INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Distributed Deduplication in a Cloud-based Object Storage System

Paulo Ricardo Motta Gomes

Dissertation submitted in partial fulfillment of the requirements for the European Master in Distributed Computing

Júri

Head:	Prof. Dr. Luís Eduardo Teixeira Rodrigues
Advisor:	Prof. Dr. João Coelho Garcia
Examiner:	Prof. Dr. Johan Montelius (KTH)

July 30, 2012

Acknowledgements

I begin these acknowledgements by expressing my gratitude to my advisor, Prof. João Garcia, that gave me freedom to pursue my own ideas and directions, while also providing me valuable feedback and guidance throughout this work.

I would also like to thank members of INESC for the availability and support with the clusters during the evaluation phase of this work.

Furthermore, I want to thank all my friends from EMDC for the help during technical discussions and funny times spent together during the course of this master's program.

Last but not least, I would like to thank my family for their immense support and encouragement: to my wife Liana, who always accompanied and supported me with patience and unconditional love; to my parents and my sister, without whom none of this would have been possible; and finally, to Liana's parents, who greatly supported me throughout this journey.

Lisboa, July 30, 2012

Paulo Ricardo Motta Gomes

A minha companheira,

Liana.

European Master in Distributed Computing EMDC

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint master's program between Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Commission via the Erasmus Mundus program.

The track of the author in this master's program has been the follows:

First and second semester of studies: **IST**

Third semester of studies: **KTH**

Fourth semester of studies (thesis): **IST**

Abstract

An increasing amount of data is being stored in cloud-based storage services and this trend is expected to grow in the coming years. This creates a huge demand for systems and algorithms that are more efficient in the use of storage resources while being able to meet necessary cloud requirements of availability and scalability. A major source of inefficiency on cloud storage systems is data duplication, since bandwidth and storage resources are wasted to transfer and store data that is already present in the system.

This thesis addresses the problem of efficiently enabling deduplication in a cloud storage system. In this context, we propose a generic architecture for a distributed object-based storage system with built-in support for data deduplication. This architecture uses a synthesis of concepts and techniques from several areas ranging from distributed systems to content-addressable storage to enable global distributed deduplication while maintaining the system's scalability, availability and reliability, critical requirements for cloud storage.

The proposed architecture was implemented as an extension to OpenStack Object Storage, an open source collaboration for cloud-based object storage. The prototype implementation was evaluated and compared against a similar system without deduplication support in different scenarios and workloads.

Keywords

Cloud Computing

Content-Addressable Storage

Data Deduplication

Distributed Storage Systems

Object-based Storage Systems

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AFS Andrew File System

ATA AT Attachment

API(s) Application Programming Interface(s)

CAS Content-Addressable Storage

DHT(s) Distributed hash table(s)

EC2 (Amazon) Elastic Compute Cloud

GFS Google File System

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IT Information Technology

JSON JavaScript Object Notation

LAN Local Area Network

NAS Network-attached storage

NFS Network File System

OSD Object-based Storage Device

PaaS Platform as a Service

REST Representational State Transfer

RPC Remote Procedure Call

S3 (Amazon) Simple Storage Service

SaaS Software as a Service

SCSI Small Computer System Interface

SQL Standard Query Language

URL Uniform resource locator

VM(s) Virtual Machine(s)

WAN Wide Area Network

WSGI Web Server Gateway Interface

XML Extensible Markup Language

1 Introduction

The widespread adoption of information technology in a world connected by the Internet is leading to a massive growth in the amount of data produced. Studies predict that the total amount of data will reach 35 ZB (10^{21} or 1 sextillion bytes) by 2020 and a significant fraction of this data is expected to live in the cloud (Gantz & Reinsel 2010; Woo 2010; Gantz & Reinsel 2011). The popularity of services like DropBox, Amazon S3, Google Cloud Storage and Windows Azure Storage, which allow users and organizations to store data in the cloud confirms this trend of moving storage from local devices to online providers.

In case data growth forecasts are confirmed, the amount of data produced will exceed the amount of digital storage available (Gantz & Reinsel 2010). This creates a huge demand for systems and algorithms that are more efficient in the use of storage resources. In the context of cloud storage, this demand must be met in addition to typical cloud requirements of scalability and availability.

A major source of storage inefficiency is data duplication. Gantz et al. (Gantz & Reinsel 2010) claims that up to 75% of all produced data is duplicated. Data deduplication is a technique that optimizes the use of storage by eliminating unnecessary redundant data. This technique has been successfully used in backup and archival storage systems[1], and its use is becoming popular for primary storage[2]. Besides improving storage utilization, deduplication can also improve throughput, latency and bandwidth utilization, since transfer of duplicated data over the network can be avoided.

In this work, we investigate how deduplication can be used in a distributed storage system to improve storage efficiency while attending cloud computing requirements.

[1] (Cox, Murray, & Noble 2002; Quinlan & Dorward 2002; EMC 2004; You, Pollack, & Long 2005; Rhea, Cox, & Pesterev 2008)

[2] (IBM Corporation 2002; EMC 2010; Bonwick 2009; Srinivasan, Bisson, Goodson, & Voruganti 2012; S3QL 2012; OpenDedup 2012)

1.1 Motivation

Cloud Computing has established itself as a solid paradigm to deliver the computing as a utility vision (Buyya, Yeo, Venugopal, Broberg, & Brandic 2009). This paradigm enables the provision of highly available, scalable and reliable on-demand services over the Internet (Armbrust, Fox, Griffith, Joseph, Katz, Konwinski, Lee, Patterson, Rabkin, & Zaharia 2009).

One of the key services offered by cloud providers is object-based storage (Amazon 2012), where customers can store and retrieve arbitrary objects using a flat identifier through a simple put/get interface. These services give clients the illusion of an unlimited storage pool that can be accessed from anywhere.

An object can contain any type of unstructured binary content, including documents, media files, virtual machine images, etc. In a cloud storage scenario, a popular object (such as song or a video) will likely be stored multiple times with different identifiers or in multiple accounts. For instance, measurements of 550 desktop file systems at Microsoft have detected that almost half of all files in these systems were duplicated (Bolosky, Douceur, Ely, & Theimer 2000).

It is also possible for different objects to share duplicated binary data among them. A study by Jin et al (Jin & Miller 2009) found that from 40% to 80% of binary data in several UNIX virtual machine images are duplicated. This pattern of duplicated data becomes even more common when dealing with an ever-growing amount of data from different users and organizations.

One way of optimizing storage utilization is by leveraging data deduplication, a technique that identifies and eliminates redundant data in a storage system. Furthermore, this technique can also improve bandwidth utilization in a distributed system, since duplicated data need not even be transferred over the network.

1.2 Objective

The goal of this thesis is to demonstrate that deduplication can be used in distributed object-based storage systems to improve storage efficiency while maintaining the system's scalability, availability and reliability, critical requirements for cloud computing.

1.3 Contributions

This work addresses the problem of efficiently enabling deduplication in object-based storage systems. More precisely, the thesis analyzes, implements and evaluates techniques to support data deduplication in a distributed storage environment with focus in the cloud computing storage scenario, where scalability, availability, and reliability are primary concerns. As a result, the thesis makes the following contribution:

- It proposes a scalable architecture for a distributed object-based storage system with built-in support for global deduplication, allowing cloud storage providers and consumers to benefit from storage space and bandwidth savings enabled by deduplication.

1.4 Results

The results produced by this thesis can be enumerated as follows:

- A prototype of the proposed solution was implemented as an extension to an important cloud-based distributed object storage platform: OpenStack Object Storage.
- An experimental evaluation of the implemented prototype, identifying the benefits and trade-offs of using deduplication in an object-based storage system with real and synthetic workloads.

1.5 Thesis overview

The thesis is organized as follows. Chapter 2 provides an introduction to the different technical areas related to this work. Chapter 3 describes the architecture and implementation of the proposed distributed object-based storage system with deduplication support. The results of the experimental evaluation study are presented in Chapter 4. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

2.1 Introduction

In this chapter we introduce and discuss fundamental concepts and techniques that are relevant for understanding the problem addressed in this thesis. We also present and discuss systems and solutions that solve similar problems or that served as inspiration for designing the solution proposed in this thesis for data deduplication in a cloud-based object storage system.

In section 2.2 we discuss the paradigm of Cloud Computing, in which the proposed solution is contained. A categorization of deduplication solutions is presented in section 2.4, introducing a taxonomy that will be used throughout the thesis and presenting relevant systems that employ deduplication. The mechanism of content-addressable storage, one of the key techniques employed in this thesis, as well as relevant systems that are based on this method are presented in section 2.5. Consistent hashing and distributed hash tables, two important technologies that were also leveraged in the proposed solution, are explored in section 2.6. Finally, in section 2.7 we provide an extensive account of existing distributed storage solutions and discuss their advantages and limitations.

2.2 Cloud Computing

Cloud Computing is a distributed computing paradigm that enables the provision of available, scalable and reliable on-demand resources over a network, primarily the Internet (Foster, Zhao, Raicu, & Lu 2008; Mell & Grance 2009). These resources can be of any type, including, but not limited to: processing power, storage, networks, platforms, applications, services, etc. They compose a shared, potentially virtualized and abstract pool of resources and are provisioned over well-defined and convenient protocols, with minimal management effort by the customer. The term “cloud” refers to the abstraction represented by the paradigm, where customers are

unaware of *how* and *where* the resources are being provisioned.

2.2.1 Service Models

The cloud computing paradigm combines several types of service offerings, which can be classified according to the level of abstraction and resource control provided by them. A widely accepted classification of cloud computing services, adopted by the United States National Institute of Standards and Technology (NIST) (Foster, Zhao, Raicu, & Lu 2008), is presented below:

Software as a Service (SaaS) This category delivers complete software applications to the end user. These applications run on the provider's or a third party's infrastructure and are accessed over a web browser or an user interface. For this reason this model provides the higher level of abstraction in comparison to other service models. The consumer has little to no control over the underlying resources needed to run the application. Examples of SaaS offerings include: Google Docs, Dropbox and Salesforce.com.

Platform as a Service (PaaS) This layer provides a software platform that allows third-party applications to be deployed on the cloud provider's infrastructure. This platform comprises programming languages, libraries and application programming interfaces (APIs) which are supported by the service provider. Some level of control is given to client applications, but this is limited to configuration settings supported by the application-hosting environment, which generally manages the underlying physical resources. Examples of PaaS providers include: Google App Engine, Microsoft Azure and Heroku.

Infrastructure as a Service (IaaS) This is the service model with the lowest level of abstraction and highest level of resource control. It provisions fundamental computing resources, such as processing, storage or networks where customers can run arbitrary software, including operating systems or custom applications. The physical resources are typically shared by multiple users, either by a virtualization hypervisor or a management middleware. Customers have control over operating systems, storage and deployed applications, but do not control the underlying physical infrastructure that runs the service. Examples of IaaS providers include: RackSpace and Amazon Elastic Compute Cloud (Amazon EC2).

The solution addressed in this thesis, cloud-based object storage, is classified as IaaS, since it deals with provisioning of a fundamental computing resource (storage). However, it is also possible to access an object storage service from more abstract layers (PaaS or SaaS). For instance, DropBox (DropBox 2012), which is a SaaS offering, stores user's files on Amazon's S3 (Google 2012), which is a IaaS offering.

2.2.2 Deployment Models

Another classification of Cloud Computing is related to the deployment model employed by the cloud infrastructure. This classification was also defined by NIST and includes the following categories:

Public Cloud The cloud is made available to the general public in a pay-as-you-go manner. It is generally owned and managed by the provider, but in some cases it may be owned and operated by a third party.

Private Cloud The cloud is exclusively available to a single organization. It may be owned and managed by the organization or by a third party.

Community Cloud The cloud is made available to a community of organizations with common interests, such as a research collaboration or a university campus. It may be owned and managed by the organizations composing the community or by a third party.

Hybrid Cloud The cloud uses a composition of the previous cloud deployment models, with a special software that provides interoperability and portability between different types of clouds.

A cloud object storage service can be made available publicly or privately. Public object storage services are typically proprietary solutions, such as Amazon S3 (Amazon 2012) and Google Cloud Storage (Google 2012). A private object storage service can be deployed with an open source object storage middleware, such as Eucalyptus Walrus (Nurmi, Wolski, Grzegorczyk, Obertelli, Soman, Youseff, & Zagorodnov 2009) or OpenStack Object Storage (OpenStack 2012). Even though we do not explicitly make a distinction between these categories, the proposed solution is more suitable to a private cloud context, since it does not handle security and authentication in its current state.

2.2.3 Cloud Storage

A primary service offered by IaaS cloud providers is persistent storage, which can be of several types according to applications' needs and storage requirements. The most important categories of cloud storage offerings are summarized below:

Block Storage Provides block storage capabilities through a low level computer bus interface, such as SCSI or ATA, over the network. An operating system using cloud-based block storage will operate on the block volume just like it would in a normal disk. This type of storage is indicated for persisting volumes on virtual machines hosted on IaaS clouds. Example: Amazon Elastic Block Store (EBS).

Object Storage Provides object storage capabilities through a simple get/put interface generally through a HTTP/REST protocol. This enables applications to store and retrieve objects from within the user-space, without the need to install and configure a file system. This type of storage is specially suitable for storing static/unstructured data, such as media files or virtual machines images. Example: Amazon Simple Storage Service (S3), Google Cloud Storage, Rackspace Cloud Files.

Database Storage Provides database storage capability, generally to a non-relational database (NoSQL), through a simple query interface. This enables applications to use a database without the need for configuring a local database and dealing with database administration. This type of storage is useful for storing structured content or application's metadata, which is typically small in size. Example: Amazon DynamoDB, Google App Engine Datastore.

This thesis addresses data deduplication in a cloud-based **object storage** system. A more detailed overview on object-based storage systems as well as relevant examples of this type of system will be presented in the next section.

2.3 Object-based Storage

Persistent binary storage has traditionally revolved around file systems that access storage devices through a block interface, such as SCSI or ATA/IDE. A Storage Area Network (SAN)

improves the scalability and throughput of file systems by allowing remote access to clusters of block-based devices, where SCSI commands are issued via IP (iSCSI (Krueger & Haagens 2002)) or fibre channel. However, the mapping of file and directory data structures to blocks on these devices still requires metadata to be kept by a file system. This limits opportunities for block sharing between multiple hosts in SAN architectures because these hosts also need to share metadata in a consistent manner, which becomes a complex problem, because they need to agree on metadata representation, space allocation policies, etc.

One way of improving file system capacity and data sharing is by allowing remote access to files through a distributed file system, such as NFS (Sandberg, Goldberg, Kleiman, Walsh, & Lyon 1985) or AFS (Howard 1988) (this paradigm is known as network-attached storage (NAS)). However, as we discuss in section 2.7.1, this approach also suffers from scalability and performance limitations like relying on some centralized component, performing static data partitioning, etc.

The object-based storage paradigm (Mesnier, Ganger, & Riedel 2003; Factor, Meth, Naor, Rodeh, & Satran 2005) proposes objects as basic storage units in storage devices as a means of overcoming the limitations presented above. An object is a variable-length data container with a file-like interface. A device that stores objects is called an object-based storage device (OSD). An OSD exposes an interface similar to that of a file system, with operations such as *read*, *write*, *create* and *delete*. However, differently from traditional file systems, OSDs operate on a flat namespace, so they don't have to manage hierarchical directories. Moreover, OSDs only provide basic storage functionality, leaving advanced features that limit scalability, like file locking or directory management, to higher level systems.

A major benefit of object-based storage, is that disk metadata management is offloaded from storage systems to OSDs. This improves data sharing between multiple hosts in comparison to block and file storage, since different hosts and applications do note need to coordinate distributed metadata management or agree on block placement policies on disk. Instead, they only need to interact with a higher level interface to access objects. Non-essential features like locking or directories are implemented by storage applications that use OSDs. This allows object storage to scale while being able to leverage data sharing, capturing benefits of both NAS and SAN architectures, without the current limitations of file or block access.

OBFS (Wang, Brandt, Miller, & Long 2004) is a storage manager for object-based storage

devices. Traditional file systems are optimized for workloads based on small files, hierarchical directories and some degree of locality. In contrast, object-based storage presents a flat namespace, with no logical relationship among objects, so traditional file system assumptions are not suitable for this workload. OBFS takes advantages of these particular differences to provide a high-performance storage system implementing an OSD interface. OBFS significantly outperforms traditional file systems like Ext2 and Ext3 on both read and write operations. Moreover, while OBFS's code is $\frac{1}{25}$ the size of XFS, it achieves significantly higher write performance and a slightly reduced read performance.

Ceph (Weil, Brandt, Miller, Long, & Maltzahn 2006) leverages a cluster of OSDs to provide a distributed file system with high capacity, throughput, reliability, availability and scalability. Ceph achieves this by decoupling data and metadata management in two distributed sets of servers. A highly adaptive distributed metadata cluster manages the file system's namespace, coordinates security, consistency and coherence. Data storage is delegated to a cluster of OSDs based on OBFS, leveraging the intelligence of these devices to reduce file system complexity.

Ceph's cluster of OSDs are collectively managed by a distributed object storage protocol (RADOS), providing a single logical object storage and namespace. RADOS delegates replication, fault tolerance and cluster expansion to OSDs in a distributed fashion. Ceph files are stripped in many objects in order to increase system throughput rates through data parallelism. Strips are distributed across many OSDs through CRUSH (Weil, Brandt, Miller, & Maltzahn 2006), a decentralized placement algorithm for replicated data. Ceph employs an adaptive directory placement algorithm to dynamically map directory hierarchy subtrees to metadata servers based on current load. This makes the load evenly distributed across metadata servers, avoiding hot spots that may degrade the system's performance.

Facebook's **Haystack** (Beaver, Kumar, Li, Sobel, & Vajgel 2010) proposes a custom object-based storage system for storing photos at Facebook. According to the authors, traditional POSIX-based file systems imposed limitations to the scalability of the previous photo storage solution. In particular, the process of fetching file system metadata on disk to open a photo introduced significant overhead to read operations. As we previously discussed, traditional file system caching based on locality is not suitable for workloads composed of random reads (such as a photo store). Moreover, unused metadata attributes, such as file permissions, unnecessarily wasted storage space when the system scaled to billions of photos and petabytes of data.

Haystack indexes photos with a simple flat identifier that contains the logical volume where the photo is stored. A directory service maps logical volumes to physical nodes. A cache service is implemented on the top of a distributed hash table to improve access to popular photos. Each HayStack node stores photos in log-based files, one for each logical volume. In order to avoid costly metadata lookups on disk, file descriptors of all logical volumes of a node are kept open in memory. Each HayStack node also maintains an in-memory mapping of photo identifiers to their location on the logical volumes, which significantly improves read throughput. In comparison with the previous solution based on NFS, HayStack improves read throughput by 400% while using 28% less storage.

In the next subsection we discuss how this paradigm for persistent storage is adopted in the context of cloud computing.

2.3.1 Cloud-based Object Storage

Object-based storage provides a simple and scalable model for cloud computing storage. While block and file-based storage are important for a wide-range of applications, a simpler object-based storage model is sufficient for storing many types of unstructured data, including media assets, application data and user-generated content. Moreover, the object-based storage interface is completely decoupled from the actual underlying storage technology, allowing data to be transparently stored by the cloud provider. This independence allows the provider to migrate data, to account for server failures, or even replace the storage infrastructure without needing to reconfigure clients.

Several cloud providers currently offer an object-based storage service: Amazon Simple Storage Service (S3), Google Cloud Storage, Windows Azure Blob Storage, Rackspace Cloud Files, etc. Despite the lack of official standards for cloud-based object storage, most existing public offerings have interfaces and feature sets similar to Amazon S3 (Amazon 2012). For this reason, and without loss of generality, we describe Amazon S3, which is considered a de facto standard for cloud storage (Carlson 2009), and assume this model of cloud-based object storage for the remainder of this thesis.

Amazon Simple Storage Service provides a pay-as-you-go object storage service for unstructured binary data. The service is accessed via HTTP through well-defined web service

interfaces, such as REST (Fielding 2000) or SOAP (Gudgin, Hadley, Mendelsohn, Moreau, & Nielsen 2003). These interfaces are directly accessible by any client that supports HTTP, allowing the service to be used by many different applications and platforms.

After registration on S3, the user is given an account and password that is used to access the service. An account can create one or more buckets (also called object containers), which are similar to a flat directory to store objects. A bucket name must be unique across the system, since Amazon uses a flat global namespace to store buckets. Using the account credentials, the client may put, get or delete objects using any of the access interfaces. An object is identified by a name and must be placed within one of the account's containers. Each object may have up to 5 terabytes in size, each accompanied by up to 2 kilobytes of metadata. An attribute defines if buckets and objects are visible to other users.

Amazon S3 exposes a public interface but is based on proprietary infrastructure. This prevents a more detailed technical analysis of its underlying software system. However, it is possible to infer some requirements and design decisions from information publicly released by the company. Some of this information is summarized in the following points:

- A blog post from Amazon's Chief Technology Officer (Vogels 2007) suggests that the Dynamo system (Hastorun, Jampani, Kakulapati, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007) is one of the technologies behind Amazon S3 (more information on Dynamo will be given in section 2.7.1.1).
- S3 was designed to provide 99.999999999% durability and 99.99% availability of objects over a given year (Amazon 2012).
- S3 ensures object durability by replicating objects across multiple data center on the initial write, and fixing replication errors when they are detected. (Amazon 2011)
- S3 objects are eventually consistent. (Amazon 2012)

The following numbers from Amazon S3 offer an insight on the scale that a public cloud-based object storage provider deals with:

- Over a trillion objects stored by June 2012 (Barr 2012a);
  - From 762 billions in 2011 (Barr 2012b);

  - From 102 billions in 2009 (Barr 2012b);
  - From 2.9 billions soon after the service started (2006) (Barr 2012b).

- Average of 650,000 requests per second. (Barr 2012b)

The numbers show that the amount of objects in a cloud storage system tend to accumulate over the years, what reinforces the need for a cloud storage solution with deduplication support.

Despite the high number of commercial offerings for cloud-based object storage, only a few non-proprietary solutions are available to the public. The Eucalyptus project (Nurmi, Wolski, Grzegorczyk, Obertelli, Soman, Youseff, & Zagorodnov 2009) offers an open source implementation of the Amazon Web Services API. **Walrus** is the Eucalyptus component analogous to Amazon S3 (object storage). However, Walrus merely provides an S3-compatible interface to a centralized storage system, which is clearly not a scalable and reliable solution for a large scale object-based storage system.

Cumulus (Bresnahan, Keahey, LaBissoniere, & Freeman 2011) proposes an S3-compatible object storage system aimed for academic use. Cumulus has a very simple design that translates S3 HTTP REST calls into lower level file system calls. In order to handle scalability, Cumulus can be configured to run as a set of replicated hosts. In this case, the replicated servers must have access to a distributed file system presenting the same view to all servers. Even though this solution is satisfactory for an academic environment, its reliance on the underlying distributed file system might limit the system's scalability, since traditional file systems are not optimized for the workload of an object-based storage system.

