot spots can be avoided, as illustrated in figure 3.7.

Get requests work for SNs the same as they work for MNs. One of the SNs accepts the responsibility of retrieving R_d^a replicas to answer to client c.

3.2.4 Consequences on consistency

The main difference in regular operations with what was described in section 2.2 is that the first MN is not always answering to metadata requests. To guarantee consistency for basic operations, we need to switch from a modified $(N, W, 1)$ quorum model for metadata to a regular (N, W, R) quorum model.

Since we have not yet implemented this algorithm, it is unclear whether or not the benefit of the added load balancing counterbalances the obligation

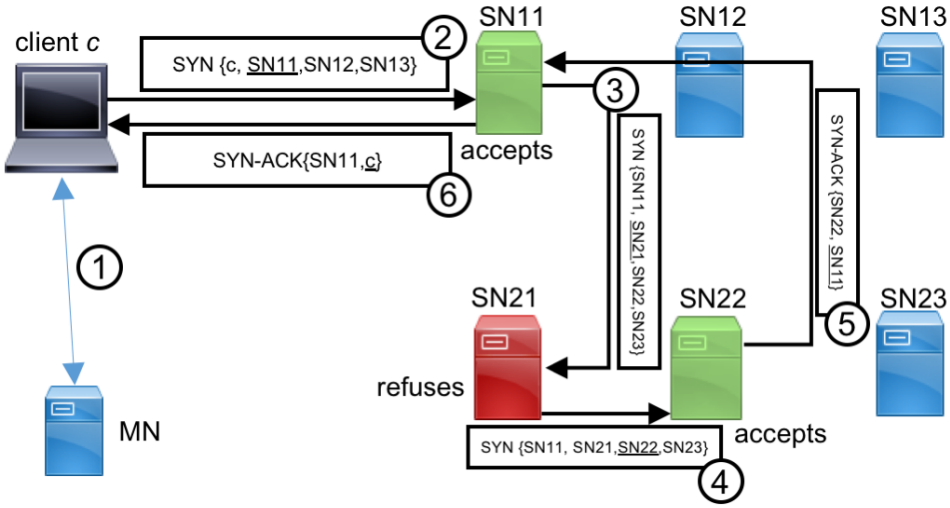


Figure 3.7: **6StorLB storage hunting. In this example with $N_d = 2, W_d^a = 2$, the first SN of the first list accepts to store and replicate the object. The first SN of the second list is under stress and refuses but the second SN of the second list accepts. Once W_d^a SNs have accepted the connection, c begins sending data to $SN11$.**

to adapt to a (N, W, R) model for the metadata interactions. However, while there is a trade-off involved for metadata operations, this trade-off does not exist for interactions with SNs themselves. Because MNs are aware of which SNs contain the up-to-date versions of objects, they can build an appropriate SR-list to be used by clients when requesting objects without risk of inconsistency, thus benefitting from this load-balancing with no negative consequences.

3.3 Conclusion

In this chapter, I presented some extensions made to 6Stor's main architecture. Section 3.1 explained why implementing a block device on top of 6Stor's object store was important, how we achieved it, and how it performed, reaching throughput lower but with the same order of magnitude than what was achieved on a local disk and comparable to state of the art products. We also showed and explained the how the size of objects containing the blocks themselves and the I/O access patterns impacted the performance of the block device. Our block device prototype showed unexpected results when varying the number of servers in the clusters, which is currently under our investigation.

Section 3.2 described how we leveraged IPv6 and segment routing technology to enable stateless, server-driven load-balancing. While the algorithms presented in this section have not been implemented, we expect it

to yield similar improvements to what was showed in the paper describing the original approach from which we took our inspiration [163]. Such an implementation would allow us to determine if the benefits from load balancing counterbalance the need to fall back to a regular (N, W, R) model for metadata operations. In any case, enabling it only for interactions with SNs enables load-balancing without negative consequences on consistency.

Contributions

The block device described in this chapter was not presented nor patented, although the results on the performance impact of object size and I/O request size are interesting. Furthermore, the block device implementation is a part of the code base that is currently being open sourced.

The SRLB adaptation to 6Stor is the object of a patent pending.

- Reducing distributed storage operation latency using segment routing techniques, patent, not yet issued [165]

Chapter 4

Guaranteeing low response time for small requests and fair throughput allocation: a Request Scheduler for Storage Servers (RS3)

Storage servers are the elementary brick of many distributed systems: from web servers (Nginx [146], Apache Traffic Server (ATS) [166]) to entire Content Delivery Networks (CDNs) (Akamai [167], Amazon Cloudfront [168], Microsoft Azure [169]) and clusters of distributed storage systems storing petabytes of data (Ceph [28], Amazon Dynamo [23], HDFS [29]). In all those cases, storage servers are usually composed of commodity hardware and only involved in storing and retrieving data.

With the increasing interest in hyper-converged architectures [170, 171], it is more and more common for multiple applications or clients to simultaneously interact with a single, large-scale architecture spanning thousands of storage servers. When that happens, every individual storage server has to store and retrieve data exhibiting a high variance in size and/or access pattern, corresponding to different types of data objects (VM disks, videos, log files, small images, container layers...).

When applications handling very different data interact with the same storage server, it is crucial to minimize the interference between them and to enforce a fair access to the underlying storage resource. For similar reasons, scheduling algorithms have been developed for various use cases, such as sharing network resources (packet scheduling), computing resources (CPU scheduling) or for efficient ordering of disk I/Os (I/O scheduling). They are essential to limit the impact of such resource contention. However, existing I/O schedulers are mostly designed for multiple processes sharing the same resource on servers.

In the cloud architectures described earlier, it is common for storage servers to run only one process per server or per disk. I/O schedulers remain essential to reorder disk block requests and guarantee an efficient usage of hard drives, but they do not have the necessary information to discriminate between applications using the storage systems themselves. This leads to

multiple applications or users contending for resources, possibly degrading global performance and experiencing significant variance in the underlying service response time [172].

This variance for individual requests in a storage system can quickly add up and have a significant impact on the service level of applications running on top of it. Indeed, such applications commonly translate end user requests in several underlying storage requests, amplifying the variance of the storage system itself [173]. This variance can, in turn, have disastrous economic consequences [20, 21, 22], since it has been shown that increasing user requests' latency significantly reduces the revenue of cloud-based services.

This chapter's main focus is to provide a solution to the two following issues that arise in storage systems:

- small requests are slowed down by concurrent large requests and thus have a response time disproportionate with their size and with a high variance, as shown later in section 4.3.3;
- users that request more data are rewarded with more resource usage which can encourage bad behavior, as shown later in section 4.3.2.

This chapter presents RS3, a request scheduler that works at the storage node granularity, and dynamically ensures that incoming requests from different users or services are handled in a *fair* manner. RS3 batches requests and slices them according to a global budget allocation. Each request class is allocated an equal share of the budget, which is reallocated to other classes whenever it is not entirely consumed during a batch.

Section 4.1 describes related work on scheduling mechanisms in general, notably in the context of packet switching and I/O schedulers, but also gives examples of system-wide schedulers. Section 4.2 describes RS3's batch allocation algorithm, how it works and what it guarantees. Section 4.3 gives a first set of results and interpretations. Section 4.4 describes optimizations taking advantage of Linux filesystem mechanisms to reduce the negative impact of RS3's batching on the total throughput of storage servers. Finally, section 4.5 presents further results aiming at fully understanding the impact of RS3's parameters, with a focus on CPU consumption. Moreover, this section presents W-RS3 (Weighted-RS3), an extension to RS3 that can be used to enforce service level agreements or quality of service contracts, and could also be used to enforce system-wide fair scheduling with more granularity.

4.1 Related work

Various scheduling techniques are used to guarantee efficiency, enforce fairness, or provide Quality of Service (QoS) with regard to specific resources,

notably network and disk access.

4.1.1 Packet scheduling

A large literature exists on packet scheduling which has been used for decades. They are essential component for insuring QoS (Quality of Service) for specific class of services or to enforce fair distribution of ressources between classes (aggregated traffic) or flows, depending on the considered granularity.

Weighted Fair Queuing (WFQ) [174], Class-Based Queuing (CBQ) [175] and Start-time Fair Queuing (SFQ) [176] are well known approaches to this problem, and have been improved upon by work such as [177]. These approaches define different traffic classes to enforce fairness or priority policies based on this classification. However, our problem, while sharing similarities, can not be solved by network level scheduling alone, since network itself is not the bottleneck in a majority of cases.

4.1.2 I/O scheduling

I/O scheduling algorithms have been developed following the improvements in disk technology, first for HDDs then for SSDs. Conventional I/O schedulers are mostly designed to reduce the impact of disk seeks and rotations with anticipation and I/O reordering techniques [178]. Some additionally make use of deadlines to take into consideration latency issues and provide QoS to applications, such as Facade [179], pClock [180] or others [181, 182, 183]. Those schedulers improve disk efficiency while maintaining acceptable maximum latency for applications. However, they do not ensure fairness between different processes I/O requests on servers.

Besides, an other family of schedulers exists, that implement the same mechanisms as their network counterpart to achieve fairness in addition to good disk performance, like Depth-based Start-Time Fair Queuing (SFQ(D)) [184] or YFQ (for Yet another Fair Queuing algorithm) [185]. Some schedulers also introduce the notion of per-task quanta to achieve fairness such as Argon [186] or the standard Linux Completely Fair Queuing (CFQ) [187].

A lot of more recent work has been done to adapt these techniques to SSDs, with specific I/O characteristics like read/write assymetry and channel parallelism. For example, Flash I/O Scheduler (FIOS) [188] uses I/O anticipation and read/write separation to ensure fairness and performance for flash devices. These characteristics are also leveraged by other work on the subject [189, 190, 191, 192], for traditional SSDs and NVMe alike.

Finally, mClock [193] provides fine-grained I/O scheduling specialized for VM disks, notably with optional minimum and maximum boundries as well as weighted throughput allocation. However, it requires precise allocation and configuration to function correctly, and by design works well with few classes with well-defined requirements and behaviors that can easily be modeled – corresponding to disk VMs, whereas our approach requires no additional configuration and works for any number of classes.

4.1.3 System-wide scheduling

Some work has also been done on system-wide scheduling. For example, [162] proposes a mechanism to ensure system-wide (rather than per-server) efficient and fair resource distribution. However, the solution requires a constant and costly rebalancing of data and relies on systematic evaluation and extrapolation of object popularity per client . While this works for slow and predictable changes of popularity, it does not work well for fast changes of popularity. Furthermore, it does not scale with the number of different clients and objects since it requires a centralized computation of the popularity estimations.

In [194], an algorithm is designed to ensure fairness in cloud systems with heterogeneous servers. The aim of this work is to optimize resource allocation for applications requiring a variable amount of various resources on pools of servers that are themselves highly heterogeneous with regard to the resource they can provide. Their focus is more on centralized multi-resource management and allocation rather than per-server scheduling specifically for storage servers.

IOFlow [195] is a Software Defined Storage (SDS) architecture that enable end-to-end policies between VMs and their associated backend storage by creating a programmable data plane. However, this approach requires a specific provisioning and a knowledge of the I/O requirements and patterns of the VMs, while this chapter aims at improving generic storage systems with no further assumption about the users.

To conclude, all these system-wide approaches are complementary to the approach we develop in the next sections: they ensure that requests are fairly allocated between servers, while our algorithm guarantees the fair treatment of these requests at the server level.

4.2 Designing RS3

For the remainder of this chapter, we define a *fair* storage system as having these characteristics:

1. users whose received throughput is limited by the storage server should all receive the same throughput;

2. users' received throughput should not depend on requests' size;

This definition of fairness applies to a theoretical perfect storage system with no fixed overhead per request. We already showed that such a fixed overhead exists in section 2.6.7, but estimating this fixed overhead or the impact of physical disk characteristics on service time is not in the scope of this chapter. A discussion on the different possible adaptations of fairness and how they would be implemented in RS3 can be found later in section 4.5.3.

With such a definition, a malicious user can not engineer its request patterns to obtain more service throughput than other users when interacting with a storage system. For example:

1. let us consider a fair storage server limited by disk I/O. A user continuously sends requests for 1MB objects. A second user continuously sends in parallel two requests for 1MB objects. The fair storage server should deliver the same throughput to both users;

2. let us again consider a storage server limited by disk I/O with two users. The first user continuously sends requests for 4KB objects. The second user continuously sends requests for 4MB objects. The first user should be able to satisfy 1000 times more requests than the second one in the same timeframe.

This definition of fairness – inspired from the max-min fairness concept introduced in [196] and broadly used in communication networks – ensures that the storage device capacity is equally allocated to all users, and that no user can increase its share of access at the expanse of another user with an equal or lower share.

4.2.1 Typical storage server implementation

Before introducing RS3, let us describe a baseline version of storage server, which we aim to build upon. A typical implementation for a storage server relies on several threads running an event-loop making `sendfile`[1] calls to read the data on a local file and copy it without additional superfluous user-space copies of the data to be sent in a socket buffer as well as additional system calls and costly context switches.

`sendfile` is typically called as soon as a socket is marked as writable in the event loop. The size of the data read on the filesystem and sent by the `sendfile` call is one of the fixed system parameters (called *read size* in the remainder of this chapter), usually between 64 and 256KB. `sendfile` is a blocking call (on the disk I/O side) and thus when multiple sockets are writable, the next one to be served is typically the first that marked itself as available. The consequence is that `sendfile` is called in a round-robin-like fashion between all simultaneous connections. This is, for example, the way NGINX

[1] `https://linux.die.net/man/2/sendfile`

[2] [146], a widely-used high-performance open-source web server, is implemented [3].

The average duration of a `sendfile` call getting its data from the disk depends on the *read size* parameter. Therefore, the duration of such a round-robin loop over every client socket grows in average linearly with the number of users. Because a new request has to wait an average of one such loop, the response time for small requests (smaller than the fixed *read size* and thus completed in a single loop) grows linearly with the number of users, even if they request much less than the *read size* parameter. Furthermore, this also means that users that aggressively send multiple requests in parallel usually receive more data since they are served multiple time per loop, at the expense of other users.

A simplified implementation of such a single-threaded storage server is presented in algorithm 1.

Algorithm 1 Standard storage server implementation

$sockfds$: Array containing active sockets
$sockfdN$: Number of active sockets
fd_{sockfd} : File descriptor of the local file associated with socket number $sockfd$
$read\_size$: Fixed size of read calls to the filesystem
function STANDARD_REQUEST_LOOP
    while $sockfd = poll(sockfds, sockfdN)$ **do**
        $sendfile(sockfd, fd_{sockfd}, read\_size)$
    end while
end function

RS3 aims at solving both these issues by sending data to users in fixed-sized (in bytes) *batches* and defining *classes* that each receive a fair share of the storage device access during each of these batches.

4.2.2 RS3's rationales

RS3 is designed to be used in a multi-tenant or multi-classes distributed storage system. For the remainder of this chapter, we discriminate requests based on their *class*. A *class* can designate a tenant, a type of data stored, an application, or any other imaginable way to differentiate between requests.

RS3 runs on every node in the storage cluster and does not rely on a centralized control plane, thus allowing it to scale horizontally with the number

[2] `http://nginx.org/`
[3] `https://github.com/nginx/nginx/blob/4bf4650f2f10f7bbacfe7a33da744f18951d416d/src/os/unix/ngx_linux_sendfile_chain.c`, line *259*, called at line *174*.

of servers and to react dynamically to very fast workload changes. However, RS3 only provides per-storage-server fairness.

4.2.3 RS3's batch budget allocation algorithm

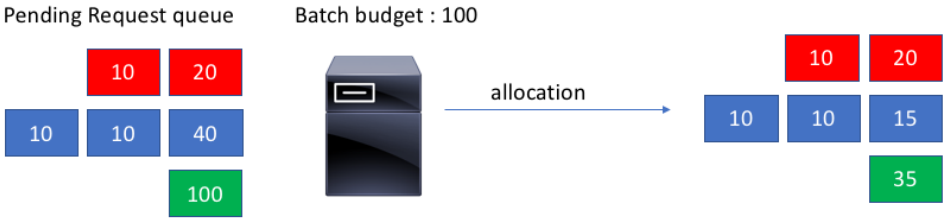
RS3 follows the same mechanism as traditional quanta-based I/O schedulers [187, 186], by allocating a fixed *budget* to the different active classes. For each batch, the budget is equally distributed to all classes. The budget left by classes requesting less than their share (or limited by their network) is then redistributed to other classes, as described in algorithm 2 and illustrated in figure 4.1.

Algorithm 2 Batch budget allocation algorithm

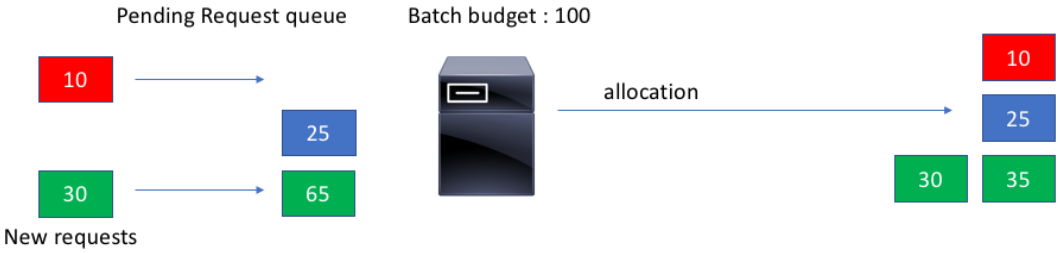
1: B: batch budget
2: N: number of active classes
3: $R[i]$: pending cumulative size of requests made by active class i, ordered from lowest to greatest
4: $T[i]$: size of the TCP send buffer associated with class i
5: $B_a[i]$: budget allocated to class i for the batch
6: R_b: Remaining budget for the batch
7: **function** BUDGET_ALLOCATION(B, N, R, T)
8: $\quad R_b \leftarrow B$
9: $\quad$ **for** $i = 0$ to $N - 1$ **do**
10: $\quad\quad B_a[i] \leftarrow min(T[i], \frac{R_b}{N-i}, R[i])$
11: $\quad\quad R_b \leftarrow R_b - B_a[i]$
12: $\quad$ **end for**
13: $\quad$ **return** B_a
14: **end function**

This algorithm has the following properties:

- the duration of a batch depends on the *batch budget* B, not on the number of active classes N;
- consequently, the lowest possible response time depends on the batch budget B and not on the number of active classes;
- large requests are not throttled *too much* in the presence of numerous small requests, as the budget not consumed by small requests is reassigned to them;
- classes that send large requests receive an equal share of the disk access (if they have sufficient network capacity);



(a) A first pass of the algorithm is done to allocate budget to classes (represented by different colors). A second pass is done per class to allocate the budget per request.



(b) New requests have arrived during the first batch. They are added to the pending requests queue and dealt with during the second iteration of the budget allocation algorithm.

***Figure 4.1:* An example of two batch budget allocations for a batch budget of 100, with 3 classes.**

- the algorithm scales in $O(N)$ with the number of active classes if the classes are already ordered by cumulative pending requests size at the beginning of the allocation computation[4], $O(N \log N)$ otherwise.

For classes that have several pending requests, a second pass of this algorithm can be made to allocate the class budget to each individual request of the class, to ensure that a class's small and latency-sensitive requests don't have to wait for a large request from the same class to complete. This is only optional and depends on how classes function. Depending on the types of objects they request, they might prefer to fulfill their requests purely sequentially. The example presented in figure 4.1 hierarchically reallocates budget inside classes rather than treating requests sequentially.

After the budget allocated to each request is calculated, a single `sendfile` call is made for each request with its allocated budget for size. After the last sendfile returns, a new batch starts, containing all the requests that have not yet been fulfilled and new requests that have arrived during the previous batch.

[4]For example, a thread can be in charge of accepting incoming connections, assigning requests to their classes and reordering them while another thread loops on the batches themselves.

4.3 First evaluation and analysis

In this section, we evaluate the impact of our algorithm on 3 factors: the fairness of the disk usage repartition between classes, the response time for small requests, and the overall throughput of the system. To this end, we compare two implementations of a storage server working with a single disk, with and without RS3. The regular implementation is based on algorithm 1: it polls [5] active, non-blocking sockets and calls `sendfile` as soon as one is writable. RS3's implementation works by batching the requests before calling `sendfile` towards non-blocking sockets once for each request in the batch as illustrated in algorithm 3. To simplify comprehension, RS3's algorithm is presented in the case where each active class has a single pending request for a single object, and thus the budget allocation phase is done in one pass.

These functions are performed by a thread while a separate thread accepts incoming connections, analyzes requests and opens corresponding file descriptors.

Algorithm 3 RS3 storage server implementation

$sockfd[i]$: Active socket descriptor for class i
$sockfdN$: Number of active sockets
fd_{sockfd} : File descriptor of the local file associated with socket number $sockfd$
$R[i]$: pending cumulative size of requests made by active class i, ordered from lowest to greatest
$T[i]$: size of the TCP send buffer associated with class i
B : Batch budget
function RS3_REQUEST_LOOP
  while *True* **do**
    $B_a = Budget\_allocation(B, sockfdN, R, T)$
    for $i = 0$ to $N - 1$ **do**
      $sendfile(sockfd[i], fd_{sockfd[i]}, B_a[i])$
    end for
  end while
end function

4.3.1 Experimental protocol

Both a regular and RS3 version of storage server are implemented using a basic custom protocol above TCP and tested on a storage node using a single hard drive. The storage server is equipped with a Toshiba 600 GB 2.5 inches 6H SAS 15K RPM hard drive and two Intel Xeon E5 CPUs (2.60GHz, 14 hyper-threaded cores). A custom client able to act as multiple classes at

[5] https://linux.die.net/man/2/poll

the same time is deployed on another identical server connected to the storage server with a 10Gbps link. Both the client and the storage servers are running on Ubuntu 18.04.1, kernel version 4.15.0.

