Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Alapan Mukherjee

Benchmarking Hadoop performance on different distributed storage systems

Master's Thesis
Espoo, June 19, 2015

Supervisor:	Assoc. Prof. Keijo Heljanko
Advisor:	M.Sc. (Tech.) Rıdvan Döngelci

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Alapan Mukherjee		
Title: Benchmarking Hadoop performance on different distributed storage systems			
Date:	June 19, 2015	**Pages:**	110
Major:	Data Communication Software	**Code:**	T-110
Supervisor:	Assoc. Prof. Keijo Heljanko		
Advisor:	M.Sc. (Tech.) Rıdvan Döngelci		

Distributed storage systems have been in place for years, and have undergone significant changes in architecture to ensure reliable storage of data in a cost-effective manner. With the demand for data increasing, there has been a shift from disk-centric to memory-centric computing - the focus is on saving data in memory rather than on the disk. The primary motivation for this is the increased speed of data processing. This could, however, mean a change in the approach to providing the necessary fault-tolerance - instead of data replication, other techniques may be considered.

One example of an in-memory distributed storage system is Tachyon. Instead of replicating data files in memory, Tachyon provides fault-tolerance by maintaining a record of the operations needed to generate the data files. These operations are replayed if the files are lost. This approach is termed lineage. Tachyon is already deployed by many well-known companies.

This thesis work compares the storage performance of Tachyon with that of the on-disk storage systems HDFS and Ceph. After studying the architectures of well-known distributed storage systems, the major contribution of the work is to integrate Tachyon with Ceph as an underlayer storage system, and understand how this affects its performance, and how to tune Tachyon to extract maximum performance out of it.

Keywords: Tachyon, HDFS, Ceph, benchmarks

Language: English

Acknowledgements

The work reported in this thesis was conducted at Aalto University, School of Science, in collaboration with CSC - IT Center for Science Ltd. It was funded by the D2I (Data to Intelligence) project.

I would like to express my gratitude to Professor Keijo Heljanko for giving me an opportunity to work in this project. His supervision, extensive knowledge of the subject, patience and practical advice were invaluable to me during the work. I am also particularly grateful to my advisor Rıdvan Döngelci, a doctoral student at Aalto University, for his help with the configurations and code modifications performed in the experiments. I want to thank my colleagues at CSC, for allowing me access to their cloud resources, and for the stimulating discussions we had during various stages of the work. Lastly, I wish to thank my family, whose constant encouragement and blessings made this work possible.

Thank you!

Espoo, June 19, 2015

Alapan Mukherjee

Abbreviations and Acronyms

HTTP	Hypertext Transfer Protocol
EMR	Elastic Map Reduce
MPI	Message Passing Interface
REST	REpresentational State Transfer
NFS	Network File System
SCSI	Small Computer System Interface
CIFS	Common Internet File System
CPU	Central Processing Unit
SAN	Storage Area Network
NAS	Network Attached Storage
DAS	Direct Attached Storage
RAID	Redundant Array of Independent Disks
RADOS	Reliable Autonomic Distributed Object Store
OSD	Object Storage Device
CRUSH	Controlled Replication Under Scalable Hashing
QEMU	QUick EMulator
RBD	RADOS Block Device
POSIX	Portable Operating System Interface
HDFS	Hadoop Distributed File System
RAM	Random Access Memory
GFS	Google File System
GB	Gigabyte
MB	Megabyte
KB	Kilobyte
PB	Petabyte
I/O	Input/Output
API	Application Programming Interface
ID	Identification
WAS	Windows Azure Storage
DNS	Domain Name Service

OT	Object Table
DFS	Distributed File System
FE	Front End
LS	Location Service
URL	Uniform Resource Locator
IP	Internet Protocol
TOR	Top-of-the-rack
FD	Fault Domain
UD	Upgrade Domain
CRC	Cyclic Redundancy Checksum
OST	Object Storage Target
BSD	Berkeley Software Distribution
LDLM	Lustre Distributed Lock Manager
MDT	Metadata Target
MGS	Management Server
LNET	Lustre Networking
OSS	Object Storage Server
ODB	Object-based disk
MGC	Management Client
MDC	Metadata Client
OSC	Object Storage Client
LOV	Logical Object Volume
VFS	Virtual File System

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Storing very large data files in distributed storage systems is necessary to provide reliability and availability. A scaling-out approach is typically used in these systems - instead of using a single high-end computer as the storage node, several computers consisting of relatively inexpensive commodity hardware are deployed for storing the data. One major implication of using commodity hardware is that hardware failures are common. This in turn means two things - 1) multiple copies of the data have to be maintained in different nodes to ensure availability of the data if one or more nodes go down, and 2) suitable frameworks have to be in place which can perform data-processing despite hardware failures, i.e. the system has to be fault-tolerant. Hadoop, which uses the map-reduce programming model, is an example of such a framework. In such an approach, the data are stored on the disk.

As the importance of speed in data-processing has increased, there is greater focus on storing the data in memory (i.e. RAM) rather than on the disk. Having data in memory results in significantly reduced query response times, which is suitable for data analytics applications, for example. Data stored in-memory can also reduce the need for data indexing. With the reduction in RAM costs, and the increased RAM space in 64-bit operating systems, in-memory-data processing is becoming an increasingly feasible form of data processing.

The shift towards in-memory data processing was highlighted by the development of Apache Spark by UC Berkeley's AMPLab. By use of in-memory data structures, it can increase the speed of data processing manyfold, compared to a disk-based framework like Hadoop. Another key step towards in-memory data processing is the development of the in-memory storage system Tachyon, at the aforementioned AMPLab. Instead of replicating multiple copies of the data, it uses a technique called lineage to maintain a log of the transformations needed to regenerate that data. In the event of data loss, these transformations are replayed to regenerate the data. Apache Ignite is yet another recent development in the field of in-memory computing. Unlike Spark, which uses RAM only for processing, Ignite provides an in-memory file system, as well as support for computing paradigms like MapReduce and MPI [2]. It has been developed as the first enterprise-level In-Memory Data Fabric by GridGain Systems. GridGain's In-Memory Data Fabric entered Apache Incubator (entry path into Apache Software Foundation for projects wanting to become a part of Apache) under the name of Apache Ignite.

Tachyon operates on top of another storage system, known as the underlying storage system. The storage systems currently supported by Tachyon are Hadoop Distributed File System (HDFS), Amazon S3, GlusterFS, and the local filesystem in a computer. This way, it allows data to be written to a filesystem while still in memory, as well as leverages the fault-tolerance of an underlayer filesystem like HDFS. The Tachyon project currently has over 40 contributors from companies like Yahoo and Intel, and is deployed by many companies.

1.1 Thesis objective and scope

The objectives of this thesis are:

1. Integrate Tachyon with Ceph Object Storage - to do this, the source code of Tachyon has to be changed to allow it use Ceph as the underlying file system, because this was not supported at the time of writing.

2. Write the files produced in intermediate stages of data-processing with Tachyon to memory, rather than to the disk - this is done to speed up data-processing. Again, this is not supported by Tachyon at the time of writing, as usually more persistent storage is used for such files.

3. Benchmark the read-only, write-only, and read-write performances of Tachyon, Ceph and HDFS with large datasets.

The main motivation of this work is to determine the performance of an open-source implementation of Amazon's Elastic MapReduce (EMR) service, which is highly effective in big data processing. EMR uses its own Hadoop framework to process large amounts of data across dynamically scalable Amazon EC2 instances, and S3 object storage to store that data. The equivalent open-source implementation of this would be an HPC (High-performance computing) cluster with compute nodes using Apache Hadoop, as well as other such frameworks, for the data processing, and storage nodes using Ceph Object Storage for data storage. The source code of Hadoop would have to be modified to point to the Ceph storage cluster instead of HDFS.

One use case of such an implementation would be in a bioinformatics pipeline. This pipeline would have a number of intermediate processing stages - of the order of ten pipeline stages. The input data may be read from Ceph, and the output data written to Ceph. However, a large number of temporary files can be produced in the intermediate stages. Writing these files to disk is time-consuming, especially when one considers the replication used in storage systems like Ceph and HDFS. Replicating temporary files which can be easily regenerated in the pipeline anyway, purely for fault-tolerance, slows down the overall speed of processing in the pipeline and is unnecessary. This is especially true for file writes, as reads from disk may be sped up by caching.

In such a scenario, writing the temporary files to memory can speed up the entire pipeline process significantly. Tachyon is the ideal choice here, as it stores data in-memory, and uses lineage, rather than replication, for regenerating lost data. This saves memory space, as well as time. Tachyon is layered on top of an persistent storage system, called the underlying file system.

For data processing, Apache Spark, instead of Hadoop, has been used in the experiments. Spark uses in-memory data storage for very fast, iterative queries. It does so by using data structures called resilient distributed datasets (RDDs). RDDs are distributed collections of objects that can be cached in memory across cluster nodes. They are automatically rebuilt after failures, using the same principle of lineage explained earlier. RDDs can be manipulated through various operations. Spark is compatible with Hadoop's storage APIs, which means it can read/write to Hadoop-supported systems such as HDFS.

1.2 Structure of the thesis

The remainder of the thesis is as follows: Chapters 2-10 describe the architectures of different types of distributed storage systems. Details of the experiments performed and results obtained are presented in Chapter 11. The work concludes with the summary and scope for further work in Chapter 12.

Chapter 2

Overview of storage systems

Three main types of storage are present - file, block and object. Each has its own advantages and specific use cases.

2.1 File storage

A file is a part of a filesystem. In a filesystem, files are organised in a hierarchical manner so that a particular file can be referred to by its path. Each file, along with its data, also has its own metadata. The metadata of a filesystem typically consists of information such as when was the file created and last modified, its size, who can access it, and so on. Filesystems are provided on our personal computers by the operating system, and we interact with them on a daily basis. Users with access to files can read or write them.

To share files securely across a network, network-attached storage (NAS) is the usually preferred option [42]. This works well in a local-area network (LAN) and in a wide-area network (WAN). The filesystem places the data on the NAS box, and implements file sharing by locking and unlocking files as necessary. While managing a small number of NAS boxes is simple, doing so with a large number of such boxes is much more difficult. The metadata for files being shared over NAS are managed completely by the file server. Although this enables cross-platform sharing, passing all the I/O though the single file server makes the file server a single-point of failure. Clients will be limited by the performance of the file server.

Filesystems are suitable for a reasonably large number of files - in the range of thousands to a few millions. However, they are difficult to use when having billions of files. Thus, files are suitable if they are relatively small in number, and are available locally.

2.1.1 Data striping

Although not restricted to only filesystems, it is worthwhile to consider an important aspect of computer storage before progressing further - data striping. Striping is the process of dividing logically sequential data into segments, and then storing the segments on different physical storage devices such as hard disks [1]. This is useful as the storage devices can be accessed concurrently by the processor. Multiple, independent requests can be served in parallel by separate disks, reducing the queuing time seen by I/O requests. Also, single requests for multiple blocks can be serviced by multiple disks in a coordinated way. This improves the effective transfer rate seen by a single request. Therefore, striping increases the overall throughput and hence performance, compared to when having all the segments on the same physical device.

There are two types of disk striping - single user and multi-user. Single user disk striping results in multiple hard disks simultaneously handling several I/O requests from one workstation. Multi-user disk striping allows several I/O requests from multiple workstations to be handled by multiple hard disks. Disk striping may be used with parity. When parity is used in striping, an extra stripe containing the parity information is stored on its hard disk and partition. When a hard disk fails, a fault tolerance driver reads content from the other disks and uses the parity block to regenerate the missing disk. While this is the main recovery techniques in redundant array of independent disks (RAID) level 5, in RAID 6, two parity blocks are used to allow for the failure of two hard disks.

2.2 Block storage

In block-level storage, raw volumes of storage are created and each block can be controlled as an individual hard drive [4]. Each block has an address, and an application retrieves a block by making a small computer system interface (SCSI) request for that address [42]. SCSI itself refers to a set of standards for physically connecting and transferring data between computers and peripheral devices. Importantly, a block has no real metadata, and only has the address as its identifier.

Unlike NAS for filesystems, where the filesystem determines where to place the data and how to access it, for block storage this task performed by the application itself. This allows the application to exercise much more granular control from a given storage array than in the case of files. This property is particularly useful for performance-centric applications such as database servers, and to allow the user to flexibly decide which filesystem to use for which server.

Examples of block-based storage in use today are direct-attached storage (DAS) and storage-area networks (SAN). DAS connects block-based storage devices directly to the I/O bus of host machines, via SCSI or SATA/SAS for example [27]. This provides high performance and security, but connectivity can be a problem. SCSI is limited by the width of the I/O bus. This concern for connectivity, and the desire to improve sharing of storage devices led to the development of SAN systems. These provide a switched fabric for enabling fast interconnection between a large number of hosts and storage devices.

Adding distance between the application and storage adversely affects the performance of block systems. Thus block storage is mostly used locally. The key points about block storage are therefore good performance, local storage, very little to no metadata, and granularity of control to the application.

2.3 Object storage

Many distributed storage systems are based on object storage, in which conventional hard disks are replaced by intelligent object storage devices (OSDs), which combine a CPU, network interface and a local cache with an underlying disk. An object consists of data along with all its metadata, all packaged together to form an object [42]. This object is given an ID that is calculated based on its content (data and metadata). Objects usually have a flat naming structure, unlike files in filesystems. Objects are located in a pool/bucket, and are referred to by their IDs. The objects may be locally present or geographically dispersed, but due to their flat naming structure, are retrieved in exactly the same way in either case. Unlike file storage, object storage does not allow writing/modifying only a part of the file. The entire object must be updated as a whole. Some of the commercial object storage systems are Amazon S3, EMC Atmos and Rackspace's Cloud Files.

Objects allow users to customise the metadata. Unlike the limited details available in filesystem metadata, one can add a lot more details to object storage metadata to open up greater opportunities for data analytics. For example, the metadata could contain details such as the type of application the object is associated with, the level of data protection desired for the object, whether it can be geographically replicated or not, when to delete it, and so on.

Object storage is ideal for storing content that can grow without bounds. Use cases include backups, archives and static web content such as images. For storing a very large amount of data across different locations with extensive metadata, object storage is ideal.