The **RADOS** subsystem implements a distributed object storage protocol used by Ceph file system (Weil, Brandt, Miller, Long, & Maltzahn 2006). It is possible to export a HTTP REST interface to RADOS allowing users to access RADOS in a cloud fashion. Since RADOS distributed design is inherently object-based, we believe it is a promising open source alternative to Amazon S3.

OpenStack (OpenStack 2012) is a collaboration project of hundreds of organizations led by NASA and RackSpace that produces an open source cloud computing platform for public and private clouds. One of OpenStack's cloud components is Object Storage, code-named Swift, which provides a scalable, reliable and available distributed object storage service. Swift's

architecture is partly based on Amazon's Dynamo, which is one of the technologies behind Amazon S3.

None of the above systems currently has support for deduplication. We have chosen OpenStack Object Storage (Swift) to implement the prototype of a cloud storage solution with built-in support for deduplication. For this reason, we present a more detailed overview on OpenStack Swift's architecture in the following subsection.

2.3.1.1 OpenStack Object Storage (Swift)

OpenStack Swift architecture is based on 3 main components: storage servers, the rings and proxy servers. We describe each of this components in the next subsections.

Storage Servers

The storage servers of OpenStack Swift can be of three types:

Object Server An object-based storage device, that stores, retrieves and deletes binary objects to/from a local storage medium. Each object is stored in the local file system using a path derived from the object's name hash.

Container Server Handles listings of objects, also called containers. Containers are similar to folders in traditional file systems but with a flat structure. Container information is stored in SQLite databases.

Account Server Handles listing of containers for customer accounts, which can have one or multiple containers. Account information is also stored in SQLite databases.

The Rings

The ring is an abstraction that maps data identifiers to storage nodes. A configurable number of bits from the hash of the object name is used as a key to lookup the nodes responsible for storing that object in the ring. A separate ring structure is used for each type of storage server, providing distinct namespaces for different server types.

Internally, the ring is constructed by dividing the hash space into equal sized partitions, which are distributed among the storage devices. Each node may hold one or more storage devices. The amount of partitions a storage device is assigned to is proportional to its capacity. When a device leaves the system, its partitions are evenly distributed across the remaining devices. Similarly, when a new device joins the system, it receives roughly the same amount of load from other devices. This scheme is a variant of consistent hashing, that will be explained in section 2.6.1.

Replication scope is also defined by the ring, that defines which devices are responsible for storing replicas of a given partition. In order to increase durability, devices are grouped in zones, which represents physical or logical separation of devices, such as: different nodes, different racks or power supplies, etc. The ring is built in a way that ensures different replicas of the same piece of data are placed in different zones.

The rings are statically managed by an administrator tool, that explicitly adds and removes devices from the ring. Each time the ring changes, it must be re-distributed among the storage nodes participating in the system.

Proxy Server

The proxy server is a gateway service that exposes the public API of the object storage service, coordinating requests from external users to the internal network of storage servers.

The proxy server supports operations on accounts, containers and objects. Each client request is authenticated and validated, and then routed to the appropriate storage node by looking up the location of the entity in the ring. The proxy also handles storage node failures: when a storage server is unavailable, it looks up a handoff node in the ring and routes the request there instead.

In order to increase availability and to avoid a single point of failure, one Swift installation may have multiple proxy servers, which will serve requests independently. A load balancer may be used to distribute load between different proxy servers.

2.4 Data deduplication

Data deduplication comprises a set of compression techniques that improves space utilization in a storage system by reducing the amount of unnecessary redundant data[1]. These techniques can also be used to avoid transfer of duplicated data between different hosts, thus improving bandwidth utilization. In this thesis we use deduplication as a key technique to achieve storage and bandwidth savings in an object-based storage system. In this section we provide an extensive account on deduplication methods and systems that use this technique.

Deduplication is generally done by identifying duplicate chunks of data and keeping just one copy of these repeated chunks. New occurrences of chunks that were previously stored are replaced by a reference to the already stored data piece. These references are typically of negligible size when compared to the size of the redundant data. The technique can be applied within a file or object (intra-deduplication) or between different files (inter-deduplication).

Mandagere et al (Mandagere, Zhou, Smith, & Uttamchandani 2008) proposes a classification for deduplication systems based on 3 axis: **algorithm**, **placement** and **timing**. We present and discuss these categories, with examples of relevant systems that employ each of the described techniques.

2.4.1 Deduplication Algorithm

The two major categories of deduplication algorithms are delta-encoding and hash-based deduplication, described below.

2.4.1.1 Delta-encoding

Delta-encoding techniques, discussed in detail in (Hunt, phong Vo, & Tichy 1996), perform deduplication based on the difference (delta) between two files. Given a reference file and another file, a delta-encoding algorithm will produce an encoded version of the second file relative to the reference file. The "diff" utility present in UNIX systems uses delta-encoding to compare two files.

[1]This does not directly apply to data that is intentionally replicated to achieve durability and availability in a distributed storage system.

Delta-encoding is typically done by comparing the contents of both files and identifying similar regions in both files. The encoded file will contain pointers to regions in the reference file when regions are shared, and the actual data when unique data is found. The encoded version is typically called a diff or a patch file. In order to reconstruct the original file, the differencing data is applied to the reference file through a simple stream matching algorithm.

The efficiency of delta-encoding deduplication is proportional to the resemblance between the file being encoded and the reference file. For this reason, it is typically applied in cases where files evolve over time, such source code tracking in revision control systems like CVS and SVN (Grune 1986; Collins-Sussman, Fitzpatrick, & Pilato 2002). Another important use case for delta-encoding is to save bandwidth on network transfers. For instance, the HTTP specification (Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, & Berners-Lee 1999) allows servers to send updated web pages in the form of diffs, in order to optimize bandwidth utilization.

2.4.1.2 Hash-based deduplication

In hash-based deduplication, each data chunk is identified by a hash of its contents (chunk fingerprint). The deduplication is performed by comparing fingerprints of fresh data with fingerprints of previously stored data. When a match is found, it means the data is already stored and must not be stored again. In this case, the data piece is replaced by a metadata reference that indicates the fingerprint of the chunk that was previously stored.

In order to ensure that any chunk will have a unique fingerprint, a secure cryptographic function, such as SHA-1 or SHA-2 (Eastlake 3rd & Jones 2001), must be used to hash chunks. This is due to the collision-resistance property of secure cryptographic functions, that make it unfeasible for two data pieces to have exactly the same digests.

Since in this type of deduplication a data chunk is shared by many distinct files/objects, the server must track the files that reference a particular data chunk (back-references). This will enable the server to garbage collect chunks that are no longer referenced by any file. Reference tracking can be done by reference counting, where each data chunk has a counter indicating the number of files that currently reference that chunk. Another way of tracking references is by explicitly storing the names of the files that track a particular chunk.

An important aspect in hash-based deduplication systems is the technique used to break

data containers (such as file or object) into chunks. This will directly influence the deduplication efficiency of a system, since the more chunks in common, the less data needs to be stored. A simple approach is to use an entire data container as the unit to perform the digest calculation: whole-file hashing. Sub-file hashing or chunk-level deduplication splits a file into multiple chunks and perform a more fine-grained deduplication. The latter approach can use static or variable chunk sizes. Each of these file splitting techniques are presented and discussed in the following subsections.

Whole File Hashing (or File-level deduplication)

This method of hash-based deduplication calculates the fingerprint of the entire file to identify duplicated content. The major benefit of this approach is increased throughput, since only a single hash calculation needs to be done per file.

This technique is particularly efficient when there are many repeated files in a dataset. For instance, a music hosting service could use whole-file hashing to identify repeated MP3 files, which would be very common for popular songs. Another use is on file systems to avoid storing multiple copies of the same file on disk, what is done in Microsoft's Single Instance Storage (Bolosky, Corbin, Goebel, & Douceur 2000).

The downside of this approach is that it may yield little benefits in datasets with low file redundancy. Additionally, a single bit change in the file will generate a different file hash, requiring the whole file to be re-stored or re-transmitted. For this reason, this method of hash-based deduplication may not be efficient for files that change often.

Fixed Chunk Hashing (or Static chunking)

In this approach, the chunking algorithm splits the file into fixed-size chunks starting from the beginning of the file. A cryptographic hash function calculates the secure digest of each chunk, and the fingerprint is used to verify if the chunk is already present in the system.

The benefits of this approach include ease of implementation and fast chunking performance. Moreover, the chunk size can be aligned with the disk block size to improve I/O performance. Example storage systems that use static chunking are Venti (Quinlan & Dorward 2002) and

Solaris' ZFS (Bonwick 2009). A study by Jin & Miller suggests that fixed-size chunking is more efficient than variable-size chunking for deduplication of Virtual Machine disk images.

In comparison to whole-file hashing, this approach has lower throughput since a hash-table must be consulted on each chunk to verify if it is already present in the system. Another downside of this method is what is known as the boundary shifting problem: if a single bit is appended to or removed from a file, the fingerprint of all chunks following the update will change. Thus, it is not resistant to changes in the deduplicated file.

Variable Chunk Hashing (or Content-defined chunking)

In variable chunking, a stream of bytes (such as a file or object) is subdivided by a rolling hash function that calculates chunk boundaries based on the file contents, rather than static locations across the file. For this reason, this method is also called content-defined chunking. After a variable-sized chunk boundary is defined, the chunk digest is calculated and used to verify if the chunk is already present in the system.

One important property of content-defined chunking is that it has a high probability of generating the same chunks even if they are placed in different offsets within a file (You & Karamanolis 2004). This makes it resistant to the chunk boundary shifting problem, because the modification of a chunk does not change the fingerprint of remaining chunks.

A popular method for variable chunking is rabin fingerprinting (Rabin 1981), which computes a modulo function across a rolling hash sliding window to define chunk boundaries. When the modulo is equal to a predetermined irreducible polynomial, a chunk boundary is determined. This approach is employed by Low Bandwidth File System (LBFS) (Muthitacharoen, Chen, & Mazières 2001) to chunk and detect similarities between files.

According to Policroniades et al. (Policroniades & Pratt 2004), variable chunk hashing is particularly efficient for deduplicating correlated data. Due to its resistance to the chunk boundary shifting problem, it is also a good choice for dealing with dynamic data since changing a few bytes from the file does not change hash digests of other chunks. The major downside of this approach in comparison with the previous hash-based techniques is that it adds significant latency overhead to the deduplication process, due to the added step of calculating chunk boundaries.

2.4.2 Deduplication Placement

The next dimension in the deduplication design space is related to where the deduplication process occurs: at the client (source) or at the deduplication appliance (target), a special purpose system that implements deduplication.

2.4.2.1 Source Deduplication (Client-based)

In source deduplication, the client performs the deduplication procedure and exchanges metadata with the server to verify data duplication. For instance, when hash-based deduplication is used, the client calculates the fingerprint of each file chunk and sends this information to the server, to verify if that data piece exists on the appliance. Only unique data chunks - chunks that weren't previously stored - need to be transferred to the server. When the chunk already exists, the server updates the reference tracking data structures and the operation is completed.

The major benefit of client-based deduplication is the potential network bandwidth savings when data is duplicated, since only small metadata and unique data needs to be transferred over the network. This is exactly the approach taken by LBFS (Muthitacharoen, Chen, & Mazières 2001) to provide a bandwidth-efficient file system. Cloud-based file systems wrappers like S3QL (S3QL 2012) and OpenDedup (OpenDedup 2012), perform client-based deduplication in order to send only unique data to the cloud provider, consequently reducing customer storage charges.

One drawback of client-based deduplication is that clients must spend computing and I/O cycles to perform deduplication, what may be a problem in devices with limited computation capacity or battery-constrained.

2.4.2.2 Target Deduplication (Deduplication Appliance-based)

In target deduplication, the deduplication procedure is performed by the deduplication appliance. In this case, the appliance exposes a default storage interface and the client transfers the entire data that needs to be stored. In some cases, the client may even be unaware deduplication is being performed. When the deduplication appliance is located in another host and the communication is done over a network, this type of deduplication may be called server-based deduplication. Venti (Quinlan & Dorward 2002) and ZFS (Bonwick 2009) are examples of storage systems that perform deduplication in the storage appliance.

Server-based deduplication is particularly beneficial for resource-constrained devices, since it consumes less computational resources from the client. Additionally, it is also a good choice to enable transparent deduplication on legacy applications or clients that cannot be upgraded, since existing client code need not be modified to support target deduplication. A major drawback of server-side deduplication is that all data needs to be transferred over the network, so this approach provides no bandwidth savings.

2.4.3 Deduplication Timing

With respect to timing, a deduplication system may perform the deduplication when the data is being inserted (synchronous), or post-process the data after it is already stored in the system (asynchronous). We present and discuss each of these approaches in the following subsection.

2.4.3.1 Synchronous (Inline deduplication)

When the deduplication procedure is performed during data insertion, before data is written to disk, it is called synchronous deduplication (also called inline, or in-band deduplication). This includes chunking the file or object, verifying if each data chunk is duplicated and generating metadata to represent the deduplicated file. The insert operation is completed after the object metadata is generated and the unique chunks are written to disk. This approach matches well client-based deduplication, since the server is able to tell immediately if given data pieces are present in the system, avoiding transfer of duplicated data between the client and the server. Systems that perform inline deduplication include: archival storage and network file system Venti (Quinlan & Dorward 2002), cloud-based file systems S3QL (S3QL 2012) and OpenDedup (OpenDedup 2012), and general purpose file system and volume manager ZFS(Bonwick 2009).

One of the benefits of synchronous deduplication is that no additional storage space is required on deduplication appliances to stage objects for post-processing deduplication. Moreover, the system needs to maintain only one protocol for retrieving data, because it only deals with objects that were already deduplicated. However, these benefits come at the cost of increased latency during insertion operations, due to the overhead added by the deduplication process. However, part of this overhead can be mitigated by leveraging parallelism, for instance, by

calculating fingerprints and transferring chunks in parallel.

2.4.3.2 Asynchronous (Offline deduplication)

In order to avoid the extra step of deduplication in the write path, and its associated overheads, it is possible to perform deduplication in an asynchronous fashion. This method is also called offline or out-of-band deduplication. In this approach, the file/object is written to the deduplication appliance as a whole to a special staging area. A background process deduplicates files that were recently inserted in the system, removing them from the staging area after the deduplication process is complete. The post-processing step is generally triggered during periods of low system load, as not to affect latency of other incoming requests.

Since this deduplication method does not add latency overhead to the data insertion operation, it is a good choice for latency-critical applications, where the ingestion speed of data is a primary concern. This is the case for commercial storage appliances, like NetApp ASIS (Alvarez 2011), EMC Celerra (EMC 2010) and IBM StorageTank (IBM Corporation 2002).

The need to keep a staging area to store unprocessed data until it is deduplicated by a background process is one of the disadvantages of asynchronous deduplication. Another related disadvantage is to keep two protocols for reading data in the storage system: one for unprocessed data and another for deduplicated data.

2.5 Content-Addressable Storage

An important paradigm leveraged in the solution proposed in this thesis is content-addressable storage (CAS), or content-addressed storage. In this technique, data elements are identified by a *content address*, which is a unique digital fingerprint permanently linked to the information content. This fingerprint is typically obtained by hashing the data with a collision-resistant cryptographic hash function (such as SHA-1 or SHA-2 (Eastlake 3rd & Jones 2001)). A basic content-addressable storage interface is shown below:

Write data *(input: binary chunk of data)* {*returns data fingerprint*}

Read data *(input: data fingerprint)* {*returns binary chunk of data*}

Check data existence *(input: data fingerprint)* {*returns true or false*}

A storage system implementing this interface maintains an index data structure that maps data fingerprints to physical locations on the storage medium. Since data elements with identical content will have the same fingerprint, they only need to be stored once, thus providing automatic data deduplication.

Content-addressable storage provides an alternative to *location-addressed storage* (LAS), where data elements are indexed by their storage location relative to a particular device. A typical example of LAS is a traditional file system, where files are disposed in a hierarchical directory structure. In contrast to content-addressable storage, in LAS systems elements with different identifiers but with identical contents are stored multiple times on disk.

A natural consequence of addressing data by the fingerprint of its contents on CAS is that changing the contents of a data element also changes its fingerprint. For this reason, CAS elements are immutable and write-once: changing the contents of a data element generates a new element with a distinct fingerprint. This makes CAS a good match for storing static data.

Since elements in CAS are only addressed by their fingerprints, systems that implement a CAS interface are typically used as building blocks to implement higher level storage systems. The latter systems need to maintain additional metadata that maps elements in higher level namespaces into content addresses.

The choice of content-addressable storage as a key technique to index data in the proposed solution was based on the following reasons:

- The automatic duplicate identification and elimination provided by the paradigm increases opportunities for storage and bandwidth savings.
- The location-independent, universal and flat namespace of data fingerprints simplifies data distribution across a set of storage nodes.
- The problem of data coherence is greatly reduced due to the use of immutable data elements, simplifying important distributed systems features like caching, replication and load balancing. (Quinlan & Dorward 2002)
- The technology provides inherent data integrity validation, since data is identified by a cryptographic digest of its contents.

Several systems employ content-addressable storage as a means of achieving automatic detection and elimination of data redundancy. These systems are from a wide range of topics, including but not limited to: backup and archival storage systems, (Venti, Foundation, Deep Store, Pastiche, EMC Centera, Hydrastor)[2], distributed file systems (LBFS, CASPER, Farsite, CernVM-FS)[3], version control systems (Git)[4], replica synchronization protocols (TAPER)[5]. A representative set of these systems that is relevant in the context of this thesis is presented in the following subsections.

2.5.1 Venti

One of the most notable CAS solutions is Venti (Quinlan & Dorward 2002), a network storage system intended for archival storage. Venti provides a simple storage interface that allows client applications to read and write arbitrary blocks of data. Upon writing a data block to Venti, the fingerprint of the block is returned to the client. The system also allows the client to pack multiple fingerprints into special types of block called pointer blocks. These blocks have their fingerprint correspondent to the hierarchical hash of the original data, allowing complex data containers (such as files or directories) to be constructed.

Venti stores blocks in an append-only log on a RAID array of disk drives in order to improve disk seek latency and provide data durability. The log provides a simple and efficient approach for storing write-once data. This log is divided into self-contained arenas, in order to simplify maintenance. Each arena stores multiple blocks and keeps a directory structure that maps fingerprints to offsets in the arena. The internal block structure keeps additional metadata and headers, in addition to the block's contents, in order to provide integrity checking and data recovery.

In order to allow blocks to be efficiently located, Venti keeps a persistent index that maps fingerprints to locations in the log. This index is divided into buckets, and each bucket is stored as a single disk block, enabling it to be retrieved using a single disk access. A fixed portion of the

[2](Quinlan & Dorward 2002; Rhea, Cox, & Pesterev 2008; You, Pollack, & Long 2005; Cox, Murray, & Noble 2002; EMC 2004; Dubnicki, Gryz, Heldt, Kaczmarczyk, Kilian, Strzelczak, Szczepkowski, Ungureanu, & Welnicki 2009)

[3](Muthitacharoen, Chen, & Mazières 2001; Tolia, Kozuch, Satyanarayanan, Karp, Bressoud, & Perrig 2003; Douceur, Adya, Bolosky, Simon, & Theimer 2002; Blomer, Buncic, & Fuhrmann 2011)

[4](Torvalds 2009)

[5](Jain, Dahlin, & Tewari 2005)

fingerprint is hashed to find the bucket responsible for storing the location of that fingerprint. The bucket is then examined using binary search to find the exact location of the requested block.

Venti leverages compression to further improve storage savings provided by CAS. Furthermore, it keeps a block cache and and an index cache to speed up block lookup operations. Venti itself is not a file system or backup system, but rather provides a storage backend for these applications. One version of the Plan 9 file system (Pike, Presotto, Thompson, & Trickey 1990) was implemented on the top of Venti and it provided significant storage savings in comparison with the previous version of the system that did not use CAS.

Even though Venti could be used as storage backend for a small scale object storage solution, it's centralized architecture is not optimized for cloud scale. Moreover, Venti relies on expensive RAID array and high speed disks for durability, making it a costly solution to be adopted at a larger scale.

2.5.2 Foundation

Foundation (Rhea, Cox, & Pesterev 2008) is a digital preservation system that employs CAS to store nightly snapshots of users' disk. Foundations was inspired by Venti's design and aims at providing an inexpensive backup system tailored for consumer user. Additionally, Foundation leverages Bloom filters to quickly detect new data and makes assumptions about the structure of duplicate data in order to increase system's throughput.

The system provides a CAS layer that stores nightly snapshots of the state of virtual machines. This layer is an optimization of Venti's design aimed at providing a higher throughput on read and write operations using commodity disks. The block index structure is summarized in an in-memory Bloom filter (Bloom 1970), in order to avoid the disk overhead of index lookup during most of the writes when data is not present in the system. Another optimization is to batch index updates in memory in order to avoid the disk overhead of updating the persistent index. In the face of failures, the index can be regenerated from the persistent data log. During reads, Foundation loads and caches entire arena directories into memory in order to leverage data locality, because data that was sequentially written to the same log-based arena will likely be read in the same order.

In addition to hash-based content-addressable storage, Foundation can operate in an alternative mode where duplicate identification is done by comparing each pair of potential duplicates byte-by-byte. In this alternative mode, blocks are identified by their offset relative to the disk log. The index structure is only used to identify potentially duplicate blocks and, thus, a weaker and faster hash function can be used, since collisions are acceptable. This mode provides faster read throughput, since naming blocks by their log offset completely eliminates the need to query the index structure. However, writes of potentially identical blocks need a full byte-by-byte comparison. Since this approach names blocks after their position in the write log, it its not a CAS solution, but rather a location-addressed storage.

Similar to Venti, Foundation could not be used as storage backend to a cloud-based deduplicated object storage solution due to its centralized architecture, which prevents the system from scaling to attend growing cloud demands.

2.5.3 CASPER

CASPER (Tolia, Kozuch, Satyanarayanan, Karp, Bressoud, & Perrig 2003) is a network file system for mobile clients derived from the Coda file system (Satyanarayanan, Kistler, Kumar, Okasaki, Siegel, David, & Steere 1990). CASPER intercepts file systems calls and tries to satisfy client requests from the system's cache. When the client is experiencing a low bandwidth connection and there is a cache miss, instead of requesting the whole file's contents from the file system server, it requests a *file recipe*. A recipe is a file synopsis much smaller than the original file that contains a list of blocks fingerprints that allow the original file to be reconstructed from its parts. The client then fetches individual data blocks from nearby Content-Addressable storage providers, possibly over Local Area Network (LAN) speeds. The remaining blocks that could not be fetched from nearby CAS providers are fetched from the original server over a wide area network (WAN). CASPER foresees a scenario where CAS providers will be readily available in different organizations. It is important to mention that the system only uses CAS providers opportunistically: when a CAS provider is not available, the file is retrieved from the file server as usual.