The storage server is pre-populated with a set of objects of varying size. To nullify the influence of the hard drive cache on the results, none of the objects is served more than once within an experiment. Furthermore, Linux's filesystem cache is emptied between each experiment. Each of the data points used to create the graphs presented in the remainder of this chapter correspond to a 60-second read test during which the response time of all requests and the CPU consumption of the storage server are measured.

4.3.2 Throughput fairness results

Firstly, we test how our storage servers react when multiple classes continuously send requests, with some of them sending multiple requests simultaneously. We want to verify that each class receives a throughput proportionnal to their request rate with the standard implementation, and equal to what other classes get with RS3. To that end, we use our client to simulate two classes. The first class has 2 threads concurrently and continuously sending requests for objects of size between 1 and 4MB. The second class has a variable number of threads doing the same. The results with a batch budget of 24KB can be found in figure 4.2.

Results displayed in figure 4.2 show that RS3 is very effective at equalizing the throughput of both classes. Indeed, for the extreme case of 16 concurrent threads for class 2 and 2 threads for class 1, class 2 utilizes 88% of the total throughput without RS3 and only 51.5% with RS3. However, there is still a small discrepancy between both classes and a slightly lower overall throughput for the storage system. The discrepancy comes from the "transition" batches between two requests: some batches are underutilized by class 1 if both the threads come at a request's end and the class therefore underutilizes its allocated budget. Furthermore, class 1's threads do not always have time to send new requests before a new batch begins. Moreover, because class 1's requests finish faster since there are less requests in parallel, there are even more such transition batches that are underutilized.

To evaluate the impact of these transition batches, we test how fairness between classes varies with the batch budget, to verify that fairness decreases when the batch budget increases. Thus, our client simulates 5 classes, each of them having a different number of concurrent threads continuously sending requests as described before, and we vary the batch budget. The results are presented in figure 4.3.

It appears that a batch budget under 32KB ensures almost perfect fairness with the current settings. As expected, the impact of transition batches

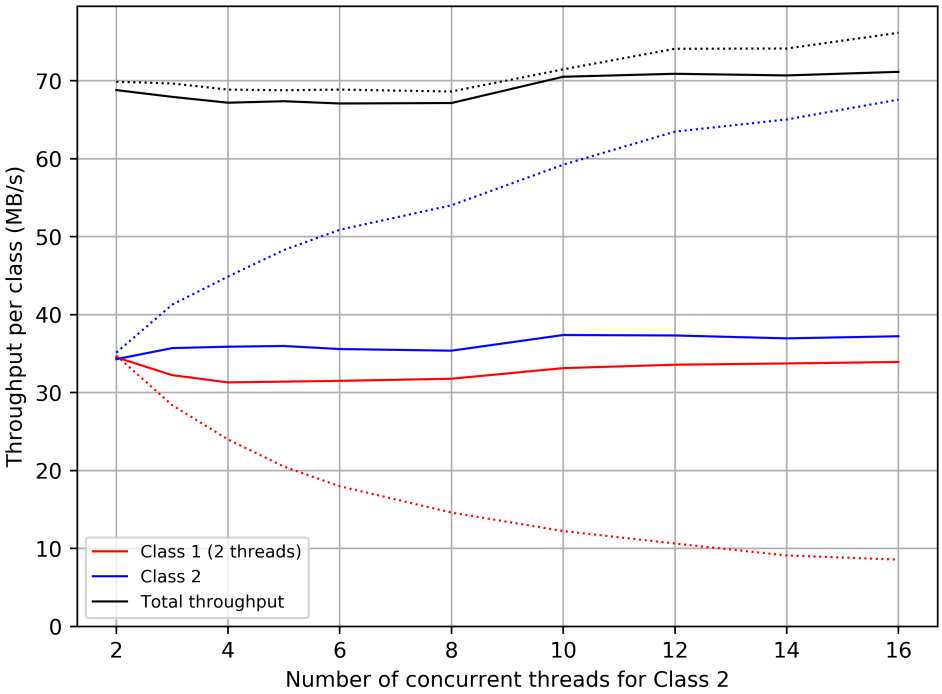


Figure 4.2: **Average throughput per class per number of concurrent threads for class 2. Plain lines correspond to results with RS3 and dotted lines to results with the standard implementation.**

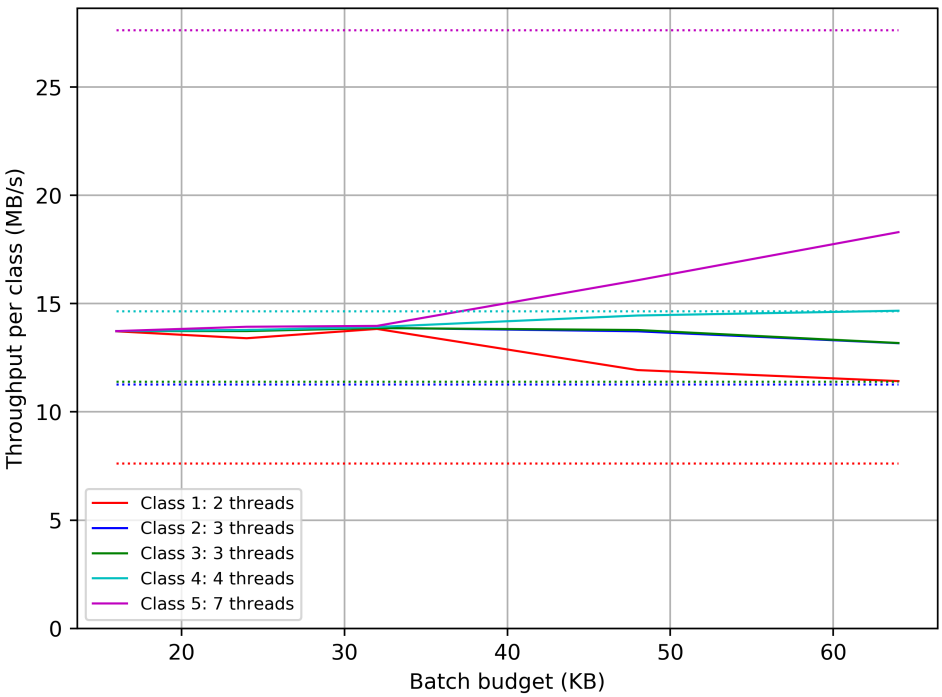


Figure 4.3: **Average throughput per class per batch budget. Dotted lines correspond to the corresponding throughput without RS3.**

is lower when batches themselves are shorter. However, higher budget allow for a slightly better overall throughput (68.64 MB/s for batches of 16KB, 70.74 MB/s for batches of 64KB).

4.3.3 Response time results

Secondly, we test how the storage servers react to an increase in the number of active classes, to measure how the response time for small increases with the number of classes, both with the standard implementation and with RS3 for different fixed *read sizes* and *batch budgets*. To test that, our client simulates a variable number of independant classes continuously sending requests of different sizes, following two specific request patterns:

- small requests (fetching 4KB objects) are sent following a Poisson process, at an average rate of 10 per second;
- large requests (corresponding to objects of size varying between 1 and 4MB) are sent continuously like described above.

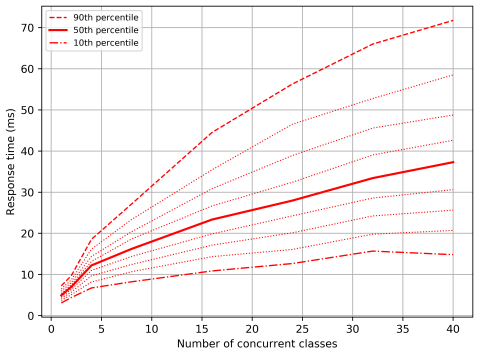
For each test, a single class sends small requests following this Poisson process and a variable amount of classes send large requests. The continuous arrival of large requests helps us simulate a case where disks are saturated and thus RS3 can play its role of allowing small requests to complete rapidly in spite of the saturation.

The results for response time tests are presented in figure 4.4. While a *read size* parameter of 4KB is almost never used, we find that it is relevant to include this result since 4KB is the smallest granularity for I/O in most filesystems, and thus a standard system with such a *read size* corresponds to a fair system. For a *read size* of 4KB, the regular implementation maintains a good distribution of response times for small requests, but still increases linearly with the number of active classes. For *read sizes* above 4KB, we see a sharp increase in response time, linearly or worse with the number of clients. This is especially relevant since most implementations of storage servers have a default parameter for *read size* between 64 and 256KB.

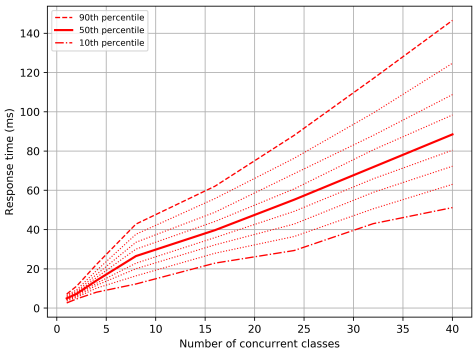
For RS3, as expected, the response time distribution increases slightly with the batch budget but at a much slower pace than the standard implementation with the number of concurrent clients. In the worst case of a batch budget of 256KB, the median is just over 40ms for 40 clients.

4.3.4 Throughput results

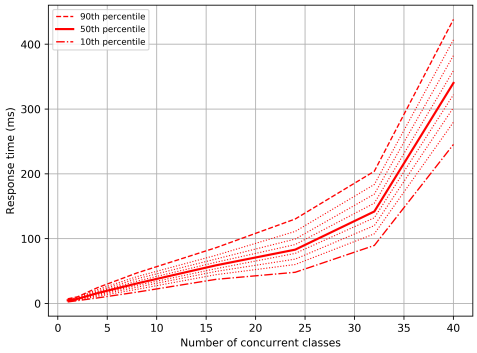
During the previous experiment, we also measured the total throughput of our storage server for each parameter set. The results are shown in figure 4.5. The results corresponding to 1 and 2 classes are not shown here since they correspond to irrelevant cases of either only 4KB requests (highly random reads) or almost only sequential reads and are not representative of the



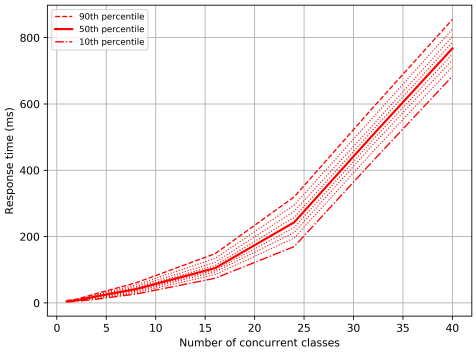
(a) Standard, *read size* = 4KB



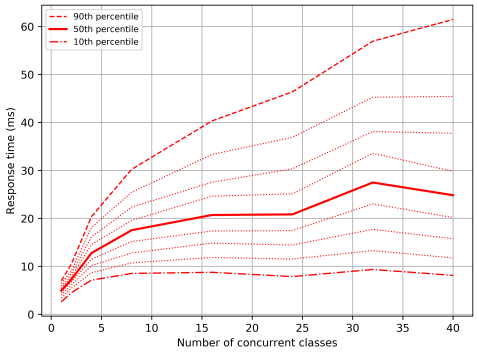
(b) Standard, *read size* = 32KB



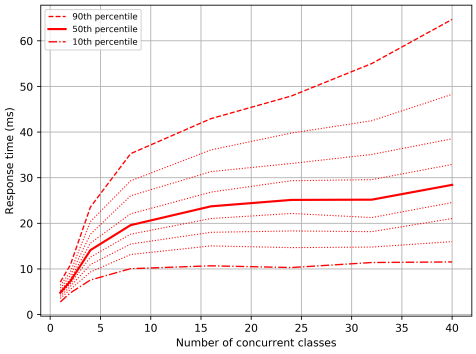
(c) Standard, *read size* = 64KB



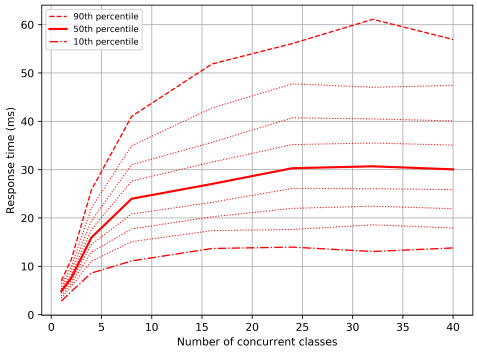
(d) Standard, *read size* = 128KB



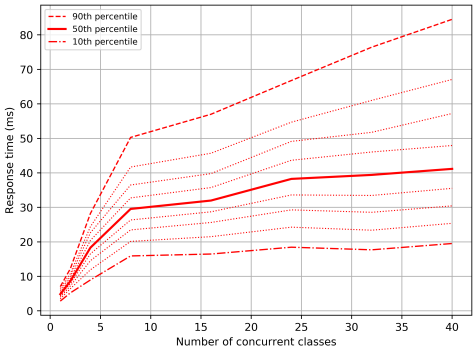
(e) RS3, *batch budget* = 32KB



(f) RS3, *batch budget* = 64KB



(g) RS3, *batch budget* = 128KB



(h) RS3, *batch budget* = 256KB

Figure 4.4: **Response time statistical distribution of 4KB requests in function of the number of clients, without (*a*,*b*) and with RS3 (*c*,*d*) for varying read and batch size.**

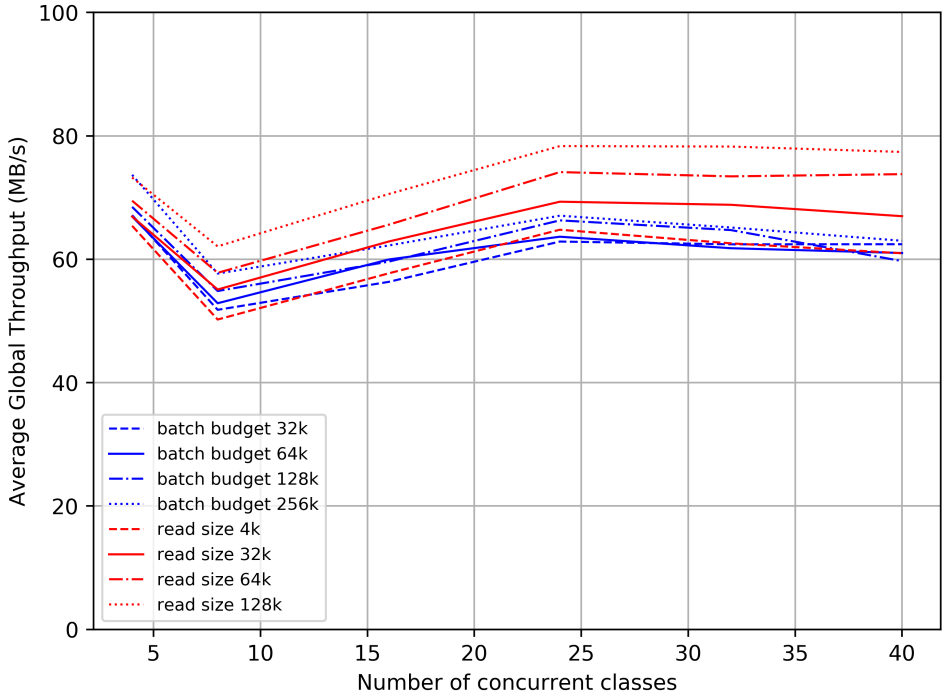


Figure 4.5: **Average total throughput as a function of the number of clients with and without RS3 for different parameter sets. RS3's results in blue, standard's in red.**

typical workload sustained by storage servers. As expected, the standard implementation displays a better total throughput for 32, 64 and 128KB *read sizes* since they have a more sequential I/O pattern than RS3.

However, for 4KB, they perform about as well as RS3, providing no real benefit over RS3 in spite of the higher reponse times for 4KB requests in most cases. The difference of global throughput between the worst case for RS3 (32KB batches) and best case for the standard storage server (128KB reads) is around 19%.

4.4 Using Linux filesystem mechanisms to improve RS3

RS3's goal is to improve fairness while maintaining the same throughput overall. Even though RS3 makes smaller individual reads than the standard implementation, the direct disk reads during the experiments from section 4.3 are still of average size almost 128KB. This is due to the Linux read-ahead [6]. Read-ahead is used by Linux to increase disk read efficiency. The default read-ahead parameter is 128KB, meaning that Linux only sends low-priority

[6]Configured in `/sys/block/<dev>/queue/read_ahead_kb`.

read requests of 128KB to the disk if possible and when it assumes a sequential read pattern, so that subsequent read requests are served directly by the filesystem cache rather than the disk. This explains that in spite of much smaller reads, RS3's implementation manages to have the same order of magnitude of overall throughput as a standard implementation: most `sendfile` reads are served directly by the filesystem cache rather than the disk itself.

However, this read-ahead is triggered only after the first read on a file: 4KB files are served directly from the disk because read-ahead does not have time to occur. As `sendfile` is blocking with regard to disk I/O (although non-blocking on the TCP socket side if called on a non-blocking socket), every small request triggers a blocking `sendfile` for both implementations. To maintain the overall throughput of the system while approaching perfect fairness as much as possible, we desynchronize disk reads and `sendfile` calls as much as possible to reduce the time spent blocking on disk I/O.

4.4.1 Sending hints to the kernel

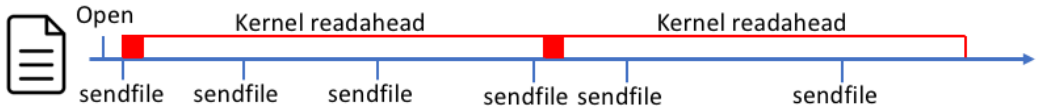
To this end, we use `posix_fadvise`[7] to send hints to the kernel to trigger disk reads before data is consumed by `sendfile`.

We use `posix_fadvise` at two different times:

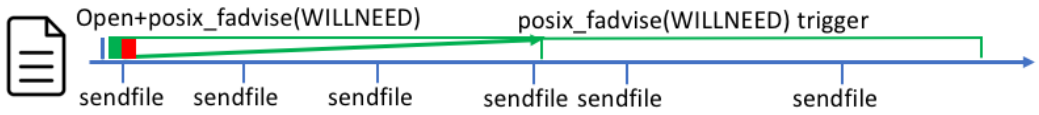
- When a request arrives and the local file is opened, an immediate call to `posix_fadvise` is made with the `POSIX_FADV_SEQUENTIAL` hint. This hint doubles the size of linux read-ahead for the file descriptor. In our case, the read patterns are always sequential since local files correspond to stored objects that are fetched as a whole.
- For every opened file, we keep an offset pointing to the last byte of the file that has been flagged as needed. This is used to call `posix_fadvise` once every few batch when needed. More precisely `posix_fadvise` is called with the `POSIX_FADV_WILLNEED` flag, that initiates a low priority nonblocking read into the filesystem cache. This is done after the computation of the batch budget allocated to the request whenever the difference between this offset and the file descriptor offset (pointing to the last byte read by `sendfile`) is lower than the specific file descriptor's read-ahead value (twice the filesystem's default parameter),

The sliding window mechanism we describe here is similar to what Linux's kernel uses for its own read-ahead mechanism at the filesystem level as described in [197]. However, we noticed during our experiments that triggering the read-ahead through `posix_fadvise` ourselves yielded better overall performance with regard to both global throughput and response time. We believe that this is due to the fact that we make small reads, leading the kernel to treat our requests as "not highly sequential" and thus maintaining very

[7] https://linux.die.net/man/2/posix_fadvise



(a) Read-ahead is done by the Linux kernel when data that is not yet in cache is requested by `sendfile`. `sendfile` is non-blocking on the network



(b) Read-ahead is done ahead of time so that `sendfile` never or very rarely blocks on disk I/O.

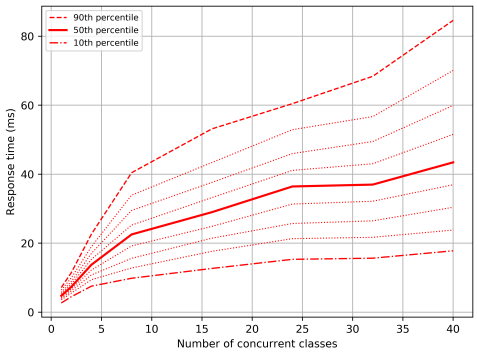
Figure 4.6: **Blocking time for `sendfile` calls without or with the use of `posix_fadvise`. Blocking disk reads are illustrated by red blocks while asynchronous, non-blocking disk reads are illustrated by green blocks.**

small read-ahead sliding windows, leading to more blocking calls than if we trigger the read-ahead ourselves. Our approach aims at increasing disk reads efficiency by compensating the way we split I/O requests and letting the I/O scheduler know in advance what data will be requested. The difference between this implementation and the previous one is illustrated in figure 4.6 for a case where the Linux read-ahead is not triggered early enough.

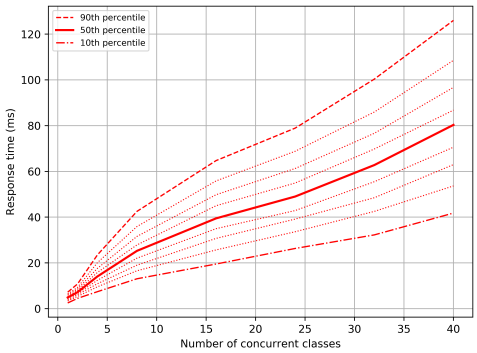
4.4.2 Response time and throughput results

We implemented this modification in both our prototypes and subjected them to the same tests than in section 4.3.3. Response time tests results are presented in figure 4.7 and throughput tests in figure 4.8. Our modifications have overall slightly increased the response time for small requests. This was to be expected since the forced read-ahead mechanism tends to favor long reads due to how I/O schedulers work. In compensation, the overall throughput has been increased for both implementations, and the difference of throughput between them has been reduced. The best case for the standard implementation (128KB reads) now outperforms the worst case for RS3 (32KB batches) by 14%. RS3's throughput for 40 active classes has been increased by 17 to 24% depending on the batch budget.