As the target is to store a very large amount of data at a low cost, frequently object storage makes use of clusters of commodity servers. This is ideal as scaling then becomes a question of simply adding more nodes. Data protection is usually achieved by replicating the objects to one or more nodes in the cluster.

Unlike a file or a block, an object can be accessed using a HTTP-based REST application programming interface (API). Calls such as GET and POST are sufficient. For file and block storage, SCSI, CIFS or NFS calls are used.

In an object storage system, space for the objects is allocated by the object store itself, and not by higher level software such as a filesystem [12]. All operations on an object, such as reading/writing at a logical location in the object and deleting the object carry a credential. The object store should verify that the user's request carries a valid credential. This allows the storage system to enforce different access rights for different parts of a volume, thus increasing the granularity of access.

Chapter 3

Ceph

Ceph is a massively scalable, distributed storage system designed to provide object, block and file storage within a single computer cluster. It scales to the exabyte level, has no single point of failure, and is fault-tolerant enough to run on commodity hardware. Usage of the platform is free of cost,and its development is open-sourced to a large community. Work on Ceph began in 2006 as a part of Sage Weil's PhD dissertation at UC Santa Cruz. Weil later created Inktank, which is the lead development contributor and sponsor of Ceph. In 2014, Inktank was acquired by Red Hat.

Ceph is an example of software-defined storage. An explanation of software-defined storage is given in the next section.

3.1 Software-defined storage

Software-defined storage refers to the storage infrastructure that is managed and automated by intelligent software, instead of by the storage hardware itself [33]. This way, infrastructure resources in a software-defined storage (SDS) system can be automatically and efficiently allocated to match the application needs of an enterprise.

By separating the storage hardware from the software that manages it, SDS enables enterprises to purchase heterogeneous storage hardware without having to worry about issues such as interoperability and under/over-utilization of specific storage resources. The software that enables a SDS environment can provide functionality such as replication, thin provisioning and snapshots. The key benefits of SDS over traditional storage are increased flexibility, automated management and cost efficiency.

Typically, SDS definitions include a form of storage virtualization to separate the storage hardware from the software that manages the storage infrastructure. The characteristics of SDS may include any or all of the following features:

1. Abstraction of logical storage devices and capabilities from the underlying physical storage systems, and in some cases pooling across multiple different implementations. Since moving data is expensive and slow compared to computation and services, pooling approaches suggest leaving data in place and creating a mapping layer to it that spans arrays. In data storage, an array is a method for storing information on multiple devices. A disk array, for example, provides increased availability, resilience and maintainability by using existing components such as controllers, power supplies, fans etc, often upto the point where single points of failure are eliminated from the design. Examples of pooling include:

    - Storage virtualization - External-controller based arrays include storage virtualization to manage usage access across the drives within their own pools.
    - OpenStack and its Swift and Cinder APIs for storage interaction, which have been applied to open-source projects as well as vendor products.
    - Parallel NFS (pNFS) - an implementation which developed within the NFS community and has expanded to other implementations.

2. Commodity hardware with storage logic abstracted into a software layer. This is also termed as a clustered file system for converged storage. Converged storage refers to co-existing virtual machines and storage on single hardware. Open-source examples of this include GlusterFS, Ceph, and VMWare Virtual SAN.

3. Scale-out storage architecture - add more commodity storage nodes to increase performance and capacity than acquire a single supercomputer. Virtually every provider says its products is scale-out, and easy-to-manage i.e. procedure of adding a new node does not depend on the size of the cluster.

4. Automation with policy-driven storage provisioning, and with service-level agreements replacing technical details.

3.2 Ceph features

Ceph provides three ways to store and access the data:

1. RADOS - the default, object storage mechanism.
2. RBD (RADOS Block Devices) - as a block device. The Linux kernel RBD driver allows striping of a block device over multiple object storage devices (OSDs).
3. CephFS - as a file, a POSIX-compliant filesystem, which allows users to use commands such as *ls* and *find* without putting a strain in the underlying object storage system. CephFS has not yet been fully developed, and is currently not recommended for production.

The key reason for Ceph's scalability is the absence of a centralized interface to the object store [10]. In many distributed storage systems, this interface provides services to a client. The interface is accessible by the user, e.g. the namenode in HDFS. This could become a bottleneck in large-scale data storage. Ceph does not have any such single interface in its architecture. Each OSD daemon and client (in CephFS) knows about other OSD daemons in the cluster. This means OSD daemons can directly interact with each other and monitors (explained later). Ceph clients can also interact directly with the OSD daemons without having to go through the aforementioned interface.

The different components of each storage type are described below:

3.3 Ceph object storage

Object storage is the fundamental part of any Ceph deployment. A Ceph object storage cluster has two types of daemons - object storage devices (OSD) and a monitor. A Ceph object storage cluster will have as its bare minimum one Ceph monitor and two OSDs for data replication.

3.3.1 RADOS

The distributed object store of Ceph is termed Reliable Autonomic Distributed Object Store (RADOS). RADOS manages the distribution, replication and migration of objects [44]. Data objects are distributed across OSDs. To direct requests, RADOS sends cluster maps to the client and the MDS (present in Ceph filesystems, described later). These cluster maps contain information about participating OSDs, their statuses and the CRUSH mapping function. A cluster map is actually a collection of five different maps - the monitor map, OSD map, placement group (PG) map, CRUSH map, and a metadata server (MDS) map. RADOS can be accessed via different interfaces:

1. RADOS gateway
2. *librados* and related C/C++ bindings
3. *rbd* and *QEMU-RBD* Linux kernel and QEMU block devices that stripe data across multiple objects

RADOS consists of object storage devices (OSDs). Along with storage, data replication is also managed by OSDs. OSDs actively collaborate with each other for data replication and recovery in the case of failures. Objects and their replicas are placed in placement groups.

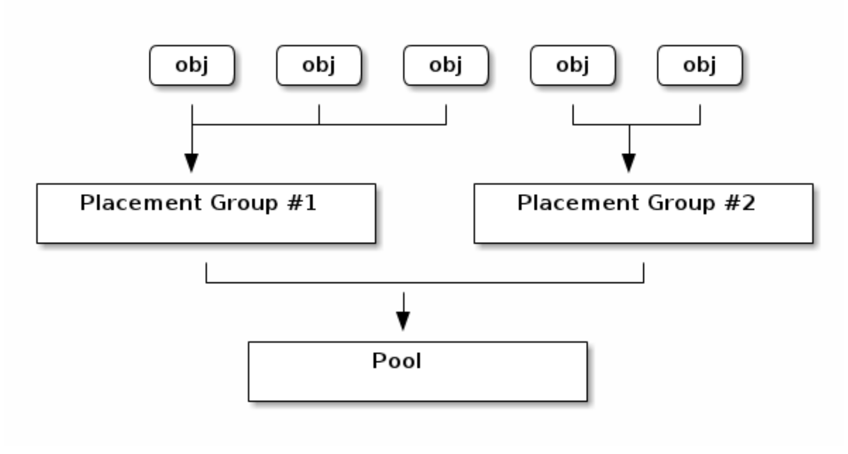

Figure 3.1: Objects are placed in a placement group [30]

Placement groups form 'pools' of objects. The CRUSH algorithm (described later) determines the placement group in which an object will be placed. The objects of a placement group are placed in different OSDs. It is important to note that, a placement group consists of several objects, while an OSD stores objects belonging to different placement groups [30].

One of the OSDs is called *primary*. This primary OSD serializes all requests to the placement group. When it receives a write request, the primary OSD forwards it to all other OSDs in the group. If it has to write locally as well, it does so only after the other OSDs have written the replicas to their

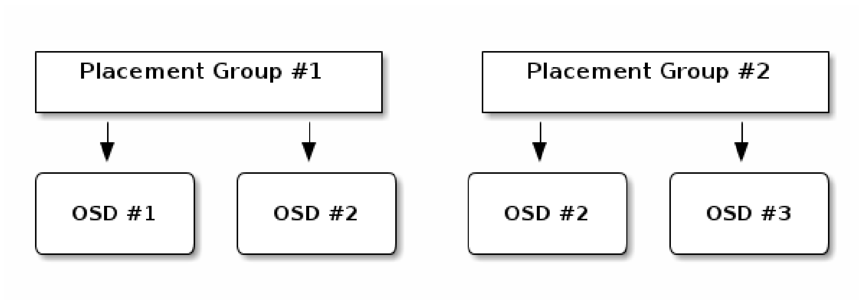

Figure 3.2: Placement groups are mapped to OSDs [30]

memories. The client (in case of a filesystem, described later) then receives an acknowledgement from the primary OSD. At this point, the data have been replicated in memory in all the replica OSDs. The client still has the data in its buffer cache. Only after the data have been committed to the disks of all the replica OSDs does the primary OSD send a *commit* notification to the client. The client can then delete the data from its buffer cache.

OSDs use btrfs as the default filesystem, although ext3 is also available. Data are asynchronously written using the copy-on-write method. This means that once-writen data are no longer directly modified. Instead, the data to be changed are written in newly allocated blocks, and the relevant metadata are accordingly updated to point to the new blocks, rather than the old ones. This ensures unsuccessful write operations can be fully rolled back. Each OSD maintains a log of object versions for each placement group in which it participates. If an OSD fails, the other OSDs of the placement group can identify missing objects by comparing the logs, and thus recover the object. Each OSD monitors the state of other OSDs in its placement group using heartbeat messages, which are sent along with replication traffic. When an OSD detects an unresponsive OSD, it notifies the monitor of the situation and, in response, receives a new cluster map from the monitor marking the unresponsive OSD as *down*. Heartbeat messages are still sent to this unresponsive OSD and, after some time, if it still does not respond, the monitor is accordingly alerted and it issues a new cluster map marking that OSD as *out*.

An OSD daemon can compare object metadata with that of its replicas in other OSDs, possibly in different placement groups [10]. This is known as *scrubbing*. OSDs can also perform deeper scrubbing by comparing data in objects bit-by-bit. While metadata scrubbing is performed daily, data scrubbing is performed on a weekly basis, and can locate bad sectors in a drive.

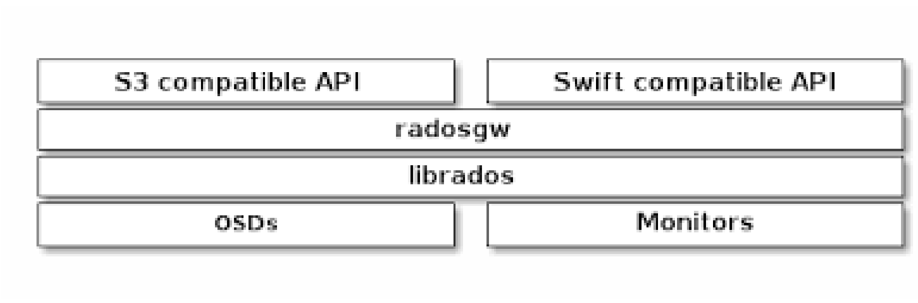

Figure 3.3: Ceph Object Storage components [10]

Ceph Object Storage uses the Ceph Object Gateway daemon (radosgw) for interacting with a Ceph storage cluster [10]. It provides interfaces compatible with OpenStack Swift and Amazon S3. Ceph Object Gateway can store data in the same Ceph storage cluster used to store data from the filesystem or block device clients. The S3 and Swift APIs share a common namespace, allowing data to be written with one API and retrieved with the other.

3.3.2 Monitor

Management of the cluster map is performed by the Ceph monitors [28]. The monitor maintains a master copy of the cluster map, and provides authentication and logging services. A client can determine the location of all monitors (there can be multiple monitors in a cluster), OSD daemons and metadata servers by connecting to one monitor and getting a cluster map. With the current copy of the cluster map and the CRUSH algorithm, the client can compute the location for any object. By computing the object locations, the client can communicate directly with the OSDs, which contributes to Ceph's scalability.

All changes in the monitor services are written by Ceph to a single PAXOS instance, and PAXOS writes the changes to a key-value store for strong consistency. The monitors query the most recent version of the cluster map during sync operations. It uses the key-value stores's snapshots and iterators (using leveldb) to perform store-wide synchronization. When object storage devices fail or new devices are added, monitors detect and create a new cluster map.

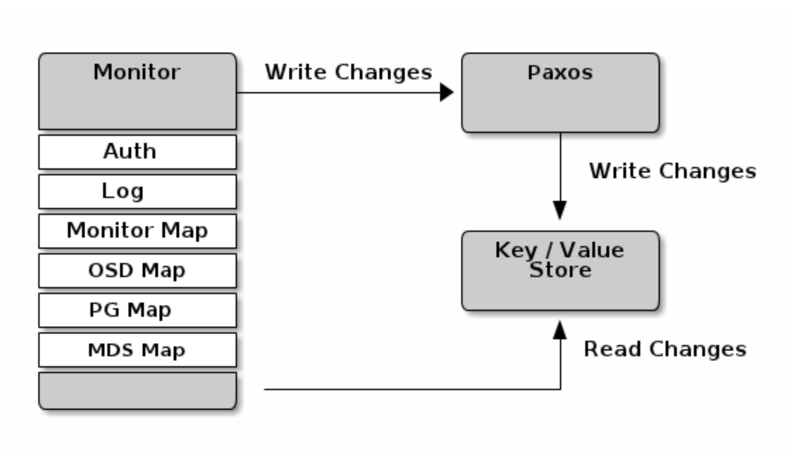

Figure 3.4: Key-value store created by PAXOS [28]

3.3.3 CRUSH

CRUSH (Controlled Replication Under Scalable Hashing) is a pseudo-random data distribution algorithm that efficiently distributes objects across a storage cluster [47]. It maps inputs such as the identifier of a placement group (a group of objects) to a list of OSDs in which the object and its replicas are to be stored. Along with the placement group identifier, it takes as input a cluster map, a set of placement rules and a replication factor. The cluster map details the available storage resources in the cluster and the logical elements from which they are built. For example, a cluster map may describe a large installation in terms of rows of server cabinets, cabinets filled with disk shelves, and shelves filled with storage devices. The data distribution policy is defined in terms of placement rules which specify how many storage devices are chosen from the cluster for the replicas and what restrictions are present for replica placement. An example of this would be a rule stating that three replicas have to be placed in devices on different server cabinets so that they do not share the same electrical circuit.