The most important contribution of CASPER is the concept of file recipes, which is a first class entity representing a file. This summarized representation allows the file to be reconstructed from its content-addressed blocks. We have used a similar structure to represent objects in the

proposed deduplicated object storage solution. Object recipes provide a way to map objects from the flat object namespace to content-addressable storage. More details on the summarized representation of the object will be given in section 3.3.2.11.

2.5.4 Commercial Solutions

One of the first commercial offerings that leverage CAS is EMC Centera (EMC 2004; Gunawi, Agrawal, Arpaci-Dusseau, Arpaci-Dusseau, & Schindler 2005). Centera offers a integrated storage appliance for data archiving that provides single instance (object-level) deduplication through content-addressable storage. This solution is based on a redundant array of commodity servers and a distributed software (CentraStar) that runs on these servers. CentraStar architecture[6] is composed of two server types: storage nodes and access nodes. The storage nodes compose a distributed hash table (DHT) and store objects on their local disks. The DHT indexes objects by their unique fingerprint and allows the system to scale to multiple storage nodes. The access nodes manage object read and write requests from clients, redirecting them to storage nodes. After a successful write, the access node returns a content address (fingerprint) that can later be used to retrieve the stored object. Objects are replicated twice before a write operation is completed in order to provide durability and availability.

HYDRAStor (Dubnicki, Gryz, Heldt, Kaczmarczyk, Kilian, Strzelczak, Szczepkowski, Ungureanu, & Welnicki 2009) is a scalable, secondary storage solution based on CAS aimed at the enterprise market. Similar to the previous system, HYDRAStor's architecture is composed of a set of frontend access nodes and a grid of backend storage nodes built around a distributed hash table. The frontend nodes exports the same block API as the backend nodes, but can be easily extended to support novel access protocols. Blocks may contain pointers to other blocks, allowing more complex data structures (like files or objects) to be stored. Differently from Centera, this system supports variable-sized blocks, what increases opportunities for duplicate identification. Moreover, the system aims at providing increased data reliability, availability and integrity through the use of erasure coding techniques.

Even though the presented systems satisfy scalability and availability requirements for a cloud-based object storage solution, they are optimized for secondary, archival storage, which

[6]This description is based on the version 2.0 of the CentraStar software analyzed by Gunawi et al.

tend to deal with writes of long data streams and where reads seldom occur. In contrast, primary storage often deals with random workloads which these systems are not optimized for. Moreover, they are based on costly proprietary licenses and the unavailability of more detailed technical information prevent them from being implemented in an accurate manner.

2.6 Distributed Hash Tables

Distributed hash tables (DHTs) provide an abstraction that maps keys onto nodes in a distributed environment. The most popular DHT protocols[7] were conceived in the context of peer-to-peer (P2P) systems, which is a distributed architecture for resource sharing over a network where nodes (peers) act as servers and clients simultaneously. A fundamental operation in these systems is locating a peer responsible for keeping a particular resource. Previous solutions either involved maintaining a centralized lookup server (Napster 1999) or broadcasting a resource lookup message to all participants in the system (Gnutella 2000). While the former approach suffers from the single point of failure problem, both approaches present significant problems for scaling to a large number of nodes. DHTs were proposed in order to fill the gap of providing a scalable, decentralized solution for efficient resource lookup in a large distributed system.

Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001) is one of the most important systems that implements a distributed hash table. It provides a simple interface that enables higher level applications to use distributed lookup functionality. Chord uses a variant of consistent hashing (explained in section 2.6.1) to assign keys to Chord nodes. Each node maintains a small routing table (no more than $O(logN)$ entries) allowing requests to be routed to the correct node within a bounded number of hops ($O(logN)$). Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures. This allows the system to correctly operate and route the requests to the correct node under a highly dynamic environment. Other structured P2P systems have a similar concept and functionality, but differ on structure, routing algorithms and performance characteristics. The most notable examples of these systems include: CAN (Ratnasamy, Francis, Handley, Karp, & Shenker 2001), PASTRY (Rowstron & Druschel 2001) and TAPESTRY (Zhao, Kubiatowicz, & Joseph 2001).

[7](Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001; Ratnasamy, Francis, Handley, Karp, & Shenker 2001; Rowstron & Druschel 2001; Zhao, Kubiatowicz, & Joseph 2001)

Several distributed storage systems employ some form of distributed hash tables to efficiently distribute data across many nodes[8]. In our solution, we leverage a distributed hash table to index data chunks and implement a distributed content-addressable storage. An important advantage of this approach is that it provides global distributed data deduplication, since data chunks with the same fingerprint map to the same set of storage nodes. Moreover, it enables the system to be both scalable and decentralized, since a DHT allows nodes to be added and removed from the system and no node is more important than any other. A few of the previously mentioned storage systems employ a similar DHT strategy to implement a distributed content-addressable storage interface: EMC Centera (EMC 2004), HYDRAStor (Dubnicki, Gryz, Heldt, Kaczmarczyk, Kilian, Strzelczak, Szczepkowski, Ungureanu, & Welnicki 2009) and DeepStore (You, Pollack, & Long 2005).

2.6.1 Consistent Hashing

A straightforward method to map a set of keys into a set of nodes is a hash function. However, a change in the number of nodes causes nearly all keys to be remapped when a traditional hash function is used (such as: $x \to (ax + b) \bmod p$, where x is the key and p is the number of nodes in the system). When used in a distributed system this means that all resources that mapped to any node N, will now be mapped to node M, and possibly become inaccessible. Even though resources can be moved across nodes, this is an expensive operation which should be done as little as possible. In a dynamic distributed environment, where nodes come and go, it is essential that the hash function causes minimal disruption when the number of nodes changes.

Consistent hashing (Karger, Lehman, Leighton, Panigrahy, Levine, & Lewin 1997) is a hashing technique that remaps only $O(K/N)$ keys (where K is the number of keys, and N the number of nodes) when the number of slots (nodes) changes. Moreover, the technique ensures that each node will be responsible for at most $(1 + \epsilon)K/N$ keys with high probability, providing a degree of natural load balancing. Most distributed hash tables use some variant of consistent hashing to efficiently map keys to nodes. The solution proposed in this thesis leverages consistent

[8] (Druschel & Rowstron 2001; Muthitacharoen, Morris, Gil, & Chen 2002; Hastorun, Jampani, Kakulapati, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007; Amann, Elser, Houri, & Fuhrmann 2008; EMC 2004; Dubnicki, Gryz, Heldt, Kaczmarczyk, Kilian, Strzelczak, Szczepkowski, Ungureanu, & Welnicki 2009; You, Pollack, & Long 2005)

hashing to increase the system's availability, since node additions and removals can be done with minimal data movements between nodes.

Consistent hashing treats the output of a hashing function as a circular ring with cardinality m, where the largest hash value $(2^m - 1)$ wraps around the smallest value (0). Each node is assigned a random m-bit identifier, where m should be large enough to make the probability of two entities having the same identifier negligible. This identifier defines the "position" of the node in the ring. The node responsible for storing a given key k is found by hashing k to yield its position in the ring and then walking the ring clockwise to find the node with identifier equal or larger to the item's position (successor). Thus, each node is responsible for the region in the ring between itself and its predecessor.

Figure 2.1 shows an example consistent hashing ring for 3 nodes. Supposing that $m = 3$, $id(A) = 2$, $id(B) = 4$, $id(C) = 6$ and we want to locate the node responsible for key k. Initially the value of k is hashed to find its location in the ring (indicated by the arrow), for instance, $hash(k) = 1$. The next node in the ring following 1 has identifier 2, so node A is responsible for storing k.

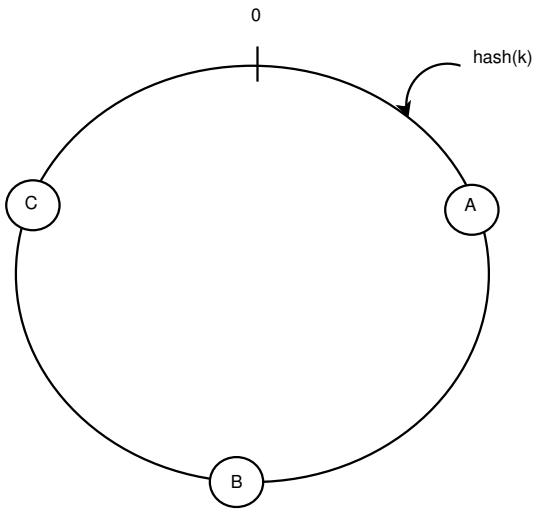


Figure 2.1: Example consistent hashing ring for 3 nodes: A, B, and C. In this example, key k will map to node A, since it is the successor node to the key's hash in the ring.

A major benefit of consistent hashing is that only the region a node is responsible for needs to be reassigned in case it leaves the system. Similarly, if a new node joins the system, it will gain responsibility over a fraction of its successor's previous region. So, there is no disruption in other regions of the ring.

In the example ring in Figure 2.1, in case node A leaves the system, node B's region will grow and occupy the region that previously belonged to A. In this new configuration, the key k which previously mapped to node A will now map to node B. Keys which previously mapped to node C will remain unchanged. In case a new node D is placed in the region between C and A, it will now become responsible for the region between C and D, which previously belonged to A. The remaining regions remain unchanged.

The basic algorithm can be enhanced by allowing each physical node to host multiple "virtual" nodes. In this approach, each virtual node has its own random identifier in the ring. This enhancement provides a more uniform coverage of the identifier space, thus improving load balance across system nodes(Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001). The downside of this optimization is that each physical node needs to store information (routing tables, etc.) for each of its virtual nodes.

2.7 Distributed Storage Systems

As storage systems grows, it becomes necessary to distribute data across a set of nodes to improve the system's capacity and performance while attending application's requirements. This is particularly important in a cloud storage environment, where the storage system needs to deal with a constantly growing amount of data from different users, and possibly organizations.

In this section we present and discuss several designs and solutions for data storage in a distributed environment. These systems provided an important theoretical background on building robust distributed storage systems that was useful for designing the solution proposed in this thesis.

2.7.1 Distributed File Systems

The initial effort to distribute storage aimed at providing access to files over a network, in a manner similar to local file storage. The most prominent examples of distributed file systems (also called network file systems) include Network File System (NFS) (Sandberg, Goldberg, Kleiman, Walsh, & Lyon 1985), Andrew File System (AFS) (Howard 1988) and Coda (Kistler & Satyanarayanan 1992). These systems follow a client-server approach where the client communicates with one or a set of servers through a well defined RPC protocol. They are also based in

the assumption that most files are read and written by a single user, so they are not optimized for concurrent access of the same files. Moreover, they heavily rely on caching to improve performance, based on the temporal locality of reference (if a file was recently referenced, it will probably be referenced again in the near future).

Network File System[9] (Sandberg, Goldberg, Kleiman, Walsh, & Lyon 1985) was designed to simplify sharing of file system resources in a workgroup composed of heterogeneous machines in an efficient, transparent and portable way. The NFS protocol doesn't distinguish clients and servers explicitly, so any "peer" can export some of its files, or access files from other peers. However, a common practice is to configure a small number of nodes to act as dedicated servers. NFS defines a stateless protocol, so all client requests must contain necessary information to satisfy the request. According to the authors, this choice simplifies crash recovery, since a client can just retry an operation when the server crashes, and the server does not need to maintain contextual information about clients. However, stateful operations such as file locking should be handled by a separate service outside NFS. The protocol does not enforce a global namespace between different servers, so there is no guarantee that all clients will have a common view of shared files. This is typically achieved by manually configuring workstations to have the same namespace.

NFS performs in-memory, block-level caching. The protocol defines a simple cache coherence mechanism based on timestamps: each file has a last modified timestamp and its blocks might be cached by clients for a given timeout T. Any reference to a cached block after T must force a validation check, where the local timestamp is compared to the server timestamp. In case a newer version is detected, the local cache is invalidated. However, this caching mechanism does not guarantee consistency across shared copies of a file, since a file may be modified in different clients during the timeout period. Moreover, verifying cache consistency against the server affects the scalability of the system, due to the increased network traffic for cache validation messages. Writes on cached blocks are asynchronously sent to the server before the file is closed.

The **Andrew File System** (Howard 1988) provides a global distributed file system with primary focus on scalability (up to tens of thousands of nodes). Differently from NFS, AFS is managed by a stateful server protocol. This protocol defines two namespaces: a shared namespace, which is identical on all workstations and a local namespace which is unique to each

[9]The original version of NFS described in Sandberg et al, 1985.

workstation. The shared namespace is partitioned into disjoint subtrees, and each subtree is statically assigned to a single server.

As opposed to NFS, AFS performs file-level caching on disk. Cache coherence is done through client callbacks: when the client caches a file from the server, the server promises to inform the client if the file is modified by someone else. Read and write operations are directed to the cached local copy, and if the cached copy is modified it is only sent to the server after being closed. These mechanisms substantially improves AFS's scalability by greatly reducing client and server interactions. Furthermore, AFS allows files to be replicated on read-only servers, improving the system's availability and load balancing. Several of AFS's features, such as cache callbacks and a stateful protocol, were integrated into NFS 4.0 (Pawlowski, Noveck, Robinson, & Thurlow 2000) to increase the protocol's scalability.

Coda (Kistler & Satyanarayanan 1992) significantly improves AFS's availability by allowing the system to operate even when servers fail or the client is disconnected from the network. This is done by replicating volumes across multiple servers and by using an optimistic replication strategy based on logical timestamps. On read operations, the client may contact any of the available servers to fetch an uncached file, but must check with other servers if a more recent version of the file is available. The server that receives the write operations must propagate the update to all available servers before completing the operation.

Coda uses vector clocks (Fidge 1988; Mattern 1989) to keep track of the update history of files on different servers. Clients may operate on cached files when no servers are available (due to network partitioning or disconnected operation), hoping that the file will not be modified by someone else. Replica conflicts are detected when vector clocks for a particular file diverge. In this case, the client is informed there is a conflict and must take an action to solve it. Since the client may be working on a stale version of the file, because of network partitioning or disconnected operation, the consistency guarantees are weaker than in AFS (eventual consistency). The cache coherence protocol of AFS is extended to work in a replicated environment.

Farsite (Adya, Bolosky, Castro, Cermak, Chaiken, Douceur, Howell, Lorch, Theimer, & Wattenhofer 2002) proposes a server-less distributed file system in order to overcome some of the limitations of server-based file systems like acquisition and maintenance costs, vulnerability to localized faults, vulnerability to unreliable, insecure or untrusted administration, etc. This is done by leveraging unused resources of insecure and unreliable client desktop machines to

provide a secure, reliable and logically centralized file system. Farsite ensures data privacy and integrity by using many cryptography techniques .Availability, scalability and reliability is ensured by file replication. Byzantine fault tolerance protocols are leveraged on Farsite to provide security and resistance to Byzantine threats. Moreover, Farsite employs single instance storage deduplication to reclaim space from duplicated files (Douceur, Adya, Bolosky, Simon, & Theimer 2002).

The **Google File System** (GFS) (Ghemawat, Gobioff, & Leung 2003) provides a large scale distributed file system optimized for Google's data processing requirements. Some of these requirements include working with very large files (from a few hundred megabytes to multi-gigabytes), dealing with workloads consisting of large streaming reads, small random reads, as well as many large sequential writes. In this environment, traditional file system's design choices, like having small blocks sizes and leveraging locality-based caching, are not optimal. Additionally, the system is tailored to work in clusters of commodity machines, where failures are norm rather than exception.

GFS has a simple distributed architecture with a single replicated master, that hosts the system's metadata, and several chunk servers that store the system's data. The master manages file and chunk namespaces, the mapping from files to chunks, and the locations of chunk replicas (the latter is also maintained by each chunk server). This metadata is kept in memory to improve system's performance, but the first two metadata types are persisted in an operation log for fault tolerance. Moreover, the master structures are replicated to the secondary master allowing the system to be recovered in the event of a crash. The choice of a single replicated master aims to increase system's simplicity, flexibility and performance. However, the main drawback of this approach is that the number of chunks, and hence the system's total capacity, is limited by the amount of memory the master has.

Files on GFS are split into 64MB fixed-size chunks, that are replicated to achieve availability. The master decides in which chunk servers chunks are replicated based on system load and rack placement, in order to improve resource utilization and fault tolerance, respectively. The client only contact the master to fetch chunk replica locations and to update metadata (e.g. create a file). All remaining data operations are handled entirely by chunk servers. This choice reduces client interactions with the master, making the system operational despite relying on a single master. Chunk servers are granted leases to be primary chunk replicas that must be renewed

periodically with the master. Primary replicas coordinates write operations with secondary replicas and define a total order on atomic operations. Data transfers are pipelined across chunk servers to maximize network bandwidth utilization.

As we previously discussed, object-based storage systems were proposed to overcome some of the scalability limitations and bottlenecks of traditional file systems. While the systems discussed in this section offer interesting insights on distributed storage, they do not fit into the object-based storage paradigm. Furthermore, while NFS is clearly not scalable to meet capacity requirements for a cloud-based storage system, AFS also does not have sufficient mechanisms for achieving necessary availability and durability requirements. Coda improves availability and durability of AFS, but still relies on static assignment of storage volumes to nodes, which is unfeasible in the context of a large scale system and leads to poor load balancing across nodes.

On the other hand, GFS and Farsite dynamically distribute their workload in a fault tolerant and scalable cluster of storage nodes. Moreover, they rely on replication and other techniques to improve the system's availability and reliability. However, these systems were designed for specific environments or workloads, and thus, are not optimized for general purpose cloud-based object storage. For instance, Farsite was designed to be executed on untrusted desktop clients, while GFS is optimized for large scale data processing applications.

2.7.1.1 Other distributed storage systems

In this section we present some relevant distributed storage solutions, with design focus on scalability, availability and reliability.

Dynamo (Hastorun, Jampani, Kakulapati, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007) is a distributed key-value storage system that aims at providing a highly reliable and available storage system to support Amazon's internal services, including Amazon S3 (Vogels 2007). Dynamo leverages several distributed system's techniques in order to build a system where all customers have a good experience, since a service disruption may have serious financial consequences. The system is based on a very simple interface based on *get* and *put* operations on key/value pairs.

Dynamo achieves incremental scalability by using consistent hashing to partition data across a dynamic set of nodes. Data is replicated into a configured number of servers (N) to ensure

availability and durability. A quorum mechanism is employed on every operation to maintain eventual consistency among replicas. This mechanism is based on two variables, R and W, which indicate the number of nodes that must be consulted before returning a read or write operation, respectively. By tuning the values of R, W and N, such that $R + W > N$, application designers can configure Dynamo based on trade-offs between performance, availability, cost efficiency and durability. Vector clocks are used to capture causality relationships between replicas. Conflicting updates are detected during reads (when two replicas are causally unrelated), and the client must reconcile them using an application-specific logic.

Dynamo uses a decentralized failure detection and ring membership protocol based on gossip, maintaining the system's management essentially decentralized. When nodes fail temporarily and a quorum cannot be formed, replicas are written to handoff nodes to ensure system's availability. When permanent node failures happen, the system employs a replica synchronization algorithm based on Merkle trees to recover failed replicas. We employ many of the techniques used in Dynamo, like consistent hashing, replication and quorum-based eventual consistency, to ensure scalability, reliability and availability of the deduplicated cloud storage solution presented in this thesis.

Google **BigTable** (Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber 2008) is a distributed storage system for structured data designed to scale to petabytes of data across thousands of commodity clusters. BigTable has a simple tabular query model based on 3 values: a key value, a column value and a timestamp. Bigtable is based on a single master architecture, replicated for availability and fault tolerance. Tables are dynamically partitioned on row ranges (called tablets) and stored in a cluster of tablet servers. Tablets are persistently stored on a GFS instance running on each tablet server. The master is responsible for assigning tablets to tablet servers, balancing system's load and garbage collecting files on GFS.

Bigtable uses a highly available and persistent distributed lock service called Chubby (Burrows 2006), which implements the Paxos algorithm (Lamport 1998). Chubby is used for many tasks, including master election, storing system's metadata and storing the bootstrap location of BigTable data. The tablet location information is stored in a 3-level hierarchy structure similar to a B+ tree. Chubby stores the location of the root tablet (which is unique), the root tablet indexes several metadata tablets, and each metadata tablets index several user tablets. This simple hierarchical structure allows the system to accommodate a sufficiently large number of

tablets. Moreover, clients only need to contact tablets servers for data operations, allowing the single master architecture to scale.

Cassandra (Lakshman & Malik 2010) is another distributed structured storage system which offers a tabular query model similar to BigTable. However, differently from BigTable, Cassandra partitions its tables across multiple nodes in a manner similar to Dynamo. Since tables in Cassandra are ordered by key, it uses an order-preserving hash function to allow queries to be efficiently answered. Durability and availability are achieved by replication, and each node stores table slices on its local disk. For brevity we omit more details on Cassandra's architecture, given its similarity with Dynamo, which was already discussed.

While BigTable and Cassandra offer interesting perspectives on distributed data storage, we do not directly leverage these systems in the proposed solution, since they're targeted at other purposes.

Conclusion

In this chapter we introduced relevant theoretical background for understanding the problem addressed in this work and explored the most relevant research work that served as inspiration for designing the solution proposed in this thesis.

We started by defining cloud computing and discussing different service and deployment models offered by this paradigm, in order to provide a contextualization of the environment in which this thesis is contained. Then, we explored the object-based storage paradigm, which solves some of the scalability limitations of traditional file systems by providing a simpler storage model and delegating low level decisions to the storage device, thus enabling a greater degree of metadata decoupling and data sharing. This paradigm was adopted as one of the central storage models for cloud computing: cloud-based object storage. We have described an object storage service that is considered a de-facto standard for this model, Amazon S3, where over a trillion objects are currently stored, what reinforces the need for a deduplicated cloud-based object storage solution. Furthermore, we have seen that most of the solutions implementing this type of service is proprietary, and none of the open solutions currently provide support to deduplication. After that, we presented the most important techniques for data deduplication, with examples of systems that employ each of the described techniques. Next, we discussed and

presented examples of content-addressable storage (CAS), an important technique employed in our solution that enables automatic data deduplication by addressing data according to its contents. However, most of the available CAS systems are either aimed at data archiving, do not attend scalability requirements necessary for a cloud deployment or are proprietary, and thus are not suited for implementing a deduplicated cloud-based object storage solution. Finally, we explored different techniques to build scalable and reliable distributed storage systems by describing and discussing important previous research work in the field of distributed storage.

The next chapter describes the architecture and implementation of the proposed solution for a distributed object-based storage system with deduplication support.

3 Cloud-based Deduplicated Object Storage

3.1 Introduction

This chapter presents a distributed object storage with built-in support for deduplication. The proposed solution is inspired by previous distributed storage works, such as Dynamo (Hastorun, Jampani, Kakulapati, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007) and Ceph (Weil, Brandt, Miller, Long, & Maltzahn 2006), as well as content-addressable storage systems, such as Venti (Quinlan & Dorward 2002) and CASPER (Tolia, Kozuch, Satyanarayanan, Karp, Bressoud, & Perrig 2003), creating a bridge between these two areas of storage technology.