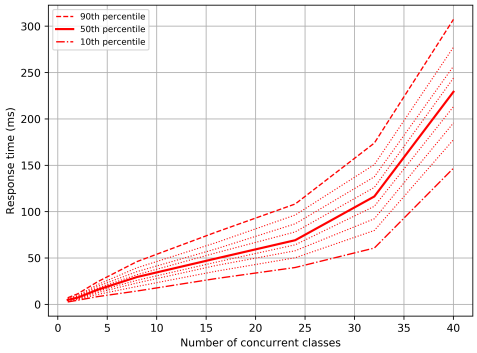
At the same time, the new implementation has also significantly reduced the standard implementation's response time for small requests for large *read sizes* (64 and 128KB), but they remain much higher than with RS3 and still scale linearly with the number of classes.



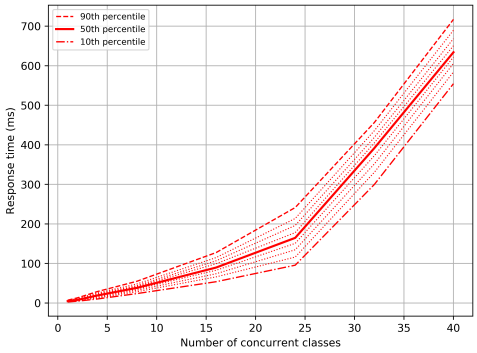
(a) Standard, *read size* = 4KB



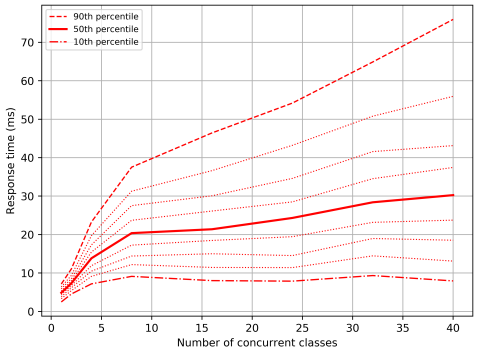
(b) Standard, *read size* = 32KB



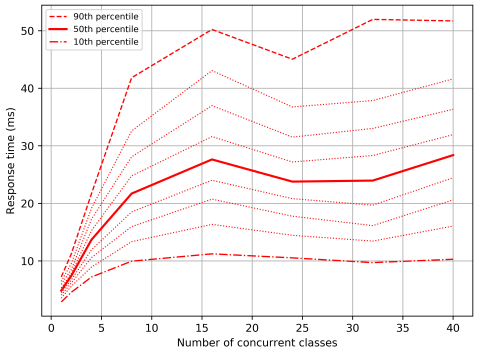
(c) Standard, *read size* = 64KB



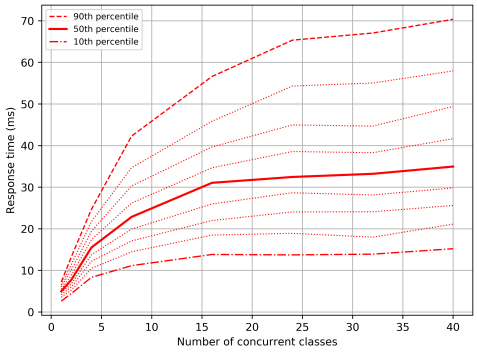
(d) Standard, *read size* = 128KB



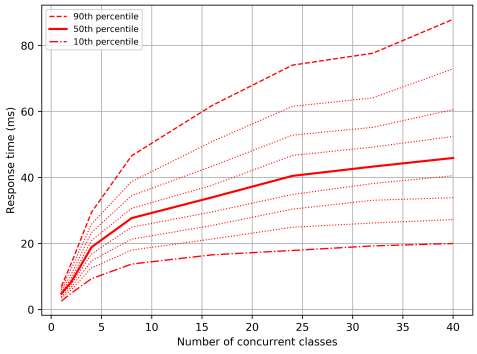
(e) RS3, *batch budget* = 32KB



(f) RS3, *batch budget* = 64KB



(g) RS3, *batch budget* = 128KB



(h) RS3, *batch budget* = 256KB

Figure 4.7: **Response time statistical distribution of 4KB requests in function of the number of clients, without (*a*,*b*) and with RS3 (*c*,*d*) for varying read and batch size.**

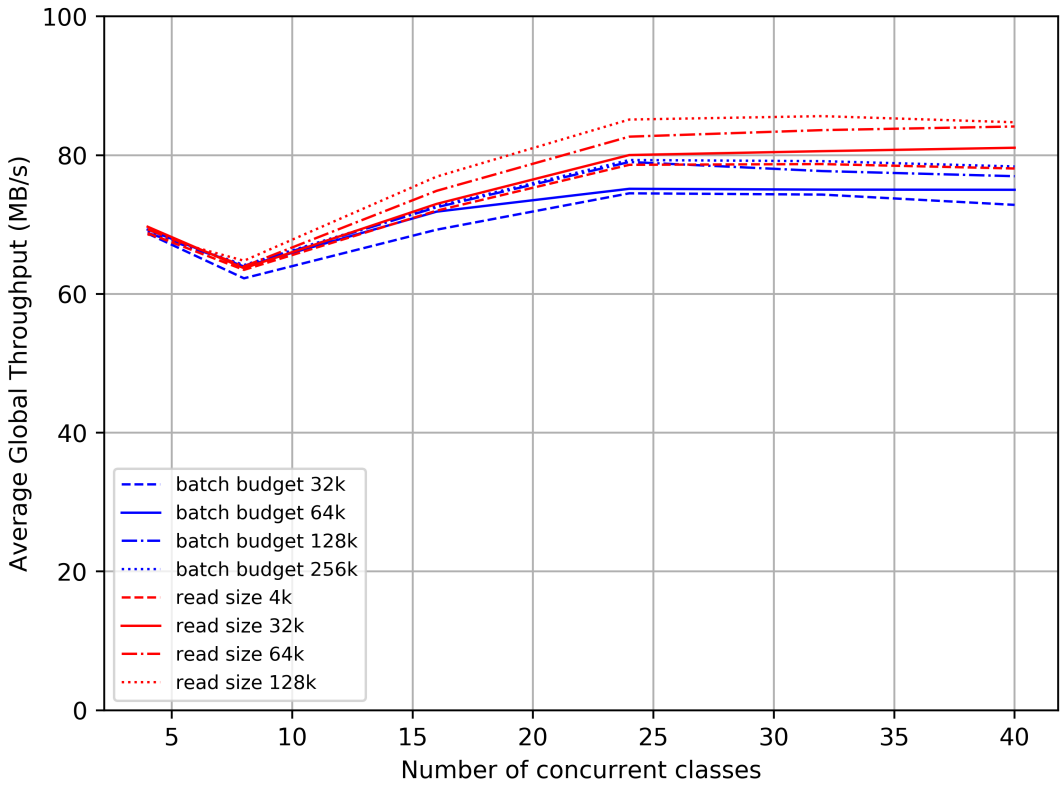


Figure 4.8: **Average total throughput in function of the number of clients with and without RS3 for different parameter sets and active classes numbers. RS3's results in blue, standard's in red.**

Both with and without this optimization, we can also observe from figures 4.4, 4.5,4.7 and 4.8 that the batch budget only has a slight impact on the total throughput and the response time distribution of small requests with the parameters we chose.

4.5 Going further with RS3

4.5.1 Evaluating batch budget's impact on RS3's performance.

We observed in figure 4.3 that lowering the batch budget provides a much better fairness, while also slightly increasing the response time for small requests and the total throughput. However, lowering the batch budget mechanically increases the CPU consumption of a storage server for two reasons:

- the batch budget allocation phase, that requires computation, has to occur more often;
- `sendfile` is called more often, which requires CPU cycles.

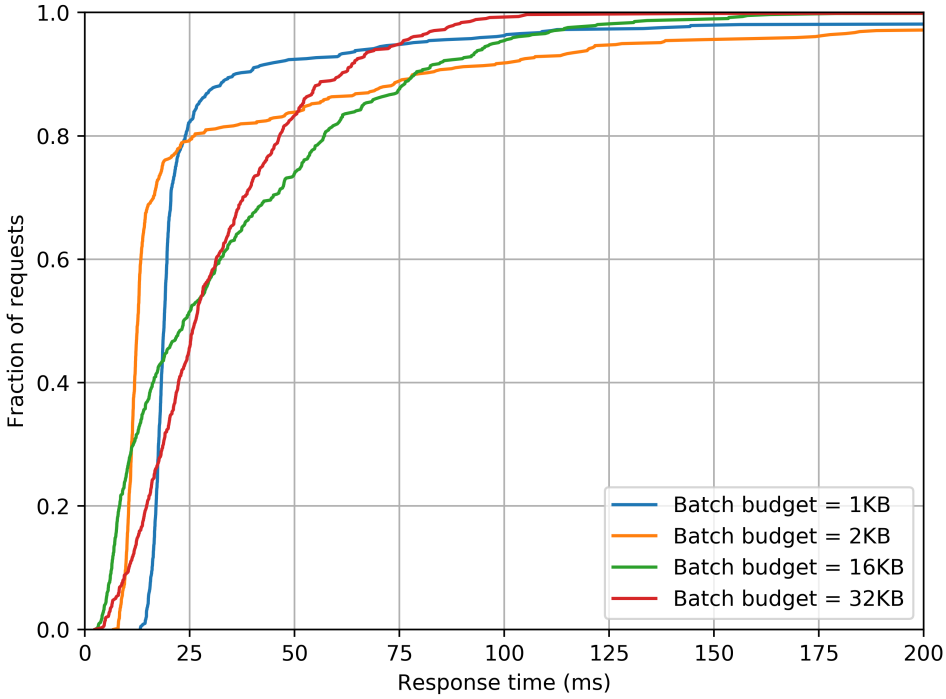


Figure 4.9: **Cumulative distribution function of 4KB requests reponse time depending on the batch budget.**

Therefore, while reducing the batch budget has a positive impact on fairness, it also has a negative impact on the overall system performance.

To evaluate this impact, we use the same setup that in section 4.3 for the 4KB requests response time. However, we set the number of concurrent classes to 40 and vary the batch budget between 1KB and 64KB. For each test, we look at the response time distribution for small requests, the overall throughput of the system, and the CPU time used by our storage server. Results are presented in figure 4.9 for the response time distribution and figure 4.10 for the throughput and CPU time.

The results of figure 4.9 show that for very low batch budgets (1 or 2 KB), the response time distribution corresponds to a very skewed long tail (80% of the requests take less than 25ms to complete but 5% of them take more than 175ms to complete) while higher batch budgets have a less skewed, shorter tailed response time distribution.

Figure 4.10 confirms that the batch budget has a significant impact on CPU consumption. Moreover, this impact is directly correlated to the total throughput of the storage system. RS3 reaches its maximum throughput (around 74 MB/s) only for batches larger than 32KB, when the CPU time amounts to less than 40% of the total time elapsed. RS3's throughput is as low as 11 MB/s for a batch budget of 1KB.

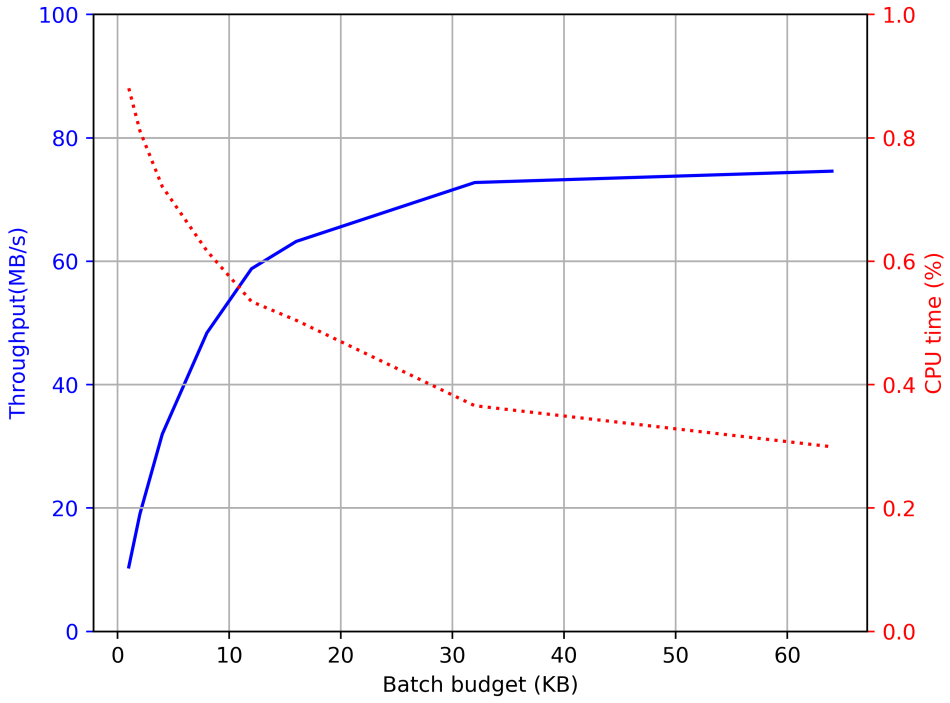


Figure 4.10: **Total throughput and storage server CPU time for 40 concurrent classes per batch budget.**

4.5.2 Tweaking RS3 to enforce policies: Weighted-RS3

The algorithm 2 presented in section 4.2 was aimed at guaranteeing perfect fairness between all classes. However, in some cases, an orchestrator may wish to assign different weights to different classes (classes can be clients with different service level agreements, applications with different priorities ...) so that some classes get a higher throughput than others in the case of contention. This can be obtained by introducing weights W_i for each class in the previous algorithm, as described in algorithm 4. Furthermore, this weighted version of RS3 could be dynamically configured by a control plane to enforce cluster-wide scheduling like described in [162].

Note that when all active classes have the same relative weight, this algorithm is strictly equivalent to the previous one. We have implemented Weighted-RS3 (W-RS3) and have tested it on our benchmark, with 3 classes having different weights and number of concurrent clients, each client thread making continuous requests for big objects (between 1 and 4MB). The throughput results can be found in table 4.1.

As expected, RS3 guarantees fairness while W-RS3 guarantees a throughput distribution almost proportional to classes' weights. W-RS3 is particularly useful, when the average global throughput of a storage system is known, to guarantee a certain amount of throughput to certain users, even in the event of network or disk saturation. For example, in the example shown in table 4.1, if the throughput of a saturated system is of 100MB/s, class 3 is

Algorithm 4 Weighted batch budget allocation algorithm

B: batch budget
N: number of active classes
$w_i = \frac{W_i}{\sum_{i=1}^{N} W_i}$: relative weight of active class i, $\sum_{i=1}^{N} w_i = 1$
$R[i]$: pending cumulative size of requests made by active class i, ordered by $\frac{R[i]}{w_i}$ from lowest to greatest
$T[i]$: size of the TCP send buffer associated with class i
$B_a[i]$: budget allocated to class i for the batch
R_b: Remaining budget for the batch
function WEIGHTED_BUDGET_ALLOCATION(B, N, R, T)
    $R_b \leftarrow B$
    for $i = 0$ to $N - 1$ **do**
        $B_a[i] \leftarrow min(T[i], R_b \frac{w_i N}{N-i}, R[i])$
        $R_b \leftarrow R_b - B_a[i]$
    end for
    return B_a
end function

			Throughput(MB/s)		
Class	Threads	Weight	Standard	RS3	W-RS3
1	4	1	30.88	23.15	12.47
2	3	2	23.48	22.9	23.48
3	2	3	16.05	21.97	30.97

Table 4.1: **Throuput per client without RS3, with RS3, and with W-RS3, with batch budget= 24KB.**

guaranteed to have an available throughput of approximatively 50MB/s.

4.5.3 Considerations on RS3 and its current implementation

Fairness The definition of fairness given in section 4.2 is widespread but simplistic. Some I/O schedulers [185, 182] introduce an additional fixed cost to every I/O request to model the average latency of HDD seeks, so that the I/O request cost is not directly proportionnal to the request size in bytes but a linear function with a fixed term. This is relevant for HDDs, displaying a high latency but low variance when serving requests, but much less for SSDs that exhibit much more variance when dealing with I/O requests and even between each others, as demonstrated in [188].

This could be adapted to RS3 by slightly modifying algorithms 2 and 4 to take into account not the cumulative *size* of requests in $R[i]$, but rather the cumulative *cost* of requests, with well-chosen cost functions depending on the underlying storage device (HDD, SSD or NVMe). Such an approach

would probably model the effective disk access time with more precision. We decided against this approach in the current presentation for two reasons:

- Modeling the performance of HDDs is not in the scope of this work and has already being extensively done [198, 199, 200, 201], even producing some disk drive performance simulator such as DiskSim [202, 203]. Thus we presented RS3 as is for the sake of simplicity. As explained, it is totally straightforward to integrate a cost function to RS3.
- Cost functions are great tools to model HDD response time because they exhibit high latency and low variance, whereas SSD response vary much more between requests and even between drive models [188].

Similarly, some approaches like [204] also include the temporal locality – the ability to hit the storage server's cache and thus make a better use of its resources – of applications in their definition of fairness, and reward such them with more throughput. We argue that it is indeed relevant while the disk is the limiting factor but not necessarily when the network is limiting.

Multi-disk RS3 A single-disk storage server is not a realistic scenario. However, running one RS3 scheduler per disk in parallel in the case of multiple disks servers provides the same guarantees when disks are the same model and RS3 configured the same way with the same batch budget:

- If the aggregate throughput of all disks is lesser than what the network can handle, there is no contention between disks and thus RS3 works as desired for every disk;
- If the aggregate throughput of all disks is higher than what the network can handle, all disks have symmetric performance and thus they naturally have access equal to the network device in average.

RS3 does not handle – in its present iteration – the case of inter-disk fairness (for instance, a class that would spread its request for data stored on multiple disks could get more throughput than another having all its data on one disk): we leave that part to external orchestration and data placement mechanisms.

Writes in RS3 The current implementation of RS3 only works for reads requests, that amount for most of the data traffic in most cases. However, write-support could be implemented the same way – with a cost function or not – by handling both read and write requests in our batches. Such an implementation would, however, order the I/O requests to separate write and read requests as it has been shown that read and write interferences reduce the effective throughput of SSDs [3, 205, 188].

4.6 Conclusion

In this chapter, we described two recurrent problems for storage systems that are shared between multiple users: the unfairness of the throughput distribution between clients that have different request rates, and the high response time encountered by small requests when storage servers are under pressure.

To solve this problem, we described RS3, a request scheduler that fairly allocates throughput between classes and sends data in fixed-size batches to guarantee low response time for small requests. We showed that RS3 fulfills the objectives of equalizing throughput between classes and allowing small requests to be satisfied rapidly (in the most extreme cases, around 30 times more rapidly than with a standard implementation), at the cost of a fraction of the total throughput.

We then explained how we optimized RS3 by sending hints to the kernel using `posix_fadvise` to reduce the time spent blocking on disk I/O. This optimization increased the overall throughput for both implementations and decreased the throughput discrepancy to around 15% between the best throughput for the standard implementation and the worst throughput for RS3 while maintaining approximatively the same response time for small requests as without this optimization for RS3.

We also explored the impact of the batch size on several metrics, notably throughput and CPU consumption. We observed that while guaranteeing a near-perfect fairness, a low batch budget ($\leq$ 32KB) also increases the CPU consumption to a point where it can have a negative impact on throughput. However, our experiments show that the batch budget only has a limited impact on total throughput as well as on the reponse time distribution for small requests (as long as the CPU does not become a bottleneck).

Finally, we described W-RS3, an extended version of RS3 that can be used to enforce Service Level Agreements (SLAs), QoS requirements or storage-system-wide resource allocation while maintaining its low-response time properties for small requests.

4.6.1 Going further

In the future, we intend to evaluate how RS3 performs in different settings, such as with SSDs, for which the CPU consumption is more of a problem [6] or under real-world workloads.

We also aim at adapting RS3 to efficiently include writes in its scheduling algorithm or work above several disks sharing the same network interface and/or CPU cores on the same server, following the leads shown in section

4.5.3

Finally, we want to provide RS3 a way to adapt its batch budget to current load conditions to always act optimally, notably by adapting the batch budget to the number of concurrent classes in order for the CPU not to become a bottleneck while also maintaining the best possible level of fairness and throughput, the key element being not to saturate the CPU.

Contributions

This work has been the object of a defensive publication by Cisco and of a paper submission.

- Fair Scheduling for low latency and high throughput storage systems, in Technical Disclosure Commons, (August 21, 2018) [206]
- Guaranteeing low response time for small requests and fair throughput allocation: a Request Scheduler for Storage Servers (RS3), paper, in preparation for submission.

Chapter 5

Replica Caching for Erasure-Code based Distributed Storage Systems

Distributed storage systems are widely used today to provide scalable and reliable storage service for companies like Facebook with Cassandra and HDFS [27, 29], Google with GFS, BigTable and Spanner [24, 25, 26], Amazon with DynamoDB [23], and others with Ceph [28, 29]. Those storage systems are deployed on generic hardware that is prone to failures and thus have to incorporate reliability mechanism to guarantee data availability when servers fail or are removed from the storage system.

Erasure codes to reduce the storage overhead To provide reliability against failures, most storage systems simply replicate data on multiple storage nodes. The standard is to replicate every data object 3 times across different disks/servers/datacenters. However, this heavily increases the hardware cost of such systems since the storage overhead of replication is high, as explained in section 1.2.3. To reduce this cost, many distributed systems have been transitionning in the last few years from replication to erasure codes to ensure data reliability.