Ceph uses this to determine the location of data storage, instead of large look-up tables, which can grow very large for a lot of data. The algorithm is consistent in that changes in node numbers result in minimal object migration to re-establish uniform object distribution.

Having the cluster map allows CRUSH to identify and address potential sources of correlated device failures. Such sources could include a shared power source or a shared network. By having information about these in cluster maps, CRUSH can determine the object locations in such a way that problems with these potential sources are avoided. For example, to address

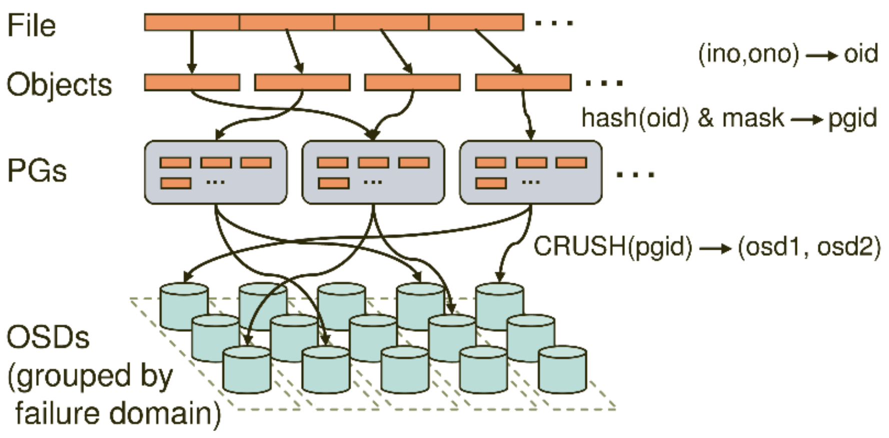

Figure 3.5: Distributed object storage [19]

the problem of device failures due to a power supply, CRUSH can place the objects in devices attached to different power supplies.

Developing a detailed CRUSH map helps in identifying failures more quickly. For example, if an OSD goes down, the map and CRUSH can identify the physical data center, room, row and rack of the host with the failed OSD in case hardware has to be replaced or onsite support is needed.

3.4 Ceph block storage

Ceph can be mounted as a thinly-provisioned block device [10]. The block device is thin-provisioned as it is allocated on a just-enough and just-in-time basis using virtualization technology, as opposed to the traditional way of allocating all blocks up front. This is built on top of the object storage system.

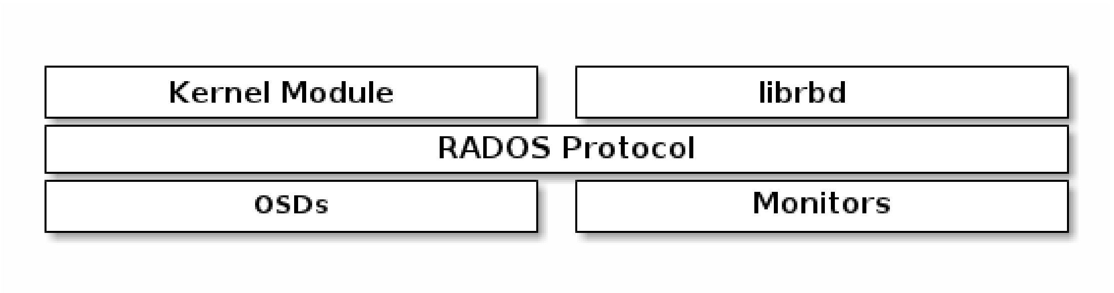

Figure 3.6: Ceph Block Device components [10]

Thus, when an application writes data to Ceph using a block device, the RADOS block device (RBD) component of Ceph automatically stripes and replicates the data across the cluster. Ceph's RBDs interact with OSDs using kernel modules or the *librbd* library. Librados is a library which allows applications to directly access RADOS and hence its capabilities. The Ceph RBDs are built on top of librados, which means it can utilise RADOS features such as taking snapshots, maintaining consistency and replication.

3.5 Ceph filesystem(CephFS)

Ceph's filesystem is built on top of the same object storage system that is responsible for object and block storage. Currently, it is not recommended to be run in production. The main components of CephFS are the metadata server and the client. The metadata server maps the directories and file names of the filesystem to objects stored within RADOS clusters. Ceph clients mount the CephFS filesystem as a kernel object or as a filesystem in user space (FUSE).

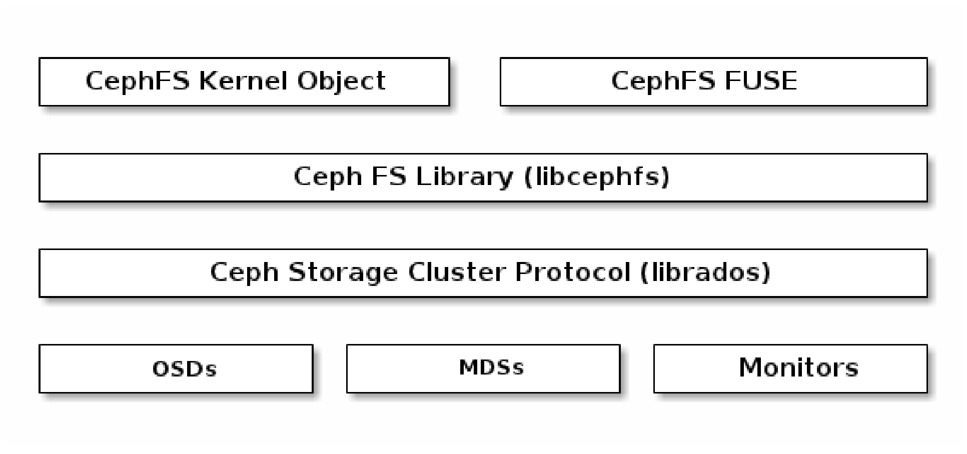
CephFS Kernel Object
CephFS FUSE
Ceph FS Library (libcephfs)
Ceph Storage Cluster Protocol (librados)
OSDs
MDSs
Monitors

Figure 3.7: CephFS architecture components [10]

The functions of these components are explained in greater details below:

3.5.1 Metadata server

The metadata server (MDS) manages the filesystem's namespace [17]. Although data and metadata are both stored in the same object storage cluster, they are managed separately to support scalability. The metadata are divided among a cluster of MDSs which actively replicate and distribute the namespace. The portions of the namespace managed by two MDSs can overlap. Periodically, the distribution of the namespace is changed by migrating responsibility for different subtrees of the structure to different MDSs. This is termed as dynamic subtree partioning, and allows MDS daemons to be added or removed at any time, depending on workloads which change dynamically. Even during large workload changes, this redistribution takes place in seconds. Large or heavily written directories are replicated for load distribution and storage. Clients are notified of relevant partitions when they communicate with the MDS.

In the Linux filesystem, basic information about a file, directory or object is stored in a data structure called index node (inode) [25]. Metadata such as file type (executable, block type etc), permissions, owner and file access are stored in the inode. The inodes are embedded in directories and stored with each directory entry, in the OSDs.

The main application of the metadata server is being an intelligent cache, because the actual metadata is eventually stored within the object storage cluster. Metadata to write is cached in a journal for a short time period, and then pushed to physical storage [17]. Due to its caching property, metadata servers can serve recent metadata back to clients. The journal is necessary for failure recovery - if the metadata server fails, the journal is replayed to ensure the metadata is safely stored on disk. This is useful when trying to bring a currently inconsistent filesystem (possibly resulting from a power failure or system crash) to its previously consistent state. For example, file deletion in Linux consists of two steps - 1) delete the directory entry and 2) mark the space for the file and its inode as free. If a power failure occurs while only either of the steps has been executed, the filesystem will be left in a corrupted state i.e. a not-yet-deleted would be marked as free, or an inode does not have the file it refers to. In such a situation, when the power is available again, the changes logged in the journal are replayed from the beginning to system returns to its earlier consistent state.

3.5.2 Client

Linux presents a common interface (virtual filesystem switch). Due to this interface, the user views Ceph as a single mount point, from which standard file I/O can be performed. The user is not aware of the underlying metadata servers, monitors and object storage devices that make up the system. The user interacts with Ceph through the client. Ceph has a user-level client, as well as a kernel-level one. The former is linked directly to the application or used via FUSE. The latter is available in the mainline Linux kernel. In many filesystems, the control and intelligence are implemented within the kernel's filesystem source itself. For Ceph, the filesystem's intelligence is distributed across the nodes. This simplifies the client interface and makes the storage system more scalable.

A standard Linux file is assigned an inode number (INO) by the metadata server, which is the unique identifier for the file. The file is then divided into a number of objects (depending on the file size). Using the inode number, the object number, and information like striping strategy and replication factor, an object identifier(OID) is created for the object. This OID is then hashed to generate a placement group ID, whereby the object is placed in a placement group. The placement group is a group of OSDs which stores an object and all of its replicas. The placement groups are mapped to OSDs by CRUSH.

For example, a client wanting to opening a */foo/bar* file for reading sends an "open for read" message to the MDS [25]. The MDS reads directory */foo* from the appropriate object storage devices (OSDs) and returns the capability for reading */foo/bar* to the client. The capability contains the inode number, the replication factor, and information about the striping strategy of a file. The client uses the inode number , striping strategy, and an offset to calculate the object identifier (OID), which is then hashed to a placement group ID. This then is mapped to OSDs by CRUSH.

CRUSH takes as input the 1) placement group ID, 2) the replication factor, 3) the current cluster map and 4) placement rules. As output, it returns an ordered list of OSD IDs to the client, which then picks the first one on the list, called the primary OSD (explained later).

By allowing Ceph clients to contact OSD daemons directly, both performance and total system capacity are significantly improved [10].

Chapter 4

OpenStack Swift

OpenStack object storage, known as Swift, is an open-source implementation of a scalable, durable, distributed object storage system using clusters of standardized servers [43]. Swift can be used in clusters ranging in size from a couple of nodes with a few hard drives as storage to thousands of geographically-distributed nodes capable of providing exabytes of storage. Swift is designed to store, among others, files, videos, virtual machine snapshots and web content. It is already widely used in production clouds of Rackspace, HP, IBM and many other private storage clusters. Swift runs on Linux distributions and standard x86 server hardware.

All objects stored in Swift have a uniform resource locator (URL) [29]. As with any other object storage system, the objects possess extensive metadata, which can be indexed or searched. All objects are replicated and the replicas are placed in regions as unique as possible. Applications store and retrieve these objects via an industry-standard RESTful HTTP API. When adding or removing hardware from the storage cluster, the data does not have to be migrated to a new storage system. There is no downtime when the nodes are being added or removed. Swift's service is highly available and partition-tolerant, whilst being eventually consistent. In eventual consistency type of storage, modifications will be made to all the replicas and become eventually (if not right away) visible to all clients.

4.1 Architecture of a Swift cluster

A Swift cluster is a collection of machines or nodes running server processes and consistency services. There are four types of server processes - *proxy*, *account*, *container*, *object*. Objects represent the actual data, and containers and accounts are used to group the objects. The server processes can be referred to as *layers* as well, e.g. the proxy layer is used to refer to the proxy server processes running in the cluster (likewise with account, container and object layers) [29]. A node running only the proxy server process is called a proxy node. Nodes running one or more of the other three processes are called storage nodes. These nodes contain the data which the requests from clients can affect, e.g. an object can be PUT in the storage drive of one of these nodes. The storage nodes will also need to run some services to maintain consistency.

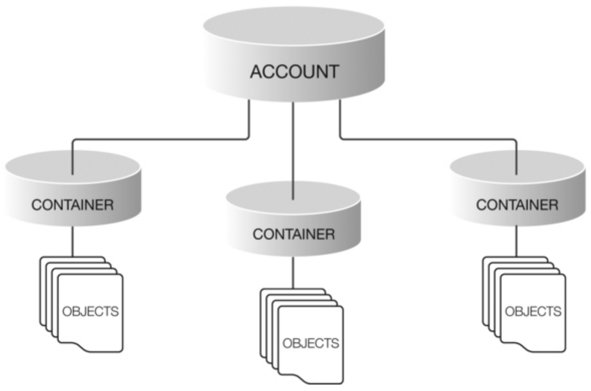

Figure 4.1: Account-container-object hierarchy [3]

A node can belong to *regions* and *zones* in a cluster [29]. These are user-defined and represent unique characteristics of a collection of nodes. These characteristics are often geographical location and points of failure, such as the power running to one rack of nodes. These ensure Swift places the data in different parts of the cluster to reduce the risk.

Regions are usually indicated by physically separate parts of the cluster, i.e. located in different geographical places. A cluster has a minimum of one region and there are many single-region clusters. Clusters with two or more regions are called multi-region clusters.

When a read request is made, the proxy server (explained in details later) prefers nearby copies of the data to those in faraway regions to reduce the latency. When a write request is made, the proxy layer, by default, writes to all locations simultaneously. An option called 'write affinity' also exists which when enabled allows the cluster to write copies locally and then transfer them asynchronously to the other regions.

Within a region, certain *zones* can be configured to isolate failure boundaries. An availability zone is defined by a set of physical hardware whose failures would not affect other zones in the region. For example, in a single datacenter, the availability zones may be different racks. A cluster should have at least one zone, and usually there are several zones in a cluster.

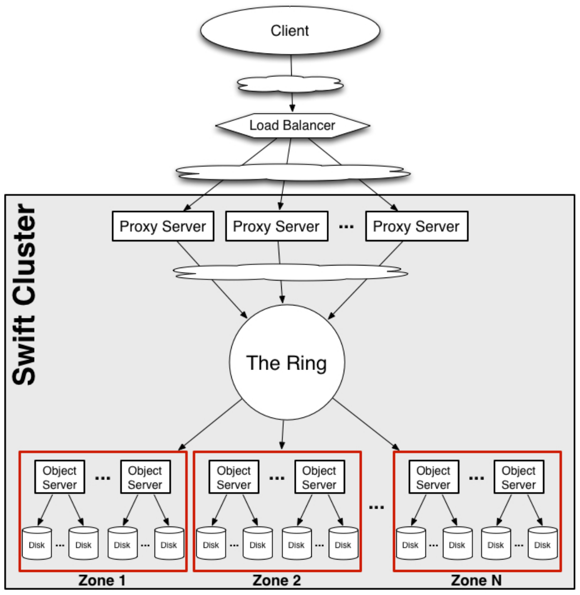

Figure 4.2: Layout diagram of Swift cluster [34]

4.1.1 Ring

Along with the components mentioned above, Swift also uses an internal data structure called the *ring*. The ring is a mapping between the names of objects on disk and their physical locations [29]. There are separate rings for accounts, containers and objects. The ring thus refers to three files that are shared among the storage and proxy nodes - object.ring.gz, container.ring.gz and account.ring.gz [14]. Along with storing the details of the objects' physical locations, the rings also compute the physical devices (hard drives) in each object, container and object in which the objects and their replicas will be stored. Finally, in case of a storage node failure, the ring determines the node to which a request is handed-off to (explained later).