We start in section 3.2 by listing the main system's requirements, extracted from discussions in previous chapters. The main contribution of this thesis, an architecture for a distributed object storage system with deduplication support is presented in section 3.3. Finally, in section 3.4 we discuss the implementation of the proposed solution as an extension to OpenStack Object Storage, an important open source solution for object storage in a cloud environment.

3.2 System Requirements

In this section we present the main requirements that were considered when designing the solution. These requirements were extracted from insights and discussions presented in previous chapters.

3.2.1 Functional Requirements

Inline Object-level deduplication

- A user must be able to choose an option to deduplicate an object as a whole when inserting it into the system.

  - When a new object is being inserted, if it is detected that the object is already stored, the system must not create an additional copy of the object, but instead return a reference to the already stored copy.

- The object must become available **after** the deduplication process is completed.
- Deduplicated objects must be retrieved in the same way as a non-deduplicated object.

Inline Chunk-level deduplication

- A user must be able to choose an option to perform chunk-level deduplication when inserting an object into the system.
  - For each new chunk of an object inserted, if it is detected that the chunk is already stored, the system must not create an additional copy of the chunk, but instead return a reference to the already stored copy.
- The object must become available **after** all chunks are transferred, and the deduplication process is completed.
- The system must allow clients to perform transfer of chunks in parallel in order to improve performance.

Flexible chunking algorithm

- The system must not impose restrictions in the method by which objects are chunked when chunk-level object deduplication is selected.
  - A default chunking method must be supported.
  - Applications may use their own application-specific chunking algorithm to yield better deduplication ratios if they wish to do so.

Flexible cryptographic hash function

- The system must support multiple secure cryptographic hash function to fingerprint entities.

  - One of the supported hash functions must be selected to perform fingerprinting of objects and chunks in the system.
  - The selected algorithm may be replaced, at the cost of having to remap all stored objects according to the new algorithm.

Flexible chunk storage

3.2.2 Non-Functional Requirements

Scalability

- The system must be able to horizontally scale storage and processing capacity to hundreds of nodes without disruption of service.

Reliability

- The system must tolerate the failure of a configurable number **N** of storage devices without data loss.

Availability

- The system must still operate given that a configurable number of storage nodes are not accessible.

Consistency

- When concurrent operations are executed on the same object, eventual consistency must be achieved.
- In case a conflict happens, the "last write wins" policy must be ensured.

Performance

- An operation done to an object that has been deduplicated as a whole must present latency and throughput performance similar to when no deduplication is used.

- When doing chunk-level deduplication, the introduction of some performance overhead is acceptable when compared to the same operation done to a non-deduplicated object.

3.3 System Architecture

In this section we will describe in detail the proposed architecture for a distributed object storage system with deduplication support.

3.3.1 Overview

The architecture is based on three components: a distributed content-addressable storage, a distributed metadata service and a client library. Figure 3.1 shows an overview of this architecture.

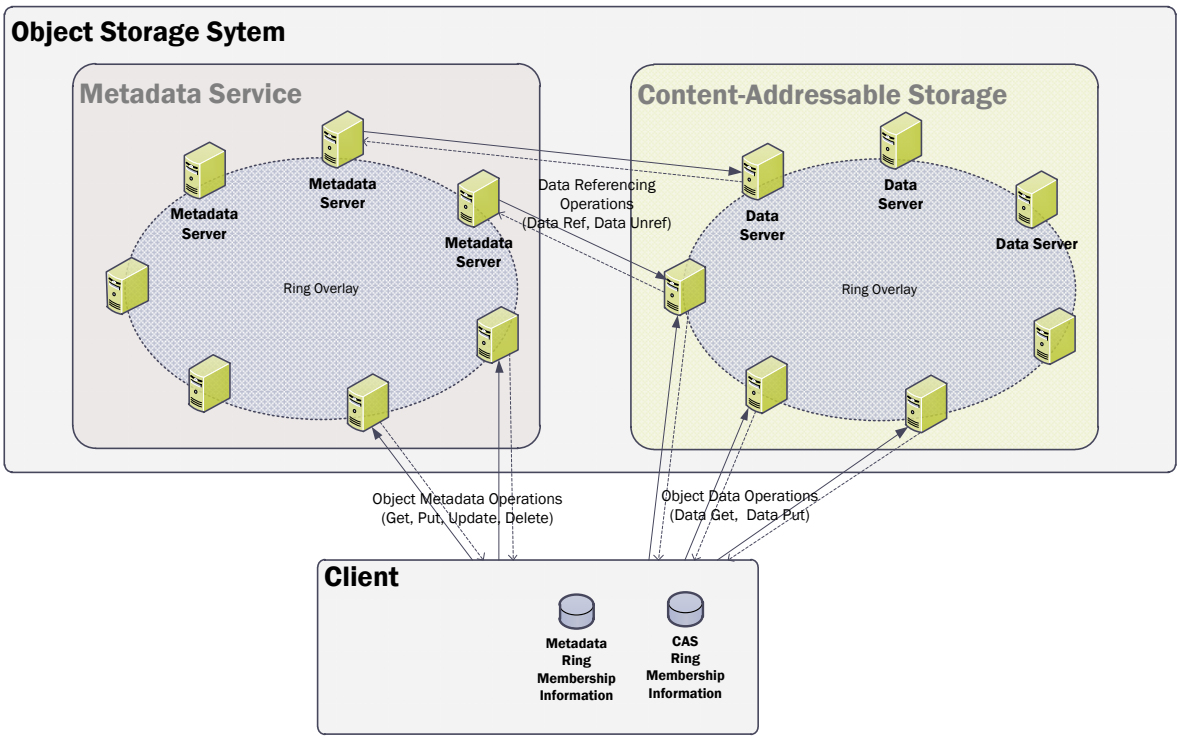


Figure 3.1: Architecture of a distributed object-based storage system with support for global deduplication

The content-addressable storage indexes data according to a cryptographic digest of its contents. This enables global distributed deduplication, because duplicated data maps to the

same node according to a distributed lookup function and it's only stored a controlled number of times (the replication factor).

The metadata service keeps information about the objects stored in the system, such as attributes (owner, modification date, etc) and references to data chunks stored in the data servers. The metadata of an object is indexed by a simple key attribute in a flat namespace, which is the object identifier.

Finally, a client library is responsible for implementing the deduplicated object storage protocol. It is important to note that a client library is not necessarily used by a human user, but most likely by another system. Examples of systems that could use this client library are: backup systems, file sharing applications, file system wrappers, document editing tools, a proxy or gateway server exposing storage functionality to end users, etc.

3.3.2 Design Decisions and Considerations

3.3.2.1 Scalability and Data Distribution

Each service is composed of a set of servers distributed in a structured overlay ring in order to provide data distribution and scalability. Each server can be located according to a distributed lookup function, which maps a data or metadata identifier to the node responsible for storing that entity. This function should preferably be a variant of consistent hashing in order to decrease the number of necessary data movements when a node fails.

This choice of design is based on previous works like Dynamo and distributed hash tables, which were discussed in the related work chapter.

3.3.2.2 Decoupling of Data and Metadata

One benefit of the separation of data and metadata handling in two separate sets of servers (that can be on the same physical host) is that it allows each type of server to optimize its operation according to the data type it stores. For instance, one possible optimization in the metadata servers is to store its data in in-memory databases or Solid State Drives (SSDs) to provide faster access performance, since typically the metadata comprises a very small fraction

of the total data size. The data server can leverage techniques for improving efficiency of data lookup and caching of popular data.

We later discovered that decoupling of data from metadata handling in different distributed servers is a good practice to increase the system's scalability which is employed by storage systems like Ceph and NFSv4.1 (Shepler, Noveck, & Eisler 2010).

3.3.2.3 Replication and Request Coordination

Data durability, availability and fault-tolerance is achieved through active replication of data and metadata to a defined number **N** of servers (replicas). In this method of replication, the same request is sent to N replicas that process each request independently. This enables the system to tolerate N-1 simultaneous node failures without data loss. Request coordination can be done by the client library or by the storage nodes. This coordination involves setting the operation timestamp and replicating data to N servers on each write operation.

When request coordination is done by the client write operations may be faster, but more bandwidth is required to simultaneously send data to all servers. Furthermore, the clocks of the servers and the client need to be synchronized to ensure replicas will be consistent (more on next section).

Another alternative is to send the request to one of the storage nodes who will perform request coordination on behalf of the client. In this case, the storage node that receives the request will timestamp it and send it to the remaining servers. This may add latency to write operations, but the internal network bandwidth (which is typically larger) can be better utilized.

3.3.2.4 Consistency Model and Concurrency Control

The system employs an eventual consistency model for object metadata based on optimistic replication. In this model, replicas of the same object may diverge but will eventually be consistent once all updates are propagated. Each metadata service is a replica manager and implements a "last write wins" conflict resolution policy based on physical clocks' timestamps. The entity that performs request coordination (the servers or the client) must have the clock synchronized with storage servers to ensure conflict resolution will be correct. This can be done

by a tool that implements a distributed network synchronization protocol, such as NTP (Mills 1985)).

The client uses a simple quorum mechanism to guarantee that reads and writes of object metadata are consistent. A metadata read or write operation is considered completed after R or W nodes have successfully replied. The client may tune R and W according to application's requirements, such that $R + W > N$. For instance, a read-intensive application could set $R = 1$ and $W = N$. A default value of R and W is $\frac{N}{2} + 1$, which yields a simple majority quorum. It is possible to relax consistency guarantees by reading from only one server, but stale metadata may be returned.

A major benefit of content-addressable storage is that it greatly simplifies concurrency and consistency control because it deals only with immutable data. Thus, changing the contents of a data segment will also change its identifier and consequently its storage location. So, it is impossible for two replicas of a data chunk with the same identifier to diverge. However, a quorum mechanism is is still used to ensure data is written to a minimum number of replicas before completing the operation to ensure durability. A client can read from one data replica, and if it exists it can be guaranteed to be consistent.

3.3.2.5 Ring Membership

The system architecture does not define how ring membership should be handled. For complete decentralization, a dynamic membership algorithm may be used, such as the algorithms used in Distributed Hash Tables, where each node keeps track of its neighbors in the ring and partitions are automatically reassigned when nodes enter or leave the system. In Dynamo, each node maintains a global view of membership state, and propagates its view to other nodes using a gossip algorithm.

Alternatively a static ring membership structure might be constructed with permanent ring information. This structure must be distributed to all nodes every time permanent membership changes. Even though this approach is suitable for small deployments, it may become a problem in very large deployments, where node replacements are very frequent.

3.3.2.6 Request Routing

Request routing to servers is done through a one-hop lookup mechanism to avoid the introduction of routing delay during requests. This requires the client to maintain updated membership information about all nodes participating in the ring. One of way of doing this is by querying the network for membership information every T seconds. Since the system is used in a controlled environment, where node churn is low, keeping routing tables updated will not add significant overhead to a system with hundreds, or even thousands, of nodes, as shown in (Gupta, Liskov, & Rodrigues 2003).

3.3.2.7 Security, Authentication and Data Integrity

We assume that the solution will be used in a trusted environment, with trusted clients and servers, such as a private cloud. Similarly, the architecture does not explicitly handle authentication of clients for simplicity purposes. However, it is straightforward to add an authentication layer on storage servers that verifies if a request is authenticated before executing an operation.

Since the architecture assumes a trusted environment, an optimization would be to add an authentication layer only on metadata servers, to enforce accounts and access permissions on objects. Data operations could remain unauthenticated, because in content-addressable storage it is necessary to have the data fingerprint to insert or retrieve a piece of data. This fingerprint can only be discovered by having the piece of data or accessing an object's metadata (given a collision-free hash function is used).

Data integrity is not handled explicitly by the system. However, the use of content-addressable storage, where data is identified by its content digests, greatly simplifies the implementation of data integrity checks. For instance, a lazy integrity check mechanism could verify if the data fingerprint matches the stored content every time it is fetched. In case a mismatch is detected, the data is marked as corrupted and a background process recovers the original data from healthy replicas.

3.3.2.8 Fault Tolerance and Replica Synchronization

The system will operate as long as the number of accessible replicas is equal to the eventual consistency quorum (R for reads or W for writes). However, doing replication to a smaller

number of nodes will affect data durability and availability, and a mechanism is necessary to ensure these properties in the face of failures. In case the number of failed or unaccessible replicas (due to network partitioning) prevents a quorum from being formed, writes are not possible and reads may return stale data.

One way of handling temporary failures is by employing a hinted handoff mechanism, as done in Dynamo. In this mechanism, if one of the replica nodes is unavailable, the replica is written to another healthy node, who will later transfer the replica to the correct node when it becomes available. In order to tolerate permanent failures and replication errors, a background process must be triggered to recover missed replicas. Dynamo uses an anti-entropy protocol (Merkle trees) to detect failed replicas and perform replica synchronization that fixes any replication divergence (such as one caused by a failed node).

In our prototype we haven't implemented such features for time constraints, however. For this reason, we assume a scenario without node failures in the experiments.

3.3.2.9 Choice of Hash Function

A secure cryptographic hash function must be used to fingerprint data before it is inserted in the storage system. In order to increase deduplication efficiency, a single function must be used system-wide, in all clients and storage servers. Since the storage system will handle at least tens of terabytes and possibly petabytes of data, it is important that the chosen function is:

- **Collision-resistant:** It should be very difficult for two different data chunks to have the same fingerprint, otherwise this could cause data-loss since the system is based on the assumption that each piece of data has a unique fingerprint.
- **Preimage-resistant:** It should be very difficult to discover the fingerprint of a data chunk without having the actual chunk. Since the system doesn't authenticate data operations, a malicious user could fetch unauthorized pieces of data by guessing its fingerprint.
- **Second-preimage-resistant:** It should be very difficult to discover two data chunks with the same fingerprint. Since the system is based in the assumption that each data chunk has a unique fingerprint, a malicious user could store bogus data with a valid fingerprint that is the same as popular data chunk (such as a song or VM image), preventing users to store or retrieve the valid data piece.

Examples of algorithms that currently meet these requirements are SHA-1 and the SHA-2 family of functions (SHA-224, SHA-256, SHA-384 and SHA-512). However, theoretical attacks against SHA-1 more efficient than brute-forcing were published in (Manuel 2008; Cochran 2007) even though they were not yet successful in practice. The MD5 algorithm is not feasible for data fingerprinting since its not collision-resistant.

The system has a configuration parameter that defines which secure cryptographic hash function is used.

3.3.2.10 Data Chunking

An important step in the deduplication process is data chunking, since it is directly related to the efficiency of redundancy detection. While fixed-size chunking can be efficient for static data types, such as media types and virtual machine images, variable content-defined chunking is more efficient for duplicate detection of dynamic files, like documents. An alternative approach is to deduplicate objects as a whole, when repeated occurrences of the same object are common in a dataset. One example where whole-object chunking yields a good deduplication efficiency is software distribution, where only a fraction of the files changes between different releases of a particular software.

In order to support a wide-range of applications, the client must have a pluggable chunking component, that can be changed according to an application's requirements and be extended to support new chunking techniques. In our prototype implementation we currently support two chunking techniques: fixed-size chunking and whole-object chunking.

3.3.2.11 Object Metadata Representation

In hash-based deduplication, an object is split into one or many chunks of data and each of these chunks is stored in the content-addressable storage and indexed by its content's fingerprint. In order to allow the reconstruction of the original object from its content-addressed chunks, a metadata representation that lists the chunk fingerprints that compose the object is necessary.

Tolia et al. (Tolia, Kozuch, Satyanarayanan, Karp, Bressoud, & Perrig 2003) proposed file recipes, which is a file synopsis containing a list of data block identifiers that compose the file

in the context of the CASPER distributed file system. Each data block identifier is called an "ingredient", and is a hash digest of the block's contents.

In this solution we propose the use of a variant of this approach to represent objects. The object recipe is a metadata file that maps an object in the flat key namespace (object identifiers) to chunks in the content-addressable storage namespace. In addition to an ordered list of chunk identifiers, the object recipe must have the following attributes:

- **Name** The name of the object in the flat key namespace.
- **Size** The total size of the object in bytes.
- **Chunking algorithm** The chunking algorithm used to divide an object into chunks.
- **Default chunk size** An optional attribute defining the default chunk size (in bytes) if fixed-size chunking is used. For variable-sized chunking, each chunk descriptor must specify its size.
- **Fingerprinting algorithm** The cryptographic hashing algorithm used to fingerprint data chunks.

In addition to default attributes, applications must be able to append to the recipe file their own application-defined attributes.

3.3.3 System parameters

The system must define the following configuration parameters:

- **Data lookup function:** A variant of consistent hashing that takes a hash number and returns a set of data nodes.
- **Metadata lookup function:** A variant of consistent hashing that takes a hash number and returns a set of metadata nodes.
- **Fingerprint hash function:** Function used to fingerprint chunks.
- **Object identifier hash function:** Function used to hash object identifiers.

- **Replication factor (N):** Number of times data and metadata must be replicated.
- **Metadata read quorum (R):** Minimum number of responses when reading an object metadata. ($R + W > N$)
- **Metadata write quorum (W):** Minimum number of responses when writing an object metadata. ($R + W > N$)
- **Data durability level (D):** Minimum number of responses before considering data written. ($1 < D <= N$)

3.3.4 Client Library

In this section we describe the functionality and operation of the client library, which is the system component that exposes deduplicated object storage functionality to other applications.

It is assumed that the client will execute in a trusted location and correctly execute the system's protocol. This assumption is more easily met in a private cloud context, where managers and administrators have control over where the client library is executed. In the context of a public cloud, the client can be part of a trusted proxy server that forwards external requests to internal storage nodes.

3.3.4.1 Client parameters

In addition to the system parameters, the client must define the following configuration parameter:

- **Chunking algorithm:** The algorithm used to split an object into chunks.

3.3.4.2 Client Operations

The proposed Application Programming Interface (API) of the client library exposes the following operations:

- **Insert Object (put object):** Inserts a new binary object into the object storage with a given identifier. This operation will overwrite previously stored objects with the same identifier.

- **Retrieve Object (get object):** Retrieves an object with a given identifier from the object storage.
- **Remove Object (delete object):** Removes an object with a given identifier from the object storage, making it unavailable.

Each of the operations is described in the following subsections.

Procedure 1 Client - Put Object (Client Coordination)

Input: *Object, Object Identifier*
Output: *Object Recipe*

```
1:  objid_hash ← id_hash(object_id)
2:  obj_uid ← objid_hash + unique_version_id()
3:  init recipe
4:  for all chunks ⊂ Object do                                   ▷ Put data
5:      fingerprint ← data_hash(chunk)
6:      data_nodes ← data_lookup(fingerprint, replication_factor)
7:      for all data_node ⊂ data_nodes do
8:          send(data_node, PUT, fingerprint, obj_uid, chunk)
9:      end for
10:     Wait durability_level responses
11:     append(fingerprint, recipe)
12: end for
13: metadata_nodes ← metadata_lookup(objid_hash, replication_factor)   ▷ Put metadata
14: ts ← clock.timestamp()
15: for all metadata_node ⊂ metadata_nodes do
16:     send(metadata_node, PUT, object_id, ts, recipe)
17: end for
18: Wait write_quorum responses
19: return recipe
```

Insert Object (Client Coordination)

An insert or put operation adds a new object to the object storage. Since the previous stored object with the same identifier is replaced by the new object, the put operation also updates an existing object. In this section we describe the procedure for inserting an object when the client performs the request coordination, as presented in Procedure 1.

The procedure starts by generating a unique object version identifier that will be used as a back-reference on data put operations (lines 1-2). Back-references are used on data servers for garbage collection of data chunks. On line 3, a metadata recipe with the objects' attributes

is initialized. In the loop of lines 4-12, the object is chunked according to the chosen chunking algorithm (fixed, variable or whole-object). Each chunk is fingerprinted by the secure hash function and this fingerprint is used to lookup the servers responsible for storing this chunk in the data ring (lines 5-6). The request must be sent simultaneously to $replication\_factor$ servers, who will ask for chunk data only if that chunk is not already stored (lines 7-9). The client must wait for *durability_level* responses (line 10) before considering the chunk inserted and appending its fingerprint to the object recipe (line 11).

After all chunks are put, the object metadata is updated on lines 13-18. The hashed object identifier is used to locate the nodes responsible for storing that object in the metadata ring (line 13) and the current timestamp is retrieved from the physical clock (line 14). The recipe is sent simultaneously to $replication\_factor$ metadata nodes including the timestamp, which will be used by the metadata servers for conflict resolution (lines 15-17). The metadata update is completed after *write_quorum* responses (lines 18). Finally, the object recipe is returned to the application in case it wants to cache it for future use (line 19).

Insert Object (Server Coordination)

Another option when inserting an object is to delegate request coordination to storage servers. In this case, the procedure is essentially the same as shown in Procedure 1. The main differences are the loops on lines 7-9 and 15-17: instead of sending data and metadata PUT operations simultaneously to *replication_server* servers, the operation is sent to only one of the servers responsible for the data or metadata entity. The server that receives the operation then timestamps the request and perform replication to the other servers.

Retrieve Object

After an object is stored in the system, the client may retrieve it using its identifier according to procedure 2. The overall approach is to get the object recipe in the metadata servers using the hash of the object ID as a lookup key (lines 1-7), and then fetch each of the object's chunks from the data servers using their fingerprint as lookup key (lines 8-13).

The metadata get operation is sent to $replication\_factor$ nodes and the responses are aggregated in a list of responses (lines 4-6). The operation is completed when *read_quorum* responses

Procedure 2 Client - Get Object

Input: *Object Identifier*
Output: *Object*

1: $objid\_hash \leftarrow id\_hash(object\_id)$ ▷ Get metadata
2: $metadata\_nodes \leftarrow metadata\_lookup(objid\_hash, replication\_factor)$
3: **init** *responses*
4: **for all** $metadata\_node \subset metadata\_nodes$ **do**
5: $responses \leftarrow send(metadata\_node, GET, object\_id)$
6: **end for**
7: $recipe \leftarrow max\_timestamp(Wait$ read_quorum $responses)$
8: **init** *object*
9: **for all** $chunk\_fingerprint \subset recipe$ **do** ▷ Get data
10: $data\_node \leftarrow data\_lookup(chunk\_fingerprint, 1)$
11: $chunk \leftarrow send(data\_node, GET, chunk\_fingerprint)$
12: $append(chunk, object)$
13: **end for**
14: **return** *object*

are retrieved since the object can be in a inconsistent state because of a failed or concurrent put operation (line 7). From these responses, the one with the maximum timestamp is considered. When a client does not mind getting stale or inconsistent data, the get metadata operation may be sent to only one metadata server.

Differently from metadata, the data get operations only need to be sent to one data node per chunk (lines 9-13), because a stored data chunk is guaranteed to be in a consistent state due to its write-once nature. Each new data chunk that is get is appended to the object container (line 12), and after all data is fetched the reconstructed object is returned (line 14).