A (k, r) erasure code splits the data into k equal-sized fragments, called *systematic fragments*, creates independant linear combinations of these fragments to generate r *parity fragments*, and stores them on different storage nodes. The r parity fragments are encoded in a way that guarantees that any k of the $k + r$ fragments can be used to reconstruct the original data object.

Thus, up to r failures are tolerated for a storage overhead of $\frac{k+r}{k}$, usually lower than $r + 1$ for a replication-based setup with a replication factor of $r + 1$ also tolerating up to r failures. For example, a 3-way replication has a storage overhead of 3 and tolerates up to any 2 simultaneous failures, while a $(10, 4)$ erasure code has a storage overhead of 1.4 and tolerates up to 4 failures.

Moreover, because data is retrieved in parallel from multiple nodes, erasure codes can speed up data retrieval compared to simple replication. However, they also induce a computation and networking overhead and thus can lower the overall performance of storage clusters:

- For every object retrieval, k TCP connections must be opened. This means that $(k-1)$ times more TCP sessions must be negociated and established in the cluster. If the data is encrypted on the wire, it has an even bigger memory and computing footprint. Network stack computation can already amount for more than 50% of the CPU time of distributed storage systems in some cases when replication is enabled rather than erasure coding [85].
- Of these k connections, the slowest one determines the completion time for a storage request. If one of the storage nodes is saturated or the network is congested for any reason, the whole request suffers from it. While network congestion and resource saturation can also happen with replication schemes, the likelihood of such an event occuring is much higher when multiple storage participate to every request.
- Depending on the type of storage device and the request rate, the physical drives' I/O rate can even decrease: a hard drive is much better at handling a big I/O request than multiple small and uncorrelated I/O requests in parallel.

How distributed storage systems cache Some distributed systems, like Ceph, offer the possibility to define a separate cache tier, deployed on storage nodes equipped with fast storage devices, e.g., NVMe or SSDs. When such a cache tier is used, all storage requests are first handled by the cache. Only when they fail are they sent to the slower storage backend. In these cases, cache hits (i.e., the requested object can be found in the cache) are dealt with as fast as possible, but cache misses (i.e. the object is not in the cache) drastically increase the time required to fetch non-cached objects, making it worse than if there was no cache tier at all because of the additional RTTs and lookups. It is thus generally not advised to define a cache tier unless the data popularity distribution is heavily skewed and the benefits of the cache counterbalance its adverse effect on cache miss performance.

Most of the distributed storage previously mentioned consist of daemons running in user-space on top of generic Linux filesystems such as XFS or ext4. They store data or data fragments as files in specific directories. As such, even when a separate caching layer is not defined, storage systems benefit from the underlying filesystem's caching mechanisms. The way filesystems cache data utilizing the available memory is well-known and is usually a variation of a LRU (Least Recently Used) cache, described later in section 5.3.2.

However, when erasure codes are used, these local files are only fragments of data. Because nodes caches operate autonomously, it is possible for some data fragments to be cached by their host's filesystem while other fragments from the same original data, stored on other storage nodes, are not. Since the data request is fulfilled only when all k fragments are retrieved, it benefits from caching only if all the fragments are simultaneously cached.

Highly popular objects probably have their fragments cached on every relevant storage node and highly unpopular objects probably have none of their fragments cached. However, moderately popular objects can have several fragments cached and others not. These cached fragments only have an indirect benefitting impact on the cluster's performance: they do not require disk I/O to be fetched and thus free the drives for other tasks. Thus, they amount mostly to cache pollution: they reduce the efficiency of caching at the scale of the storage system.

The aim of this work is to evaluate the impact that a traditional filesystem-level caching policy can have on an erasure-coded distributed storage system, particularly in terms of cache hit, cache waste, and request response time, then to propose a new caching algorithm taking into account the erasure coded nature of the fragments stored locally.

The rest of this chapter is organized as follow: section 5.1 gives an overview of the work that has been done on the topic of erasure codes and specifically what caching strategies have been developped to adapt to their characteristics. Section 5.2 describes the problematic in depth as well as our proposed caching policy. Simulation and experimental performance evaluations are described in section 5.3 and section 5.4 respectively. Finally, a conclusion is given in section 5.5.

5.1 Related Work

The topic of caching itself has been extensively researched for a long time now in the context of generic storage systems as a means to increase system performance [207, 208]. However, few research has been done on the specific use-case of caching in erasure-coded systems, probably because they have been deployed for only a few years.

The impact of erasure coding on latency and performance, however, has been extensively studied. Research has been done to identify the impact of interacting with multiple storage nodes – in erasure-coded setups – rather than single ones – in replicated setups – with regard to latency [209, 210, 159, 160].

For example, the authors in [160] showed than when requests are flooded to all servers storing a fragment of a (k, r)-erasure coded object –rather than most approaches that only fetch the systematic fragments–, the expected download time is reduced since it is enough to wait for the k fastest fragments. They also showed that it is possible to quantify the trade-off between storage overhead and download time. However, flooding requests to all storage nodes storing an object fragment has the negative impact of further increasing the number of active connections. Besides, it can trigger useless disk reads when fragments are read on a storage server but not returned fast

enough to be used for the object reconstruction.

Meanwhile efforts are still being carried out to predict more accurately the latency of erasure-coded systems despite their inherent superior complexity to replication-based systems [209].

As previously explained, using erasure coding in distributed storage systems raises new challenges. One of these is to guarantee that object segments are either all available or all unavailable in storage nodes' caches. To solve this issue, the authors in [211] suggest a cooperative caching approach, where each storage node communicates its local cache content to the other nodes. Then, based on these informations, each node updates its local cache to ensure that all the object segments are available in cache nodes, or that none are.

While this solves the cache pollution problem, it requires extensive communication and communication between nodes and is subject to race conditions when caches update quickly, especially for large scale systems where full-mesh inter-nodes communication does not scale. Furthermore, it does not reduce the overhead of maintaining several connections when retrieving all the cached fragments, which we saw reduces the benefit of the cache itself.

In [212], the authors describe a way to cache only a subset of fragments from an object based on their geographical position. They focus on a storage system spanning several continents and storing the data fragments for every object on different continents. In this setup, they show the advantage of caching only the most remote fragments relative to each region in the region's dedicated cache. On the contrary, our approach focuses on single-region storage systems where there is no inherent heterogeneity in fragment access, which makes such optimizations irrelevant. Moreover, it supposes the existence of a separate cache tiering in each region whereas our approach focuses on taking advantage of the default caching capabilities of every generic storage server. However, both approaches are complementary when taken region-per-region.

While the solutions described previously are based on caching only systematic segments, some research work [213, 214, 215] investigated how to cache additionally generated parity segments in addition to systematic fragments.

For example, the authors in [214] propose to cache d erasure coded segments to obtain a $(d+k, r)$ code and cache the d coded segments in a caching proxy. The object is then reconstructed form these d fragments in the proxy and the fastest $k-d$ fragments retrieved from the k storage nodes storing the systematic fragments. However, this approach requires heavy computation to optimize the fragments' placement. Moreover, this approach also uses a separate cache tier acting as a gateway between clients and storage servers whereas ours just use generic storage server caching capabilities, and just

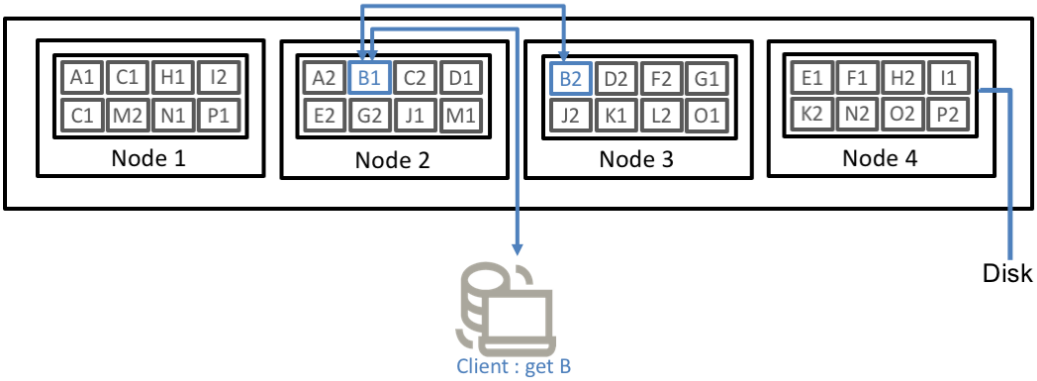


Figure 5.1: **Client requesting object B from a $(2, r)$ erasure coded distributed object store (parity fragments not represented).**

acts on a regular LRU strategy with no synchronization or placement optimization whatsoever.

The authors in [215] propose a way to provide resilient caching using erasure codes. They argue that high-performance, large-scale storage environments require resiliency even in their in-memory caching layer, and propose to use erasure codes to provide this resiliency. They then propose to load balance between caches by reaching out to $k + \delta$ in-memory fragments of objects rather than k to fetch the fastest and reduce the tail latency of requests.

In addition, distributed object stores like Ceph [28] either use standard filesystem caching mechanisms or dedicate a separate storage pool for storage. While the performance are good for objects that are cached, this heavily degrades the performance for objects that are not cached since requests have to go through the entire request process for the cache before trying it again in the storage backend.

5.2 Caching and Popularity In Distributed Storage Systems

5.2.1 System Architecture

In this work, we consider an object-based distributed storage system using a (k, r) erasure code to guarantee reliability. The way clients keep track of where objects' fragments are stored as well as the precise type of erasure code used is out of the scope of this work.

We assume that the client itself interacts only with one storage node that fetches all the necessary fragments before returning the whole object to the client. While this is not optimal (it is more efficient for the client to directly interact with all storage nodes), almost every distributed storage system uses

a form of gateway acting as a proxy for clients. We merely here make the assumption that every storage node can act as a gateway. Without loss of generality we assume for the remainder of this chapter that the client interacts with the storage node storing the first fragment of an object when making a request for this object. Furthermore, for the sake of simplicity, we will assume for the remainder of this chapter that all objects – and thus fragments – are of the same size.

Figure 5.1 describes an example of such a storage system using a $(2, r)$ erasure code. This basic setup, that we will use as an example for the remainder of this chapter, consists of 4 storage nodes having a drive capacity of 8 fragments. We note here that we do not represent the r parity fragments that are also stored on the storage nodes but never accessed outside of reparation purposes, which is out of the scope of our work.

In this example, the storage system stores 16 objects labeled A to P, which fragments are randomly distributed on the storage nodes, with the only constraint that 2 fragments of the same object cannot be stored on the same node. The figure illustrates how a client retrieves the object B in such a setup. The client connects to the Node 2 (storing $B1$), which fetches the required fragment $B2$ from Node 3 and then send the full object to the client.

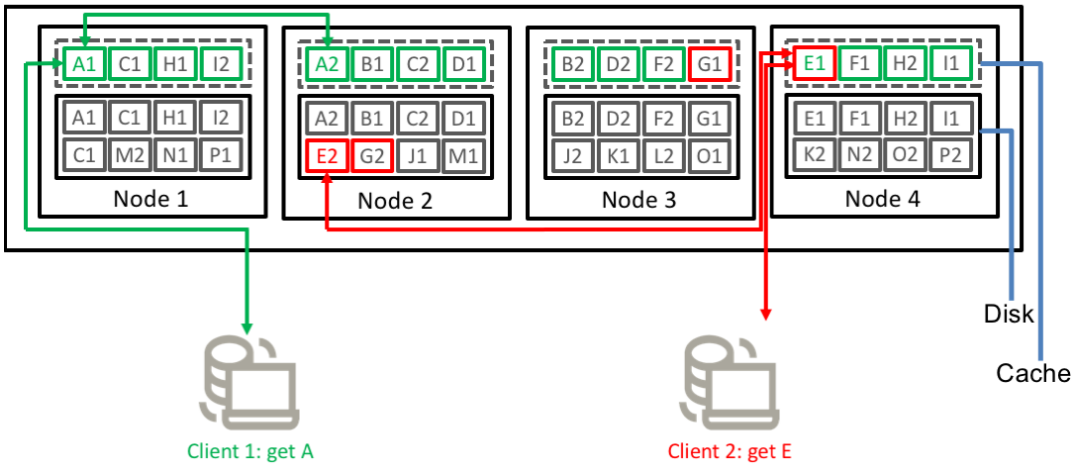
5.2.2 Object Caching

In such a system, each storage node's filesystem caches its most popular fragments. This can lead to objects having a fraction only of their fragments served by some storage nodes' cache while the rest are served by other storage node's disk. This doesn't benefit the client as the request is only completed when the slowest fragment is retrieved since all k fragments are required to reconstruct the object.

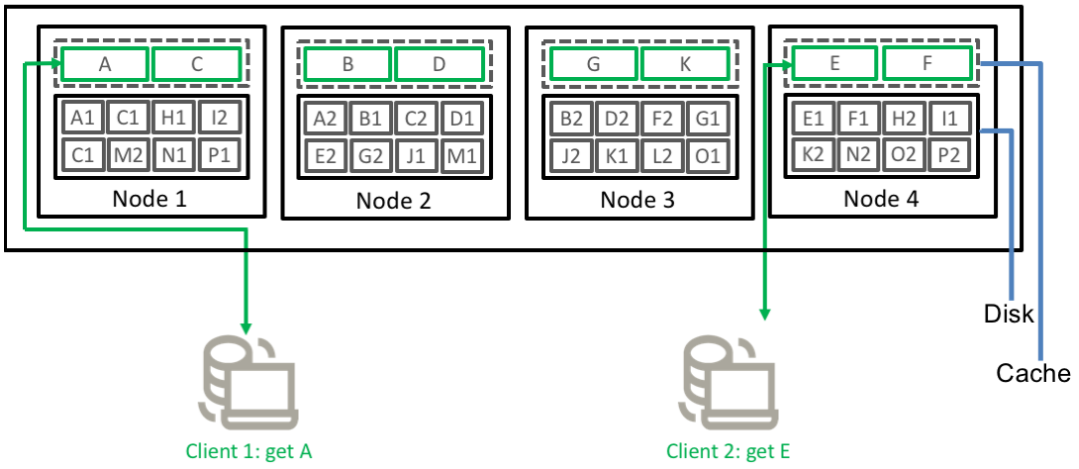
Thus, we propose that instead of caching objects as fragments, they should be cached as full replica. To that end, we modify the filesystem's caching layer so that when the decision is taken to cache a local file corresponding to a fragment, the following happens instead:

- If the fragment is the corresponding object's first fragment, the storage node retrieves the whole object as it normally would and caches it in addition to sending it to the client [1]. If required, it evicts a previously cached object from the cache.
- If the fragment is not the corresponding object's first fragment, it does nothing more than just retrieving the fragment to send it to the storage node requesting it.

[1] This can be done either by modifying the filesystem's caching layer code, or disabling the filesystem's cache and developping a custom caching layer.



(a) Legacy filesystem caching: client 2 gets E1 from cache but E2 from disk – the cache does not speed up object fetching.



(b) Full replica caching: client 2 gets E from storage node 4's cache directly.

Figure 5.2: **Fragments caching versus full replica caching. Caches represented in dotted lines.**

The difference between the legacy filesystem cache and our approach is illustrated in figure 5.2, on the setup previously presented in figure 5.1. In this example, we suppose that objects A to P are ordered by popularity, with A the most popular and P the least one. The fragments are positioned randomly since the system does not know how popular objects are before they are stored. We also suppose that the storage nodes' caches are perfect and store exactly their 4 most popular fragments, or the 2 objects corresponding to their two most popular primary fragments. In this example with traditionnal filesystem caching, fragments $E1$ and $G1$ are cached while $E2$ and $G2$ are not cached. As such, $\frac{1}{8}^{th}$ of the cache of our storage system is wasted. Instead, storage nodes 3 and 4 could respectively cache $O1$ and $O2$, or even better, $K1$ and $K2$.

However, figure 5.2 also shows that our approach does not necessarily caches only the most popular content: K is cached in spite of H,I and J being more popular. This is unavoidable without more orchestration or rebalancing as long as objects' popularity is unknown prior to them being stored.

Following this example, let us define what we mean for the remainder of this chapter by *cache hit ratio* and *cache waste ratio*:

- **Cache hit ratio:** the cache hit ratio is the ratio of object requests that are fully served by the storage system's cache. When fragments are cached, a request is counted as a cache hit only if all its fragments are cached, since only then does it benefit the client in term of latency.
- **Cache waste ratio:** when fragments are cached, we define it as the ratio between the number fragments cached from objects which do not have all their fragments cached and the total number of fragments cached in the system. It corresponds to the percentage of cache space that is dedicated to fragments not benefitting the client in term of latency.

5.3 Theoretical Evaluation

In this section, we evaluate the impact of our full replica caching policy with a simulation. To model objects' popularity, we choose to use the modified Zipf-law model described in section 5.3.1. Our simulation evaluates the cache hit and cache waste ratios of both ours and the traditionnal approach to caching for different parameter sets.

5.3.1 Popularity Model

A commonly observed popularity distribution for data access on the web is the modified Zipf-law [216, 217, 218, 219, 220, 221]. Such a popularity distribution describes several *popularity classes* that have different relative popularities following a power law. Each class has a popularity weight of $\frac{C}{i^{\alpha}}$, with i the index of the class, α the Zipf parameter identifying the skewness of the

distribution, and $C = \frac{1}{\sum_i \frac{1}{i^\alpha}}$. In each class, all objects have the same popularity. Thus, the popularity of an object x is given by $P(x) = \frac{C}{i(x)^\alpha \times N(i(x))}$, with $i(x)$ the class corresponding to object x and $N(i)$ the number of objects in class i. $P(x)$ corresponds to the probability of randomly picking x when following the given probability distribution, and is equivalent to picking a class at random following an α modified Zipf-law and then picking an object with a uniform probability distribution in that class. The higher the value of α, the higher the popularity difference between popular and unpopular classes, called the popularity distribution *skewness*.

5.3.2 System Model

For the remainder of this section, we simulate an environment with the following characteristics:

- N_o objects of same size are stored amongst N_s servers. Each object is split in k systematic fragments. Each of these k fragments is randomly assigned to a different storage server.
- The N_o objects follow a modified Zipf popularity law of parameter α with N_c classes, each class i containing a N_c^i objects, with $N_c^i > 0, \sum_i N_c^i = N_o$. The objects are ordered by popularity.
- Each server has a cache of fixed size able to store N_c fragments or $\frac{N_c}{k}$ objects. These caches can be configured to be *perfect* or *LRU*.

Perfect Cache In a perfect cache, the popularity distribution of all objects is perfectly known at any instant, which allows to explicitely cache the most popular fragments/objects. Based on whether a full object is cached or a set of fragments, we can distinguish two cases:

- If fragments are cached, every server's cache is initially filled with its most popular fragments.
- If full objects are cached, every server's cache is initially filled with the most popular objects for which they store the first fragment.

LRU Cache Each LRU cache is initially empty and is progressively filled as requests come in. The objects are ordered in node caches by last request time. The fragment/object with the oldest last request time is evicted when a new fragment/object is cached. Similarly to the perfect cache, whether fragments or objects are cached, the request process varies.

- If fragments are cached, the following happens when a fragment is requested:
  - If the fragment is already in the cache, it is moved at the beginning of the queue.

Parameter	Value
Number of requests : N_r	10^6
Number of objects : N_o	10^5
Number of servers : N_s	100
Number of fragments : k	10
Cache capacity : N_c	$\{0.01, 0.05, 0.1\}$
Number of class of popularity	4
Class repartitions	$[1,1,4,4]$, $[1,1,1,1]$
Zipf exponent: α	$[0, 0.1, 0.2, \ldots, 1.5]$

Table 5.1: **Simulation Parameters Settings**

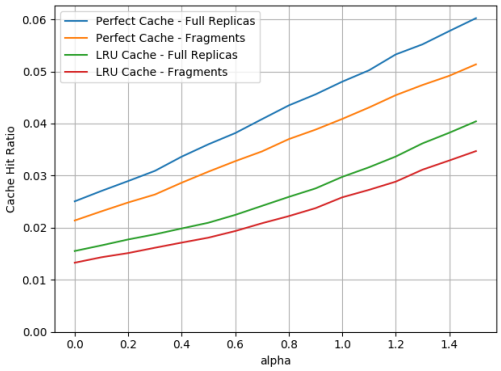
  - If the fragment is not in the cache, the requested fragment is placed at the beginning of the queue. Additionally, if the cache is already full, the last fragment of the queue is removed from the cache.

- If full replicas are cached, the following happens when a fragment is requested:
  - If the fragment is not the first fragment of the corresponding object, nothing happens.
  - If it is the first fragment and the object is already in the cache, the object is placed at the beginning of the queue.
  - If it is the first fragment and the object is not in the cache, the object is placed in the cache at the beginning of the queue. Additionally, if the cache was already full, the last object of the queue is removed from the cache.

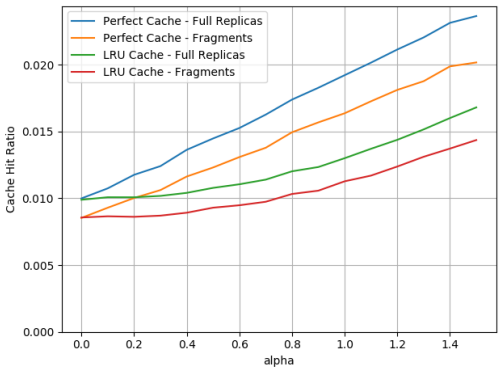
5.3.3 Performance Evaluation

In this section, we evaluate by simulation the impact of our caching policy on distributed storage systems in terms of *cache hit ratio* and *cache waste ratio*. The simulation is carried out for a storage system storing 10^5 objects across 100 servers.