The server process do not themselves modify the rings - this is done by an external utility called the ring-builder. When it receives an object to store, Swift computes an MD5 hash of the object's full name (including the container and account names). A part of the hash is a partition number, and the object ring has the map which maps each partition number to a specific physical device. The relation between storage nodes, disks and partitions is thus this - a storage node has disks, and a partition is represented as a directory on each disk. The length of the part of the hash kept for determining the partition number depends on the number of partitions configured for the Swift cluster. If n bits of the hash are kept, the number of partitions is 2^n.

Partitions are the smallest unit of storage in Swift. Data are added to partitions, consistency processes will check partitions, partitions can be moved to new drives etc. While the size and number of partitions in disk drives do not change, the drives housing the partitions do. The more drives there are in a cluster, the fewer are the partitions per drive. For example, 2 drives with 150 partitions in total would each have 75 partitions. A new drive would result in each drive having 50 partitions, thus keeping the total number of partitions constant. The constant number of partitions ensures predictable behavior of the system when it is scaled up.

4.1.2 Proxy layer

The proxy server processes represent the 'face' of the cluster to the clients - the clients can communicate only with these processes. All requests and responses to and from the proxy use HTTP verbs and response codes [29]. At least two proxy servers are needed for redundancy - should one fail, the other takes over.

When a request is received by the proxy server, the proxy server verifies the request and looks up the ring to determine the correct storage nodes and partitions in which the data are stored. The request is then sent to the storage nodes concurrently. If one of the partitions and/or nodes is unavailable, the proxy uses the ring to identify an appropriate hand-off node to which the request will be forwarded. The proxy server writes to a majority of the replicas in the nodes before returning a success message to the client.

4.1.3 Account layer

The account server handles requests regarding metadata of the account or the containers within that account [29]. The information is stored by the account server process in SQLite databases on disk. Statistics are also tracked, just as what containers are there in the account, and total storage space used up in the account server.

4.1.4 Container layer

The container layer handles listings of objects. It does not where the objects are, but only what objects are present within that container. The listings are stored as SQLite database files (like account servers) [29]. These database files are replicated across the cluster in a way similar to object replication. It stores statistics similar to the account layer as well.

4.1.5 Object layer

The object server process is responsible for storing,retrieving and deleting objects. The objects are stored as binary files on the storage nodes with metadata stored in the fileâs extended attributes (xattrs). For this to happen, the underlying filesystem for object servers has to support xattrs in files. The objects are stored on the drives using a path which consists of its associated partition and the operation's timestamp. The timestamp allows the object server to store multiple versions of an object while providing only the latest version for a typical download (GET) request.

4.2 Consistency, availability and partition tolerance

Before proceeding further, it is pertinent at this point to consider the features of distributed databases, or 'datastores', as they are often termed. They do not have the ACID (Acidity, Consistency, Isolation and Durability) properties of centralised, relational databases (RDBMS).They are also known as NoSQL databases. However, this term is getting outdated as some SQL support is now present in cloud datastores. Several computers are involved in storing data, whilst communicating over a potentially faulty network.

Three features of datastores, as described by Eric Brewer in 2000 [7] are:

1. Consistency - all nodes have a consistent view of the distributed datastore, i.e. all database servers having the replicated copies store the same version of the same.

2. Availability - the datastore can be updated by new transactions and a request is received for every datastore request about whether the request was successful or not.

3. Partition tolerance - Despite loss of network connectivity, the datastore can be queried and updated.

Brewer stated that it was not possible for a distributed datastore to possess all three of the properties at the same time, but only two. A datastore that is consistent and available (CA) cannot be connected over a network (i.e. is centralised and cannot be partition-tolerant). Being consistent and partition-tolerant (CP) results in updates being disallowed if the network goes down, thus losing availability. Finally, being available and partition-tolerant (AP) means the datastore in inconsistent, i.e.different interconnected database servers store different contents.

4.3 Consistency services in Swift

Consistency services run on nodes supporting account, container and/or object server processes to ensure the integrity and availability of the data, even during failures. The two main consistency services are auditors and replicators [29].

Auditors run in the background of every storage node in a Swift cluster and scan the disks continuously to ensure that the data have not been corrupted in any way. There are individual account auditors, container auditors and object auditors which support their corresponding server processes. If an error is found, the auditor moves the corrupted object to a quarantine area.

Replicators also run in the background of all storage nodes running account, container and object services. Each replicator examines its local node and compares the accounts, containers and objects with those in other nodes in the cluster. If the other nodes have old or missing copies of data, the replicator sends copies of its local data to those nodes. The replicators only push their own local data out to other nodes; they do not pull in remote copies if their local versions are stale or missing. The replicator also manages object and container deletions. Object deletion initially creates a zero-byte tombstone file which is the latest version of the object.This is replicated to other nodes and the object is deleted from the local node. A container can be deleted only if it does not contain any objects. An empty container is marked as deleted and the replicator pushes this version out.

4.3.1 Specialized consistency services

Alongside the auditors and replicators, certain specialized services run in the storage nodes as well, including container and object updaters, object expirers and account reapers [29]. The container updater service supports accounts by updating container listings in the accounts, as well as the account metadata (i.e. object count, container count and the bytes used). The object updater supports containers, but as a backup service. The object server process is the primary process to support containers, and only if it fails with the object updater take over and update object listings in the containers and container metadata(object count and bytes used).

The object expirer deletes data that is marked by the replicator as expired. The account reaper service monitors accounts in a node. When it finds an account marked as deleted, it empties that account of any containers and objects it may contain. The reaper has a configurable delay value, that indicates the period of time for which the reaper should wait before it starts deleting data, to prevent erroneous deletions. However, the algorithms deployed by the consistency services are not fully described, and therefore the fault tolerance of this part of Swift is not well understood.

4.4 Replication for fault tolerance

Like other distributed storage systems, Swift relies on replication to protect against losing data when there is a failure. Failures are common in clusters spanning different datacenters and geographic locations [29].

Partitions are replicated. The most common replication count is three. When the Swift rings are initially created, every partition is replicated and each replica is placed as uniquely as possible in the cluster. If the rings are being rebuilt, the new positions (if at all required) of the replicated partitions will have to be recalculated. Thus, when it is said that the proxy server determines the storage nodes for requested data, more specifically it seeks out the partitions which store the data and its replicas.

Partition replication also involves designation of handoff drives. These are needed when a storage node fails. When drives in a node fail, the replicator/auditor services notice and push the missing data to handoff nodes. The chances of all replicas becoming corrupted before the consistency services notice the failures are very small, thus giving Swift its durability.

4.4.1 Unique-as-possible algorithm

Swift assigns partitions using the unique-as-possible algorithm, to ensure that data are distributed evenly over the defined spaces (regions, zones, nodes and disks) [29]. The algorithm identifies the least-used space in the cluster to place a partition. It begins by looking at the least used region. If all the regions contain a partition then it looks for the least-used zone, followed by server (IP:port), and finally the least-used disk and places the partition there. This formula also attempts to place the replicas as far away from each other as possible. Once the placements of all partitions have been determined, the account, container and object rings are created.

4.5 Eventual consistency type of storage

Swift favours availability and partition tolerance over consistency. Even if multiple nodes in the cluster fail, the data will remain available to the client, as replicas are maintained [29]. Being eventually consistent means that updates will eventually be made to all the replicas, whilst still providing data to the client should any node fail. As it is available and partition-tolerant, it will continue to allow clients access to objects requested even as the network is down due to some reason. However, the copy the client receives may not be the most up-to-date one, as the copies may not have become consistent yet. For an object storage system, however, performance and scalability are often most important, and so the available-and-partition-tolerant model works well. Strong consistency is not the most critical factor for a highly scalable system. While transactional data must be strongly consistent, backup files, log files and unstructured data need not necessarily be so. The latter are usually present in object storage systems, and hence some consistency can be sacrificed for performance and scalability.

4.6 HTTP Requests in Swift

HTTP requests are made by clients to the Swift storage system using a RESTful API. The request is made up of at least three parts - 1) a HTTP verb (GET/PUT/DELETE) , 2) storage URL, 3) authentication information, and optionally 4) any data/metadata to be written. The verb the action desired on the object - such as PUTting the object in cluster, GETting account information etc [29]. The storage URL is in the form:

```
https://swift.example.com/v1/account/container/object
```

This URL has two parts - cluster location and storage location. The former specifies the place in the cluster where the request should be sent and the latter where the requested action must occur in that place. In the given URL, the cluster location would be `swift.example.com/v1` and the object storage location `account/container/object`.

4.7 Comparing Swift and Ceph

Openstack Swift offers only object storage, while Ceph provides object, block and filesystem. Ceph chooses consistency and partition tolerance over availability. Swift was designed to be eventually consistent (similar to S3) [6]. It favours availability and partition tolerance, instead of consistency.

Being strongly consistent is the reason why Ceph is able to provide block storage. The consistency ensures that all data to be written is written on all the disks before sending the acknowledgement to the client. Swift, on the other hand, does not need to be strongly consistent as it provides only object storage. Being eventually consistent means that when hardware fails (which is likely in a cluster), Swift will fall back to providing high availability to the data, i.e. ensuring it can be updated via transactions. One scenario where the objects become eventually consistent is when reading objects that were overwritten when some hardware component had failed. This eventual consistency allows Swift to be deployed across wide geographic areas.

As it is written in C++, Ceph is highly optimized for performances and, due to its design, allows clients to communicate directly with object storage devices (OSDs). Swift is written in Python and this allows it to be integrated with different types of middleware to incorporate more specific features. It can also be plugged in different authorization systems.

Swift is used in large-scale public clouds, including those offered by companies such as Rackspace, HP and Cloudwatt. Ceph is not as powerful as the full scale Python WSGI (Web Server Gateway Interface - an interface between web servers and web applications developed in Python) and does not allow modularity. The shared filesystem feature in Ceph is still currently being developed and not quite ready for production. The S3 API is well-defined, but cannot be used with other middleware possible with Swift.

Considering the use cases of the two, Ceph is the obvious choice for block and file storage requirements, as these are not supported by Swift. However, for only object storage, Swift is better. Ceph does provide object storage through its object storage gateway radosgw, which is good enough when used with the supported S3 API or Swift API, but does not provide a fully featured object storage system. Also, objects stored using the radosGW gateway will not be accessible from the block storage system.

Chapter 5

Google File System

Google File System (GFS) is the distributed filesystem developed at Google for storing large amounts of data from its data-intensive applications, primarily the search engine. It is designed to be fault-tolerant while running on commodity hardware (scaling out of storage) using the Linux operating system. Despite the enormous amount of data required to be stored, it provides high performance to a large number of clients.

There are a number of assumptions, based on which the GFS system has been designed [15]. Firstly, as commodity hardware is being used, hardware failures are common. Hence, the data needs to be backed-up in another place so that if one copy is lost on account of a hardware failure, another copy can be referred to for processing. Secondly, files are very large in number, of the order of several millions, and data of the order of several exabytes are being generated every year. Scalability of the storage system is thus crucial. Thirdly, two main types of reads are performed - large streaming reads, typically of the order of hundreds of kilobytes (KBs) (or more than 1 megabyte (MB)), and smaller random writes of a few KBs. Fourthly, sequential writes which append data to files, rather than overwrite existing data in them, are common. Seldom are files modified once written. Thus, random writes within a file are virtually non-existent. Fifthly, it is important to realise that multiple processes (possibly one per node in a cluster) concurrently append to the same file. Hence atomicity of updates (i.e. either all of the data are appended to the file, or none at all), along with synchronization of updates are important. This is to prevent partial updates by conflicting processes. Finally, high bandwidth is needed more than low latency. It is important to be able to process a large amount of data, than look to do it within a fixed short time limit.

In the following sections, the original architecture of GFS (from the late

'90s), its problems, and the new architecture have been discussed.

5.1 The original architecture

A GFS cluster consists of several machines, or nodes. Each of them usually has Linux has its operating system and runs a user-level server process. One node in the cluster is termed the *master*, and the others *chunkservers*. Chunkserver and client processes can run on the same node, as long as machine resources permit it, and the resulting lower reliability of the application code is acceptable [15].

A single file can contain many documents, such as Web documents. Files are divided into *chunks* of size 64MB, and each chunk is assigned a unique 64 bit identifier (called a *chunk handle*) by the master node at chunk creation time. Chunkservers store the chunks as Linux files on local disks [15]. They read or write chunk data specified by the tuple of chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers, with the default being three, which can be configured to be otherwise by users.

The master maintains all the filesystem metadata in its memory. Metadata in this kind of a filesystem are of three types - file and chunk namespace, file-to-chunk mapping, and the chunk and its replicas' locations. The namespaces and mappings are stored on a log file, termed *operation log*, to create persistence. This log file is stored on the master's local disk and the replicated on other nodes. The importance of the log file is that apart from being a persistent record of metadata, it also maintains a logical time line, in which the times at which chunks and files are created are written. This helps to uniquely identify different versions of files and chunks. The chunk locations are not stored persistently. Instead, it asks other chunkservers about their chunks during master startup and also a chunkserver when it joins the cluster. This way, the master and chunkservers do not have to stay in sync as chunkservers join or leave the cluster, restart and fail, all of which are likely to happen in a cluster of hundreds or thousands of servers. In addition, the master performs activities such as garbage collection of orphaned chunks, chunk lease management, and migration of chunks between chunkservers. Finally, it sends on a periodic basis 'heartbeat' messages to chunkservers to give them instructions and collect information about their states.