Remove Object

The algorithm to remove an object from the object storage system is very simple, as shown in Procedure 3. Initially, the hash of the object identifier is used to look up the metadata servers where the object recipe is stored (lines 1-2). A delete message with the object identifier and the current timestamp is sent to each of the $replication\_factor$ nodes (3-6). The timestamp is necessary for conflict resolution, in case delete and put operations are concurrent. The operation is completed after $write\_quorum$ responses are received.

One important aspect of the delete operation is that only metadata servers need to be contacted. The data chunks are not deleted immediately because they might be shared by

other objects. Instead, a background process in the metadata servers unreference the chunks referenced by the deleted object. If a chunk is not referenced by any object for a long time, it is eventually deleted by a garbage collection process running in the data servers.

Procedure 3 Client - Delete Object

Input: *Object Identifier*

1: $objid\_hash \leftarrow id\_hash(object\_id)$ ▷ Remove metadata
2: $metadata\_nodes \leftarrow metadata\_lookup(objid\_hash, replication\_factor)$
3: $ts \leftarrow clock.timestamp()$ ▷ Put metadata
4: **for all** $metadata\_node \subset metadata\_nodes$ **do**
5: $send(metadata\_node, DELETE, object\_id, ts)$
6: **end for**
7: $Wait$ write_quorum $responses$ ▷ Data is unreferenced by metadata servers

3.3.4.3 Parallelization

It is possible to improve performance of object "puts" and "gets" by processing data in parallel, since chunks within an object are stored independently from each other in different data servers.

In object puts (Procedure 1), the loop of lines 4-12 can be delegated to multiple threads, so fingerprint calculation and chunk upload is done in parallel. The implementation must take care to place fingerprints in the correct order in the object recipe, to enable the object to be correctly reconstructed from the recipe. A wait barrier must be placed before line 13 to ensure that the object recipe is only inserted after all chunks have been successfully upload.

In object gets (Procedure 2), chunk downloads can be done in parallel (lines 9-13). In this case, the implementation must care to place chunks in the correct order when reassembling the object.

3.3.5 Data Servers

The data servers implement a distributed content-addressable storage, where each piece of data is indexed by a hash digest of its contents, also called fingerprint. The fingerprint is used by the consistent hashing algorithm to route data operations to a subset of the data nodes. Thus, operations in the same chunks of data maps to the same set of data servers, enabling global deduplication of data. The components that compose a data server are shown on Figure 3.2.

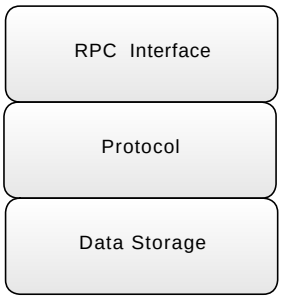


Figure 3.2: Data Server internal components

The remote procedure call (RPC) interface exposes the data server API to clients over a network. The protocol component implements the procedures described in this section. The storage component performs on-disk operations for data reading and writing. This component can be as simple as a file system wrapper that writes each data chunk to a different file to a storage folder on disk, or a complex component that leverages caching, on-memory indexes and disk structure knowledge to speed up data operations.

3.3.5.1 Data Server Operations

The proposed API of data servers has the following operations:

- **Insert Data Chunk (data put):** Stores a binary chunk of data of arbitrary size if it's not already stored (write-once). This operation adds a back-reference to the object that is referencing the inserted data.
- **Retrieve Data Chunk (data get):** Retrieves an binary chunk of data by its fingerprint.
- **Unreference Data Chunk (data unref):** Removes a back-reference from a data chunk.

Each of the operations is explained in the following subsections.

Insert Data Chunk

The Insert Data Chunk (data put) operation shown on Procedure 4 stores a new data chunk in the content addressable storage if it was not previously stored. It also creates a back-reference to the object that references this data chunk for data deletion purposes.

Initially, the data server tries to lookup the data with the given fingerprint to check if that data chunk is already stored in the server (line 2). In case the chunk is already stored, a back-reference for the object that is referencing this piece of data is created (line 3). This back-reference is a unique object version identifier that can later be removed if the chunk is no longer referenced by the object. The operation is completed on line 4 and the client does not need to send the data over the network.

When the data chunk is not present in the data store, the server requests the chunk's contents to the client (line 6). After the data transfer is complete, the server verifies if the fingerprint matches the hash that was computed for the data, to prevent a corrupted or malicious chunk to be stored with an incorrect fingerprint (line 7). Finally, the data is written to the data storage (line 8) and the back-reference to the object is created (line 9). If no error occurs, a success response is returned to the client (line 10).

Procedure 4 Data Server - Data Put

Input: *Data Fingerprint, Object Unique Version Identifier, Client Callback*
Output: *SUCCESS if the operation was executed successfully*

```
1:  ds ← DataStorage
2:  if ds.lookup(fingerprint) then
3:      ds.create_reference(fingerprint, object_uid)        ▷ Data is already stored
4:      return SUCCESS
5:  else
6:      data ← send(client, REQUEST_DATA, fingerprint)
7:      validate(data, fingerprint)
8:      ds.write(fingerprint, data)                         ▷ New Data
9:      ds.create_reference(fingerprint, object_uid)
10:     return SUCCESS
11: end if
```

Retrieve Data Chunk

The retrieve data chunk operation is a stub to the data storage component, which returns the chunk identified by the fingerprint if it is present in the data store, or a error message if the requested chunk is not found.

Unreference Data Chunk

This operation removes an object back-reference from a chunk with a given data identifier (fingerprint). This operation is called by clients and metadata servers when a data chunk is no longer referenced by the current version of an object recipe, such as when an object is deleted or updated. More details on back-references and data deletion will be given in the next section.

3.3.5.2 Data deletion

Differently from traditional storage systems, where a delete operation is made available to clients, in block-sharing systems data deletion is not performed directly by clients due to its write-once nature. A mechanism is necessary to identify when a data block can be safely deleted. Our initial approach was to maintain reference counts for each stored data chunk and update the counts as objects are added, removed or updated. However, as pointed out in (Efstathopoulos & Guo 2010), maintaining reference counts in a distributed environment is a cumbersome task. Since chunks are replicated, the reference counts among different copies of the same chunk need to be consistent, what requires the chunk referencing operation to be transactional. Furthermore, any lost or repeated update may cause a chunk to be mistakenly kept or deleted. Additional bookkeeping is necessary to detect such problems, making the solution even more complex.

Mack et al. (Macko, Seltzer, & Smith 2010) proposed the use of back-references in write-once file systems to track files that reference shared blocks in order to support data deletion or relocation. In this context, back-references are inverted references that link each data chunk to the objects that reference them. A simple implementation keeps a persistent list of object unique version identifiers for each chunk stored in a data server. When new objects that reference a particular data block are added or updated, the object unique version identifier is added or removed from the list of back-references. This is a better solution since it can be easily implemented and is immune to repeated reference updates. A downside of this approach is that more storage space is required to keep back-references, what must be taken into account when analyzing the metadata overhead.

A local garbage collection process can monitor when a particular chunk has no back-references in a data server and trigger a distributed garbage collection process for that specific chunk. This process takes a union of back-references for all replicas of that chunk and verify

if it contains zero back-references. In positive case, the chunk is temporarily locked for writing and is simultaneously deleted from all data servers. If back-references are not consistent across replicas, the metadata servers of the referenced objects are checked to fix inconsistencies.

3.3.6 Metadata Servers

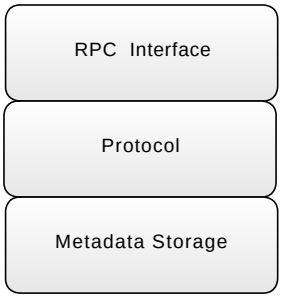


Figure 3.3: Metadata Server internal components

The metadata server stores and manages replicas of object recipes. Consistency between object replicas is ensured by a simple conflict resolution algorithm based on physical timestamps. A component view of the metadata server's internal architecture is shown on Figure 3.3. The main difference from the data server architecture is that it uses a data storage component optimized for metadata operations.

3.3.6.1 Metadata Server Operations

The proposed Application Programming Interface (API) of the metadata servers exposes the following operations:

- **Insert Object Recipe (metadata put):** Inserts a new object recipe in the metadata server with a given identifier. This operation overwrites an object recipe with the same identifier stored with a smaller timestamp attribute.
- **Retrieve Object Recipe (metadata get):** Retrieves an object recipe with a given identifier from the metadata server.
- **Remove Object Recipe (metadata delete):** Removes an object recipe with a given identifier from the data server, making it unavailable.

Insert Object Recipe

The metadata put operation, shown on Procedure 5, inserts or replaces a metadata recipe of an object with a given identifier. Additionally, this operation enforces the "last write wins" conflict resolution policy for concurrent metadata writes.

The algorithm starts by fetching the current version of the object recipe from the metadata store if it exists (line 2). On the next step (line 3), in case a previous recipe exists, the timestamp of the new recipe is compared with the timestamp of the previous version in order to enforce the "last write wins" policy. If the timestamp is older than the current stored version, it means a more recent version was already written and this put should be discarded. In case it's a new or more recent version being put, the recipe is written to the metadata store (line 4).

When a previous version of the object recipe exists, the data server must call a procedure that will remove back-references from the previous object version on data servers, in order to allow deletion of unused chunks (lines 5-7). This procedure can be executed by a background process in order to avoid adding latency to the put operation, since chunk unreferencing does not need to be performed immediately.

Procedure 5 Metadata Server - Metadata Put

Input: *Object Identifier, Client Timestamp, Object Recipe*
Output: *SUCCESS if the operation was executed successfully*

```
1: ms ← MetadataStorage
2: prev_obj ← ms.lookup(obj_id)
3: if NOT prev_obj OR timestamp > prev_obj.timestamp then      ▷ Conflict Resolution
4:     ms.write(obj_id, recipe, timestamp)                     ▷ Write new recipe
5:     if prev_obj then
6:         remove_references(prev_obj)
7:     end if
8: end if
9: return SUCCESS
```

Retrieve Object Recipe

The metadata get operation is a stub to the metadata storage component, which will lookup the recipe indexed with the object identifier and return it to the client if it exists.

Remove Object Recipe

The metadata delete operation calls the **Insert Object Recipe** with an empty object recipe and the timestamp provided by the client. Due to the "last write wins" policy, if the delete is the most recent operation, the current object recipe with the specified identifier will be replaced by the empty metadata. The operation will also unreference data chunks referenced by the deleted recipe.

Since an empty object metadata cannot be submitted directly by the client, an empty recipe will be interpreted by the metadata server as a tombstone, resulting in an error if tried to be accessed by the client.

3.4 Implementation

The proposed architecture was implemented as an extension to OpenStack Object Storage. OpenStack is a collaboration project of hundreds of organizations led by NASA and RackSpace that produces an open source cloud computing platform for public and private clouds. One of OpenStack's cloud components is Object Storage, code-named Swift, which provides a distributed object storage service.

The choice of OpenStack Swift as a platform to implement the deduplicated object storage architecture proposed in this thesis was based on the following reasons:

- It is an important and recognized solution for cloud object storage in industry, supported by many prominent organizations worldwide.
- It is open source, which allows code to be modified and redistributed.
- Deduplication is not supported by OpenStack Swift in its current state, and the result of this project can be contributed to the community.
- OpenStack Swift current components matches well the architecture proposed in this thesis, which allows common code to be reused.

In section 3.4.1 we present an enhanced architecture for OpenStack Object Storage, with built-in support for deduplication. Relevant implementation details of the proposed architec-

ture's components: metadata servers, data servers, proxy servers and client library, are presented on sections 3.4.2, 3.4.3, 3.4.4 and 3.4.5, respectively.

3.4.1 OpenStack Object Storage Extended Architecture

OpenStack Object Storage current architecture has 3 main components: the rings, storage servers and proxy servers. The rings determine in which storage server data resides, providing a variant of consistent hashing that routes requests to storage nodes. Storage servers maintain information about accounts, containers and objects. The proxy server is a gateway service that coordinate requests from clients routing them from an external network to the internal network of storage servers. More details on the Swift architecture are found on section 2.3.1.1.

The implemented prototype extends Swift by introducing two new types of storage servers: data servers and metadata servers, which were described in sections 3.3.5 and 3.3.6. These servers are part of two new overlay rings that leverage the existing ring structure of Swift: metadata ring and data ring. The metadata ring distributes the object namespace across the metadata servers. The data ring implements the distributed content-addressable storage, mapping data chunk fingerprints to data servers responsible for storing them.

The new server types, data and metadata servers, may entirely replace the former object servers in a Swift installation. In this case, the system will only support deduplicated object storage. However, it is possible to install the new servers in addition to the previous object servers, and support both protocols of Swift (backwards compatibility). In this case, a flag determines whether an object uses the old or the new protocol, and the proxy server routes the request to the correct set of servers. The implementation of metadata and data servers are discussed in sections 3.4.2 and 3.4.3, respectively.

A custom client was implemented to provide deduplication functionality using the new set of servers. Moreover, the proxy server was modified to support operations on data and metadata servers. The implementation of the proxy server and the client are discussed in sections 3.4.4 and 3.4.5, respectively.

Openstack Object Storage is developed in the Python language and is composed of multiple daemons implementing different services. All communication between client and services, and between services is done through the Hypertext Transfer Protocol (HTTP) protocol through

a RESTful interface. The Representational State Transfer (REST) style defines a series of guidelines and constraints to web services design. Due to the use of HTTP, all communication between Swift components is synchronous.

3.4.2 Metadata Servers

The metadata server reuses the previous Swift object server module. The module is a python web server based on the Web Server Gateway Interface (WSGI) standard. The server uses HTTP as a communication protocol, exposing the following HTTP interface:

HTTP PUT /<device>/<partition>/<account>/<container>/<object>
Implements the **Insert Object Recipe** operation, creating a new version of the object specified in the Uniform Resource Locator (URL).

HTTP GET /<device>/<partition>/<account>/<container>/<object>
Implements the **Retrieve Object Recipe** operation, retrieving a recipe of the object specified in the URL.

HTTP HEAD /<device>/<partition>/<account>/<container>/<object>
Similar to HTTP GET but does not return the message body. This method is useful to retrieve the timestamp of an object recipe without returning the recipe, so the client can decide from which server to fetch the most recent recipe.

HTTP DELETE /<device>/<partition>/<account>/<container>/<object>
Implements the **Remove Object Recipe** operation, which removes the recipe of the object specified in the URL.

The /<device>/<partition> prefix in the HTTP URL indicates in which consistent hashing partition (equivalent to virtual node) the chunk must be stored, and is necessary because a data server may have multiple devices, each of them hosting multiple partitions. The /<account>/<container> terms indicates in which account and container the object is listed.

These methods implement the protocols described in the Metadata Server Operations section (3.3.6.1). In the following subsections we present relevant implementation details of the metadata servers.

3.4.2.1 Recipe Format

```
{
    "meta": {
        "name": "image.jpeg",
        "length": 153600,
        "creation_date": "2012-05-21 15:01:38.913645"
    },
    "recipe": {
        "algo": "fixed",
        "hash_type": "sha-1",
        "default_length": 65536,
        "list": [
            {
                "id": "356a192b7913b04c54574d18c28d46e6395428ab"
            },
            {
                "id": "da4b9237bacccdf19c0760cab7aec4a8359010b0"
            },
            {
                "id": "77de68daecd823babbb58edb1c8e14d7106e83bb",
                "length": 22528
            }
        ]
    }
}
```

Listing 3.1: JSON Object Recipe

The object recipes are represented in the prototype as JavaScript Object Notation (JSON) objects. This choice was based on the format simplicity, widespread adoption in the Cloud computing community and compatibility with the Representational State Transfer (REST) architecture, in which OpenStack Swift is based.

A sample JSON object recipe for an image object is shown on listing 3.1. The described object length is 150KB, and it was chunked into 2 fixed-size chunks of 64KB and the last chunk is 22KB. Additionally, the hashing algorithm used was SHA-1 and a application-defined attribute "creation_date" indicates the date when the object was created.

3.4.2.2 Metadata Storage

Object recipes are stored as plain JSON files in the file system. A hash derived from the object identifier is used to lookup the object recipe from a flat directory in the file system. Important attributes such as timestamp are embedded in the file's extended attributes, which is a feature of some file systems that enables applications to associate metadata with files.

We also implemented an alternative metadata storage module that uses document database MongoDB (MongoDB 2012). The choice of MongoDB was primarily due to its use of JSON as document type, what enables efficient storage and retrieval of object recipes which are based on this format. One feature of MongoDB that can be leveraged to improve write performance is to batch metadata writes on memory before flushing it to disk. However, as some of the replicas may fail during a metadata write operation, a mechanism is necessary to synchronize replicas when this feature is used.

The alternative metadata storage component uses the single server setup of MongoDB, so there is no communication between MongoDB instances in different servers. One MongoDB database is hosted per device and the object recipes of each partition are stored in the same collection, which is analogous to a table in a relational database, in order to simplify synchronization of collections between different servers in the face of node failures.

3.4.2.3 Conflict Resolution

The default conflict resolution mechanism of the Swift Object Server is used when the file-based metadata storage strategy is selected. In this approach, during **HTTP PUT** operations, the object recipe is atomically inserted in the file system using the timestamp as file name suffix and all previously stored files with a smaller timestamp are removed from the object storage folder. During **HTTP GETs**, the contents of the object directory are lexically ordered and the file recipe with the highest timestamp suffix is returned to the client.

When MongoDB is used as storage backend on Metadata Servers, the **HTTP PUT** operation performs a conditional insert operation that atomically compares the timestamp provided by the client with the timestamp of the stored recipe and only inserts the new recipe if its timestamp is more recent than the stored version. This is done through the **upsert** operation, that updates the existing entity if it exists and matches the condition, or inserts a new entity in the database if it does not exist.

3.4.3 Data Servers

The data server was implemented as an extension to the existing Swift object server. The following HTTP interface is exposed by data servers:

HTTP PUT /<device>/<partition>/<fingerprint> Implements the **Insert Data Chunk** operation, adding the binary chunk with the fingerprint specified in the URL to the data server, if the data is not already stored. This method also creates a back-reference to an object version that must be specified in the HTTP headers of the request.

HTTP GET /<device>/<partition>/<fingerprint> Implements the **Retrieve Data Chunk** operation, retrieving a binary chunk of data with the fingerprint specified in the URL from the data server.

HTTP UNREF /<device>/<partition>/<fingerprint> Implements the **Unreference Data Chunk** operation, which removes a back-reference to an object from the binary chunk of data with the fingerprint specified in the URL.

The implementation of these methods follow the algorithms described in the Data Server Operations section (3.3.5.1). In the following subsections we present relevant implementation details of the data servers.

3.4.3.1 Data Storage

The data server prototype relies on the underlying file system to store and locate chunks. The server writes each received data chunk to an individual file on disk. Since a data server will

typically host millions or even billions of data chunks, the file system must have the ability to store a very large number of files.

The extended file system version 4 (ext4), limits the maximum number of files by the size of the inode table, which is defined at creation time (Mathur, Cao, Bhattacharya, Dilger, Tomas, & Vivier 2007). However, this means a large space must be reserved for the inode table, that may never be consumed if the file system does not grow to the expected size. XFS (Trautman & Mostek 2000) solves this problem by using dynamic allocation of inodes, limiting the number of files only by the amount of disk space available.

Each chunk is stored in the device specified in the request URL in a flat partition directory. The folder structure used to store chunks is based on a fixed prefix of the chunk fingerprint. This optimization avoids storing all chunks in a single directory, what would affect performance on chunk put and get operations as the number of files stored in a directory grows. For instance, if the configured size of the prefix is 3, the chunk with SHA-1 fingerprint "047595d0fae972fbed0c51b4a41c7a349e0c47bb" is stored in the following directory:

/<device>/<partition>/chunks/047/047595d0fae972fbed0c51b4a41c7a349e0c47bb

The file system directory structure is used as index to locate chunks on disk. Even though this scheme is satisfactory for storing large chunks (when whole-object deduplication is used), it may introduce the chunk-lookup disk bottleneck problem for small chunks. Since every chunk lookup requires at least one random disk access to find the location of the chunk on disk, the seek overhead may dominate operation times when inserting or retrieving objects composed of many small chunks.

However, we expect this problem to be alleviated by the fact that the chunks of an object will be spread among many servers, and by the network overhead when transfers are done over slow connections. Several works have published solutions that leverage caching, bloom filters and chunk locality to mitigate the disk bottleneck problem (Zhu, Li, & Patterson 2008; Lillibridge, Eshghi, Bhagwat, Deolalikar, Trezise, & Camble 2009; Bhagwat, Eshghi, Long, & Lillibridge 2009). Some of these solutions can be integrated to this architecture to further increase chunk lookup efficiency.

3.4.3.2 Insert Data Chunk Operation

The **Data Put** operation requires data to be transferred only when a chunk does not exist in the data server, due to the write-once nature of content-addressable storage. This allows bandwidth to be saved, by requiring that only new chunks are transferred over the network on HTTP PUT operations.

The HTTP 1.1 specification defines the "Expect: 100- continue" header, that allows the client to defer the transfer of the payload of an HTTP request until the server requests it. This mechanism fits well the behavior of the data put operation. The sequence diagram for a HTTP PUT request on the data server is shown on figure 3.4.

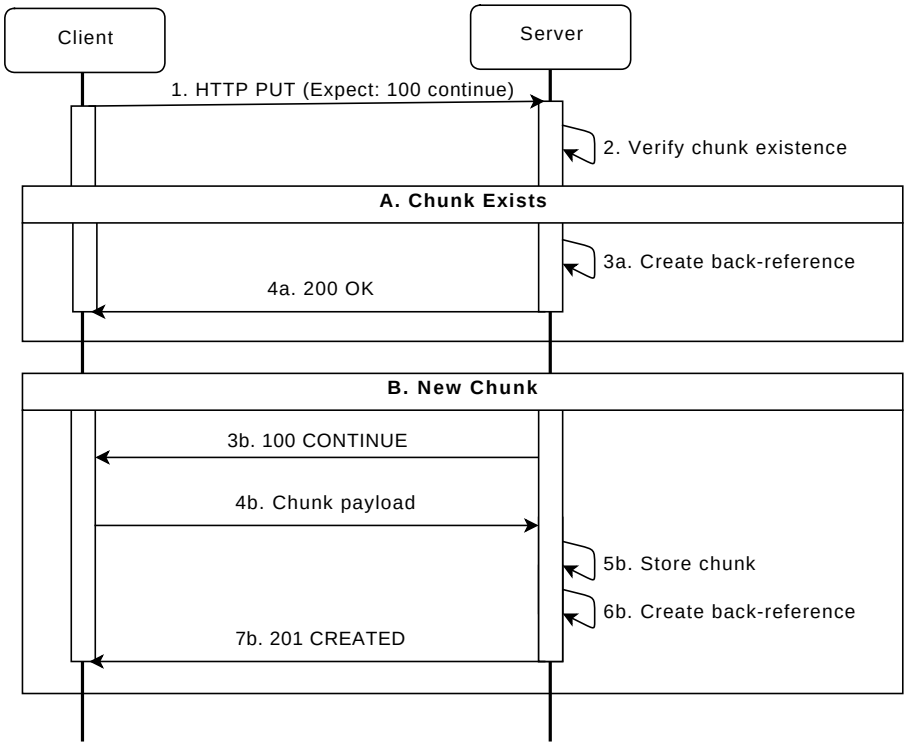


Figure 3.4: Sequence diagram of HTTP PUT operation on Data Servers

On step 1, the HTTP PUT request is sent to the data server containing the "Expect: 100-continue" header. The server verifies if the chunk with the requested fingerprint (given on the request URL) is present in the local storage (step 2). In case it exists, a back-reference for that chunk is created with the object unique identifier specified in the request headers (step 3a). Finally, a successful HTTP response "200 OK" is returned to the client on step 4a.