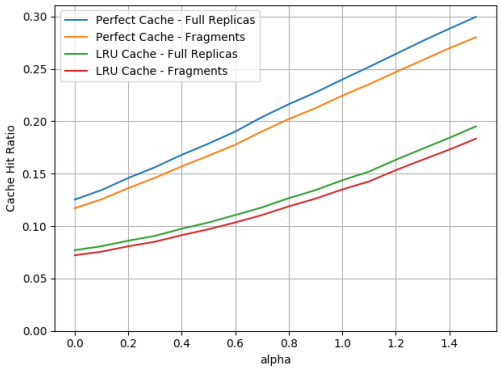
The different parameter sets used for our simulation are given in table 5.1. The cache capacity is given as a fraction of the average number of fragments stored by a server: a cache capacity of 0.01 means that a server's cache can hold $0.01 \times \frac{10^5}{100} = 10$ objects – or $10 \times k$ fragments depending on the caching policy. For each parameter set, we simulate 10^6 consecutive requests following a Zipf-like popularity distribution of variable class repartition and skewness parameter α. A class repartition of $[1,1,4,4]$ corresponds to 10% of the objects in the most popular class, 10% in the second most popular class, 40% in the third and 40% in the least popular one.



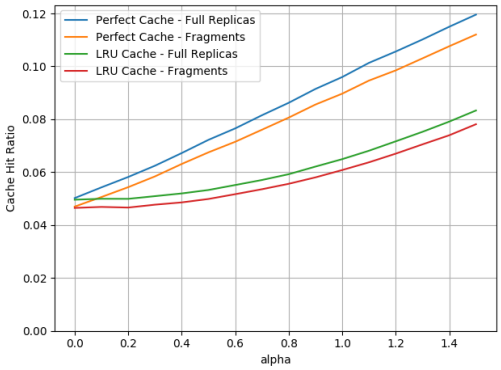
(a) Cache capacity: 0.01, Class repartition: $[1, 1, 4, 4]$



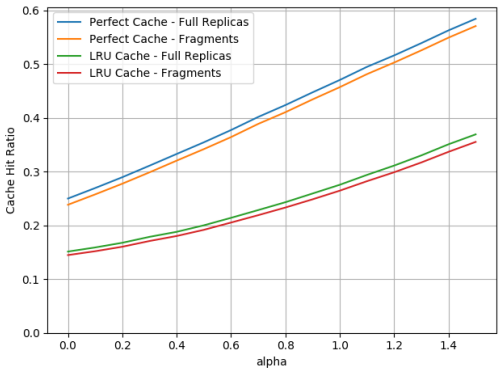
(b) Cache capacity: 0.01, Class repartition: $[1, 1, 1, 1]$



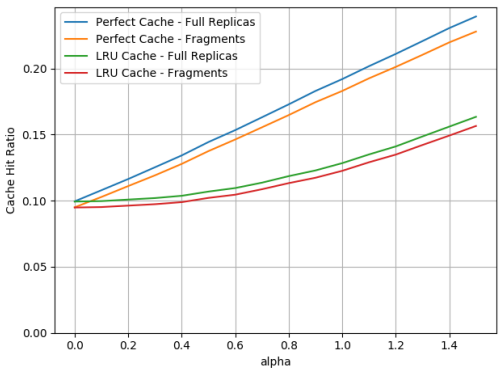
(c) Cache capacity: 0.05, Class repartition: $[1, 1, 4, 4]$



(d) Cache capacity: 0.05, Class repartition: $[1, 1, 1, 1]$



(e) Cache capacity: 0.1, Class repartition: $[1, 1, 4, 4]$



(f) Cache capacity: 0.1, Class repartition: $[1, 1, 1, 1]$

Figure 5.3: **Cache hit ratio for various class repartitions and cache capacities. Results are shown for LRU and perfect cache, and for regular fragment and full replica caching.**

5.3.4 Results and evaluation

The cache hit ratio obtained during these experiments is given in figure 5.3, and the cache waste when the traditionnal fragment caching approach is used is given in figure 5.4. As shown in figure 5.3, our approach has a better cache hit ratio in every case. However, the magnitude of the improvement depends on the parameters, and mostly on the cache capacity. As expected, the cache hit ratio increases in all cases with the cache capacity and with the Zipf skew parameter α. Furthermore, it is also higher for a $[1,1,4,4]$ repartition than for a $[1,1,1,1]$ repartition, as expected again since there is less dispersion between popular objects in such a repartition. Note that with a repartition of $[1,1,1,1]$ and $\alpha = 0$, all objects are equaly as popular, and therefore the cache hit ratio at these points for a perfect cache is equal to the cache capacity.

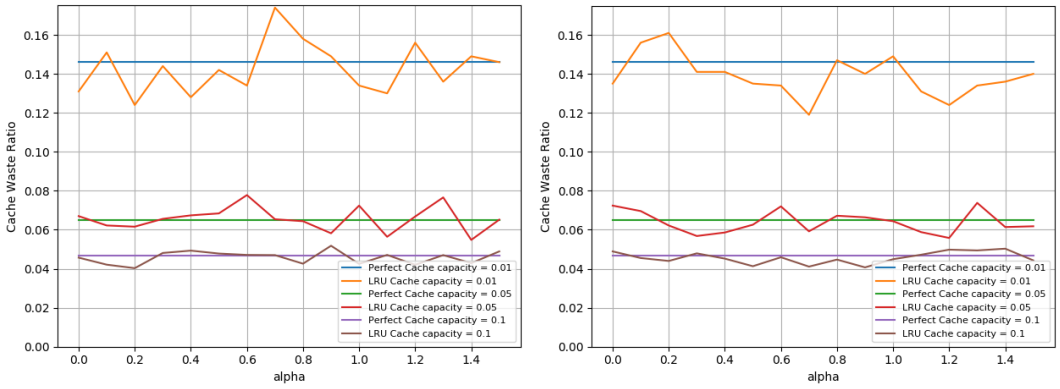
Our caching approach increases the cache hit ratio itself between $\sim 4\%$ in the worst case and $\sim 18\%$ in the best case, with this value decreasing when the cache capacity increases. The numbers show that the cache capacity is the only real factor determining the cache hit ratio difference between a regular fragment caching approach and our replica caching approach.

This correlation between cache capacity and cache hit ratio improvement is easily explained: the only objects for which our policy matters is the ones that are popular enough to have a few fragments cached but not all of them. This does not apply either to very popular objects nor to very unpopular objects. While the threashold popularity depends on the cache capacity, the number of objects in this situation – and thus the cache hit gain from our algorithm – does not. Thus, while the cache hit ratio increases with the cache capacity for both approaches, the number of objects that are fully cached with our approach and not with a regular fragment cache stays approximatively the same, reducing the relative difference between both caching algorithms.

This fact is perfectly illustrated in figure 5.4: the cache waste ratio, defined in section 5.2.2, diminishes when the cache capacity augments, but is virtually unaffected by changes in popularity skewness (both when the class repartition and when α changes). Note that the cache waste ratio does not depend on α or the class repartition for the perfect cache, since the most popular fragments are explicitely cached, regardless of the skewness of the distribution.

5.4 Experimental Evaluation

After running simulations, we wanted to test the impact of our caching scheme on a real system. To do that, we implemented logical independant storage servers on a single server managing 23 10K rpm HDDs –one per disk– and submitted them to the same type of workload that we simulated in the previous section.



(a) Class repartition: $[1, 1, 4, 4]$ *(b)* Class repartition: $[1, 1, 1, 1]$

***Figure 5.4:* Cache waste ratio for various class repartitions and cache capacities. Results are shown for LRU and perfect cache, only when fragment caching.**

5.4.1 Experimental setup

Our experimental setup has several characteristics:

- Since we only want to measure the impact of caching full replicas versus fragments and not the impact of the encoding and decoding of data itself, we only store generic fragments and objects in the storage servers instead of actual data.
- Each storage server stores precisely one generic fragment and one full object in memory, and several generic fragments on disk. There are more fragments on each disk than what the disk cache is able to hold. The storage servers cycle through these on-disk fragments as illustrated in figure 5.5 for each request so that we maintain a disk cache hit ratio of 0, since we only want to measure the impact of the filesystem cache layer.
- Each storage server maintains a virtual LRU structure that is updated after every request and uses object or fragment name as identifier.
- On a cache hit, a storage server sends the generic fragment or object (depending on the caching policy) to the client or storage node requesting it directly from memory.
- On a cache miss, a storage server sends a generic fragment from its disk, cycling through them between requests. These interactions are summarized in table 5.2.

Each storage server keeps track of the cache hit ratio for requests coming through them. Like in the previous section, a request is considered a cache hit only when all fragments are local cache hits on the corresponding storage nodes when fragments are cached.

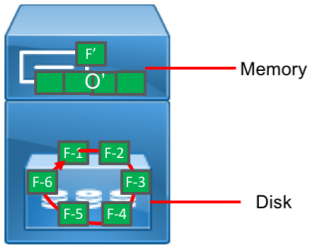


Figure 5.5: **Storage server implementation: a single generic fragment and object in memory, and enough generic fragments on disk to cycle through them without ever hitting the disk cache.**

	Request from Client	Request from Storage Server
Fragment Cache	• Gather generic fragments from servers • Send full generic object to client • Update local fragment cache	• Send generic fragment to server • Update local fragment cache
Replica Cache	• If cached, send full generic object to client • Else, gather generic fragments from servers • Update local object cache	• Send generic fragment to server

Table 5.2: **Functions performed by storage servers when receiving a request.**

Parameter	Value
Continuous sequential requests during : t	120 seconds
Number of objects : N_o	23×10^3
Number of servers : N_s	23
Object size : O_s	500KB
Number of fragments : k	5
Cache capacity : N_c	0.05
Number of class of popularity	4
Class repartitions	$[1, 1, 4, 4]$
Zipf exponent: α	$[0, 0.2, 0.4, \ldots, 1.6]$

Table 5.3: **Benchmark Parameters Settings**

A client is implement and deployed on the same physical server. Like in the previous section, the client send requests randomly following a modified Zipf-law. In addition to the cache hit ratio measured by storage servers, the client measures the time requests take to complete, to measure the performance impact of our replica caching policy in addition to its cache hit ratio improvement.

For each parameter set given in table 5.3, the client continuously sends sequential requests until a fixed amount of time has elapsed.

5.4.2 Results and Evaluation

The evolution of the obtained cache hit ratio of both approaches in function of α is displayed in figure 5.6. Our full replica caching policy has a slightly better cache hit ratio. However, the difference appears to be slim and the apparent variance prevents us from drawing additional conclusions.

Figure 5.7 presents response time histograms for all the experiments. These histograms show not only the effect of increased skewness on cache hit ratio, but also the effect of our caching policy on response time when cache hits occur. While figure 5.6 displays only a slight difference in cache hit ratio, it is apparent on figure 5.7 that caching full replicas has much more benefit in terms of response time than caching all the fragments from an object.

Cache hits correspond to the leftest bins on the histograms (lowest response time). For the histograms corresponding to our replica caching approach, the cache hits are clearly separated from the cache misses and display a reponse time of $\sim$ 8 ms on average, whereas the cache hits from the fragment caching approach display an average response time of $\sim$ 10 ms, equal to the fastest cache miss reponse times.

We think that comes from the additional network and computing overhead and their effect on performance: the variance and delay introduced by

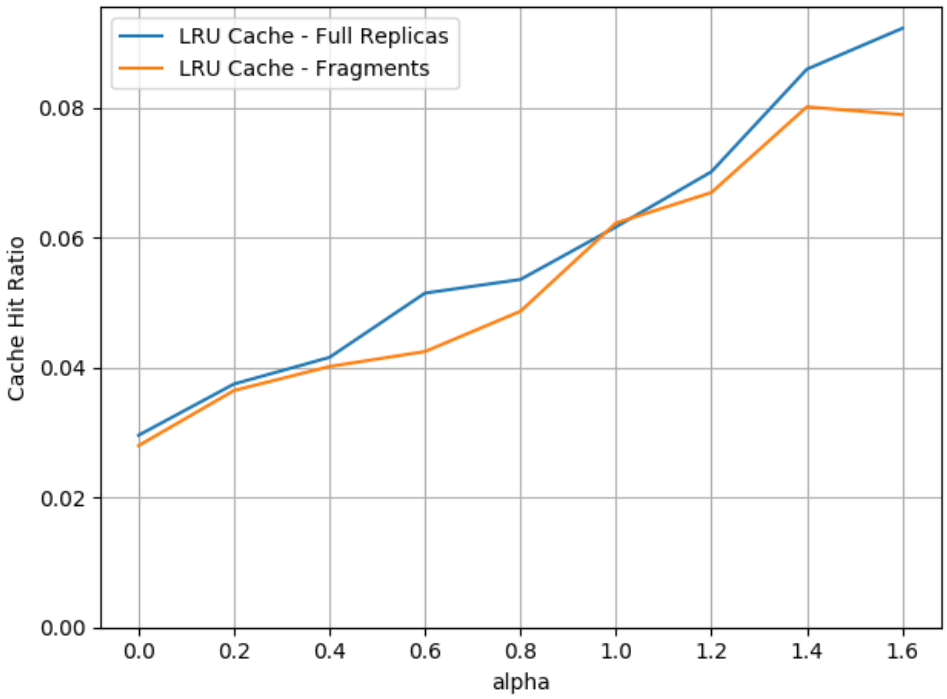


Figure 5.6: **Cache hit ratio for different values of α, with a LRU cache.**

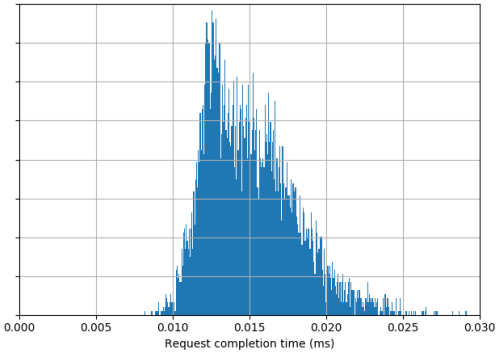
fetching fragments – even cached – from other servers almost counterbalances the positive effect of the cache itself.

5.5 Conclusion

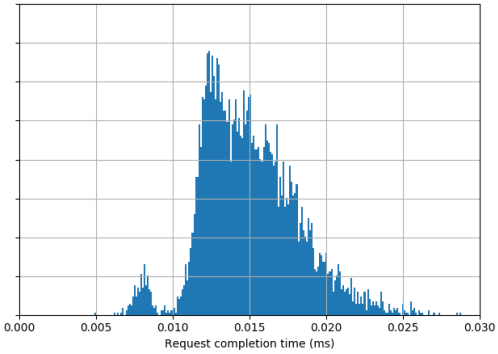
In this chapter, we explained why relying on traditionnal filesystem caching mechanism is not optimal for storage systems using erasure codes as a reliabilty mechanism. We presented a straightforward solution, which consists of caching entire objects or nothing, thereby avoiding cases where only some fragments of an object were cached.

A simulated setup showed that our solution improved the cache hit ratio of such systems in every case (between 4 and 18% depending on the parameters). We also showed that our approach's benefit depended monstly on the cache capacity, not so much on other factors such as the popularity distribution skewness or the exact caching policy which have no visible effect on the cache waste ratio of the traditionnal fragment caching solution.

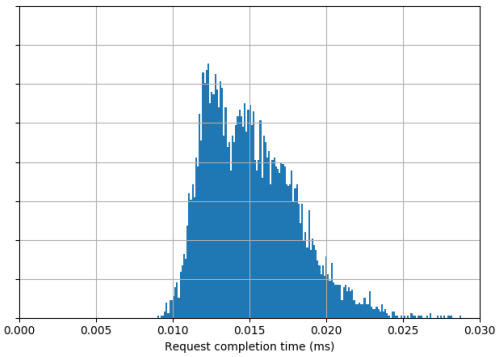
In a second phase, our implementation not only validated the previous result, but also showed that caching fragments on every node rather than full replicas at once had an adverse effect on cache hit response time. Our implementation showed that cache hits on full replicas had a better request response time than cache hits on all fragments of an object, because of the



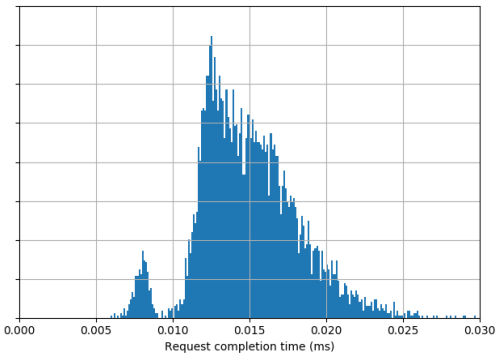
(a) Fragment cache, $\alpha = 0.0$



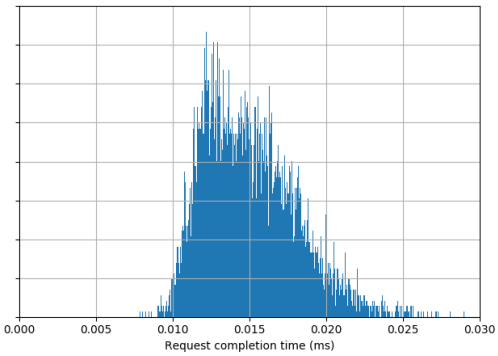
(b) Replica cache, $\alpha = 0.0$



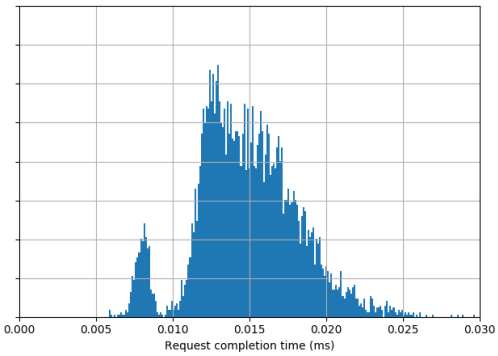
(c) Fragment cache, $\alpha = 0.4$



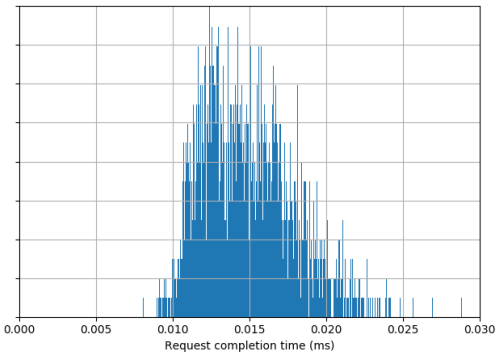
(d) Replica cache, $\alpha = 0.4$



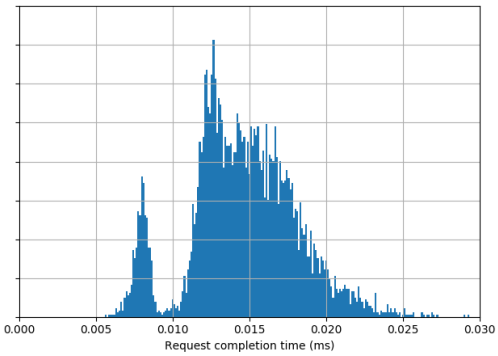
(e) Fragment cache, $\alpha = 1.0$



(f) Replica cache, $\alpha = 1.0$



(g) Fragment cache, $\alpha = 1.6$



(h) Replica cache, $\alpha = 1.6$

Figure 5.7: **Response time histograms for different values of α.**

additional networking and computing delays suffered when retrieving the fragments from other servers.

Contributions

The work presented in this chapter is the object of a patent and of a paper submission.

- Hybrid distributed storage system to dynamically modify storage overhead and improve access performance, patent, not yet issued [222]
- Replica Caching for Erasure-Code based Distributed Storage Systems, paper, in preparation for submission.

Conclusion

The goal of this dissertation was twofold. First, to present the limitations of existing distributed storage systems in chapter 1 and describe 6Stor, an architecture inspired from their strengths to avoid their drawbacks in chapters 2 and 3. Second, to point out several specific interactions of such systems and propose mechanisms to improve them, and thus their throughput and latency guarantees in chapters 4 and 5. While the different chapters of this dissertation might seem uncorrelated, they all gravitate towards the same goal: to help distributed storage systems face the explosive growth of data requiring storage and distribution.

- Chapter 1 gave a brief overview of the different types of distributed storage systems. It described several architectures, their advantages and drawbacks, and then explained how those systems dealt with node failures by implementing reliability mechanisms. These same mechanisms introduce consistency issues that are dealt with in numerous ways. Finally, this chapter presented a short list of widely used distributed storage systems and their reliability and consistency mechanisms. Most distributed storage system limitations presented in this chapter have something to do with scalability issues, whether it is the single master bottleneck or the data redistribution spanning hours or days when petabytes of data have to be moved between storage nodes.

- In chapter 2, I presented 6Stor, a distributed object store that combines the distributed aspect of DHTs with the metadata layer offered by master-slave storage systems. 6Stor also leverages IPv6 capabilities to provide scalability and fast bootstraps for joining storage nodes. The architecture was thoroughly described and its performance evaluated against a similar object store – Ceph. 6Stor was also designed to limit the number of software layers, which was evaluated through measurements showing the overhead of using HTTP instead of simple TCP.

- Chapter 3 presented two extensions to 6Stor. Firstly, I explained why distributed storage systems often implement a block device API since many applications rely on them. I then described the process we went through to implement such a block device on top of 6Stor, and presented a benchmark of its performance for different I/O size and parameters. Secondly, I explained how we could leverage 6Stor's IPv6 capabilities through the use of segment routing to enable load-balancing,

thus reducing the latency of 6Stor's basic operations, especially when under heavy load.

- In chapter 4, I explained why traditionnal I/O and packet schedulers were not sufficient to ensure an efficient and fair repartition of storage resources between different applications or users sending requests to individual storage servers. I then introduced RS3, a request scheduler deployed on storage servers using batching to fairly allocate storage resources between request classes. I showed that RS3 allows small requests to be handled much faster than without RS3 when faced with concurrent large requests, and that RS3 almost equalizes the throughput of classes simultaneously sending consecutive requests, even when some of them sent them at a faster rate that others. Such scheduling mechanisms are becoming essential in a time where hyper-convergence becomes the norm and brings hundreds of containers, VMs and applications on the same servers, relying on the same storage backends.