For a simple read operation, the client initially translates the file name and byte offset specified by the application to a chunk index within the file. It then sends a request to the master containing the file name and chunk index. The master replies with the corresponding chunk handle and replica

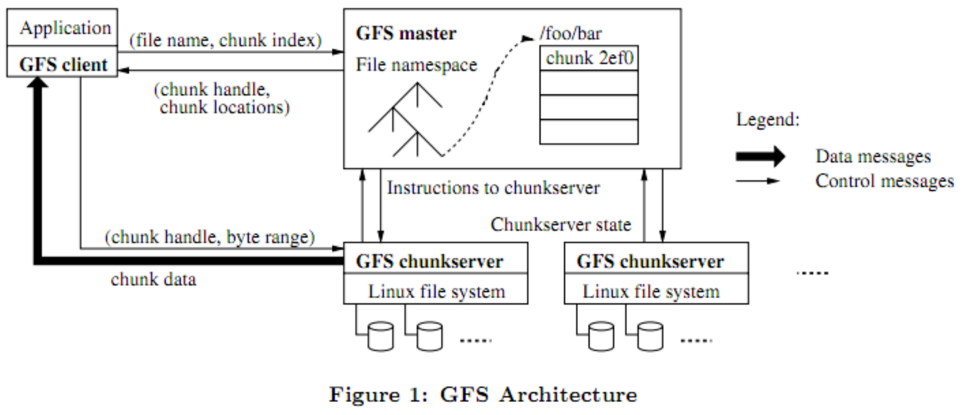

Figure 5.1: GFS architecture [15]

locations. The client caches these using the file name and the chunk index as the key. The caching is done for a limited time, and for many subsequent operations, the client contacts the chunkserver directly, i.e. without the master's involvement. The client sends a request to one of the chunk replicas, usually the closest one (to reduce network bandwidth consumption). The request consists of the chunk handle and a byte range within the chunk. Further reads of the same chunk require no further client-master interaction unless the client cache has expired or the file is reopened. This way, the master's involvement in reads and writes is minimized.

5.1.1 Mutations and leases

An operation which changes the data or metadata of a chunk is termed a *mutation* [15]. Each mutation is performed at all of the chunk's replicas. The master server grants permission to a chunkserver (termed the *primary*) for a limited period of time for modifying the chunk. This time period is termed a *lease*. During this time, no other chunkserver is allowed to modify the chunk. The lease initially is of 60 seconds duration. However, as long as the chunk is being mutated, the chunkserver requests and is granted lease extensions.

The write process to modify a chunk is described below:

1. The client asks the master which chunkserver holds the lease currently for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses.

2. The master replies with the identity of the primary chunkserver holding the primary chunk, and the locations of the secondary chunkservers,

which are the chunkservers storing the replicas. The client caches this information for future mutations. It contacts the master again only when the primary chunkserver does not respond or it no longer holds the lease.

3. The client pushes the data to all the replicas. Each chunkserver having the chunk and its replicas store the data in an internal LRU (Least Recently Used) cache until the data is written or is aged out.

4. Once all the replicas have acknowledged receiving the data, the client sends a write request for the data to the primary chunkserver. It can so happen that the primary chunkserver receives requests to mutate the same chunk from multiple clients. In that case, the chunkserver assigns serial numbers to each request, and performs them in serial number order. The primary chunkserver performs the write.

5. After writing it, the primary chunkserver forwards the write requests to all the other chunkservers. These secondary servers perform the writes in the same serial number order assigned by the primary chunkserver.

6. After completing the writes, the secondary chunkservers send acknowledgements to the primary chunkserver. To preserve atomicity, the changes made by the writes are not saved until all the chunkservers acknowledge.

7. The primary chunkserver then replies to the client, reporting it of any errors which could have occurred during the write. If a write fails at the primary, the request is not assigned a serial number and is not forwarded. Thus, if the change cannot be made to the primary chunk, neither can it done to the replicas. If, however, the primary chunkserver manages to write successfully, but only a subset of the secondary chunkservers manage to do the same, the application code retries on the failed chunkservers, before starting from the beginning of the write all over again.The primary chunkserver thus coordinates writes to its own chunk as well as its replicas.

5.1.2 Atomic record appends

GFS supports atomic record append operation for multiple clients to concurrently operate on a single file with atomicity [15]. In traditional writes, the client suggests the offset where data is to be written. As a result when concurrent writes happen, the region may contain data fragments from different clients. In a GFS record append the client specifies only the data. GFS appends the data to the file at least once atomically (i.e. as one continuous sequence of bytes) at an offset chosen by GFS and returns that offset to the client. If the append causes the chunk data to exceed the chunk boundary the primary chunkserver pads the chunk to the maximum size of 64 MB and requests the secondary chunkservers replicas to do the same to their replicas. The primary chunkserver then replies the client to try the operation on next chunk. If the size is not exceeded the primary chunkserver appends the data and requests the secondary chunkservers to do the same, at the exact same offset as itself. If any error occurs it requests the client to try again.

5.1.3 Snapshots

GFS also provides a snapshot feature [15]. The snapshot operation creates a copy of a file or a directory tree, while minimizing any interruptions to mutations being made to the original during the operation. This is used to create copies of large data sets, and even copies of those copies, or to maintain a record of the current state before experimenting with changes that can be later committed or rolled back easily.

Copy-on-write techniques (explained earlier) are used to implement snapshots. When a master receives a request for creating a snapshot of a file, it first checks if there is already a lease for any of the chunks of that file. If yes, it revokes the lease. This means that if another client wants to write to that chunk, it will first have to contact the master to find the lease holder. This will give the master time to create a copy of that chunk to which the writes will be made.

After the lease for a chunk has expired or is revoked, the master logs the operation to disk. It then duplicates the metadata for the source file or the directory tree. The metadata of the newly created snapshots point to the same chunks as the source files.

After a snapshot operation, when a client wants to write to a chunk of the file from which a snapshot was created, it sends a request to the master to find the current lease holder. The master, upon finding that the lease being requested is for a chunk that was snapshotted, asks each chunkserver which has a replica of that chunk to create a new chunk. Having the new chunk on the same chunkserver as the replicas mean that changes can be copied locally and not over the network. The master then grants one of the chunkservers a lease on the newly created chunk, and replies to the client, which performs the write. It does not know that it is writing to a newly created chunk.

5.2 Shortcomings of the original architecure

The original design of GFS has proved to be remarkably durable over a decade, given that Internet usage in general and Google's operations in particular have scaled more exponentially than anyone could have imagined in the late '90s. However, as the amount of data being accessed over the Internet increases, problems with the original architecture have become all-too apparent.

The first issue is with the single master. The master in GFS, as discussed earlier, has to store the metadata for the chunks in its memory. Memory here refers to main memory (RAM), used in order to provide faster responses to client requests. With a single master, there's only a finite number of files which the system can accomodate, as the memory of the master cannot store more metadata for the file chunks beyond its own capacity. This problem is particularly apparent with small files. Having a large number of small files creates a strain on the master as it has to store a lot of metadata for files which eventually take up a relatively small amount of space of the chunkservers' disks. This is due to the way GFS has been designed - metadata objects are stored for file chunks of 64 MB size. Whether the file is large or not, the metadata is stored for the same chunk size. For example, suppose the size of a metadata object is 150 bytes [11]. If the input file is of size 1 GB, it is broken into 16 64 MB chunks. The amount of metadata stored for this 1 GB file is (150 * 16) = 2.4 KB. Now supposing there are 10,500 files, each of 10 kB size. As the file size is smaller than the chunk size, each file would occupy 1 chunk. The amount of metadata for this many files is (150 * 10500) = 1.575 MB, although the files themselves occupy a relatively smaller space on the chunkservers' disks. When a client makes a request for a large number of such small files, the single master becomes a bottleneck for operations. The situation is worse when multiple clients request the master for this large number of small files.

As Google's Sean Quinlan discusses in an interview with the Association for Computer Machinery(ACM), the main reason for the single server was to deliver applications to users within a short space of time [26]. Use of a single master allowed applications such as the search engine to be built far quicker than it would have with distributed masters. Some of the engineers involved in creating GFS later went on to make BigTable, the distributed database, and that took many years.

A related problem with one master is the presence of a single point of failure in the architecture. If the master goes down, the whole storage system becomes non-operational. As Quinlan explains, GFS originally lacked an automatic failover system if the master went down [26]. One had to manually restore the master, meaning GFS service was absent for upto an hour. Later, automatic failover was added, but even then, there was a noticeable service outage. The delay was brought down from several minutes to about 10 seconds.

This delay highlights the other issue with the design. While delay of a few seconds is acceptable for batch-oriented applications like web crawling and indexing, it is clearly not so for user-facing applications [26] such as GMail. It has been mentioned earlier that one of the assumptions made during GFS development was that high bandwidth was more important than low latency. Given the way the Internet is used now and Google's own applications, this is clearly not the case. It has been shown in studies that user traffic to a webpage is inversely proportional to the page load time.

5.3 Colossus - the GFS upgrade

In 2010, Google remodeled its search mechanism to Caffeine. Caffeine moves the back-end indexing system of Google from MapReduce (the distributed data processing framework developed at Google) to BigTable, which the company's distributed database platform. It uses BigTable to create a kind of database programming model which allows search indexes (indexing of web documents to speed up their retrieval) to be updated without building them from scratch, as was the case with MapReduce.

Caffeine uses an upgraded version of GFS, termed Colossus. While GFS was built for batch operations (background operations performed, whose results are moved to a live website), Colussus is built for ”realtime” services, where processing is nearly instantaneous [8]. Earlier, Google would use GFS and MapReduce to build a new search index every few days and later hours. Now the index is updated with new information in real time, using Caffeine and Colossus.

To allow quicker updates, Colossus make use of multiple masters distributed over the clusters in a datacenter. This solves the single-point-of-failure problem. It also reduces the size of the data chunks from 64 MB down to 1 MB. This lets the storage system store far more files across a larger number of machines.

Chapter 6

Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a sub-project of the Apache Hadoop project. This project by the Apache Software Foundation is designed to provide a fault-tolerant storage system capable of running on commodity hardware. HDFS is the open-source version of GFS (described earlier) and has been written in Java for the Hadoop framework. It is designed to store very large data sets reliably, and to stream those data sets at high bandwidths to user applications [38]. In a Hadoop cluster, data are partitioned and worked on by several (upto thousands) of hosts. In such a cluster, computation capacity, storage capacity and I/O bandwidth are scaled simply by adding more commodity servers. These servers both host directly attached storage as well as execute user application tasks.

Each input file is split into blocks, each block being of size 64 MB by default. The block size can be configured to be otherwise by the user. Since commodity hardware is used for storage, component failures are common. Therefore, like GFS, also here the blocks are replicated on different hosts. By default, the number of hosts on which the blocks are replicated is three - typically one written on the local disk of a host, another on the local disk of another host on the same rack, and another on the disk of a host in a different rack. Data are thus replicated to at least two racks. The replication factor can be configured to be otherwise by the user. Files can be served to the client if the blocks are present in at least one location.

6.1 Architecture

As with GFS and other distributed storage systems like Lustre, HDFS stores application data and metadata separately. The metadata are stored on a dedicated server called the *namenode*. Application data are stored on other servers called *datanodes*. All these servers are connected to each other and communicate through TCP-based protocols.

6.1.1 Namenode

The HDFS namespace is a hierarchy of files and directories [38]. Files and directories have their metadata stored in the form of inodes. Inodes store attributes of the file/directory such as file modification and access times, permissions and disk space quotas. The namenode maintains the namespace tree and the mapping of file blocks to datanodes, where the blocks are physically stored. The usual design is to have a single namenode in each cluster, although more recent designs make it possible to have more than one namenode.

The inode data and the list of blocks for a file comprise the metadata of the file called the *image* [38]. Each file thus has an image in the namenode. The persistent record of the image written to the namenode's local file system is called *fsimage*, a checkpoint of the namespace and edits (explained later). The namenode also stores the modification log of the image in the local file system. This log is termed *edits*, which is a journal. Before making any changes to the image, the changes to be made are written to the journal. As explained earlier, this is useful in events such as power failures, which can leave the metadata in an inconsistent state. Reading the journal then results in replaying the changes to be made to the image, thus bringing the metadata and data to its earlier consistent state. For redundancy, copies of the fsimage and edits files can be stored on other servers. The locations of the blocks change over time and are not part of the persistent fsimage content.

6.1.2 Datanode

The actual data blocks are stored in the datanodes. There can be thousands of datanodes in a cluster. Each block is represented by two files in the local filesystem of the datanode [38]. One file contains the data itself, and the other has metadata for the block which consists of checksums for the data and the block's *generation stamp*. It is important to note here that the actual length of the block on the disk is equal to the size of the file. This means that a file of size 36 MB will require a single 64 MB block, but the actual space occupied on the local disk is 36 MB. Extra padding of 28 MB will not be done to bring the file up to the nominal block size. However, the namespace metadata in the namenode will treat the file as a single block. Files larger than 64 MB will be considered as two blocks or more in the namespace metadata.

During startup each datanode connects to the namenode and performs a *handshake*. This is done to verify the *namespace ID* and *software version* of the datanode. Both of these have to match that of the namenode, otherwise the datanode will have to shut down. The namespace ID is assigned to the file system when it is formatted. This ID is persistently stored in all nodes of the cluster. All nodes in a cluster thus have the same namespace ID. Nodes with different IDs cannot join the cluster, thereby preserving the integrity of the file system. A newly initialized datanode, without any namespace ID, can join the cluster and is assigned the ID of that cluster. The software versions also have to be the same for all nodes in a cluster. Different, incompatible versions may cause data corruption and loss. Different versions can result when hosts do not shut down properly prior to the software upgrade or are not available during the upgrade. In a cluster of thousands of hosts, it is possible to overlook hosts with different versions. Hence performing the check during startup is very important.

After the handshake, the datanode *registers* with the namenode, to receive a *storage ID* from the latter. This is an internal identifier of the datanode and is persistently stored by the datanode. This storage ID makes it possible for the datanode to be recognized even if it starts up with a different IP address or port. After registering with the namenode, the storage ID received never changes.