When the data piece does not exist, the server sends a "100 CONTINUE" message to the

client, indicating that the request payload must be transferred (step 3b). The client sends the chunk data on step 4b and after the chunk is received and the fingerprint is validated, the server writes it to the storage component on step 5b. The back-reference is created on step 6b, and a HTTP response "201 CREATED" is returned to the client on step 7b, indicating the chunk was successfully created in the server.

3.4.3.3 Back-References

A back-reference is a string identifier that references a unique version of an object and serves the purpose of garbage collection of unreferenced chunks. A back-reference must be specified in the "X-Chunk-Obj-Ref" HTTP header during a **HTTP PUT** or **HTTP UNREF** operation on Data Servers. We have evaluated several strategies for storing back-references on data servers. Each strategy is briefly described below:

Text (append-only) Maintain an append-only back-references file per data chunk, storing back-references as plain text.

Pickle (append-only) Maintain an append-only back-references file per data chunk, storing back-references as Pickle (python library) serialized objects.

Shelve Maintain a Shelve (python library) persistent dictionary per data chunk to store back-references as dictionary keys.

MongoDB Maintain a MongoDB document collection per data chunk to store back-references as JSON objects.

SQLite Maintain a relational table on lightweight database SQLite to associate back-references with chunk fingerprints.

The main criteria in the evaluation of strategies for storing back-references was write latency, since it directly affects the latency of **HTTP PUT** on data servers, which is a critical operation because it is part of the object insertion workflow. In order to evaluate this, we executed a custom benchmark that creates 10.000 random back-references for 1000 data chunks. In the benchmark, back-references were 32 byte-long strings. The results of this experiment are shown on Table 3.1.

Strategy	Average Latency (ms)	Standard Deviation (ms)
Text	0.02	0.01
Pickle	0.03	0.02
Shelve	19.97	24.94
MongoDB	0.07	1.77
SQLite	85.08	38.95

Table 3.1: Latency of back-reference create operation using different strategies for 10.000 back-references of 1000 data chunks. Each back-reference is a 32-byte string representing the unique version identifier of an object that references a data chunk. The Text and Pickle file-based strategies had a much better performance in comparison to structured data storage alternatives (Shelve, MongoDB and SQLite).

The results show that file-based strategies are more efficient than structured persistence for storage of back-references. We have selected the text-based append-only strategy for storing back-reference due to its superior performance and lower disk space footprint in comparison with the Pickle strategy.

The append-only nature of the text-based strategy requires that two files are maintained per data chunk: one for references, and another for "unreferences". When a new back-reference is created during a **HTTP PUT** operation, it is appended to the references file. The **HTTP UNREF** operation appends a back-reference to the "unreferences" file. A background process must periodically merge both files by removing the entries of the references file that are also in the "unreferences" list.

3.4.4 Proxy Server

In the OpenStack Object Storage architecture, clients cannot contact storage servers directly. For this reason, the Proxy Server forwards external requests to storage servers based on the HTTP request URL. In order to be consistent with Swift's architecture we have enabled access to the new storage servers through the Proxy Server. The supported URLs in the extended proxy server and their associated storage servers are shown on table 3.2.

The standard version of the Proxy Server currently supports URLs 1-3. Our prototype extends the Proxy Server with support to URLs 4 and 5. The /<version> prefix indicates which version of the Swift protocol is being used.

ID	URL	Server Type
1	/<version>/<account>	Account Server
2	/<version>/<account>/<container>	Container Server
3	/<version>/<account>/<container>/<object>	Object Server
4	/<version>/<account>/<container>/<object>?dedup	Metadata Server
5	/<version>/<fingerprint>?dedup	Data Server

Table 3.2: Proxy server URLs and associated storage servers

The fourth URL indicates a metadata operation that should be routed to a metadata server. The only difference from URL 3 is the *?dedup* REST attribute that allows the proxy server to distinguish between an operation on an ordinary object and on a deduplicated object. Upon receiving an URL on this format, the proxy server queries the metadata ring structure and routes the request to the appropriate metadata server partition and device based on the hash of the object name.

The fifth URL indicates an operation on a data chunk, with a *?dedup* attribute to allow the server to distinguish between URLs 5 and 1. Similar to the previous case, the Proxy Server queries the data ring and routes the request to the appropriate data server partition and device based on the hash digest (fingerprint) contained in the URL.

3.4.5 Client

The client enables users and applications to leverage global distributed deduplication functionality on OpenStack Object Storage. The client implements the protocol presented in section 3.3.4, providing a simple *get/put/delete* interface to retrieve, store or remove objects from the object storage system.

The proxy server could act as a deduplication client instead, enabling transparent deduplication to external users. However, we decided to implement the client as a standalone python module since it gives applications more flexibility to embedded the client on them according to their needs. The client application can be used in terminal mode, or imported by another application to be used as an auxiliary library.

In order to be coherent with Swift's design, all the traffic between the client and the storage servers is relayed through the proxy server. This will also allow a fair experimental comparison

between deduplication-enabled Swift and the unmodified version of the system, since in both versions the traffic must go through a proxy server. However, it is straightforward to extend the client to enable direct access to storage servers, by querying the ring data structure on each remote operation.

3.4.5.1 Design and Operation

The client is a HTTP client of proxy and storage servers. Initially we have used default python libraries to handle HTTP requests: httplib2 and urllib2. However, these libraries did not provide advanced features, such as support for "Expect: 100-Continue" behaviour, necessary for **data put** operations. For this reason we have chosen the PyCurl library, which is a python binding to the UNIX Curl utility which provides a comprehensive set of HTTP features.

Client functionality is implemented by the Client class, which accepts the following configuration parameters on its constructor:

Proxy Server URL The proxy server URL that must be contacted to perform HTTP calls.

Deduplication-enabled Whether deduplication should be enabled or not for operations on objects. Default value: True.

Chunking method The algorithm to chunk objects. Currently supports: whole-file chunking and fixed-size chunking. Default value: fixed-size chunking.

Chunk length The length of the chunk when fixed-size chunking is used. Default value: 4096B.

After the class is instantiated, the application may execute any of the three supported operations on objects: **put**, **get** and **delete**. Each of the operations takes the object name as parameter. The **put** and **get** operations also take a local file system path as parameter, from where the object must be read or where it should be written.

A persistent HTTP connection between the client and the proxy server is maintained between subsequent operations. The application may also change configuration parameters at runtime or override default parameters on each operation.

Summary

This chapter presents the design and implementation of a distributed object-based storage system with built-in support for deduplication.

The system's architecture is composed of two services: a data service and metadata service. The data service implements a content-addressable storage, where data is indexed according to a fingerprint of its contents. The metadata service stores object recipes, which is a summarized representation of an object allowing it to be reconstructed from its content-addressed parts. Each service is implemented by a set of servers composing a distributed hash table. A consistent hashing algorithm maps data fingerprints and object identifiers to the correct server responsible for storing that entity. The client application chunks objects and send their data to the data service and the object recipe to the metadata service, allowing it to be retrieved at a later time. The architecture was implemented as an extension to OpenStack Object Storage, an important solution for cloud-based object storage.

In next chapter we conduct a performance evaluation of the implemented prototype.

4 Evaluation

4.1 Introduction

In this chapter we present the methodology used to evaluate the solution and discuss obtained results. We evaluate several metrics in different scenarios with the objective of identifying trade-offs and overheads that affect performance and efficiency of a cloud-based object storage system with deduplication support.

The remainder of this section is structured as follows. The experimental settings and datasets used in the experiments are described in section 4.2. We analyze the metadata overhead incurred in the system for different deduplication strategies and workloads on section 4.3. The storage utilization and space savings of using deduplication in different datasets and its associated trade-offs are evaluated in section 4.4. Finally, we evaluate the performances of chunk-level deduplication in section 4.5, and object-level deduplication in section 4.6

4.2 Experimental Settings

The prototype was implemented as an extension to OpenStack Object Storage (Essex release - 1.4.6). We modified OpenStack Swift to support two modes of operation: normal mode, which is the unmodified version of the software, and deduplication-enabled mode, which implements the architecture proposed in this thesis. For simplicity, we refer to each of this modes throughout this chapter as "Swift-unmodified" (the former) and "Swift-dedup" (the latter). In order to focus solely on the performance of object operations, we disabled operations on containers and accounts in both modes of operation. These features only affect listings of objects per account or container, which are not the focus of this evaluation.

The experiments were executed in 2 clusters, each of them with the configuration specified in Table 4.1. The cluster was configured as a 17-node Swift installation (the full capacity was

# Nodes	10
Processor	2x 4-Core Intel Xeon E5506@2.13GHz
Memory	16GB DDR3-1066 ram
Disk	1TB 7200rpm SATA hdd
Operating System	Ubuntu 10.04.

Table 4.1: Clusters configuration

not used due to the unavailability of some nodes), deployed according to the diagram shown in Figure 4.1.

The clusters are interconnected by a high speed network switch, and host 8 Swift storage servers each. The configured object replication factor was set to 2 in the experiments, unless otherwise specified. Each cluster composes a replication zone, so every piece of data (object, chunks or metadata) is replicated in both clusters during write operations. Access to storage nodes is done through a proxy server installed in one of the nodes of cluster A. A client application executes in the same machine as the proxy server, transmitting data to the server through the loopback network interface, so the client-proxy communication overhead is negligible. Unless otherwise specified, the OpenStack Object Server extension uses the configuration summarized in table 4.2.

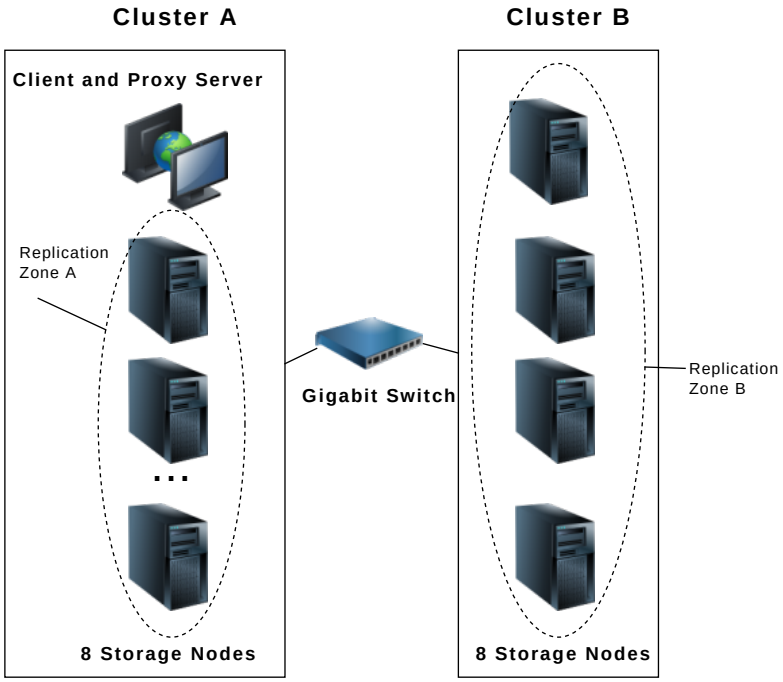


Figure 4.1: Deployment of OpenStack Object Storage in 2 clusters

Fingerprint hash function	SHA-1
Object identifier hash function	MD-5
Replication factor (N)	2
Operations Quorum (R, W, D)	2
Metadata Storage Strategy	File-based

Table 4.2: OpenStack Object Server configuration

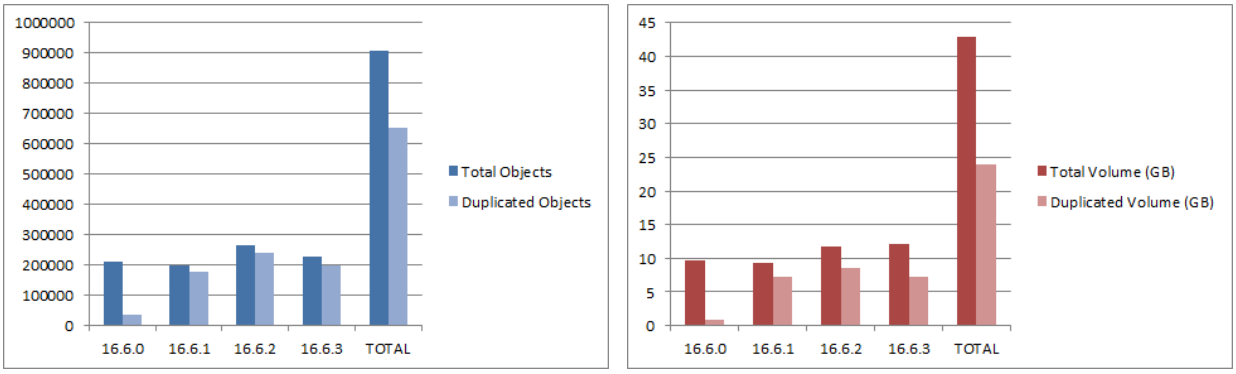
The presented results are an average of at least 3 executions of each experiment. Standard deviation results are presented in error bars when they are non-negligible.

4.2.1 Datasets

Real and synthetic workloads were used to evaluate the implemented prototype. In this subsection we describe each of the datasets used throughout the evaluation.

4.2.1.1 Scientific Software

This dataset is composed of software releases from the ATLAS experiment (Aad et al. 2008) at CERN, made available through the CernVM project (CERN 2012). Each release has on average 10 gigabytes distributed in hundreds of thousands of files. There is a large quantity and volume of duplicated objects among different releases, as seen in Figures 4.24.2a and 4.24.2b, what creates an opportunity for object-level deduplication.



(a) Total and duplicated number of objects of each release and all dataset

(b) Total and duplicated volume of objects of each release and all dataset

Figure 4.2: Duplication statistics of CERN ATLAS software releases used in the evaluation

4.2.1.2 Virtual Machine Images

A virtual machine image is a binary file containing the complete contents and structure of a given operating system in a specific format. Several studies have demonstrated the effectiveness of applying chunk-level deduplication to VM images (Nath, Kozuch, O'hallaron, Harkes, Satyanarayanan, Tolia, & Toups 2006; Jin & Miller 2009; Zhang, Huo, Ma, & Meng 2010; Jayaram, Peng, Zhang, Kim, Chen, & Lei 2011).

We selected some virtual machine images to evaluate the performance of chunk-level deduplication in the proposed system. The VMs are built for the Virtual Box system (Watson 2008) and were downloaded from http://virtualboxes.org/images/. The details of each virtual machine are summarized in table 4.3.

Index	**Operating System and Version**	**Uncompressed image size**
1	Ubuntu 6.06.1 (x86)	2.4 GB
2	Ubuntu 7.10 (x86)	2.5 GB
3	Ubuntu 8.04 (x86)	2.6 GB
4	Ubuntu 8.10 Server	1.1 GB
5	Ubuntu 8.10 (x86)	2.68 GB
6	Ubuntu 9.04 (x86)	2.5 GB
7	Ubuntu 9.04 (amd64)	3.6 GB
8	Ubuntu 9.10 alpha	2.52 GB
9	Ubuntu 9.10 final	2.36 GB
TOTAL		22 GB

Table 4.3: Details of virtual machine images used in the evaluation

4.2.1.3 Synthetic workload

In order to evaluate the throughput of the solution we used synthetic workloads generated with the standard Unix random number generator (/dev/urandom). The workload generator receives an object size as input and generates a file with the specified size from bytes randomly generated.

The natural entropy of the random generator function decreases the probability of unwanted duplicated data, which may affect throughput's test where the level of data duplication must be be controlled.

4.3 Metadata Overhead Analysis

There are two sources of metadata overhead in the system: object recipes and chunk back-references. The amount of metadata stored per object is calculated as follows:

$$metadata\_size = recipe\_size + backreferences\_size$$

Where,

$$recipe\_size = attributes\_size + chunk\_references\_size$$

$$chunk\_references\_size = num\_chunks \times fingerprint\_size$$

$$backreferences\_ = num\_chunks \times obj\_identifier\_size$$

So,

$$metadata\_size = attributes\_size + num\_chunks \times (fingerprint\_size + obj\_identifier\_size)$$

So, we can calculate the metadata overhead per object relative to the object size with the following equation:

$$overhead = \frac{metadata\_size}{data\_size}$$

$$= \frac{attributes\_size + num\_chunks \times (fingerprint\_size + obj\_identifier\_size)}{data\_size}$$

Based on this, we analyze the metadata overhead incurred in the system for chunk and object-level deduplication in the following subsections.

4.3.1 Chunk-level deduplication

In chunk-level deduplication, an object is divided into several chunks to increase the probability of duplicate detection. For this reason, $num\_chunks \times (fingerprint\_size + obj\_identifier\_size) >> attributes\_size$, so we can simplify the metadata overhead equation to:

$$overhead \approx \frac{num\_chunks \times (fingerprint\_size + obj\_identifier\_size)}{data\_size}$$

Furthermore, when fixed-size chunking is used to split an object, its size can be computed as:

$$data\_size = num\_chunks \times chunk\_size$$

When variable sized chunking is used, the average chunk size can be used to calculate an approximated object size. This is a reasonable extrapolation for the purpose of this analysis, since systems that employ variable chunking typically impose a maximum and minimum chunk sizes, what reduces chunk size variability (You, Pollack, & Long 2005; Jin & Miller 2009). Based on that, we can rewrite the overhead formula as:

$$overhead \approx \frac{num\_chunks \times (fingerprint\_size + obj\_identifier\_size)}{num\_chunks \times chunk\_size}$$

What brings us to:

$$overhead \approx \frac{fingerprint\_size + obj\_identifier\_size}{chunk\_size}$$

This equation illustrates the relationship between (average) chunk sizes and metadata overhead when chunk-based deduplication is used.

OpenStack Swift uses an MD5 digest of the object name as object identifier, which is represented as a 32 byte Unicode string. Furthermore, in our experiments we use the SHA-1 algorithm to fingerprint chunks, which is represented as a 40 bytes string. Based on these values, we calculated the metadata overhead for different chunk sizes, shown in table 4.4.

As can be seen from the table, the system adds very little metadata overhead for chunk sizes equal or greater than 4KB. For chunk sizes of 1KB the system imposes a larger metadata overhead. However, smaller chunk sizes yield better deduplication ratios, since more chunks are shared between different objects, so we expect this overhead to be superseded by large storage space savings when very small chunks are used. One way of improving this overhead is by storing fingerprints in binary encoding, rather than text format. In this case, a SHA-1 fingerprint would

Chunk size	Metadata Overhead
1KB	7%
4KB	1.8%
16KB	0.4%
64KB	0.1%

Table 4.4: System metadata overhead for different chunk sizes when using chunk-level deduplication

only occupy 160 bits, instead of the current 40 bytes.

4.3.2 Object-level deduplication

Object-level deduplication is a special case of chunk-level deduplication where there is only one chunk with size equal to the object size. Based on that, we can rewrite the overhead equation as:

$$overhead = \frac{attributes\_size + fingerprint\_size + obj\_identifier\_size}{data\_size}$$

The attributes section of the JSON object recipe (example shown on listing 4.2) is limited to 200 bytes in our prototype. As we previously mentioned, the object identifier size is 32 bytes and the SHA-1 fingerprint is 40 bytes. This adds up to a total metadata size of 272 bytes per object. Even for a modest 20KB object, this represents only a 1.3% metadata overhead. This low level of metadata overhead enables the solution to be adopted for object-level deduplication even for datasets with modest levels of object duplication, when object sizes are much larger than this overhead.

4.4 Storage Utilization

The storage utilization of the system is proportional to the level of deduplication of the stored data. The object-based storage system is agnostic to the chunking technique used by applications to split objects. This allows applications to use techniques and parameters that best match their datasets to yield good deduplication ratios.

In this section we analyze the potential storage savings that can be achieved when using the solution, and what factors influence the deduplication efficiency of a dataset. This analysis provides a background that will be useful in the remainder of this evaluation.

4.4.1 Chunk-level deduplication

In this analysis we evaluate how the chunk size influences the storage savings of the VM image dataset presented in table 4.3. This was done by chunking all the virtual machines in the dataset using different chunk sizes and comparing the observed deduplication ratios and predicted storage savings. The metadata overhead was based in the analysis made in the previous section and the overheads presented in table 4.4. The results are shown in Figure 4.3.

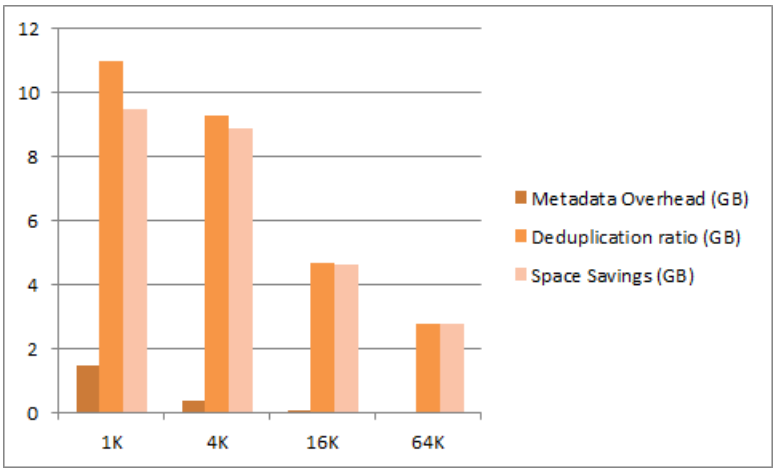


Figure 4.3: Metadata overhead, deduplication ratio and space savings obtained when storing all virtual machines of table 4.3 with different chunk sizes (Total volume: 22GB)

From the figure it is possible to observe a trade-off between chunk size, deduplication ratio and metadata overhead. Smaller chunks yield a better deduplication ratio but imply more metadata overhead. In this particular case, the best chunk size is 4KB, since it has a good balance between deduplication ratio and metadata overhead. Even though 1KB chunks provide slightly better space savings, the additional performance penalty on data fragmentation and throughput does not compensate for additional 1GB of storage savings (as we discuss in section 4.5). These results confirm previous findings that deduplication can yield very good storage savings of virtual machine images.