- Finally, chapter 5 explained how to improve traditionnal filesystem's caching on individual distributed storage systems servers when erasure codes are used instead of replication to provide reliability. The approach presented requires no synchronization and uses the traditionnal LRU mechanisms used in most caching systems. I showed that it increases the cache hit ratio of such storage systems by caching only full replicas of data objects instead of only fragments. Furthermore, experiments showed that cache hits were more beneficial when a full replica was cached than when fragments were cached on different storage servers. Erasure codes are becoming the norm as a reliability mechanism since the amount of data generated by humankind grows faster than what the hardware can currently handle. It is therefore essential to make sure that generic caching mechanisms are as efficient as possible in this setup.

Through this work, I showed that there are several directions to explore in order to improve the performance, scalability and flexibility of distributed storage systems. There are obviously many more steps to take in everyone of these directions, and in others. To this end, every chapter presented some clues as to what these next steps in their respective directions could be.

A specific area worth further investigation is the use of DPDK and SPDK. DPDK and SPDK are libraries used to build custom user-space network and storage stacks respectively. Ceph is currently investigating how to use SPDK in their custom data store named Bluestore that replaces the local filesystem by custom user-space code providing only what is necessary for such a data store (cutting the filesystem journaling, extended attributes, ACLs...), and we believe that we could even leverage DPDK with 6Stor's IPv6 capabilities in addition to SPDK to create such an optimized user-space layer providing

zero-copy storage directly from network with the least possible computing overhead.

We also want to explore the possibility of custom placement policies allowed by 6Stor's metadata layer, allowing different objects from the same cluster to have different storage representations (replication or erasure codes) on different storage tiers.

Appendix A and B present work on containers and file servers carried out in the PIRL in the context of the chairs with Telecom ParisTech and Ecole Polytechnique. Both those appendix refer to contributions made during 5-months research internships from Ecole Polytechnique's students. While the topics are not directly related to the rest of this PhD dissertation, I found that they were still relevant to the subject at hand and thus were a worthwhile inclusion in this manuscript.

Appendix A

Predictive Container Image Prefetching

Disclaimer

This work presented here is currently the object of a patent application. The container statistics presented here have been gathered and compiled by Maxime Larcher in the context of a research internship.

A.1 Motivations

Containers are processes packaging code and dependancies used as standalone executable packages of software deployed to run applications. Container images contain everything they need to run including libraries, code, system tools, settings ... They are mainly used as more lightweight alternatives to VMs used to rapidly deploy, upscale or downscale applications and services in cloud environments. Containers images are run on container engines (such as Docker or Kubernetes) that play a similar role to an hypervisor for VMs. Their properties have made them the heart of cloud service deployment innovation for the last few years. However, they are not always exploited to the best of their capabilities.

A container must be *pulled* –fully downloaded– on a node before it can be *run*. Therefore, containers are supposed to be as light as possible and package only what they require. However, in reality, most popular container images contain a lot of data not read during the container execution – and thus useless to download. This has the unfortunate consequence of slowing the start time of most containers. This section presents a way to reduce this unnecessary download overhead and speed up the container start time. For the remainder of this section, we take the specific example of docker containers.

A.2 Storage and containers

Docker containers have three ways to interact with storage:

- Users can mount a host directory within the container, for example the source directory of code files for a compiler to provide binaries or some access to data the container has to process;
- Users can also attach a separate *Docker volume* with persistent data to a docker container when running it;
- The container's root file system is mounted by the Docker storage driver and is composed from several *layers*. All relevant layers are pulled with a container, and assembled in a single read-only layer when a container is run for the first time on a host. A non-persistent Copy-On-Write (COW) layer – where directories and files created or modified are created – is then created each time a container is run.

The two first storage interactions are optional and correspond to data that is already on the sever. However, as explained before, the last one requires every node on which the container is deployed to pull all layers and assemble them. This is problematic for two reasons:

- Some containers are typically scheduled to be deployed on different servers over time, and are run for a short time every time. Furthermore, the complete image is not necessarily kept on the server between executions;
- One of the reasons containers are in use is the short time they take to deploy. The pull phase artificially increases the "time to first byte" of the container – which is the time between the `docker run` (or equivalent) command and the time the container becomes functional – when the image is not already on the server.

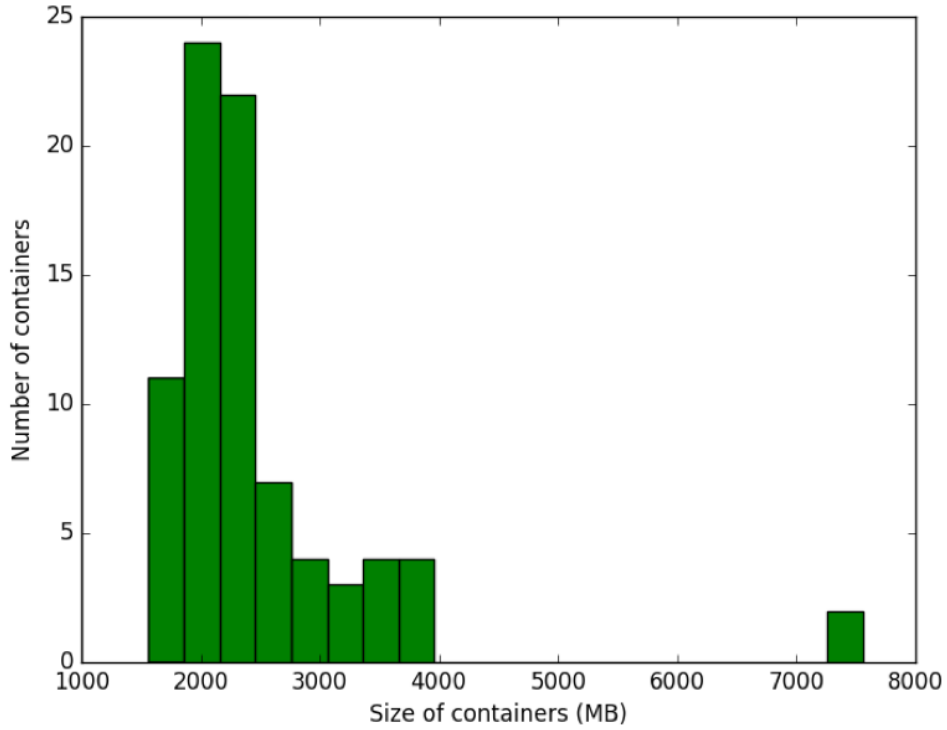
Docker containers themselves are composed of multiple hierarchicaly organized read-only layers and a single, temporary writeable layer that serves as a root filesystem for the container during its execution.

A.3 Some statistics about popular container images

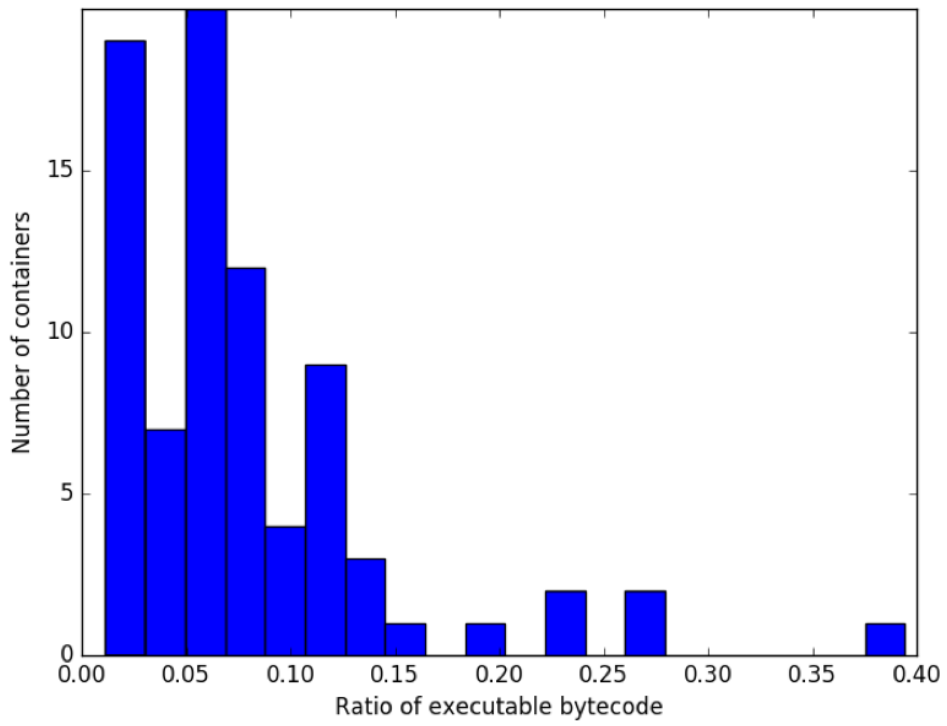
While the pull time is incompressible with regard to the data required by the container to run, most container images are not optimally built and thus contain a lot of never-accessed data that slows container pull time. Furthermore, this useless data wastes storage space on servers themselves.

To evaluate the scale of this problem, we analyzed 81 of the 100 most popular container images from Docker Hub [1] at the time (05/2017). We measured the size and number of executable files in those images as well as the number of layers they are made of. The results are presented in figure A.1.

[1] `https://hub.docker.com/`



(a) Container size distribution among the 81 of the most popular docker images.



(b) Distribution of the proportion of executable files among 81 of the most popular docker images.

***Figure A.1:* Size and executable proportion distributions for 81 of the most popular docker images.**

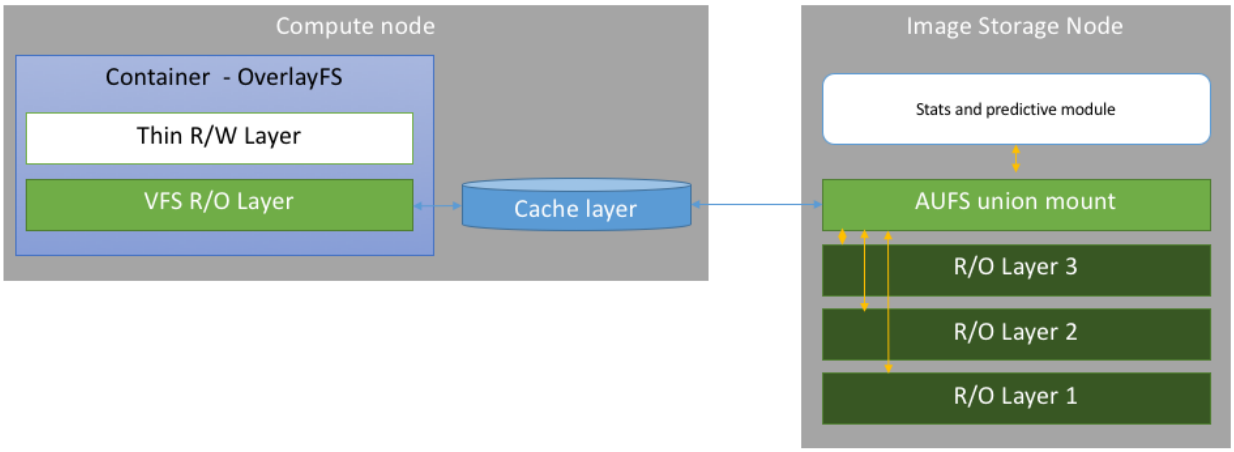


Figure A.2: **OPCISS Architecture.**

The results show that most containers are of size between 2 and 3GB. On average, only 8% of this size is made of executables. Furthermore, it has recently been shown [223] (albeit on an older range of containers) that only 6.4% of the images' data is actually read by these containers in average. The same study shows that pulling a container's image accounts for 76% of the "time to first byte" in average. This work proposes a way to lazily pull only the required data rather than the whole images. It also improves cache sharing between container images having layers in common.

A.4 Optimized Predictive Container Image Storage System (OPCISS)

In addition to the statistics presented in the previous section, [223] showed that 99% of the reads incurred when a container is run can be serviced by the cache if the container has been previously run on the same server. This indicates a very high predictability during container execution. Therefore, we propose the following architecture (see figure A.2) that adds a predictive component to the lazy prefetching described in [223].

An *Image Storage Node* is deployed, that stores or pulls the full images of containers to be executed in the environment. Containers are deployed on *Compute Nodes* that do not pull full container images before running them. Instead, they execute them and fetch the required data directly from a local *Image Storage Node* storing full images for all the containers used in the environment. There can be one or several such nodes per datacenter or compute node sets depending on the number of compute nodes and/or containers.

OPCISS works as follows:

- The R/O layers are presented as a AUFS union mount on the image storage node;

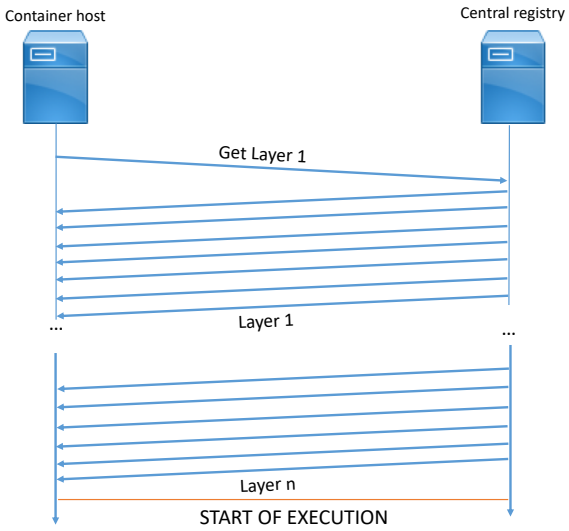
- On the compute node, each read request goes through a local cache and then as a NFS request to the image storage node if there is no cache hit;
- On the image storage node, a predictive modules keeps track of what blocks and files are requested per container image, and in what order;
- After sufficient training, this predictive module can send blocks to compute nodes even before they are requested, further reducing the spin up time for containers.

The difference between the traditionnal approach, Slacker's approach [223] and ours is illustrated in figure A.3. Because of the highly deterministic nature of container execution shown in [223], we expect OPCISS's predictive module to highly efficiently prefetch the correct blocks for the compute nodes.

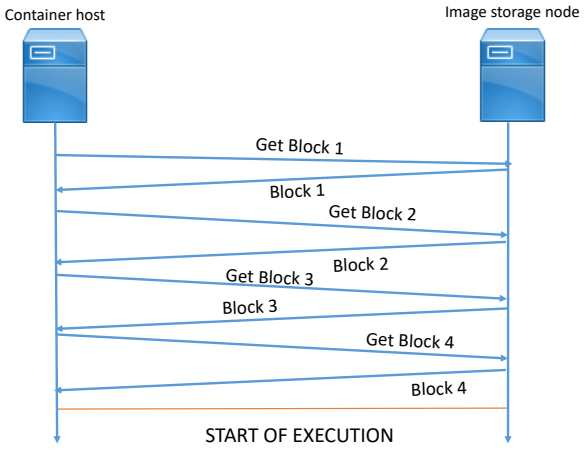
Contributions

A patent describing OPCISS is currently being reviewed by the US Patent Office

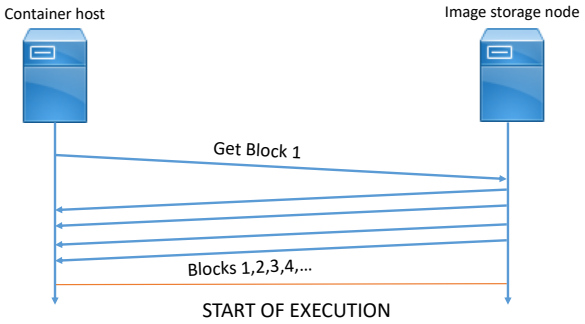
- Predictive container image storage system for fast container execution, patent, not yet issued [224]



(a) Traditionnal approach: all layers are fully downloaded.



(b) Slacker's approach: only requested blocks are downloaded from a local image storage node.



(c) OPCISS's approach: only requested blocks are downloaded from the image storage node. Furthermore, blocks that have a high probability of getting requested next are also sent to the Compute Node.

Figure A.3: **Comparison of spin up strategies for containers.**

Appendix B

Vectorizing TCP data handling for file servers

Disclaimer

The work presented here is currently the object of a patent application. I participated to the patent elaboration and to the idea of vectorizing TCP's packet handling by sharing a single memory copy of data segments between connections. The precise segment-oriented architecture and implementation in VPP, however, are mostly the work of Clément Durand in the context of a research internship.

B.1 Motivations

Most Content Delivery Networks (CDNs) and storage systems use the HTTP protocol to deliver data to clients. While HTTP can use UDP-based protocols such as the Quick UDP Internet Connection protocol (QUIC), TCP amounts for 90% of the Internet traffic [225]. Most applications use their choice OS's kernel implementation of TCP, but it has been shown that it is possible to outperform the standard implementation [226, 227, 228, 229], notably by using Linux foundation's (formerly Intel's) DPDK [230]. Notably, it is possible to make optimizations when the same content is being delivered to multiple clients at once.

DPDK is a core component of FD.io's VPP, a performant [231] virtual switch/router running in user-space. VPP is an open source project that has been used in Cisco products for fifteen years. It has been designed to leverage the processor's instruction cache by batching packets organized in vectors. To this end, VPP is organized as a graph node, each node corresponding to a function carried on the packets it receives. Each graphe node's code is written to fit into the instruction cache and vector of packet pointers are passed between the graph nodes instead of packet per packet processing. Packet themselves are initially put in buffers which indices are transmitted from graph node to graph node to avoid memory copies. An example of VPP graph is presented in figure B.1.

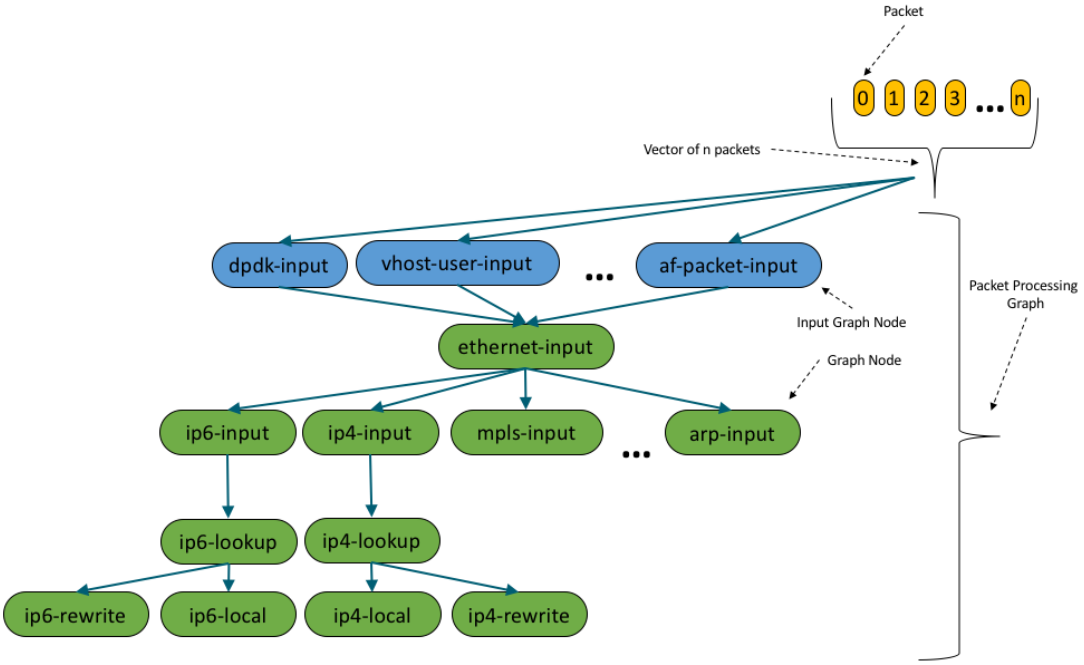


Figure B.1: **VPP Architecture. A Vector of packets come through an input node, and are then handled by graph nodes performing atomic network functions.**

This work aims at applying the same concepts to build a custom TCP stack optimized to distribute content to several clients without incurring memory copies, by having every TCP connection for the same content sharing *data segments*. This is not possible with the traditionnal byte-stream-oriented TCP socket API.

B.2 State of the art

DPDK allowed work such as mTCP [227] to deploy an efficient TCP stack in user-space. In [226], a dedicated web server is designed that generates all the TCP packets (including TCP, IP and link-layer headers) associated with an HTTP request as soon as the initial HTTP header is processed. However, these packets are not shared between different connections for the same content, which incurs more memory footprint, memory copies and processing.

B.3 Segment-oriented TCP in VPP

This section describes how a segment-oriented TCP stack can be implemented as VPP nodes to reduce the number of memory copies during TCP packet handling.