A datanode informs the namenode about the blocks in its possession by sending the latter a *block report*. This report is specific to each individual block and contains the *block id*, *generation stamp* and block length. The first block report is sent by the datanode immediately after its registration. Subsequent block reports are sent every hour. This way, the namenode is updated on where to find what in the cluster. From time-to-time, datanodes send periodic *heartbeat* messages to the namenode to inform the latter that they are still alive, and that the blocks they host are available. The default heartbeat interval is three seconds. If the namenode does not receive a heartbeat signal from a datanode for ten minutes, it considers that datanode to be down and the blocks it hosts unavailable. The namenode then schedules creation of new replicas on other datanodes.

The diagram below is a high-level description of the architecture. Same-coloured blocks in the datanodes indicate the replicas. In the diagram, the number of replicas is shown as three.

Heartbeats from a datanode also contain information such total storage capacity in a node, amount of storage in use, and the number of ongoing data transfers [38]. This information is used by the namenode when making space allocation and load balancing decisions. When replying to the heartbeat messages, the namenode can include instructions for the datanodes. The

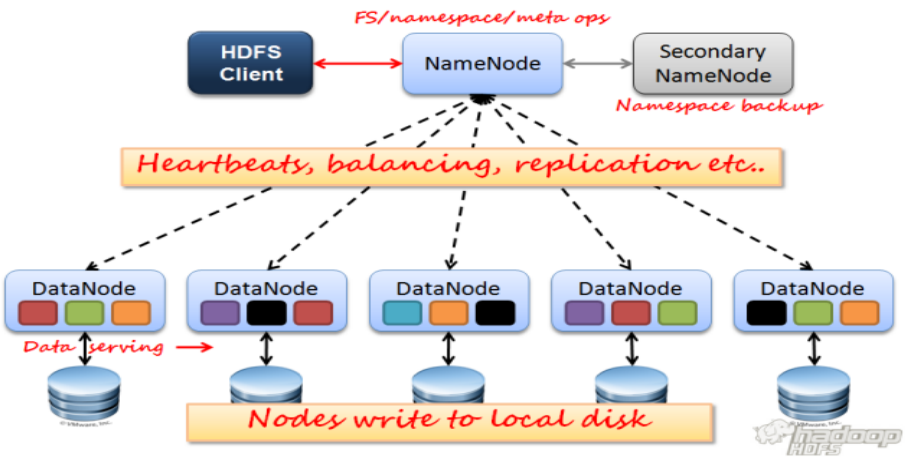

Figure 6.1: HDFS Architecture [48]

instructions include commands to replicate blocks to other nodes, remove local block replicas, re-register or shut down the node, and send an immediate block report. The namenode can process thousands of heartbeats per second without affecting other namenode operations.

6.1.3 Client

The HDFS client is a code library that exports the HDFS interface. This code library is used by user applications to access the file system [38]. The user references files and directories by paths provided in the namespace. HDFS supports operations to read, write, delete files as well as create and delete directories.

When an application reads a file, the client contacts the namenode to identify the datanodes that are currently storing the blocks for the file. After the namenode replies with the list of datanodes, the client contacts the nearest datanode directly and requests transfer of the blocks.

The client which opens a file for writing is granted a lease for the file; this means no other client can write to the file. The writing client can renew the lease by sending heartbeat messages to the namenode. The writer's lease does not prevent other clients from reading the file - a file can have many concurrent readers. When writing data, the client requests the namenode to nominate a group of three datanodes (by default), to host the block and its replicas. The selected datanodes form a pipeline, in such a way so that the distance between the client and the last datanode is minimized. Bytes are pushed by the client into the pipeline as a sequence of packets. These bytes are first stored in a buffer at the client side. When the buffer is full (its size is typically 64 kB), the data are pushed to the pipeline. The first datanode receives the data in small portions of 4 kB size. The portions are written to its local hard disks and transfers that portion to the second datanode in the pipeline. The second datanode repeats the process, flushing its data portion to the third datanode and so on until the data is replicated the same number of times as specified. After data has been written to all the datanodes, an acknowledgement is sent to the client. When all the replicas are written, the client moves on to the next block of the file. The datanodes confirm the creation of the block replicas to the namenode.

As stated earlier, HDFS allows a file to be read even while it is being written. When reading such a file, the length of the last block of the file is unknown to the namenode.The reading client asks one of the replicas for the latest length before starting to read the content.

HDFS provides an API that exposes the locations of the file blocks. This allows applications like the MapReduce framework to schedule a task at the data locations, thus reducing the consumption of valuable network bandwidth due to transfer of large amounts of data over the network.

6.1.4 Checkpoint node

Before discussing the utility of the checkpoint node, it is instructive to study the fsimage and edits files in more detail. It has been mentioned earlier that a persistent record of the namespace image in the namenode written to disk is called the fsimage file [38]. Also, a journal called ‘edits’ has been described as a log for recording changes to be made to the file system metadata. These changes must be persistent. For example, creating a new file in HDFS or changing a file’s replication factor causes the namenode to make an entry in the journal regarding this [5]. For each client-initiated transaction, the changes to be made are recorded in the edits journal, and the journal is flushed and synched before committing the changes to the HDFS client [38]. During startup the namenode initializes the namespace image from the fsimage file, and then replays changes from the journal (written before shutdown) until the image is up-to-date with the state of the file system before shutdown. The new version of the namespace image then is flushed out to a new fsimage on disk, and the old edits log truncated since all its transactions have been applied to the persistent fsimage. This process is called a checkpoint, and results in a new fsimage and empty edits file. [5] The fsimage file is never modified by the namenode. It is completely replaced during restart, when requested by the administrator, or by the checkpoint node.

The reason new fsimages are periodically created is that this protects the metadata represented by the namespace image. The system can start from the most recent fsimage if all other persistent copies of the checkpoint and the journal are unavailable.

Another benefit of generating new checkpoints is that it helps to keep the size of the journal in check [38]. HDFS clusters run for long periods of time without restarts, resulting in the edits journals becoming very long. Very large journals may lose data or contain corrupted data. By generating a new fsimage, the empty journal created ensures the length of the journal file is reset, i.e. set to 0 again. It is considered good practice to create a checkpoint on a daily basis [38].

The checkpoint node downloads the fsimage and edits files from the namenode, merges them locally, and uploads the new image back to the active namenode. The checkpoint node usually runs on a different machine than the namenode as its memory requirements are the same as that of the namenode.

6.1.5 Backup node

Along with providing the same checkpointing functionality as the checkpoint node, the backup node maintains an in-memory (i.e. in RAM), up-to-date copy of the file system namespace that is always synchronized with the active namenode state [38]. It accepts the stream of file system edits specified in the journal from the namenode and persists these logs to disk. Along with that, it also applies these edits to its own copy of the namespace in memory, thus creating a backup of the namespace.

Unlike the checkpoint node, the backup node does not need to download the fsimage and edits files from the namenode to create a checkpoint, as it already has the up-to-date state of the namespace in its own memory. It is thus more efficient than the checkpoint node as it only needs to save the namespace in the local fsimage file, and reset edits to create the empty journal. Since the backup node also maintains a copy of the namespace in its RAM, its memory requirements are the same as that of the namenode.

The namenode registers one backup node at a time. No checkpoint node may be present if a backup node is already in use.

6.2 HDFS robustness

HDFS operations are adversely affected by namenode failures, datanode failures and network partitions [5].

The datanodes periodically send heartbeat messages to the namenode, thereby assuring the latter that they are working properly. Loss of connectivity between the datanodes and namenode can result due to network partitions. This connection loss results in heartbeat messages no longer reaching the namenode. When the namenode detects a datanode that fails to send it heartbeat messages, it marks that datanode as dead and does not forward any I/O requests to it. Data stored in a dead datanode is not available to HDFS any more. Due to datanode failures, the number of replicated copies of a file may be lesser than that configured. The namenode initiates re-replication of the blocks when it detects the disparity. Re-replication can arise due to other reasons as well - the replication factor having been increased, or a replica may become corrupted.

The client software performs checksum calculations on each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. Network faults, buggy software or problems in storage devices can result in a block of data requested from a datanode arriving to the client in a corrupted state. The client verifies that the checksum of the data it received from the datanode matches the one stored in the checksum file. If not, the client can retrieve that block from another datanode with a replica of that block.

The importance of the checksum and journal (or fsimage and edit respectively) files in storing metadata has been described earlier. Corruption of these files can cause HDFS to become non-functional. Hence the namenode can be configured to store multiple copies of those two files. Changes made to either the checksum or journal file causes all their copies to get updated synchronously. This synchronous update can reduce the rate of namespace transactions per second that the namenode can support. However, this degradation is tolerable as HDFS applications are not really metadata-intensive in nature - they are data-intensive. When the namenode restarts, it selects the latest consistent checksum and journal to use.

Chapter 7

Windows Azure Storage

Windows Azure Storage (WAS) is a massively scalable cloud storage system that allows customers to store large amounts of data for any length of time. The storage system can be accessed from anywhere in the world, by applications running in the cloud, on servers, desktop/laptop computers, mobiles or tablets [9]. It has been in production in Microsoft since November 2008, for application such as serving music, video and game content, managing medical records etc. Several customers outside Microsoft also use WAS.

WAS stores data in three forms - blobs (user files), tables (structured datasets), and queues (message delivery). These three are used in combination in many applications. One example of that is incoming and outgoing data being stored as blobs, message flows during blob-processing taking place through queues, and the intermediate and final data being stored in tables or blobs. Customer data is stored across multiple data centers separated hundreds of miles apart. This geographic replication provides protection against natural disasters, thus facilitating disaster recovery.

To reduce storage cost, many customers are served from the same shared storage infrastructure [9]. The workloads of different customers are combined together so that significantly less storage needs are provisioned that than if the workloads had their own dedicated hardware.

7.1 Global namespace in WAS

One of WAS's key features is a global namespace which allows clients to access all of their storage in the cloud and scale to any amount when necessary [9]. The storage namespace has three parts - an account name, a partition name and an optional object name. Data in a storage stamp (explained later) is accessible via an URI of the form `http(s)://AccountName.{service}.core.`

`windows.net/PartitionName/ObjectName`.

Here, the **AccountName** is the account name selected by the customer for accessing the storage system and is a part of the DNS host name. This name is used to locate the primary storage cluster and data center where the data is stored. The primary storage stamp (explained later) is where all requests go to reach data for that customer account. An application can have several such account names to store its data in different locations. While the account name identifies the storage cluster where the data is located, the **PartitionName** is needed to identify the storage node within that cluster. This name is needed to scale out access to the data across storage nodes based on traffic needs.

7.2 WAS architecture

The main feature of WAS is its scaling ability - it can store and provide access to a very large amount of data (exabytes and beyond). A high-level diagram of the architecture is given below:

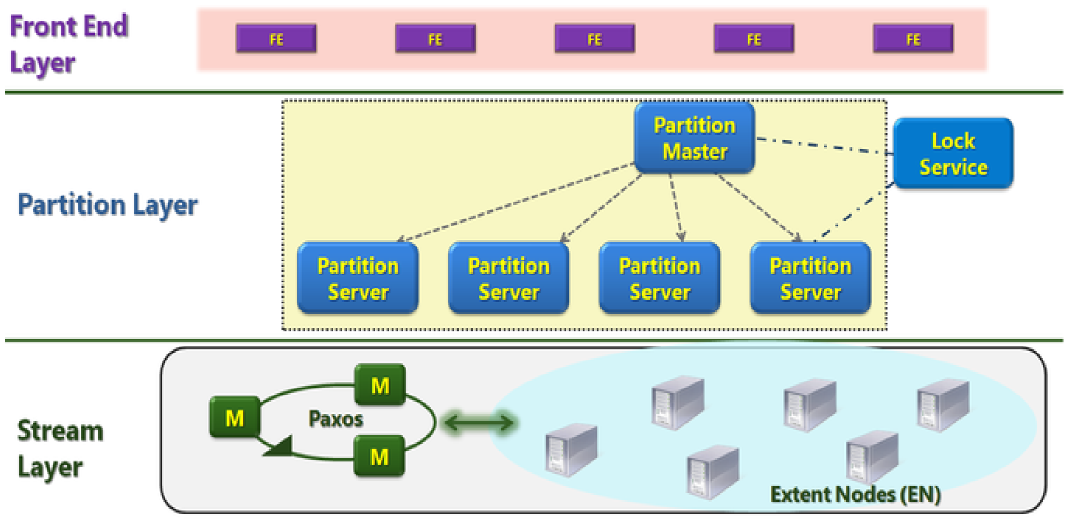

Figure 7.1: Windows Azure storage stamp layout - a high-level architecture [40]

The production system of WAS consists of the following components:

7.2.1 Storage stamps

A storage stamp is a multi-rack cluster of storage nodes, where each rack has redundant networking and power to handle faults [9]. Each cluster usually stores 10-20 racks with 18 disk-heavy storage nodes per rack. The earliest generation of storage stamps store approximately 2 PB (petabytes) of data each, while later generations store upto 30 PB of storage each. Not all of this storage is utilized, however - usually a storage stamp is around 70% utilized in terms of capacity, transactions and bandwidth, with the limit being about 80%. 20% is kept as reserve for a) disk short stroking to gain better seek time and higher throughput by utilizing the outer tracks of the disks, and b) to continue providing storage capacity and availability in the event of a rack failure within the stamp. When a storage stamp reaches 70% utilization, the location service migrates accounts to different stamps using inter-stamp replication.

A storage stamp consists of three layers:

1. Partition layer - The partition layer has several functions. Firstly, it handles the high-level data structures (blobs, tables and queues) [9]. Secondly, it provides a scalable object namespace. Thirdly, it provides transaction ordering and strong consistency for objects. Fourthly, it stores object data on top of the distributed file system layer (explained later). Finally, it caches object data to reduce disk I/O.

   An important internal data structure provided by the partition layer is called the object table (OT). The OT is a massive table which can be of several petabytes in size. These are dynamically broken into RangePartitions (also called partition ranges), based on traffic load on the table. A RangePartition is a contiguous range of rows in an OT. Every row in the OT is a part of a RangePartition. The rows in OTs are non-overlapping.