4.4.2 Object-level deduplication

In order to illustrate the storage space savings achieved by object-level deduplication in a dataset with a large number of duplicated objects, we compare the storage utilization of Swift-unmodified and Swift-dedup when all software releases of the CERN-ATLAS dataset (presented in section 4.2.1.1) are inserted in both systems sequentially. The results are presented in figure 4.4. The vertical lines represent the boundary of each release (the version is indicated in bold).

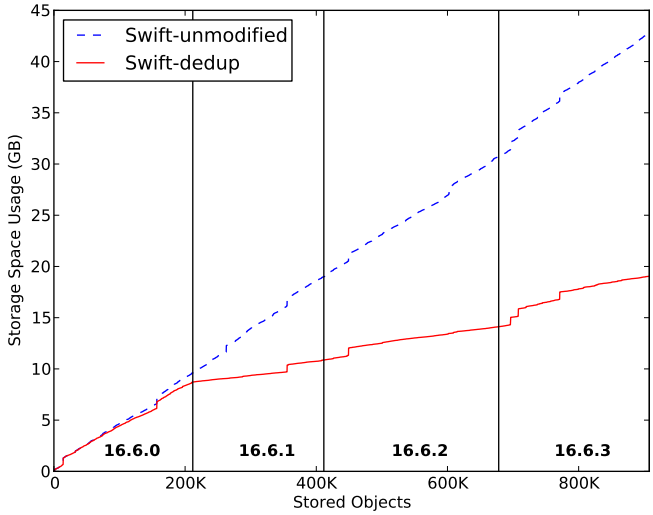


Figure 4.4: Growth of data in the system as new objects from CERN-ATLAS dataset are stored with object-level deduplication. The Swift-dedup version presents a slow volume growth rate, since duplicated objects are identified and deduplicated. In contrast, Swift-unmodified maintains a steady growth rate, since repeated objects are stored multiple times with different identifiers. After all releases are stored, the deduplication-enabled version of Swift uses about only half of the storage space of its unmodified counterpart due to the large amount of duplicated objects in this dataset.

After the first release (16.6.0) is inserted, there is a very slow growth in the volume of stored data in Swift-dedup. This happens because the subsequent releases have on average 70% of duplicated volume, so additional copies of repeated objects are not kept by the system. In contrast, if objects are inserted without deduplication support in Swift-unmodified, the repeated objects are stored several times, what severely impacts the storage utilization of the system. The metadata overhead was omitted from the figure, since it represented less than 1% of the total amount of data.

4.5 Chunk-level deduplication performance

We evaluate the performance of data operations when an object is split into multiple chunks in terms of throughput. This metric measures the amount of data a system can process in a given amount of time. We conducted experiments to determine which factors and trade-offs affects the system's throughput, and present them in the next subsections.

4.5.1 Chunk size and Concurrent Transfers

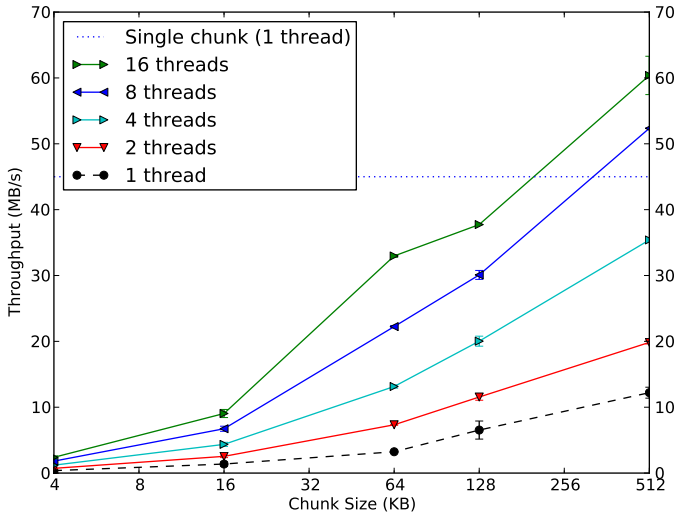


Figure 4.5: Write throughput for storing a 100MB object with different chunk sizes and concurrent write threads. Smaller chunk sizes yield a lower throughput due to disk seek overhead, which limits processing rate. Transfer parallelism can increase write throughput in a rate proportional to the size of the chunk. The horizontal dotted line on the top shows the throughput of the system when the object is written as a single 100MB chunk, as done in the unmodified Swift version. By combining chunk sizes with data parallelism, it is possible to achieve throughput rates larger then writing the object as a single chunk.

Objective: This experiment evaluates how the chunk size affects the throughput of put operations in comparison to inserting the object as single chunk (as done in unmodified Swift). Furthermore, we evaluate how transfer parallelism can increase the throughput of the system, since this mode of transfer is not available in unmodified Swift, where objects must be inserted or retrieved sequentially.

Experiment description: In this experiment, the client inserts a 100MB object using different

number of concurrent write threads and different chunk sizes. Each data chunk is unique to avoid performance gains from deduplication (we evaluate the throughput of duplicated chunks in section 4.5.1). We then compare the observed throughput rates when varying these parameters.

Discussion: The results for this experiment are shown in Figure 4.5. The lower dotted line represents the evolution of the write throughput when the chunked object is written sequentially with different chunk sizes. The limited throughput of small chunk sizes is due to HTTP communication overhead and disk seek overhead.

At least two random disk seeks are necessary to write a chunk in the prototype: one or more to verify if the chunk is already stored in the file system, and another one to move the disk head to the write position. For instance, a typical disk seek time is 10ms (Dean 2009), so a maximum of 100 disk seeks can be done per second. This yields a maximum of 400KB/s raw write throughput for chunks of 4KB, if write and communication times are not considered. Furthermore, a typical HTTP request in our prototype has 250 bytes, which means that about 6% more data needs to be transferred over the network if chunks of 4KB are used. By increasing the size of the chunk, more data can be written in the same amount of disk seeks, so the throughput is increased. However, the communication overhead still influences data transfer capacity, and the disk and network bandwidth are not fully utilized due to the small volume of the chunk.

When a chunked object is written using a single thread, the throughput is always inferior to that of writing the object as a single chunk (top horizontal line), because in the latter case the full network and disk bandwidth are utilized (45MB/s), and disk seek and communication overhead are negligible. Parallelism can significantly improve performance of write throughput for a chunk of the same size, as seen in the upper points for the same chunk sizes. The gains from parallelism are proportional to the size of the chunk: the larger the chunk, the greater the performance gain (as seen from the bigger spacing between parallel points when chunk sizes grow). By combining chunk sizes and transfer parallelism, it is possible to achieve throughput rates superior to that of writing the object as a single chunk (as can be seen from the case with 8 and 16 threads for a 512KB chunk). This is leveraged in systems like Ceph and pNFS (parallel NFS) to improve the system's throughput.

Even when parallelism is used, throughput rates for small chunks are only a small fraction of the throughput when writing the object as a single chunk (as done in unmodified Swift). Since best deduplication ratios are typically achieved with small chunk sizes, there is a large performance penalty for efficiently deduplicating objects. This makes the system less appealing to latency-critical applications, where the data transfer speed is of primary concern.

We expect this penalty to be less severe when disks are replaced by SSDs, driven by availability of this technology and lower costs, since disk seeks in SSDs are two or more order of magnitudes faster than in magnetic disks. Furthermore, it is possible to increase the write throughput of the solution by reducing or eliminating the amount of disk seeks necessary to write a chunk in a storage node. An in-memory index can eliminate the need of a disk seek to verify if the chunk is present in the system. A log structured file system (Rosenblum & Ousterhout 1992) can be used to write chunks sequentially to a continuous stream, thus eliminating the need for a random seek during disk write. Another improvement is to batch multiple chunks in a single request to the same node, in order to decrease the effect of communication overhead since more data can be sent in fewer requests.

Read performance: Data reads present a similar behavior in this experiment, since reads of small chunks are also dominated by communication overhead and random seeks. This happens because we perform the mapping of fingerprints to disk block locations by using a file named after the fingerprint. This makes reads very expensive, since one or more seeks are necessary only to find the location of a chunk on disk, and another seek is necessary to read the actual data.

One way to improve read performance, is by maintaining an in-memory index mapping chunks to block locations on disk. In this case, additional seeks to find the location of the chunk on disk are eliminated, but one random seek is still necessary to read the chunk from disk. This problem is inherent of deduplication, since best deduplication ratios are achieved by chunking data, and these chunks are distributed in random positions of the disk.

It is possible, however, to decrease the number of disk seeks necessary to read an object by leveraging locality during writes: chunks of the same object are written to a sequential

location on disk, and reads would require only a disk seek to retrieve multiple chunks. In this case, the client would need to batch multiple chunk operations in a single request to the server. Even though this approach could improve the performance of reads for non-duplicated objects, the fragmentation problem will still occur for shared chunks, since they will be distributed among many of these sequential locations.

4.5.2 Storage nodes and Concurrent Transfers

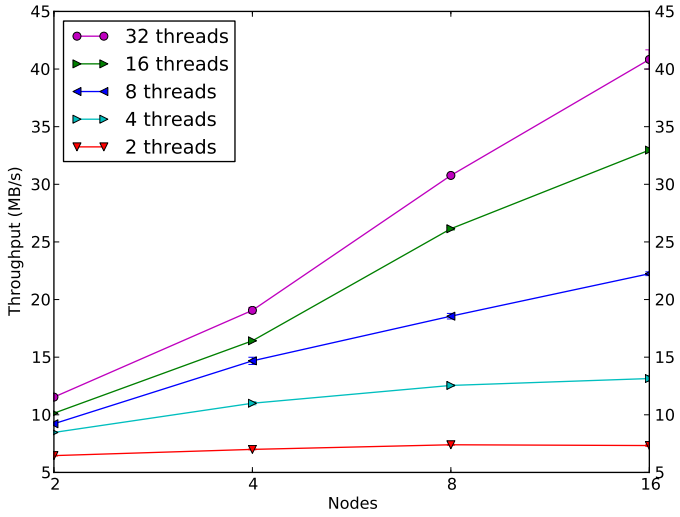


Figure 4.6: Write throughput for storing a 100MB object with different number of storage nodes and concurrent write threads. The throughput grows logarithmically as the number of storage nodes increases for the same number of writer threads. After the maximum capacity is reached (for instance, for 2 and 4 threads), the throughput is oblivious to an increase on the number of storage nodes. Throughput for the same number of storage nodes can be increased by adding more write threads, but is limited by the processing capacity of nodes to process a growing amount of concurrent requests.

Objective: Transfer parallelism can significantly improve the system's throughput, as seen in the previous section. In this experiment, we evaluate how adding more storage nodes affects the throughput of the system when object chunks are written in parallel.

Experiment description: The client writes a 100MB object with a fixed chunk size of 64KB in parallel, using different numbers of threads. We increase the number of nodes in the system to verify how throughput is affected by different numbers of concurrent writers

when the number of nodes changes. Each data chunk is unique to avoid performance gains from deduplication (we evaluate the throughput of duplicated chunks in section 4.5.1).

Discussion: Figure 4.6 shows the results of this experiment. For the same number of writers throughput grows logarithmically as more storage nodes are added. This happens because the previous load is evenly distributed across the new nodes, which then can serve requests with less queuing. After the maximum throughput is reached for a given number of threads (as seen in the lines of 2 and 4 threads for any number of nodes), the throughput is oblivious to the number of storage nodes, since the existing capacity is sufficient to serve that number of concurrent requests. When the number of nodes is fixed, it is possible to increase the throughput by increasing the number of threads (vertical dots). The larger the number of nodes, the higher throughput growth is achieved with more concurrent threads (as seen in the bigger spacing between points when more threads are added for a fixed number of nodes). However, after the processing capacity of all nodes is exhausted, the throughput will start to degrade when more threads are added, since the nodes will become overloaded with more requests than what they can support.

4.5.3 Duplicated chunks

Objective: One benefit of deduplication, is that bandwidth is saved by avoiding the transfer of duplicated chunks over the network. Instead, only a small back-reference is sent and appended to the list of back-references of the duplicated chunk. In this experiment we evaluate to what extent this mechanism influences on the write throughput of duplicated chunks when compared to that of unique chunks, for different chunk sizes.

Experiment description: In this experiment, an object of 100MB is written to the system in two scenarios: with only duplicated chunks (e.g. chunks that are already present in the system), or with only unique chunks (eg. chunks that are not present in the system). We vary the chunk sizes and compare the effect of chunk sizes on the throughput of duplicated and unique chunks.

Discussion: Figure 4.7 shows the throughput for writes of duplicated and unique chunks. For small chunk sizes (4KB to 16KB), there is very little difference in the throughput of unique and duplicated chunks. Similar to the performance bottleneck of unique chunks,

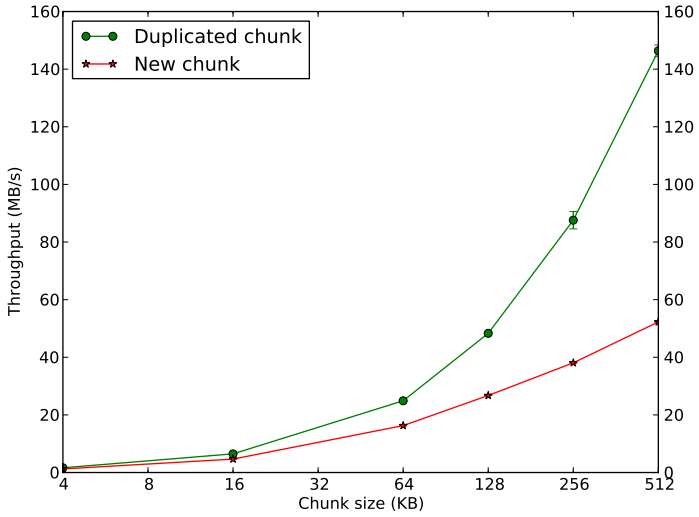


Figure 4.7: Comparison of write throughput of the system for duplicated and unique chunks, for different chunk sizes. For small chunk sizes, communication and disk seek overhead dominates writes of both new and duplicated chunks. The throughput of duplicated chunks grows indefinitely as the size of the chunk grows, since a larger volume of data can be processed at a near-constant cost (of writing a small back-reference to disk).

throughput of small duplicated chunks are also penalized by the overhead of disk seeks and HTTP communication. At least 2 random seeks are involved in the write of a duplicated chunk: one two find the location of the chunk file on disk, in order to verify that it exists, and another to write the back-reference files (which is located in the same directory as the chunk). Since the back-reference is only a 32 bytes string, the write time is negligible when compared to the seek time. Random seeking and communication overhead also dominates writes of unique small chunks, so the amount of duplicated and unique chunks that can be processed in the same amount of time is basically the same for small chunk sizes.

The throughput of duplicated chunks grow linearly as the size of the chunk grows, since the time to write a duplicated chunk is constant (the seek overhead), so it is possible to process a larger volume of data in the same amount of time. However, higher deduplication ratios are achieved with smaller chunks, and when these are used there is little to no performance improvement for inserting an object with a large deduplication ratio, when compared to an object composed of unique chunks. Throughput of duplicated chunk writes can be improved by reducing or eliminating the amount of disk seeks required to identify a duplicate and to write back-references. We discussed a few ways this can be done in

section 4.5.1 (in-memory index, SSDs, log-based file systems).

4.5.4 Proxy Server Bottleneck

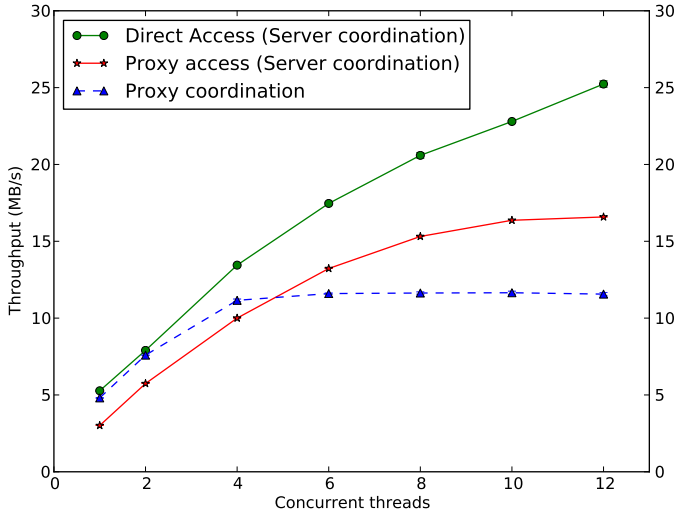


Figure 4.8: Comparison of different access strategies in terms of write throughput. Coordination done by the proxy server has significant limitations when the number of concurrent writers grow. A significant improvement can be seen by delegating coordination to storage nodes, but still relaying data through the proxy server. Maximum throughput is achieved by sending data to storage nodes directly, completely bypassing the proxy server.

Objective: In the current architecture of OpenStack Swift, all data is relayed through a proxy server, which coordinates requests and replicates data to storage servers. This is not a major problem in the unmodified version of Swift since an object can only be retrieved sequentially. However, as data access parallelism is enabled to increase the throughput on data operations, the proxy server may quickly become a bottleneck as it needs to handle more requests concurrently. Since the proxy performs request coordination and replication, an alternative approach is to move this responsibility to the first storage server that receives the request, thus, offloading the proxy. A third approach is to completely remove the proxy server and send data directly to storage nodes, that will perform coordination on behalf of the client. We compare each of these strategies in this experiment.

Experiment description: In this experiment, the client writes a 100MB object with a fixed chunk size of 64KB using different numbers of concurrent write threads and different access

methods (proxy coordination, via proxy (but server coordination) and direct access). Each data chunk is unique to avoid performance gains from deduplication. We then compare the observed throughput when varying these parameters.

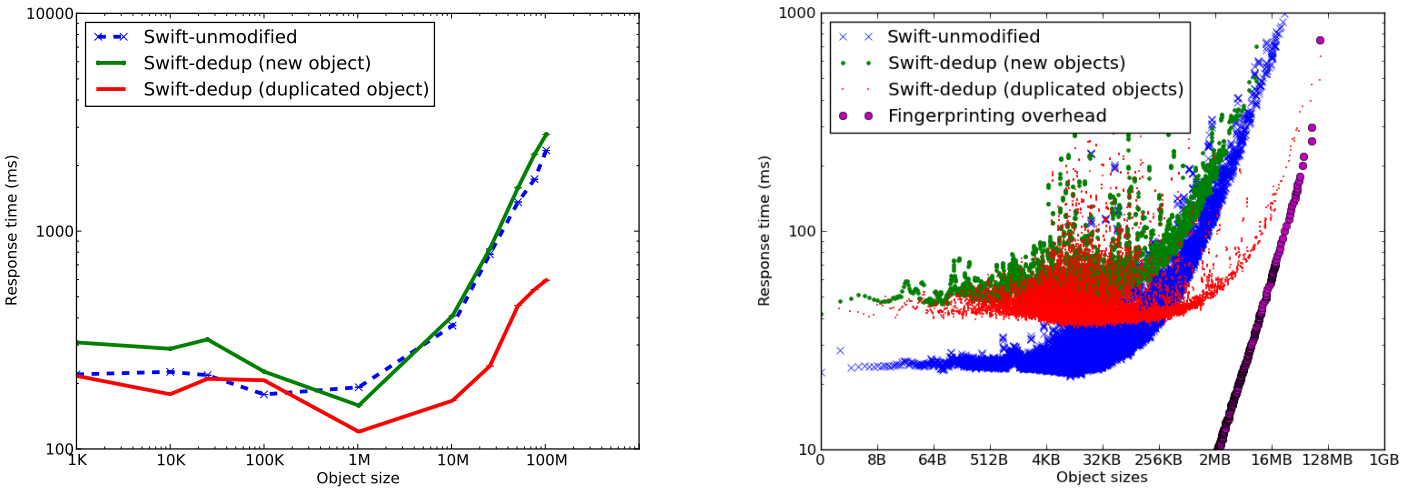
Discussion: Figure 4.8 presents the results for this experiment. The proxy server capacity is already exhausted after only 4 concurrent write threads (blue dotted line), which clearly shows the limitation of the proxy server for handling multiple concurrent writes. The proxy server capacity can be further improved by offloading request coordination to storage nodes (middle red line). Besides improving throughput, this strategy can also improve bandwidth utilization, as the network links between the storage nodes are used to replicate data. However, for a small number of concurrent threads (2 and 4), the proxy coordination is still better than storage server coordination via proxy. This is due to the additional hop in the data path before replication is done. Superior throughput is achieved when the proxy server is bypassed altogether (upper green line). This strategy improves throughput by removing the proxy server from the data path and distributing coordination load among the storage servers. A similar pattern happens for reading data, but the proxy coordination approach can handle more data read requests concurrently before exhausting its capacity, since read is a less intensive operation (no replication is involved).

4.6 Object-level deduplication performance

In object-level deduplication, the throughput of data transfers are not influenced in comparison to the unmodified version of Swift, since objects are put or retrieved as a single chunk. However, an additional metadata step is necessary to convert the object name into its unique fingerprint. For this reason, we evaluate the performance of object-level deduplication in terms of response times, to see how this additional step affects performance of the deduplication-enabled version of Swift.

4.6.1 Write Performance

Objective: This experiment evaluates how the additional steps of fingerprint calculation and metadata put affect response times of object insertions in the proposed enhancement of Swift. Furthermore, we compare difference in response times between duplicated objects



(a) Synthetic workload

(b) CERN-ATLAS workload

Figure 4.9: Object insertion times for object-level deduplication

and new objects, in order to evaluate how performance is improved when objects are deduplicated.

Experiment description: We performed two experiments to evaluate the write performance:

A) In a controlled environment with 16 storage nodes and replication=2 where synthetic objects with sizes growing exponentially from 1K to 100M were put sequentially in the system.

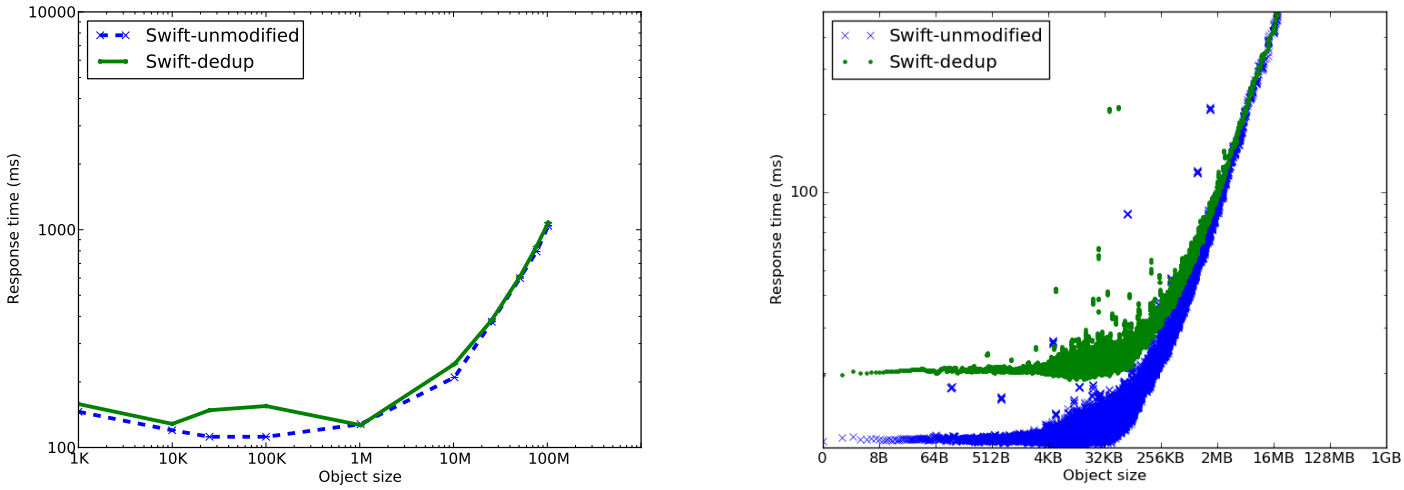
B) In a concurrent environment with 9 storage nodes and replication=3 where all objects from the CERN-ATLAS dataset (shown on section 4.2.1.1) were randomly put by 4 concurrent threads in order to verify how performance is affected when the system load is increased.