POSIX TCP socket APIs echo the byte-stream nature of TCP. The user sends and receives bytes rather than packets. The translation between a byte

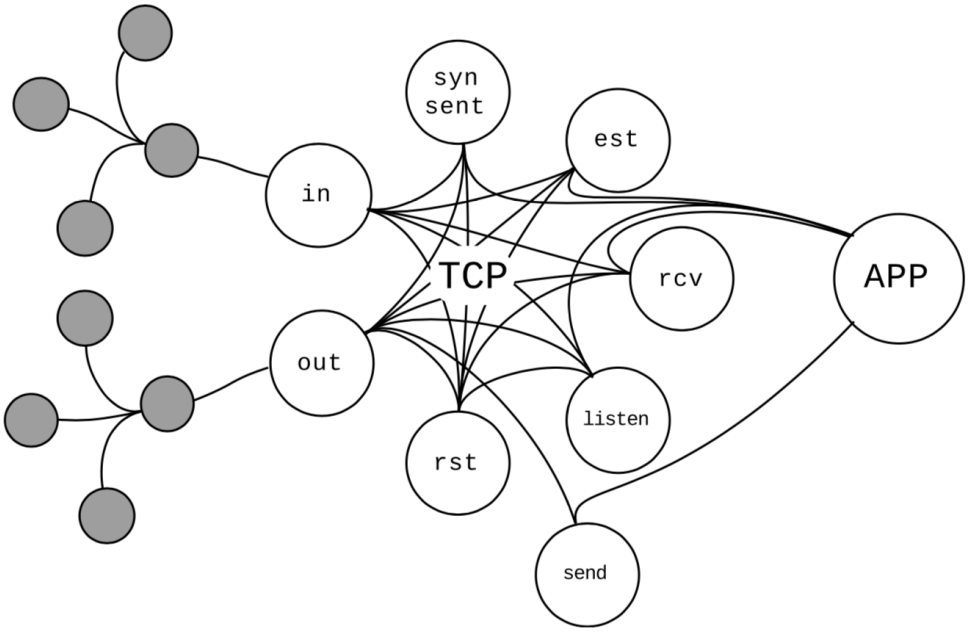


Figure B.2: **Application as a VPP node, passing pointers to data segments to the VPP TCP graph nodes.**

stream and TCP packets require memory copies from user space data buffers to kernel space packet buffers. A solution to this problem is to enable an application to directly send pointers to data segments in a VPP graph containing TCP nodes, as illustrated in figure B.2. In this example, the application is a graph node that, as any other VPP graph node, sends pointers to its data segments that are handled by the VPP TCP nodes.

B.4 Zero-copy file server

Static file servers often directly map the delivered files to memory to send them faster through TCP. However, this still requires memory copies since the whole file is contiguous in memory and leaves no room for TCP headers. Therefore, we propose to pre-compute the data similarly to what is done in [226], except that the pre-computation splits the data in segments fit for the TCP segment-oriented API presented previously, as illustrated in figure B.3. Instead of a contiguous file in memory, the file is pre-packetized in a list of segment buffers that can be moved without copy through a VPP graph. The TCP API just has to chain the TCP headers to these segments when clients require them, without copying the segments themselves for each TCP connection. The combination of the segment-oriented TCP API and this implementation of file server require zero memory copies from disk to network.

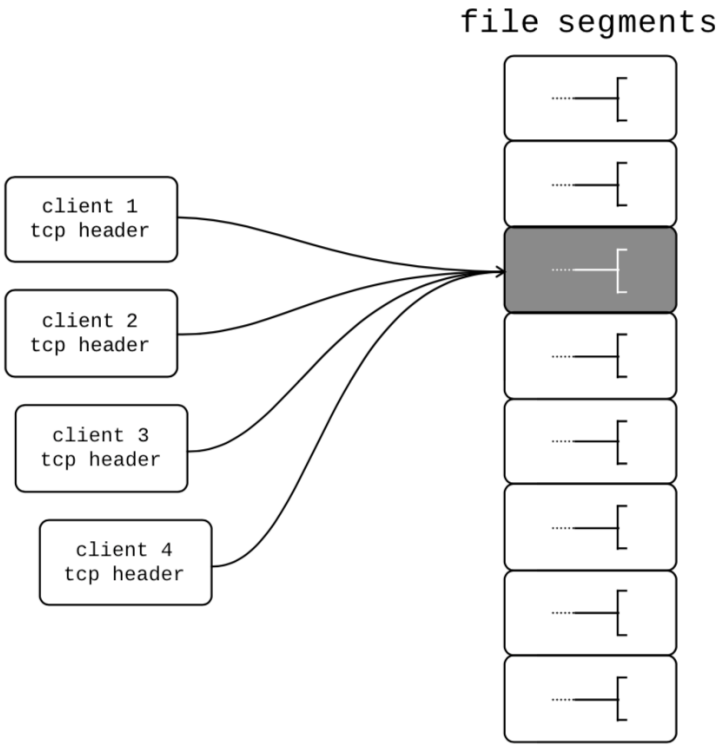


Figure B.3: **File is pre-packetized in file segments. The file server uses the same segments for every connection and just has to chain the TCP header and compute the checksum.**

Contributions

This appendix presented a way to create a zero memory copy TCP file server, using a single memory copy of content to deliver it to all users. However, the same mechanisms could be used in other, more complex storage systems.

A patent describing the zero-copy file server is currently being reviewed by the US Patent Office.

- Data stream pipelining and replication at a delivery node of a content delivery network, patent, not yet issued [232]

Bibliography

[1] Fritz Kruger SanDisk Fellow. *CPU Bandwidth – The Worrisome 2020 Trend Guest Blog*. 2016. URL: https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/.

[2] I Atkin. "Getting the Hang of IOPS, an Introduction to Disk Performance". In: *Symantec Connect Whitepaper* (2012).

[3] Feng Chen, David A Koufaty, and Xiaodong Zhang. "Understanding intrinsic characteristics and system implications of flash memory based solid state drives". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 37. 1. ACM. 2009, pp. 181–192.

[4] Qiumin Xu et al. "Performance analysis of NVMe SSDs and their implication on real world databases". In: *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM. 2015, p. 6.

[5] Mark Hachman. *Notebook hard drives are dead: How SSDs will dominate mobile PC storage by 2018*. 2015. URL: https://www.pcworld.com/article/3011441/storage/notebook-hard-drives-are-dead-how-ssds-will-dominate-mobile-pc-storage-by-2018.html.

[6] Mihir Nanavati et al. "Non-volatile Storage". In: *Queue* 13.9 (Nov. 2015), 20:33–20:56. ISSN: 1542-7730. DOI: 10.1145/2857274.2874238. URL: http://doi.acm.org/10.1145/2857274.2874238.

[7] Rob Davis. *The Network is the New Storage Bottleneck*. 2016. URL: https://www.datanami.com/2016/11/10/network-new-storage-bottleneck/.

[8] Bernard Marr. *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*. 2018. URL: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#3ce481e960ba.

[9] Kaushal Amin. *Big Data Overview 2013-2014*. URL: https://www.slideshare.net/kmstechnology/big-data-overview-2013-2014.

[10] James Zetlen. *Google's datacenters on Punch cards*. 2013. URL: https://what-if.xkcd.com/63/.

[11] Sean Gallagher. *NSA "touches" more of Internet than Google*. 2013. URL: https://arstechnica.com/information-technology/2013/08/the-1-6-percent-of-the-internet-that-nsa-touches-is-bigger-than-it-seems/.

[12] Richi Jennings. *NSA's Huge Utah Datacenter: How Much Of Your Data Will It Store? Experts Disagree...* 2016. URL: https://www.forbes.com/sites/netapp/2013/07/26/nsa-utah-datacenter/#53852dc45d9c.

[13] Ren Wu. *Deep learning meets heterogeneous computing*. 2014.

[14] Pamela Vagata and Kevin Wilfong. *Scaling the Facebook data warehouse to 300 PB*. 2014. URL: https://code.fb.com/core-data/scaling-the-facebook-data-warehouse-to-300-pb/.

[15] Tom Fastner. *Extreme Analytics at eBay*. 2011.

[16] Liz Tay. *Inside eBay's 90PB data warehouse*. 2013. URL: https://www.itnews.com.au/news/inside-ebay8217s-90pb-data-warehouse-342615.

[17] Richard Brueckner. *Sanger Institute Deploys 22 Petabytes of Lustre-Powered DDN Storage*. 2013. URL: https://insidehpc.com/2013/10/sanger-institute-deploys-22-petabytes-lustre-powered-ddn-storage/.

[18] Tim Cutts. *Managing Genomics Data at the Sanger Institute*. 2013.

[19] Adam Kawa. *Hadoop Operations Powered By ... Hadoop*. 2014.

[20] Ron Kohavi and Roger Longbotham. "Online experiments: Lessons learned". In: *Computer* 40.9 (2007).

[21] Greg Linden. "Geeking with Greg: Marissa Mayer at Web 2.0.(2006)". In: *URL http://glinden. blogspot. com/2006/11/marissa-mayer-at-web-20. html.[Online* ().

[22] Wolfgang Lehner and Kai-Uwe Sattler. "Data Cloudification". In: *Web-Scale Data Management for the Cloud*. Springer, 2013, pp. 1–12.

[23] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 29–43. ISSN: 0163-5980. DOI: 10.1145/1165389.945450. URL: http://doi.acm.org/10.1145/1165389.945450.

[25] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: http://doi.acm.org/10.1145/1365815.1365816.

[26] James C. Corbett. "Spanner: Google's Globally-Distributed Database". In: ().

[27] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[28] Sage A. Weil et al. "Ceph: A Scalable, High-performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 307–320. ISBN: 1-931971-47-1. URL: http://dl.acm.org/citation.cfm?id=1298455.1298485.

[29] K. Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[30] Kirk McKusick and Sean Quinlan. "GFS: Evolution on Fast-forward". In: *Commun. ACM* 53.3 (Mar. 2010), pp. 42–49. ISSN: 0001-0782. DOI: 10.1145/1666420.1666439. URL: http://doi.acm.org/10.1145/1666420.1666439.

[31] K.V. Shvachko. "HDFS scalability: The limits to growth". In: 35 (Jan. 2010), pp. 6–16.

[32] Dong-Yun Lee et al. "Understanding write behaviors of storage backends in Ceph object store". In: *Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology*. Vol. 10. 2017.

[33] *Ceph Community Newsletter, October 2018 edition*. 2018. URL: https://ceph.com/community/ceph-community-newsletter-october-2018-edition/.

[34] Ceph. *Ceph-Crimson*. URL: https://github.com/ceph/ceph/projects/2.

[35] Spencer Shepler et al. *Network file system (NFS) version 4 protocol*. Tech. rep. 2003.

[36] windows-sdk content. *Microsoft SMB Protocol and CIFS Protocol Overview - Windows applications*. 2018. URL: https://docs.microsoft.com/fr-fr/windows/desktop/FileIO/microsoft-smb-protocol-and-cifs-protocol-overview.

[37] *Apple Filing Protocol Programming Guide*. 2012. URL: https://developer.apple.com/library/archive/documentation/Networking/Conceptual/AFP/Concepts/Concepts.html.

[38] John H Howard et al. *An overview of the andrew file system*. Carnegie Mellon University, Information Technology Center, 1988.

[39] *OpenAFS*. URL: http://www.openafs.org/.

[40] Mahadev Satyanarayanan et al. "Coda: A highly available file system for a distributed workstation environment". In: *IEEE Transactions on computers* 39.4 (1990), pp. 447–459.

[41] Peter Braam, Michael Callahan, Phil Schwan, et al. "The intermezzo file system". In: *Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*. 1999.

[42] Ian Clarke and Supervisor Dr Chris Mellish. *A Distributed Decentralised Information Storage and Retrieval System*. Tech. rep. 1999.

[43] Ian Clarke et al. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". In: *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Ed. by Hannes Federrath. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 46–66. ISBN: 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_4. URL: https://doi.org/10.1007/3-540-44702-4_4.

[44] Wikipedia. *Gnutella — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Gnutella&oldid=837420609. [Online; accessed 27-June-2018]. 2018.

[45] Wikipedia. *Kazaa — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Kazaa&oldid=841649382. [Online; accessed 27-June-2018]. 2018.

[46] Xuemin Shen et al. *Handbook of Peer-to-Peer Networking*. 1st. Springer Publishing Company, Incorporated, 2009, pp. 118–119. ISBN: 0387097503, 9780387097503.

[47] Ion Stoica et al. "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications". In: *IEEE/ACM Trans. Netw.* 11.1 (Feb. 2003), pp. 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407. URL: http://dx.doi.org/10.1109/TNET.2002.808407.

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

[49] Antony Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware 2001*. Ed. by Rachid Guerraoui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 329–350. ISBN: 978-3-540-45518-9.

[50] Ben Yanbin Zhao, John Kubiatowicz, Anthony D Joseph, et al. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing". In: (2001).

[51] R. R. Noel and P. Lama. "Taming Performance Hotspots in Cloud Storage with Dynamic Load Redistribution". In: *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. 2017, pp. 42–49. DOI: 10.1109/CLOUD.2017.15.

[52] *Using the pg-upmap*. URL: http://docs.ceph.com/docs/mimic/rados/operations/upmap/.

[53] Salman Niazi et al. "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 89–104. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi.

[54] Cade Metz. "Google Remakes Online Empire with 'Colossus'". In: *Wired [Online]. Available: http://www. wired. com/2012/07/google-colossus/* (2012).

[55] Michael Ovsiannikov et al. "The Quantcast File System". In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1092–1101. ISSN: 2150-8097. DOI: 10.14778/2536222.2536234. URL: http://dx.doi.org/10.14778/2536222.2536234.

[56] Frank Schmuck and Roger Haskin. "GPFS: A Shared-disk File System for Large Computing Clusters". In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. FAST'02. Monterey, CA: USENIX Association, 2002, pp. 16–16. URL: http://dl.acm.org/citation.cfm?id=1973333.1973349.

[57] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. "The Global File System". In: *The Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*. Vol. 2. College Park. 1996, pp. 319–342.

[58] *The Lustre File System*. URL: http://doc.lustre.org/lustre_manual.xhtml.

[59] O. Rodeh and A. Teperman. "zFS - a scalable distributed file system using object disks". In: *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.* 2003, pp. 207–218. DOI: 10.1109/MASS.2003.1194858.

[60] S. Yang, W. B. Ligon III, and E. C. Quarles. "Scalable distributed directory implementation on orange file system". In: *IEEE International Workshop for Storage Network Architecture and Parallel I/Os*. 2011.

[61] Atul Adya et al. "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 1–14. ISSN: 0163-5980. DOI: 10.1145/844128.844130. URL: http://doi.acm.org/10.1145/844128.844130.

[62] Juan Benet. "IPFS-content addressed, versioned, P2P file system". In: *arXiv preprint arXiv:1407.3561* (2014).

[63] Sage A Weil et al. "Rados: a scalable, reliable storage service for petabyte-scale storage clusters". In: *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*. ACM. 2007, pp. 35–44.

[64] Kalen Delaney. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2000.

[65] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. "Replication, load balancing and efficient range query processing in DHTs". In: *International Conference on Extending Database Technology*. Springer. 2006, pp. 131–148.

[66] John Byers, Jeffrey Considine, and Michael Mitzenmacher. "Simple load balancing for distributed hash tables". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 80–87.

[67] Sage A. Weil et al. "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188582. URL: http://doi.acm.org/10.1145/1188455.1188582.

[68] Peter M Chen et al. "RAID: High-performance, reliable secondary storage". In: *ACM Computing Surveys (CSUR)* 26.2 (1994), pp. 145–185.

[69] Irving S Reed and Gustave Solomon. "Polynomial codes over certain finite fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.

[70] M Celebiler and G Stette. "On increasing the down-link capacity of a regenerative satellite repeater in point-to-point communications". In: *Proceedings of the IEEE* 66.1 (1978), pp. 98–100.

[71] Rudolf Ahlswede et al. "Network information flow". In: *IEEE Transactions on information theory* 46.4 (2000), pp. 1204–1216.

[72] JeanPierre Guédon and Nicolas Normand. "The Mojette transform: the first ten years". In: *International Conference on Discrete Geometry for Computer Imagery*. Springer. 2005, pp. 79–91.

[73] Nicolas Normand, Andrew Kingston, and Pierre Évenou. "A geometry driven reconstruction algorithm for the Mojette transform". In: *International Conference on Discrete Geometry for Computer Imagery*. Springer. 2006, pp. 122–133.

[74] SAS Fizians. "RozoFS: a fault tolerant I/O intensive distributed file system based on Mojette erasure code". In: *Workshop Autonomic Oct*. Vol. 16. 2014, p. 17.

[75] KV Rashmi et al. "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 331–342.

[76] KV Rashmi, Nihar B Shah, and Kannan Ramchandran. "A piggybacking design framework for read-and download-efficient distributed storage codes". In: *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 331–335.

[77] Korlakai Vinayak Rashmi, Nihar B Shah, and P Vijay Kumar. "Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction". In: *IEEE Transactions on Information Theory* 57.8 (2011), pp. 5227–5239.

[78] KV Rashmi et al. "Explicit construction of optimal exact regenerating codes for distributed storage". In: *arXiv preprint arXiv:0906.4913* (2009).

[79] Yunnan Wu, Alexandros G Dimakis, and Kannan Ramchandran. "Deterministic regenerating codes for distributed storage". In: *Allerton Conference on Control, Computing, and Communication*. 2007, pp. 1–5.

[80] Dimitris S Papailiopoulos et al. "Simple regenerating codes: Network coding for cloud storage". In: *INFOCOM, 2012 Proceedings IEEE*. IEEE. 2012, pp. 2801–2805.

[81] Alessandro Duminuco and Ernst Biersack. "Hierarchical codes: How to make erasure codes attractive for peer-to-peer storage systems". In: *Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on*. IEEE. 2008, pp. 89–98.

[82] David JC MacKay. "Fountain codes". In: *IEE Proceedings-Communications* 152.6 (2005), pp. 1062–1068.

[83] M. Asteris and A.G. Dimakis. "Repairable Fountain codes". In: *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. 2012, pp. 1752–1756. DOI: 10.1109/ISIT.2012.6283579.

[84] Alexandros G Dimakis, Vinod M Prabhakaran, and Kannan Ramchandran. "Distributed Fountain Codes for Networked Storage." In: *ICASSP (5)*. 2006, pp. 1149–1152.

[85] Stig Telfer StackHPC. *CEPH DAY BERLIN - CEPH ON THE BRAIN!* 2018. URL: https://www.slideshare.net/Inktank_Ceph/ceph-day-berlin-ceph-on-the-brain.

[86] Julio Araujo, Frédéric Giroire, and Julian Monteiro. "Hybrid approaches for distributed storage systems". In: *International Conference on Data Management in Grid and P2P Systems*. Springer. 2011, pp. 1–12.

[87] Armando Fox and Eric A. Brewer. "Harvest, Yield, and Scalable Tolerant Systems". In: *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*. HOTOS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 174–. ISBN: 0-7695-0237-7. URL: http://dl.acm.org/citation.cfm?id=822076.822436.

[88] Eric A. Brewer. "Towards Robust Distributed Systems (Abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 7–. ISBN: 1-58113-183-6. DOI: 10.1145/343477.343502. URL: http://doi.acm.org/10.1145/343477.343502.

[89] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

[90] E. Brewer. “CAP twelve years later: How the "rules" have changed”. In: *Computer* 45.2 (2012), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37.

[91] Newer SQL. “Errors in Database Systems , Eventual Consistency , and the CAP Theorem”. In:

[92] D. J. Abadi. “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story”. In: *Computer* 45 (Jan. 2012), pp. 37–42. ISSN: 0018-9162. DOI: 10.1109/MC.2012.33. URL: doi.ieeecomputersociety.org/10.1109/MC.2012.33.

[93] Jim Gray. “The Transaction Concept: Virtues and Limitations (Invited Paper)”. In: *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. VLDB ’81. Cannes, France: VLDB Endowment, 1981, pp. 144–154. URL: http://dl.acm.org/citation.cfm?id=1286831.1286846.

[94] Theo Haerder and Andreas Reuter. “Principles of Transaction-oriented Database Recovery”. In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: http://doi.acm.org/10.1145/289.291.

[95] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[96] David Bermbach and Jörn Kuhlenkamp. “Consistency in distributed storage systems”. In: *Networked Systems*. Springer, 2013, pp. 175–189.

[97] Haifeng Yu and Amin Vahdat. “Design and evaluation of a conit-based continuous consistency model for replicated services”. In: *ACM Transactions on Computer Systems (TOCS)* 20.3 (2002), pp. 239–282.

[98] Francisco J Torres-Rojas and Esteban Meneses. “Convergence through a weak consistency model: Timed causal consistency”. In: *CLEI electronic journal* 8.2 (2005).

[99] David Bermbach and Stefan Tai. “Eventual consistency: How soon is eventual? An evaluation of Amazon S3’s consistency behavior”. In: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM. 2011, p. 1.

[100] Eric Anderson et al. “What consistency does your key-value store actually provide?” In: *HotDep*. Vol. 10. 2010, pp. 1–16.

[101] Wojciech Golab, Xiaozhou Li, and Mehul A Shah. “Analyzing consistency properties for fun and profit”. In: *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM. 2011, pp. 197–206.

[102] Swapnil Patil et al. “YCSB++: benchmarking and performance debugging advanced features in scalable table stores”. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 9.

[103] Muntasir Raihan Rahman et al. “Toward a Principled Framework for Benchmarking Consistency.” In: *HotDep*. 2012.

[104] Hiroshi Wada et al. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective." In: *CIDR*. Vol. 11. 2011, pp. 134–143.

[105] Kamal Zellag and Bettina Kemme. "How consistent is your cloud application?" In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 6.

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

[107] Leslie Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[108] Leslie Lamport. "Paxos Made Simple". In: (2001), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[109] L. Lamport and M. Massa. "Cheap Paxos". In: *International Conference on Dependable Systems and Networks, 2004*. 2004, pp. 307–314. DOI: 10.1109/DSN.2004.1311900.

[110] Robbert Van Renesse and Deniz Altinbuken. "Paxos Made Moderately Complex". In: *ACM Comput. Surv.* 47.3 (Feb. 2015), 42:1–42:36. ISSN: 0360-0300. DOI: 10.1145/2673577. URL: http://doi.acm.org/10.1145/2673577.

[111] Roberto De Prisco, Butler Lampson, and Nancy Lynch. "Revisiting the Paxos algorithm". In: *Distributed Algorithms*. Ed. by Marios Mavronicolas and Philippas Tsigas. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 111–125. ISBN: 978-3-540-69600-1.

[112] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 305–319.

[113] Kyle Kingsbury. *Jepsen: etcd and Consul*. URL: https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul.