   There are several types of object tables used by the partition layer. The Account Table stores metadata and configuration for each storage account assigned to the stamp [9]. The Blob Table stores all blob objects for all accounts in the stamp. The Entity Table stores all entity rows for all accounts in the stamp. The Message Table stores all messages for all accounts' queues in the stamp. The Schema Table maintains a record of the schema of all the OTs. The Partition Map Table keeps track of the current RangePartitions for all OTs and which RangePartition is being served by which partition server. This table is used by the front-end servers to forward requests to the appropriate partition servers. The primary key for the Blob, Entity, and Message Tables is a composite of three properties - account name, partition name and object name.

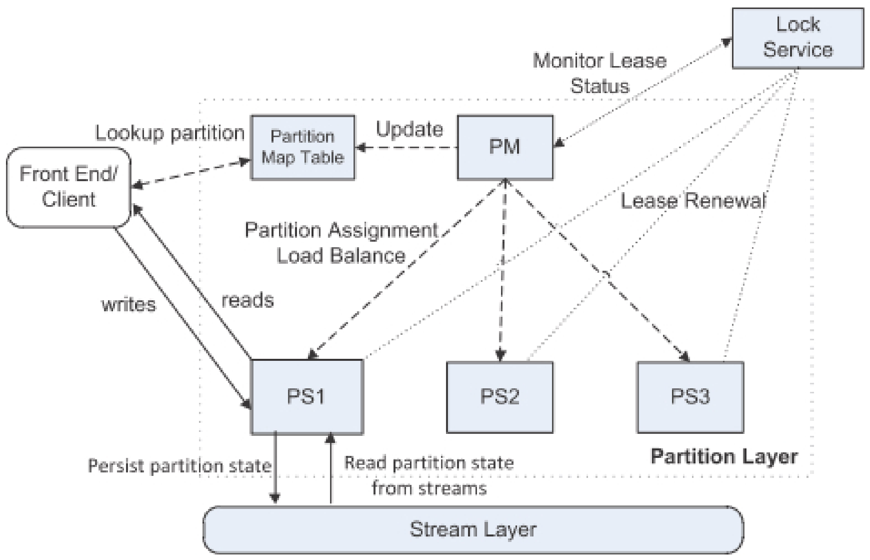

Figure 7.2: Partition layer [18]

The partition layer has three main architectural components - the partition master, partition servers, and the lock service. The partition layer has a master system for assigning RangePartitions to the partition servers and load balancing [46]. The partition master (also called the partition *manager*) constantly monitors the overall load on each partition server as well as the individual partitions, and uses this for load balancing. The partition master keeps track of and splits the massive OTs into RangePartitions and then assigns each RangePartition to a partition server. The partition master stores this assignment in the Partition Map Table. It also ensures that each RangePartition is assigned to exactly one partition server at any point of time, and that they do not overlap. Each storage stamp has several instances of the partition master. These instances contend with each other for a leader lock that is stored in the Lock Service (explained later). The master with the lease is the active master instance which controls the partition layer.

Each partition server is assigned a set of object partitions (blobs, tables and queues). It stores all the persistent states of the partitions into streams and maintains a memory cache of the partition state for efficiency. The system guarantees that no two partition servers can serve the same RangePartition at the same time by using locks with the lock service (explained next). This way, a partition server provides strong consistency and ordering of concurrent transactions to objects in the RangePartition it is serving. A partition server can concurrently serve multiple RangePartitions from different OTs.

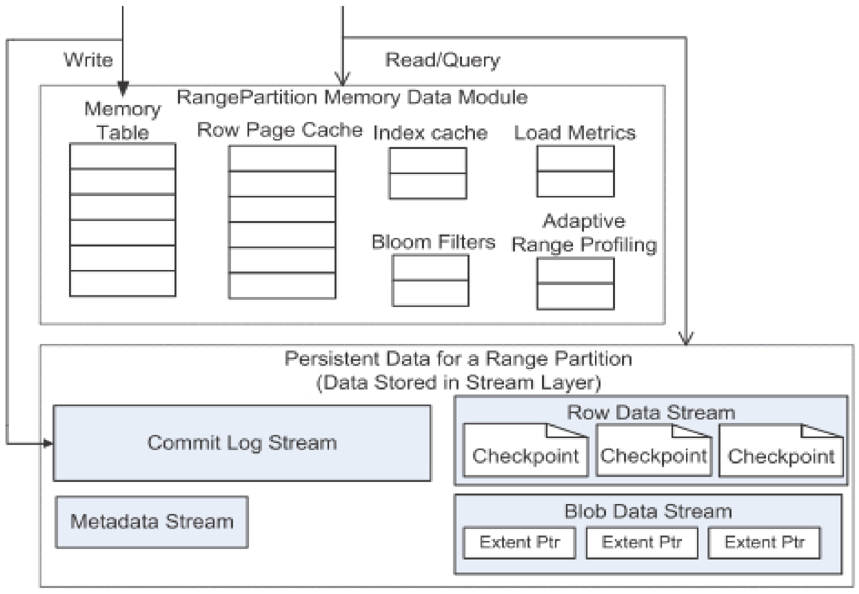

Figure 7.3: RangePartition data structures [18]

A partition server serves a RangePartition by maintaining a set of in-memory data structures and a set of persistent data structures in streams [9]. A RangePartition uses a log-structured merge tree to maintain its persistent data. Each RangePartition consists of its own set of streams in the DFS layer. However, the underlying extents can be pointed to by different streams in different RangePartitions. Each RangePartition comprises of the following streams (which are persistent data structures) -

- Metadata Stream - This is used by the partition master to assign a partition to a partition server. This is the root stream for a partition server. It contains enough information for the partition server to load a RangePartition. This information includes the names of the commit log and data streams (explained next), as well as the pointers (extent + offset) into those streams for where to start operating in those streams, e.g. where to start processing in the commit log stream.

- Commit Log Stream - This commit log is used to store the recent insert, update and delete operations applied to the RangePartition since the last checkpoint was generated for this range.
- Row Data Stream - This stores the row checkpoint data and index for the RangePartition.
- Blob Data Stream - It is used by the Blob Table to store the blob data bits.

Each of the streams listed above is a separate stream in the DFS layer owned by an Object Table's RangePartition. Apart from the Blob Table, all other OTs have RangePartitions with only one data stream. Along with the persistent data structures, a partition server has the following in-memory data structures [9]:

- Memory Table - This is the in-memory version of the commit log stream for a RangePartition, containing all the recent updates which have not yet been checkpointed to the row data stream. During a lookup, the memory table is referenced to find recent updates to the RangePartition.
- Index Cache - This caches the checkpoint indexes of the row data stream.
- Row Data Cache - This is a read-only memory cache of the checkpoint row data pages. When a lookup occurs, both the row data cache and the memory table are checked, with preference given to the memory table. The index and row caches are separate to ensure that as much of the main index is cached in memory as possible for a given RangePartition.
- Bloom filters - If the data is not found in the memory table or the row data cache, the index/checkpoints in the row data stream need to be searched. A bloom filter is kept for each checkpoint, which indicates if the row being accessed *may* be in the checkpoint.

The Lock Service (LS) uses the Paxos algorithm for selecting a leader among the project master (PM) instances.

2. Distributed file system layer - The distributed file system (DFS) (originally known as the *stream*) layer stores the bits on disk and is responsible for distributing and replicating the data across many servers to keep the data durable within the stamp [9].

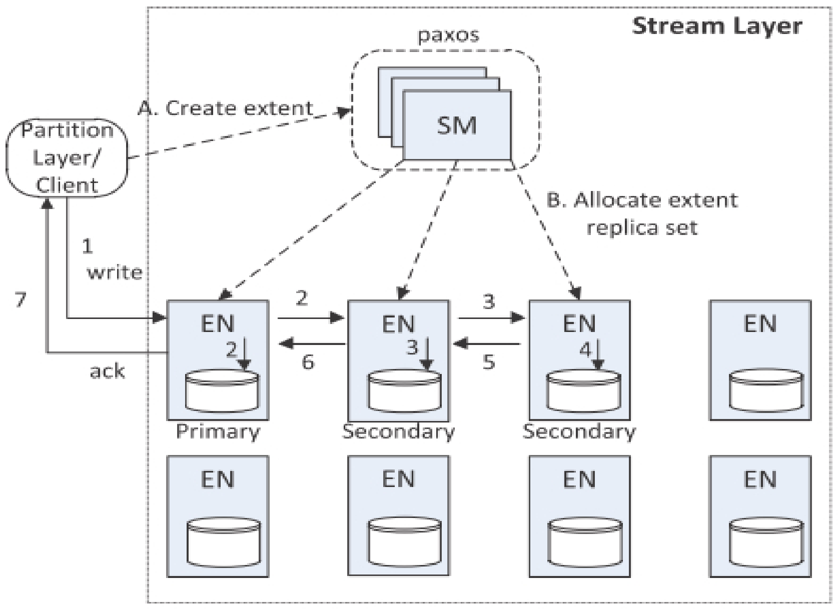

Figure 7.4: Stream layer [18]

The DFS layer can be thought of as a distributed file system layer within a stamp. The data is stored in the DFS layer, but is accessible from only the partition layer. Thus, when assigning partitions to different partition servers, no data is actually moved around on disk. This is because the data itself is stored in the DFS layer and is accessible from any partition server. The actual data reading/writing is done in the DFS layer by the partition layer. All writes are append-only. The partition layer can open, close, delete, rename, read, append to, and concatenate large files called 'streams'.Streams in WAS are ordered lists of pointers to storage chunks called 'extents'. Each extent has a set of blocks that were appended to it. A block here is the minimum unit of data in reading and writing. Data is appended as one or more concatenated blocks to an extent. Blocks in an extent do not all have to be of the same size.

Clients (partition servers in the partition layer) perform the appends in terms of blocks and controls the size of each block. A client read specifies an offset to a stream or extent, and the stream layer responds by reading as many blocks as needed from the offset to complete the read.

The DFS layer handles the data at a lower level, i.e. it knows how to store and replicate the bits in the extents, but does not understand higher level object constructs (i.e. blobs, queues and tables) and their semantics. The storage system ensures that all partitions are always served. Partition servers (daemon processes in the partition layer) and DFS servers are co-located on each storage node in a stamp.

3. Front-end layer - The front-end (FE) layer consists of a series of stateless servers which receive incoming requests [9]. Upon receiving a request, the FE layer looks up the AccountName, authenticates and authorizes the request, an forwards the request to a partition server in the partition layer, based on the PartitionName in the request. The FE layer maintains a partition map which records the partition name ranges and maps partition names to partition servers. The partition map is cached and used to determine which partition server to forward a request to. The FE servers cache frequently accessed data for efficiency and stream large objects directly from the DFS layer.

7.2.2 Location service

The location service (LS) manages the storage stamps, and the account namespace across the stamps [9]. The LS allocates accounts to storage stamps and manages them across the storage stamps for disaster recovery and load balancing. The location service itself is located in two different geographic locations to provide redundancy. WAS has its data centers distributed in different locations all over three geographic regions - North America, Europe and Asia [9]. The data center in each location has several storage stamps in one or more buildings. To scale up to provide more storage, the LS can easily add new regions, new locations to a region, or new stamps to a location. Practically, this means adding more storage stamps to the desired location's data center and adding them to the LS. The LS then allocates new storage accounts to the new stamps and migrate (for load balancing purposes) old, existing accounts in other storage stamps to the new ones.

The process for storing data in WAS begins with an application requesting a new account for storing data [9]. The application specifies the desired location from available options for storing the data, based on its needs, e.g. US North. The LS then chooses a storage stamp within that location as the primary stamp for that account. It does this by analysing the load information in all stamps in that location, such as network and transaction utilization, and the amount of data in the stamp currently. The LS follows this up by storing account metadata information in the chosen storage stamp, which tells the stamp that it has to start taking traffic for the assigned account. Finally, the LS updates the DNS to allow requests from the name allow requests with the URL `https://AccountName.service.core.windows.net/` to the assigned storage stamp's virtual IP.

7.3 Consistency, availability and partition tolerance in WAS

The CAP theorem (described earlier) states that a distributed system cannot have all three properties of consistency, availability and partition tolerance at the same time. WAS, however, claims to provide all three of these properties in a storage stamp, due to the layered architecture and the overall system design.

The append-only (in writes) data model of the DFS layer provides high availability in the face of network partitions and other such failures. On top of the DFS layer, the partition layer provides strong consistency. This layering allows the decoupling of nodes responsible for producing strong consistency from nodes ensuring high availability, i.e. those which store the actual data.

The kind of network partitioning seen in a storage stamp is when a top-of-rack (TOR) or a node fails. When a TOR switch fails, the affected rack will no longer be used for traffic. The DFS layer instead uses extents on available racks to allow streams to continue writing. In addition, the partition layer reassigns the RangePartitions to partition servers on available racks to allow all of the data to continue to be served with high availability and strong consistency.

However, it must be kept in mind that the strong consistency is guaranteed within only a particular geographic region. Across multiple such regions the consistency guarantees are weaker.

7.4 Fault tolerance in WAS

As with other distributed storage systems, the large scale of WAS means failures are common. To handle failures in storage, one can use replication or erasure coding. The former involves adding more copies, and the latter uses parity to recover the lost data. In other words, replication involves reading the same data from another location, while erasure coding recreates the data. Erasure coding is advantageous in that it saves space.

WAS uses two types of replication schemes:

1. Intra-stamp replication - This type of replication is used in the DFS layer. Multiple copies of data are stored synchronously within a region

for durability against hardware failures. The data are replicated at the bit level. To ensure durability, the transaction is replicated synchronously across three different storage nodes across different fault and upgrade domains. A fault domain (FD) can be considered as nodes belonging to the same physical rack. These nodes represent a single unit of failure. An upgrade domain (UD) is a group of nodes that will be upgraded together during the process of service upgrade (called a *roll-out*). Spreading the three replicas across the UDs and FDs means the data is available even if hardware failure occurs in a single rack and when nodes are upgraded during a rolling upgrade.

Updates to the storage account are synchronously replicated to three storage nodes in the storage stamp and success returned only when all three replicas are successfully replicated. Along with this, cyclic redundancy checksums (CRCs) of the data are stored to ensure correctness. The CRCs are periodically read and validated to detect bit rot - random errors occurring on the disk media over time.