Discussion: The results of both experiments, shown in figure 4.9, are similar: the performance difference between putting a new object in both versions of Swift (green and blue lines in both graphs) is in the order of a few milliseconds, which is the extra time required to perform a fingerprint calculation and a recipe put operation in the Swift-dedup version. This difference is more impacting for very small objects (smaller than 1MB), because the time it takes to insert the metadata is roughly the same time as to insert the data (due to network and disk seek overhead). Furthermore, the fingerprinting overhead is negligible for objects smaller than 2MB, and after that it grows linearly with the object size, since

the entire object must be read in order to calculate its fingerprint before it is inserted in the system (as seen in the lower magenta line of Figure 4.9b).

One interesting fact is that results slightly differ in the performance of puts of small duplicated objects: while in test A (synthetic workload) the insertion of duplicated objects smaller than 1MB is roughly the same for both versions, in test B (CERN workload), Swift-unmodified slightly outperforms Swift-dedup (difference between red and blue lines). After investigating the fact, we discovered that in test A, the location of the object data was still cached in the directory lookup cache of the file system, so no seek was necessary to verify that the duplicated object was present in the system. In contrast, in test B, there was no time locality among repeated objects, so one or more disk seeks were necessary to find the location of the chunk on disk. This difference illustrates a potential performance gain of moving the chunk index to memory.

4.6.2 Read Performance



(a) Synthetic workload

(b) CERN-ATLAS workload

Figure 4.10: Object retrieval times for object-level deduplication

Objective: During object reads, the main difference between both systems (Swift-dedup and Swift-unmodified) is that in Swift-dedup an object recipe is fetched from metadata servers to locate the fingerprint of the requested object. After this is done, the fingerprint is used to retrieve the object from data servers. We evaluate this overhead in this section.

Experiment description: These experiments were executed in the same settings described in the previous section. After objects were put in the system, we executed read operations to retrieve the stored object.

Discussion: The results of both experiments (shown in figure 4.10) are similar: Swift-unmodified has a slightly superior performance for very small objects, since the time to transfer the object recipe is roughly the same time to transfer the actual data. Nevertheless this difference becomes negligible after the data transfer dominates the response time (after 1MB size). We believe this is a positive result, since a significant fraction of objects in a cloud storage system will likely be at least 1MB in size. One way of further improving performance for metadata writes it to store object recipes in SSDs drives. Since metadata is only a tiny fraction of the system's total data, it becomes a feasible option to store them in more expensive, but faster disks.

Conclusion

This chapter described the methodology used to evaluate the solution and discussed the results obtained in the evaluation.

The analysis of the metadata overhead for chunk-level deduplication concluded that the overhead is proportional to the average chunk size. Since smaller chunk sizes typically yield higher deduplication ratios, we expect this overhead to be superseded by the storage savings achieved through deduplication. When object-level deduplication is used, the metadata overhead is fixed on a couple hundreds of bytes, which becomes a negligible amount for objects in the order of a few megabytes. This allows object-level deduplication to be adopted even for datasets with modest levels of deduplication, if such datasets are composed of medium objects (larger than 1MB).

The potential storage savings when using the solution was evaluated for different datasets and chunking parameters. For chunk-level deduplication, smaller chunks have better deduplication ratios but impose more metadata overhead, and the optimal storage savings depends on this trade-off. Object-level deduplication storage savings are directly proportional to the amount of duplicated volume within a dataset.

We evaluated the throughput of the solution when chunk-level deduplication is used and identified some limitations for small chunk sizes, which typically yield best deduplication ratios. In particular, the overhead of HTTP communication and random disk seeks are limiting the rate in which small chunks are processed. These limitations prevent the system from being adopted in its current state by latency-critical applications. However, we expect this overhead to be smaller when SSD disks are used, since random seeks are two or more orders of magnitude faster than in magnetic disks. Furthermore, we intend to reduce the amount of random seeks necessary to process operations by storing data in a log-structured file system and maintaining an in-memory chunk index. We identified through experiments that throughput can be improved by increasing the amount of parallelism on data operations, and also by increasing the total number of storage nodes in the system. We also identified that the Swift proxy server significantly limits the scalability of the system in terms of throughput when data operations are done in parallel. This limitation can be solved by allowing clients to communicate directly to storage nodes.

Finally, the performance overhead of object-level deduplication was evaluated in terms of response times. On object insertions, the response time overhead is incurred due to fingerprint calculation and the additional step of inserting an object recipe. While the overhead imposed by the object recipe insertion is only significant to small objects ($< 1MB$), the fingerprint calculation overhead only becomes significant to large objects ($> 100MB$). In practice, this means that medium objects, which account for a significant fraction of the objects stored, will notice very little response time overhead during insertions. During object retrievals, the additional step of fetching the object recipe doesn't add significant response time overhead when objects are larger than 1MB, since data transfer time typically dominates the response time of fetching these objects. These findings suggest that object-level deduplication is a very low overhead option to save storage space in a cloud-based object storage system when most of the dataset is composed of medium objects.

In the next chapter we conclude this thesis and provide pointers for future work.

5 Conclusions

The object-based storage paradigm offers a simple and powerful model for cloud computing storage which addresses some of the scalability limitations of traditional file systems. This paradigm gives users the illusion of an unlimited storage pool that can be accessed from anywhere. As an increasing amount of data is stored in "the cloud", it becomes necessary to optimize the utilization of storage resources while still meeting requirements of availability and scalability which are key characteristics of cloud computing. In this thesis we propose, implement and evaluate a design that embodies global deduplication in a distributed object-based storage system as a means to improve storage efficiency of a cloud storage provider.

The proposed design is based on a distributed hash table architecture that maps pieces of data to distributed storage nodes based on a unique fingerprint extracted from the data's contents. This distributed content-addressable storage mechanism enables automatic deduplication of data, because data pieces with the same contents are mapped to the same nodes, and thus, stored only once. Based on this primitive, the client library lets applications choose the best way to chunk their data to yield best deduplication ratios, since each dataset has unique characteristics. After the object data is stored in the DHT, the client generates a piece of metadata which allows the object to be reconstructed from its parts. This object "recipe" is stored in a distributed set of metadata servers, allowing the object to be retrieved at a later time. The solution provides availability and durability by leveraging replication. The DHT architecture based on consistent hashing allows the system to scale the number of storage nodes with minimal service disruption. A prototype of the solution was implemented as an extension to OpenStack Object Storage, an open source solution for cloud-based object storage.

The solution is able to provide significant storage savings with low metadata overhead depending on dataset characteristics. However, the evaluation has identified relevant problems in the prototype's throughput performance when chunk-level deduplication is used for small chunk sizes, which typically yield the best deduplication ratios. In particular, random disk

seeks and communication overhead are limiting the throughput of the solution. This prevents adoption of the system on its current state by latency critical applications. However, it is possible to significantly improve the throughput of the solution by using faster storage technologies, such as SSDs, by reducing the amount of random disk seeks during reads and writes, and by batching data requests together. When deduplication occurs at the object level, the solution is able to satisfy writes and particularly reads with low latency overheads for objects larger than 1MB.

5.1 Future Work

We plan to address the limitations identified in the evaluation of the system's throughput for small chunk sizes. In particular, we want to decrease the number of random seeks necessary to read and write data chunks, which are imposing a significant overhead on chunk operations. Furthermore, we want to investigate ways to reduce the amount of data fragmentation in the system by leveraging some form of locality and co-location of multiple chunks of the same object, since this problem also influences the throughput of the solution.

References

Aad, G. et al. (2008). The ATLAS Experiment at the CERN Large Hadron Collider. *JINST 3*, S08003.

Adya, A., W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, & R. P. Wattenhofer (2002). Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, New York, NY, USA, pp. 1–14. ACM.

Alvarez, C. (2011). Netapp deduplication for FAS - deployment and implementation guide. Technichal Report TR-3505 Version 8, NetApp.

Amann, B., B. Elser, Y. Houri, & T. Fuhrmann (2008). Igorfs: A distributed p2p file system. In *Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, P2P '08, Washington, DC, USA, pp. 77–78. IEEE Computer Society.

Amazon (2011, May). Amazon web services: Overview of security processes. Technical report, Amazon Web Services.

Amazon (2012). Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/. [Online; accessed 30-Jun-2012].

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, & M. Zaharia (2009). Above the clouds: A berkeley view of cloud computing. Technical report.

Barr, J. (2012a, June). Amazon Web Services Official Blog. http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillion-objects.html. [Online; accessed 30-Jun-2012].

Barr, J. (2012b, April). Amazon Web Services Official Blog. http://aws.typepad.com/aws/2012/04/amazon-s3-905-billion-objects-and-650000-requestssecond.html. [Online; accessed 30-Jun-2012].

Beaver, D., S. Kumar, H. C. Li, J. Sobel, & P. Vajgel (2010). Finding a needle in haystack: facebook's photo storage. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Berkeley, CA, USA, pp. 1–8. USENIX Association.

Bhagwat, D., K. Eshghi, D. D. Long, & M. Lillibridge (2009, September). Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09*, pp. 1–9. IEEE.

Blomer, J., P. Buncic, & T. Fuhrmann (2011). CernVM-FS: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management*, NDM '11, New York, NY, USA, pp. 49–56. ACM.

Bloom, B. H. (1970, July). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*(7), 422–426.

Bolosky, W. J., S. Corbin, D. Goebel, & J. R. Douceur (2000). Single instance storage in windows 2000. In *In Proceedings of the 4th USENIX Windows Systems Symposium*, pp. 13–24.

Bolosky, W. J., J. R. Douceur, D. Ely, & M. Theimer (2000, June). Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. *SIGMETRICS Perform. Eval. Rev. 28*(1), 34–43.

Bonwick, J. (2009, November). ZFS deduplication (Jeff bonwick's blog). https://blogs.oracle.com/bonwick/entry/zfs_dedup. [Online; accessed 29-Jun-2012].

Bresnahan, J., K. Keahey, D. LaBissoniere, & T. Freeman (2011). Cumulus: an open source storage cloud for science. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, New York, NY, USA, pp. 25–32. ACM.

Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, Berkeley, CA, USA, pp. 335–350. USENIX Association.

Buyya, R., C. S. Yeo, S. Venugopal, J. Broberg, & I. Brandic (2009, June). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst. 25*(6), 599–616.

Carlson, M. A. (2009). Snia cloud storage:standards and beyond. Technical report, Storage Networking Industry Association - SNIA. [Online; accessed 30-Jun-2012].

CERN (2012). CernVM - software appliance. http://cernvm.cern.ch/. [Online; accessed 15-May-2012].

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, & R. E. Gruber (2008, June). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*(2), 4:1–4:26.

Cochran, M. (2007). Notes on the wang et al. 2^{63} sha-1 differential path. Cryptology ePrint Archive, Report 2007/474.

Collins-Sussman, B., B. W. Fitzpatrick, & C. M. Pilato (2002). *Version Control with Subversion*. O'Reilly.

Cox, O. P., C. D. Murray, & B. D. Noble (2002). Pastiche: making backup cheap and easy. In *In OSDI: Symposium on Operating Systems Design and Implementation*, pp. 285–298.

Dean, J. (2009). Software engineering advice from building large-scale distributed systems. `http://research.google.com/people/jeff/stanford-295-talk.pdf`.

Douceur, J. R., A. Adya, W. J. Bolosky, D. Simon, & M. Theimer (2002). Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, Washington, DC, USA, pp. 617–. IEEE Computer Society.

DropBox (2012). DropBox. https://www.dropbox.com/help/7/en. [Online; accessed 30-Jun-2012].

Druschel, P. & A. Rowstron (2001). Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pp. 75–80.

Dubnicki, C., L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, & M. Welnicki (2009). HYDRAstor: a scalable secondary storage. In *Proccedings of the 7th conference on File and storage technologies*, Berkeley, CA, USA, pp. 197–210. USENIX Association.

Eastlake 3rd, D. & P. Jones (2001). Us secure hash algorithm 1 (sha1). *RFC 3174* (3174).

Efstathopoulos, P. & F. Guo (2010). Rethinking deduplication scalability. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, HotStorage'10,

Berkeley, CA, USA, pp. 7–7. USENIX Association.

EMC (2004). EMC Centera. http://www.emc.com/.

EMC (2010, March). Achieving storage efficiency through EMC Celerra data deduplication. White Paper, EMC.

Factor, M., K. Meth, D. Naor, O. Rodeh, & J. Satran (2005). Object storage: the future building block for storage systems. In *Proceedings of the 2005 IEEE International Symposium on Mass Storage Systems and Technology*, LGDI '05, Washington, DC, USA, pp. 119–123. IEEE Computer Society.

Fidge, C. J. (1988). Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference 10*(1), 56â66.

Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, & T. Berners-Lee (1999). Rfc 2616, hypertext transfer protocol – http/1.1.

Fielding, R. T. (2000). *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.

Foster, I., Y. Zhao, I. Raicu, & S. Lu (2008, November). Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, pp. 1–10. IEEE.

Gantz, J. & D. Reinsel (2010, May). The digital universe decade - are you ready? Technical report, International Data Corporation.

Gantz, J. & D. Reinsel (2011, June). Extracting value from chaos. Technical report, International Data Corporation.

Ghemawat, S., H. Gobioff, & S. Leung (2003, October). The google file system. *SIGOPS Oper. Syst. Rev. 37*(5), 29–43.

Gnutella (2000, January). Gnutella website. http://www.gnutella.com/.

Google (2012). Google Cloud Storage. http://cloud.google.com/products/cloud-storage.html. [Online; accessed 30-Jun-2012].

Grune, D. (1986). Concurrent versions system, a method for independent cooperation. Technical report, IR 113, Vrije Universiteit.

Gudgin, M., M. Hadley, N. Mendelsohn, J.-J. Moreau, & H. F. Nielsen (2003, June). Soap version 1.2 part 2: Adjuncts. W3C Recommendation.

Gunawi, H. S., N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, & J. Schindler (2005, May). Deconstructing commodity storage clusters. *SIGARCH Comput. Archit. News 33*(2), 60–71.

Gupta, A., B. Liskov, & R. Rodrigues (2003). One hop lookups for peer-to-peer overlays. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, Berkeley, CA, USA, pp. 2–2. USENIX Association.

Hastorun, D., M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007). Dynamo: amazon's highly available key-value store. *IN PROC. SOSP 41*, 205—220.

Howard, J. H. (1988). An overview of the andrew file system. In *in Winter 1988 USENIX Conference Proceedings*, pp. 23–26.

Hunt, J., K. phong Vo, & W. F. Tichy (1996). An empirical study of delta algorithms.

IBM Corporation (2002, January). IBM white paper: IBM storage tank – a distributed storage system.

Jain, N., M. Dahlin, & R. Tewari (2005). Taper: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, Berkeley, CA, USA, pp. 21–21. USENIX Association.

Jayaram, K. R., C. Peng, Z. Zhang, M. Kim, H. Chen, & H. Lei (2011). An empirical analysis of similarity in virtual machine images. In *Proceedings of the Middleware 2011 Industry Track Workshop*, Middleware '11, New York, NY, USA, pp. 6:1–6:6. ACM.

Jin, K. & E. L. Miller (2009). The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, pp. 7:1–7:12. ACM.

Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, & D. Lewin (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, New York, NY, USA, pp. 654–663. ACM.

Kistler, J. J. & M. Satyanarayanan (1992, February). Disconnected operation in the coda file system. *ACM Trans. Comput. Syst. 10*(1), 3–25.

Krueger, M. & R. Haagens (2002). Small computer systems interface protocol over the internet (iscsi) requirements and design considerations.

Lakshman, A. & P. Malik (2010, April). Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*(2), 35–40.

Lamport, L. (1998, May). The part-time parliament. *ACM Trans. Comput. Syst. 16*(2), 133–169.

Lillibridge, M., K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, & P. Camble (2009). Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies*, Berkeley, CA, USA, pp. 111–123. USENIX Association.

Macko, P., M. Seltzer, & K. A. Smith (2010). Tracking back references in a write-anywhere file system. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, Berkeley, CA, USA, pp. 2–2. USENIX Association.

Mandagere, N., P. Zhou, M. A. Smith, & S. Uttamchandani (2008). Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, Companion '08, New York, NY, USA, pp. 12–17. ACM.

Manuel, S. (2008). Classification and generation of disturbance vectors for collision attacks against sha-1. stephane.manuel@inria.fr 14187 received 4 Nov 2008.

Mathur, A., M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, & L. Vivier (2007, June). The new ext4 filesystem: Current status and future plans. In *Proceedings of the 2007 Linux Symposium*, pp. 21–33.

Mattern, F. (1989). Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pp. 215–226. North-Holland.

Mell, P. & T. Grance (2009). The nist definition of cloud computing. *National Institute of Standards and Technology 53*(6), 50.

Mesnier, M., G. R. Ganger, & E. Riedel (2003, August). Object-based storage. *Comm. Mag. 41*(8), 84–90.

Mills, D. (1985, September). Network Time Protocol (NTP). RFC 958. Obsoleted by RFCs 1059, 1119, 1305.

MongoDB (2012). MongoDB document database. http://www.mongodb.org/. [Online; accessed 30-Jun-2012].

Muthitacharoen, A., B. Chen, & D. Mazières (2001, October). A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev. 35*(5), 174–187.

Muthitacharoen, A., R. Morris, T. M. Gil, & B. Chen (2002). Ivy: A read/write peer-to-peer file system. pp. 31–44.

Napster (1999). http://www.napster.com/.

Nath, P., M. A. Kozuch, D. R. O'hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, & M. Toups (2006). Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. *IN PROC. USENIX ANNUAL TECHINCAL CONFERENCE 3*, 363—378.

Nurmi, D., R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, & D. Zagorodnov (2009). The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, Washington, DC, USA, pp. 124–131. IEEE Computer Society.

OpenDedup (2012). A userspace deduplication file system (SDFS). http://code.google.com/p/opendedup/. [Online; accessed 30-Jun-2012].

OpenStack (2012). OpenStack Object Storage. http://www.openstack.org/software/openstack-storage/. [Online; accessed 30-Jun-2012].

Pawlowski, B., D. Noveck, D. Robinson, & R. Thurlow (2000). The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000.*

Pike, R., D. Presotto, K. Thompson, & H. Trickey (1990). Plan 9 from bell labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pp. 1–9.

Policroniades, C. & I. Pratt (2004). Alternatives for detecting redundancy in storage systems data. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, Berkeley, CA, USA, pp. 6–6. USENIX Association.

Quinlan, S. & S. Dorward (2002). Venti: A new approach to archival storage.

Rabin, M. O. (1981). Fingerprinting by random polynomials. *Technical Report TR1581 Center for Research in* (TR-15-81), 15–18.

Ratnasamy, S., P. Francis, M. Handley, R. Karp, & S. Shenker (2001, August). A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev. 31*(4), 161–172.

Rhea, S., R. Cox, & A. Pesterev (2008). Fast, inexpensive content-addressed storage in foundation. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, Berkeley, CA, USA, pp. 143–156. USENIX Association.

Rosenblum, M. & J. K. Ousterhout (1992, February). The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*(1), 26–52.

Rowstron, A. I. T. & P. Druschel (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, London, UK, UK, pp. 329–350. Springer-Verlag.

S3QL (2012). A full-featured file system for online data storage. http://code.google.com/p/s3ql/. [Online; accessed 30-Jun-2012].

Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, & B. Lyon (1985). Design and implementation or the sun network filesystem.

Satyanarayanan, M., J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, & C. Steere (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39*, 447–459.

Shepler, S., D. Noveck, & M. Eisler (2010, January). Network file system (NFS) version 4 minor version 1 protocol. RFC, Internet Engineering Task Force.

Srinivasan, K., T. Bisson, G. Goodson, & K. Voruganti (2012). idedup: latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, pp. 24–24. USENIX Association.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001, August). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev. 31*(4), 149–160.

Tolia, N., M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, & A. Perrig (2003). Opportunistic use of content addressable storage for distributed file systems. *IN PROCEEDINGS OF THE 2003 USENIX ANNUAL TECHNICAL CONFERENCE*, 127—140.

Torvalds, L. (2009). Git - fast version control system. http://git-scm.com/.

Trautman, P. & J. Mostek (2000). Scalability and Performance in Modern Filesystems. White paper, SGI, Mountain View, CA.

Vogels, W. (2007). Amazon's dynamo. `http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html`.

Wang, F., S. A. Brandt, E. L. Miller, & D. D. E. Long (2004). Obfs: A file system for object-based storage devices. In *IN PROCEEDINGS OF THE 21ST IEEE / 12TH NASA GODDARD CONFERENCE ON MASS STORAGE SYSTEMS AND TECHNOLOGIES, COLLEGE PARK, MD*, pp. 283–300.

Watson, J. (2008, February). Virtualbox: bits and bytes masquerading as machines. *Linux J. 2008*(166).

Weil, S. A., S. A. Brandt, E. L. Miller, D. D. E. Long, & C. Maltzahn (2006). Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, Berkeley, CA, USA, pp. 307–320. USENIX Association.

Weil, S. A., S. A. Brandt, E. L. Miller, & C. Maltzahn (2006). Crush: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA. ACM.

Woo, B. (2010, October). Virtual storage delivers a Three-Dimensional approach to storage scaling. Technical report, International Data Corporation.

You, L. L. & C. Karamanolis (2004). Evaluation of efficient archival storage techniques.

You, L. L., K. T. Pollack, & D. D. E. Long (2005). Deep Store: An Archival Storage System Architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, Washington, DC, USA, pp. 804–8015. IEEE Computer Society.

Zhang, X., Z. Huo, J. Ma, & D. Meng (2010, September). Exploiting data deduplication to accelerate live virtual machine migration. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pp. 88 –96.

Zhao, B. Y., J. D. Kubiatowicz, & A. D. Joseph (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA.

Zhu, B., K. Li, & H. Patterson (2008). Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, Berkeley, CA, USA, pp. 18:1–18:14. USENIX Association.