[114] etcd io. *etcd-io/etcd*. 2018. URL: https://github.com/etcd-io/etcd.

[115] Colin J. Fidge. "Timestamps in message-passing systems that preserve partial ordering". In: 10 (Feb. 1988), pp. 56–66.

[116] Friedemann Mattern. "Virtual Time and Global States of Distributed Systems". In: *PARALLEL AND DISTRIBUTED ALGORITHMS*. North-Holland, 1988, pp. 215–226.

[117] *GlusterFS*. 2011. URL: https://redhatstorage.redhat.com/products/glusterfs/.

[118] V Srinivasan et al. "Aerospike: architecture of a real-time operational DBMS". In: *Proceedings of the VLDB Endowment* 9.13 (2016), pp. 1389–1400.

[119] Rusty Klophaus. "Riak core: Building distributed applications without shared state". In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM. 2010, p. 14.

[120] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. Boston, MA, USA. 2010.

[121] *Announcing The Ceph Foundation*. 2018. URL: https://ceph.com/community/announce-the-ceph-foundation/.

[122] Sorin Faibish et al. *Lustre file system*. US Patent 9,779,108. 2017.

[123] K. R. Krish, A. Anwar, and A. R. Butt. "hatS: A Heterogeneity-Aware Tiered Storage for Hadoop". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2014, pp. 502–511. DOI: 10.1109/CCGrid.2014.51.

[124] *OpenIO*. URL: https://www.openio.io/wp-content/uploads/2016/12/OpenIO-CoreSolutionDescription.pdf.

[125] *NVFUSE: a NVME user-space filesystem*. URL: https://github.com/nvfuse/nvfuse.

[126] Ilias Marinos et al. "Disk, Crypt, Net: Rethinking the Stack for High-performance Video Streaming". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 211–224. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098844. URL: http://doi.acm.org/10.1145/3098822.3098844.

[127] Marcelo Bagnulo Braun and Jon Crowcroft. *SNA: Sourceless Network Architecture*. Tech. rep. University of Cambridge, Computer Laboratory, 2014.

[128] Glenn Deen et al. *Using Media Encoding Networks to address MPEG-DASH video*. Internet-Draft draft-deen-naik-ggie-men-mpeg-dash-00. Work in Progress. Internet Engineering Task Force, July 2016. 8 pp. URL: https://datatracker.ietf.org/doc/html/draft-deen-naik-ggie-men-mpeg-dash-00.

[129] Suman Srinivasan and Henning Schulzrinne. "IPv6 Addresses as Content Names in Information-Centric Networking". In: ().

[130] Bengt Ahlgren et al. "A survey of information-centric networking". In: *IEEE Communications Magazine* 50.7 (2012).

[131] Luca Muscariello et al. *Hybrid Information-Centric Networking*. Internet-Draft draft-muscariello-intarea-hicn-00. Work in Progress. Internet Engineering Task Force, June 2018. 21 pp. URL: https://datatracker.ietf.org/doc/html/draft-muscariello-intarea-hicn-00.

[132] G Carofiglio. *Mobile video delivery with Hybrid ICN*. Tech. rep. Cisco, Tech. Rep, 2016.

[133] T Bonald, S Oueslati-Boulahia, and J Roberts. "IP traffic and QoS control: the need for a flow-aware architecture". In: *World Telecommunications Congress*. Citeseer. 2002.

[134] Weibin Zhao, David Olshefski, and Henning Schulzrinne. "Internet quality of service: An overview". In: *Columbia University, New York, New York, Technical Report CUCS-003-00* (2000).

[135] Alexandros G Dimakis et al. "A survey on network codes for distributed storage". In: *Proceedings of the IEEE* 99.3 (2011), pp. 476–489.

[136] Daniel Ford et al. "Availability in Globally Distributed Storage Systems." In: *Osdi*. Vol. 10. 2010, pp. 1–7.

[137] Tian Luo et al. "hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems". In: *Proc. VLDB Endow.* 5.10 (June 2012), pp. 1076–1087. ISSN: 2150-8097. DOI: 10.14778/2336664.2336679. URL: http://dx.doi.org/10.14778/2336664.2336679.

[138] S. Kaneko et al. "A Guideline for Data Placement in Heterogeneous Distributed Storage Systems". In: *2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*. 2016, pp. 942–945. DOI: 10.1109/IIAI-AAI.2016.162.

[139] Y. Qin et al. "Data placement strategy in data center distributed storage systems". In: *2016 IEEE International Conference on Communication Systems (ICCS)*. 2016, pp. 1–6. DOI: 10.1109/ICCS.2016.7833566.

[140] C. F. Wu et al. "File placement mechanisms for improving write throughputs of cloud storage services based on Ceph and HDFS". In: *2017 International Conference on Applied System Innovation (ICASI)*. 2017, pp. 1725–1728. DOI: 10.1109/ICASI.2017.7988272.

[141] Benoit Claise et al. "Ipfix protocol specification". In: *Interrnet-draft, work in progress* (2005).

[142] Wikipedia. *NetFlow — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=NetFlow&oldid=847024174. [Online; accessed 27-June-2018]. 2018.

[143] Mohammed HAWARI. *TCP Fast Open Cookie for IPv6 prefixes*. Internet-Draft draft-hawari-tcpm-tfo-ipv6-prefixes-00. Work in Progress. Internet Engineering Task Force, July 2015. 5 pp. URL: https://datatracker.ietf.org/doc/html/draft-hawari-tcpm-tfo-ipv6-prefixes-00.

[144] Sacha Goedegebure. *Big Buck Bunny*. URL: https://peach.blender.org/.

[145] Tim Baumann. *Valkaama project*. URL: http://www.valkaama.com/.

[146] Will Reese. "Nginx: the high-performance web server and reverse proxy". In: *Linux Journal* 2008.173 (2008), p. 2.

[147] *wrk - a HTTP benchmarking tool*. URL: https://github.com/wg/wrk.

[148] *What is VPP?* https://wiki.fd.io/view/VPP/What_is_VPP%3F.

[149] C. Filsfils et al. "The Segment Routing Architecture". In: *2015 IEEE Global Communications Conference (GLOBECOM)*. 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2015.7417124.

[150] *Segment Routing*. URL: http://www.segment-routing.net/.

[151] IJsbrand Wijnands et al. *Multicast Using Bit Index Explicit Replication (BIER)*. RFC 8279. Nov. 2017. DOI: 10.17487/RFC8279. URL: https://rfc-editor.org/rfc/rfc8279.txt.

[152] Guillaume Ruty, Andre Surcouf, and Jean-Louis Rougier. "6Stor: A Scalable and IPv6-Centric Distributed Object Storage System". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 2017. URL: https://www.usenix.org/conference/fast17/poster-sessions.

[153] G. Ruty, A. Surcouf, and J. L. Rougier. "Collapsing the layers: 6Stor, a scalable and IPv6-centric distributed storage system". In: *2017 Fourth International Conference on Software Defined Systems (SDS)*. 2017, pp. 81–86. DOI: 10.1109/SDS.2017.7939145.

[154] Guillaume Ruty et al. "An initial evaluation of 6Stor, a dynamically scalable IPv6-centric distributed object storage system". In: *Cluster Computing* (2019). ISSN: 1573-7543. DOI: 10.1007/s10586-018-02897-8. URL: https://doi.org/10.1007/s10586-018-02897-8.

[155] Andre Surcouf, Guillaume Ruty, and William Mark Townsley. *Distributed object storage*. US Patent App. 15/408,129. 2018.

[156] Andre Surcouf et al. *Delivering content over a network*. 2016.

[157] Eric A Brewer. "Kubernetes and the path to cloud native". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM. 2015, pp. 167–167.

[158] *Object Storage for Capacity-Intensive Workloads, Exabyte Scale*. URL: https://cloudian.com/.

[159] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. *The MDS queue: Analysing latency performance of codes and redundant requests*. Tech. rep. Technical Report, 2013.

[160] Gauri Joshi, Yanpei Liu, and Emina Soljanin. "On the delay-storage trade-off in content download from coded distributed storage systems". In: *IEEE Journal on Selected Areas in Communications* 32.5 (2014), pp. 989–997.

[161] David Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. ACM Press. DOI: 10.1145/258533.258660.

[162] David Shue, Michael J Freedman, and Anees Shaikh. "Performance Isolation and Fairness for Multi-Tenant Cloud Storage." In: *OSDI*. Vol. 12. Hollywood, CA. 2012, pp. 349–362.

[163] Yoann Desmouceaux et al. "6LB: Scalable and Application-Aware Load Balancing with Segment Routing". In: *IEEE/ACM Transactions on Networking* 26.2 (2018), pp. 819–834.

[164] Clarence Filsfils et al. *Segment Routing Architecture*. RFC 8402. July 2018. DOI: 10.17487/RFC8402. URL: https://rfc-editor.org/rfc/rfc8402.txt.

[165] Guillaume Ruty et al. *Reducing distributed storage operation latency using segment routing techniques*. 2018.

[166] *Apache Traffic Server - Overview*. URL: http://trafficserver.apache.org/.

[167] *Learn - Akamai Documentation*. URL: https://learn.akamai.com/en-us/products/media_delivery/index.html.

[168] AWS CloudFront. "Amazon cloudfront". In: *URL: http://aws. amazon. com/cloudfront* (2014).

[169] Marshall Copeland et al. "Microsoft azure and cloud computing". In: *Microsoft Azure*. Springer, 2015, pp. 3–26.

[170] Kerrie Meyler et al. *Microsoft Hybrid Cloud Unleashed with Azure Stack and Azure*. Sams Publishing, 2017.

[171] Sara Davis. *The Era Of Hyperconvergence: Simplifying Your Data Center Operations*. 2017. URL: https://www.forbes.com/sites/forbesagencycouncil/2017/09/08/the-era-of-hyperconvergence-simplifying-your-data-center-operations/#339f5a612a01.

[172] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. "On the performance variability of production cloud services". In: *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE. 2011, pp. 104–113.

[173] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[174] Alan Demers, Srinivasan Keshav, and Scott Shenker. "Analysis and simulation of a fair queueing algorithm". In: *ACM SIGCOMM Computer Communication Review*. Vol. 19. 4. ACM. 1989, pp. 1–12.

[175] Sally Floyd and Van Jacobson. "Link-sharing and resource management models for packet networks". In: *IEEE/ACM transactions on Networking* 3.4 (1995), pp. 365–386.

[176] Pawan Goyal, Harrick M Vin, and Haichen Chen. "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks". In: *ACM SIGCOMM Computer Communication Review*. Vol. 26. 4. ACM. 1996, pp. 157–168.

[177] Ali Ghodsi et al. "Multi-resource fair queueing for packet processing". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 1–12.

[178] Sitaram Iyer and Peter Druschel. "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O". In: *ACM SIGOPS Operating Systems Review*. Vol. 35. 5. ACM. 2001, pp. 117–130.

[179] Christopher R Lumb, Arif Merchant, and Guillermo A Alvarez. "Façade: Virtual Storage Devices with Performance Guarantees." In: *FAST*. Vol. 3. 2003, pp. 131–144.

[180] Ajay Gulati, Arif Merchant, and Peter J Varman. "pClock: an arrival curve based approach for QoS guarantees in shared storage systems". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 35. 1. ACM. 2007, pp. 13–24.

[181] Robert Abbott and Hector Garcia-Molina. *Scheduling real-time transactions with disk resident data*. Princeton University. Department of Computer Science, 1989.

[182] Anna Povzner et al. "Efficient guaranteed disk request scheduling with fahrrad". In: *ACM SIGOPS Operating Systems Review*. Vol. 42. 4. ACM. 2008, pp. 13–25.

[183] AL Reddy and Jim Wyllie. "Disk scheduling in a multimedia I/O system". In: *Proceedings of the first ACM international conference on Multimedia*. ACM. 1993, pp. 225–233.

[184] Wei Jin, Jeffrey S Chase, and Jasleen Kaur. "Interposed proportional sharing for a storage service utility". In: *ACM SIGMETRICS Performance Evaluation Review* 32.1 (2004), pp. 37–48.

[185] John Bruno et al. "Disk scheduling with quality of service guarantees". In: *Multimedia Computing and Systems, 1999. IEEE International Conference on*. Vol. 2. IEEE. 1999, pp. 400–405.

[186] Matthew Wachs et al. "Argon: Performance Insulation for Shared Storage Servers." In: *FAST*. Vol. 7. 2007, pp. 5–5.

[187] Jens Axboe. "Linux block IO—present and future". In: *Ottawa Linux Symp*. 2004, pp. 51–61.

[188] Stan Park and Kai Shen. "FIOS: a fair, efficient flash I/O scheduler." In: *FAST*. 2012, p. 13.

[189] Kai Shen and Stan Park. "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs." In: *USENIX Annual Technical Conference*. 2013, pp. 67–78.

[190] Matias Bjørling et al. "Linux block IO: introducing multi-queue SSD access on multi-core systems". In: *Proceedings of the 6th international systems and storage conference*. ACM. 2013, p. 22.

[191] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. "NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs." In: *HotStorage*. 2016.

[192] Byunghei Jun and Dongkun Shin. "Workload-aware budget compensation scheduling for NVMe solid state drives". In: *Non-Volatile Memory System and Applications Symposium (NVMSA), 2015 IEEE*. IEEE. 2015, pp. 1–6.

[193] Ajay Gulati, Arif Merchant, and Peter J Varman. "mClock: handling throughput variability for hypervisor IO scheduling". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association. 2010, pp. 437–450.

[194] W. Wang, B. Li, and B. Liang. "Dominant resource fairness in cloud computing systems with heterogeneous servers". In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 2014, pp. 583–591. DOI: 10.1109/INFOCOM.2014.6847983.

[195] Eno Thereska et al. "IOFlow: A Software-defined Storage Architecture". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 182–196. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522723. URL: http://doi.acm.org/10.1145/2517349.2522723.

[196] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*. Vol. 2. Prentice-Hall International New Jersey, 1992.

[197] Fengguang Wu et al. "Linux readahead: less tricks for more". In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 273–284.

[198] C. Ruemmler and J. Wilkes. "An introduction to disk drive modeling". In: *Computer* 27.3 (1994), pp. 17–28. ISSN: 0018-9162. DOI: 10.1109/2.268881.

[199] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. "A detailed simulation model of the HP 97560 disk drive". In: (1994).

[200] Keith Muller and Joseph Pasquale. "A high performance multi-structured file system design". In: *ACM SIGOPS Operating Systems Review*. Vol. 25. 5. ACM. 1991, pp. 56–67.

[201] Chandramohan A Thekkath, John Wilkes, and Edward D Lazowska. "Techniques for file system simulation". In: *Software: Practice and Experience* 24.11 (1994), pp. 981–999.

[202] Greg Ganger, B Worthington, and Y Patt. "The DiskSim simulation environment (v4. 0)". In: *Parallel Data Lab, http://www. pdl. cmu. edu/DiskSim/Online-document* (2009).

[203] Vijayan Prabhakaran and Ted Wobber. "SSD extension for DiskSim simulation environment". In: *Microsoft Reseach* (2009).

[204] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. "PARDA: Proportional Allocation of Resources for Distributed Storage Access." In: *FAST*. Vol. 9. 2009, pp. 85–98.

[205] Shimin Chen. "FlashLogging: exploiting flash devices for synchronous logging performance". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, pp. 73–86.

[206] Guillaume Ruty, Jerome Tollet, and Aloys Augustin. "FAIR SCHEDULING FOR LOW LATENCY AND HIGH THROUGHPUT STORAGE SYSTEMS". In: (2018). URL: https://www.tdcommons.org/dpubs_series/1427.

[207] Atul Adya et al. "Fragment reconstruction: Providing global cache coherence in a transactional storage system". In: *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*. IEEE. 1997, pp. 2–11.

[208] Miguel Castro et al. "HAC: Hybrid adaptive caching for distributed storage systems". In: *ACM SIGOPS Operating Systems Review*. Vol. 31. 5. ACM. 1997, pp. 102–115.

[209] Yu Xiang et al. "Joint latency and cost optimization for erasure-coded data center storage". In: *IEEE/ACM Transactions on Networking (TON)* 24.4 (2016), pp. 2443–2457.

[210] Longbo Huang et al. "Codes can reduce queueing delay in data centers". In: *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 2766–2770.

[211] Shripad J Nadgowda et al. "C2P: Co-operative Caching in Distributed Storage Systems". In: *International Conference on Service-Oriented Computing*. Springer. 2014, pp. 214–229.

[212] Raluca Halalai et al. "Agar: A caching system for erasure-coded data". In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 23–33.

[213] Tianqiong Luo, Vaneet Aggarwal, and Borja Peleato. "Coded caching with distributed storage". In: *arXiv preprint arXiv:1611.06591* (2016).

[214] Vaneet Aggarwal et al. "Sprout: A functional caching approach to minimize service latency in erasure-coded storage". In: *IEEE/ACM Transactions on Networking* 25.6 (2017), pp. 3683–3694.

[215] KV Rashmi et al. "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding." In: *OSDI*. 2016, pp. 401–417.

[216] Lee Breslau et al. "Web caching and Zipf-like distributions: Evidence and implications". In: *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 1. IEEE. 1999, pp. 126–134.

[217] Bernardo A Huberman et al. "Strong regularities in world wide web surfing". In: *Science* 280.5360 (1998), pp. 95–97.

[218] Lada A Adamic and Bernardo A Huberman. "Zipf's law and the Internet." In: *Glottometrics* 3.1 (2002), pp. 143–150.

[219] Mark E Crovella, Murad S Taqqu, and Azer Bestavros. "Heavy-tailed probability distributions in the World Wide Web". In: *A practical guide to heavy tails* 1 (1998), pp. 3–26.

[220] Ganesh Ananthanarayanan et al. "Scarlett: coping with skewed content popularity in mapreduce clusters". In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 287–300.

[221] Qi Huang et al. "An analysis of Facebook photo caching". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 167–181.

[222] Andre Surcouf et al. *Hybrid distributed storage system to dynamically modify storage overhead and improve access performance*. 2018.

[223] Tyler Harter et al. "Slacker: Fast Distribution with Lazy Docker Containers." In: *FAST*. Vol. 16. 2016, pp. 181–195.

[224] Guillaume Ruty et al. *Predictive container image storage system for fast container execution*. 2017.

[225] DongJin Lee, Brian E Carpenter, and Nevil Brownlee. "Media streaming observations: Trends in udp to tcp ratio". In: *International Journal on Advances in Systems and Measurements* 3.3-4 (2010).

[226] Ilias Marinos, Robert NM Watson, and Mark Handley. "Network stack specialization for performance". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 175–186.

[227] EunYoung Jeong et al. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems." In: *NSDI*. Vol. 14. 2014, pp. 489–502.

[228] Oliver Spatscheck et al. "Optimizing TCP forwarder performance". In: *IEEE/ACM Transactions on Networking (TON)* 8.2 (2000), pp. 146–157.

[229] R Rajesh, Kannan Babu Ramia, and Muralidhar Kulkarni. "Integration of LwIP Stack over Intel (R) DPDK for High Throughput Packet Delivery to Applications". In: *Electronic System Design (ISED), 2014 Fifth International Symposium on*. IEEE. 2014, pp. 130–134.

[230] *Home*. URL: https://www.dpdk.org/.

[231] David Barach et al. "High-Speed Software Data Plane via Vectorized Packet Processing". In: *IEEE Communications Magazine* (2018).

[232] Pierre Ppfister et al. *Predictive container image storage system for fast container execution*. 2017.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Vers un meilleur passage à l'échelle et une plus grande flexibilité pour les systèmes de stockage distribué

Mots clés : Stockage Distribué, Datacenters, Cloud

Résumé : Les besoins en terme de stockage, en augmentation exponentielle, sont difficilement satisfaits par les systèmes de stockage distribué traditionnels. Même si les performances des disques continuent à s'améliorer, les systèmes de stockage distribué actuels peinent à suivre le croissance du nombre de données requérant d'êtres stockées, notamment à cause de l'avènement des applications de big data. Par ailleurs, l'équilibre de performances entre disques, cartes réseau et processeurs a changé et les suppositions sur lesquelles se basent la plupart des systèmes de stockage distribué actuels ne sont plus vraies.

Cette dissertation explique de quelle manière certains aspects de tels systèmes de stockages peuvent être modifiés et repensés pour faire une utilisation plus efficace des ressources qui les composent. Elle présente 6Stor, une architecture de stockage nouvelle qui se base sur une couche de métadonnées distribuée afin de fournir du stockage d'objet de manière flexible tout en passant à l'échelle. Elle détaille ensuite un algorithme d'ordonnancement des requêtes permettant à un système de stockage générique de traiter les requêtes de clients en parallèle de manière plus équitable. Enfin, elle décrit comment améliorer le cache générique du système de fichier dans le contexte de systèmes de stockage distribué basés sur des codes correcteurs avant de présenter des contributions effectuées dans le cadre de courts projets de recherche.

Title : Towards more scalability and flexibility for distributed storage systems

Keywords : Distributed Storage, Datacenters, Cloud

Abstract : The exponentially growing demand for storage puts a huge stress on traditionnal distributed storage systems. While storage devices' performance keep improving over time, current ditributed storage systems struggle to keep up with the rate of data growth, especially with the rise of cloud and big data applications. Furthermore, the performance balance between storage, network and compute devices has shifted and the assumptions that are the foundation for most distributed storage systems are not true anymore.

This dissertation explains how several aspects of such storage systems can be modified and rethought to make a more efficient use of the resource at their disposal. It presents 6Stor, an original architecture that uses a distributed layer of metadata to provide flexible and scalable object-level storage, then proposes a scheduling algorithm improving how a generic storage system handles concurrent requests. Finally, it describes how to improve legacy filesystem-level caching for erasure-code-based distributed storage systems, before presenting a few other contributions made in the context of short research projects.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France