2. Inter-stamp replication - Inter-stamp replication is used to provide geographical redundancy against geographical disasters such as earthquakes, which are rare (compared to hardware failures). It performs asynchronous replication by replicating data across stamps, thus providing disaster recovery. This is done by the partition layer, and is configured for an account by the location service. The replication is done at the object level (blobs, queues and table data). Thus, while inter-stamp replication is focused on replicating objects and the transactions applied to those objects, intra-stamp replication replicates blocks of disk storage which make up those objects. Either the whole object is replicated or the recent changes are replicated for a given account.

   The stamps across which the data are replicated are frequently located hundreds of miles apart. In case of a disaster in the primary region, the replicate data in the secondary regions remain durable. In intra-stamp replication, success is returned only once all three replicas are successfully persisted. In inter-stamp replication, after the updates are committed to the primary stamp (three times), they are asynchronously replicated to the secondary stamps. On the secondary stamp, the updates are again committed to a three-replica set before returning to the primary stamp. Having three replicas in each of the locations means each location can recover by itself from common failures (disk/node/rack/TOR failing) without having to communicate with other locations. Usually, the customer select the primary and secondary locations. Commonly, there is one primary and one secondary location.

It is crucial to achieve intra-stamp replication with low latency, since the primary stamp is one which is referred to for all user requests initially (due to closer proximity). The focus of inter-stamp replication is optimal use of the network bandwidth between stamps with an acceptable amount of delay [9].

The WAS stream layer is an append-only distributed file system. Streams are very large files, which are split into units of replication called extents. Extents are replicated before they reach their target sizes. Once they reach their target sizes, they become immutable (become *sealed*) and then erasure coding is applied in place of replication. Earlier, Reed-Solomon 6+3 was the convention for erasure coding - a sealed extent was split into six pieces, and these were coded into three redundant parity pieces.

WAS is robust enough to handle node failures in each of the three main layers [46]:

1. Partition server failure - If the system determines that a partition server is unavailable, it immediately reassigns partitions it was serving to other partition servers, and the partition map for the front-end servers is updated with the new changes.

2. DFS server failure - If the storage system determines that a DFS server is unavailable, the partition layer stops using that server for reading and writing while it is unavailable. Instead, the partition layer uses the other DFS servers which contain the replicas of the data. If a DFS server is down for too long, additional replicas of the data are generated to have a healthy number of replicas for durability.

3. Front-end server failure - If a front-end server is unresponsive, the load balancer realizes this and redirects incoming requests to other, available front-end servers, until the failed server comes back up again.

Chapter 8

Lustre

Lustre is an open-source distributed parallel file system developed and maintained by Sun Microsystems Inc. Its high scalability makes it popular in scientific supercomputing, as well as in sectors such as oil and gas and finance. It presents a POSIX interface to its clients with parallel access capabilities to the shared file objects. The name 'Lustre' is an amalgam of the words 'Linux' and 'cluster'.

The ability of a Lustre file system to scale capacity and performance for any need reduces the need to deploy multiple file systems, such as one for each compute cluster [21]. Storage management is simplified by avoiding the need to copy data between compute clusters. Along with aggregating storage capacities of many servers, the I/O throughput is also aggregated and scales with additional servers. Throughput and/or capacity can be easily increased by adding servers dynamically.

8.1 Features of Lustre

A Lustre installation can be scaled up or down by changing the number of client nodes, disk storage and bandwidth [21]. Scalabilty and performance are dependent on the available disk and network bandwidth, as well as the processing power of the servers in the system.

The notable Lustre software features are:

(a) **Performance-enhanced ext4 file system** - Lustre uses an improved version of the ext4 file system to store data and metadata. This version, called *ldiskfs*, has been enhanced to improve performance and provide additional necessary functionalities.

(b) **Interoperability** - The Lustre file system can run on a variety of CPU architectures and mixed-endian clusters and is interoperable between successive major Lustre software releases.

(c) **POSIX compliance** - POSIX itself does not say anything about how a file system will operate on multiple clients [13]. However, Lustre conforms to the most reasonable interpretation of what the single-node POSIX requirements would mean in a clustered environment.

For example, the coherency of read and write operations are enforced through the Lustre distributed lock manager. Thus, if applications on different nodes try to read from or write to the same part of a file at the same time, they would see consistent results. In a cluster, most operations are atomic so clients never see stale data or metadata, with two exceptions:

- *atime* updates - *atime* is the file access time in Linux systems. This time value gets updated when a file is opened, as well for other operations like grep, cat, head etc. It is not practical to maintain fully coherent *atime* updates in a high-performance cluster file system. Clients will refresh a file's *atime* value value whenever they read or write objects of that file from the object storage targets (OSTs), but will do only local *atime* updates for reads from cache.
- *flock/lockf* - POSIX and BSD flock/lock system calls will be completely coherent across the cluster, using the Lustre lock manager, but are not enabled by default as of now.

Lustre supports mmap file I/O as well.

(d) **Object-based architecture** - Clients are isolated from the on-disk file structure enabling upgrading of the storage architecture without affecting the client.

(e) **Disaster recovery tool** - Lustre provides an online distributed file system check, called Lustre File System Check (LFSCK) that can restore consistency between storage components in case of a major file system error. A Lustre file system can operate even in the presence of file system inconsistencies, and LFSCK can run while the file system is in use. Thus LFSCK is not required to complete before returning the file system to production.

(f) **Byte-granular file and fine-grained metadata locking** - Many clients can read and modify the same file or directory concurrently. The Lustre distributed lock manager (LDLM) ensures that files are coherent between all clients and servers in the file system. The metadata target (MDT) LDLM manages locks on inode permissions and pathnames. Each OST has its own LDLM for locks on file stripes stored on itself, which scales the locking performance as the file system grows.

8.2 Architecture

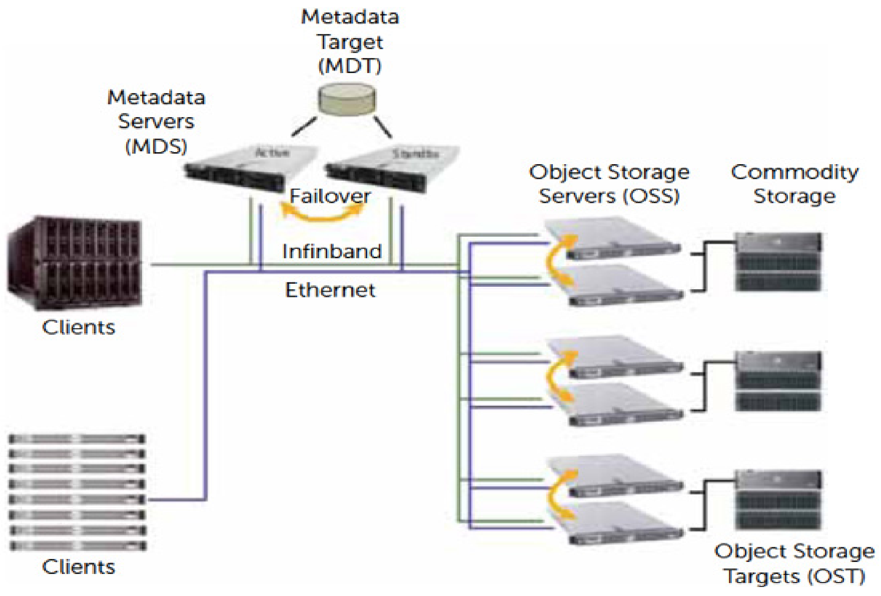

Figure 8.1: Lustre file system architecture [23]

An installation of Lustre includes a management server (MGS) and one or more Lustre file systems interconnected with Lustre networking (LNET) [21]. The MGS stores configuration information for all the Lustre file systems in a cluster and provides this information to other Lustre components. Each Lustre target contacts the MGS to provide information, and Lustre clients contact the MGS to retrieve information. While it is preferable that the MGS has its own storage space so that it can be managed independently, it can also be co-located and share storage space with a metadata server.

The components of a Lustre file system are:

8.2.1 Metadata target

For Lustre software of version 2.3 or earlier, each file system has one metadata target (MDT) [21]. From 2.4 onwards, each file system has at least one MDT, and can have more as well. The MDT stores metadata (e.g. filenames, directories, permissions, and file layout) on storage attached to a metadata server.

8.2.2 Metadata server

The metadata server (MDS) makes metadata stored in one or more MDTs available to Lustre clients [21]. Each MDS manages the names and directories in the Lustre file system(s) and provides network request handling for one or more local MDTs.

The Lustre file system has a unique inode for every regular file, directory, symbolic link and special file [24]. When a Lustre inode represents a regular file, the metadata only holds references to the file data objects stored on the OSTs, instead of references to the actual file data itself. In Lustre, creating a new file causes the client to contact a MDS, which creates an inode for the file and then contacts the OSTs to create the file objects that will store the actual file data. This is in contrast to other file systems, where creating a new file causes the file system to allocate an inode and set some of its basic attributes. Once, the MDS has directed the OSTs to create the objects for the newly created file, subsequent I/O to the file is done directly between the client and the OST, without the intervention of the metadata server. The metadata server is only updated when additional namespace changes related to the new file are required.

The MDS views each file as a collection of data objects striped on different OSTs, as mentioned earlier [45]. A file object's layout information is defined in the *extended attribute* (EA) of the inode. The EA, in essence, describes the mapping between file object IDs and the corresponding OSTs. This information is also known as *striping EA*.

8.2.3 Object storage servers

Object storage servers (OSSs) provide file I/O service and network request handling for one or more local object storage targets (OSTs) [21]. An OSS usually serves between two and eight OSTs, up to 16 TB each. A typical configuration is an MDT on a dedicated node, two or more OSTs on each OSS node, and a client on each of a large number of compute nodes.

8.2.4 Object storage targets

An OSS node can have one or more object storage targets (OSTs). OSTs handle all of the interaction between client data requests and the underlying physical storage. Within the OST, data is read from and written to underlying storage known as object-based disks (OBDs). The total data capacity is the sum of all individual OST capacities [45]. Storage in the OSTs is actually not limited to disks because the interaction between the OST and the actual storage device is done through a device driver. The behavior of the device driver mask the specific identity of the underlying storage system that is being used. This enables Lustre to utilise existing Linux file systems and storage devices for its underlying storage, whilst providing the flexibility required to integrate new technologies such as smart disks and other types of file systems. For example, Lustre currently provides OBD device drivers that support data storage within journaling Linux file systems such as ext3, ReiserFS, and XFS. Using the journaling mechanisms which come with these filesystems further increases the reliability and recoverability of Lustre. Lustre can also be used with specialized 3rd party object storage targets.

OSTs provide a flexible model for adding new storage to an existing Lustre file system. New OSTs can be easily brought online and added to a pool of OSTs that a cluster's metadata server can use for storage. Similary, new OBDs can be easily added to the pool of underlying storage associated with any OST.

One of the reasons attributed to the high performance of a Lustre file system is its ability to stripe data across multiple OSTs in a round-robin fashion [21]. Striping causes segments or 'chunks' of data in a file to be stored on different OSTs. Users can optionally configure for each file the number of stripes, stripe size, and OSTs that are used. Striping is useful when a single OST does not have enough free space to hold an entire file. Striping can also be used to improve performance when the total bandwidth to a single file exceeds the bandwidth of a single OST.

A RAID 0 pattern is used in which data is 'striped' across a number of objects [21]. Each stripe has a definite size, called the *stripe_size*. The number of objects in a single file is called the *stripe_count*. Each object contains chunks of data from the file. When the chunk of data being written to a particular object exceeds the *stripe_size* value, the next chunk of data in the file is stored on the next object.

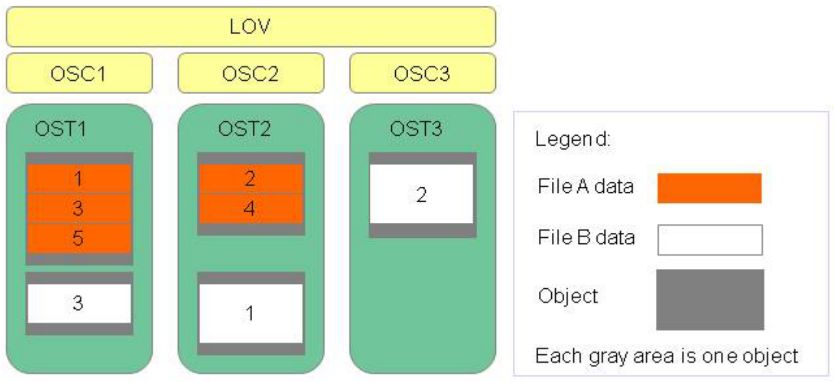

Figure 8.2: Files striped with a stripe count of 2 and 3 with different stripe sizes [22]

Default values for stripe_count and stripe_size are set for the file system [21]. The default value for stripe_count is 1 stripe for file and that of stripe_size is 1 MB. These values may be set by the user on a per-directory or per-file basis.

Since the file system metadata and file data are stored in different locations (MDS and OST respectively), file system updates involve two distinct operations - metadata updates on the MDS, and file data updates on the OSTs [24]. File system namespace operations are performed on the MDS so that they do not interfere with operations that manipulate file data. As mentioned earlier, once the MDS has identified the storage location of a file, all subsequent file I/O operations directly between the client and the OSTs. Metadata servers provide significant scope for performance optimization. For example, metadata servers can maintain a series of pre-allocated objects on various OSTs, thereby speeding up the file creation process.

8.2.5 Clients

Lustre clients are computational, visualization or desktop nodes that are running Lustre client software, allowing them to mount the Lustre file system [21]. The client software includes a management client

(MGC), a metadata client (MDC), and multiple object storage clients (OSCs). A logical object volume (LOV) aggregates the OSCs to provide transparent access across all OSTs. Hence, a client sees a single, synchronized namespace. Several clients can write to different parts of the same file simultaneously. At the same time, other clients can read from the same file. Being a POSIX-compliant filesystem, Lustre presents a unified file system interface such as open() and read() to the client. In Linux, this is achieved through Virtual File System (VFS) layer [45].

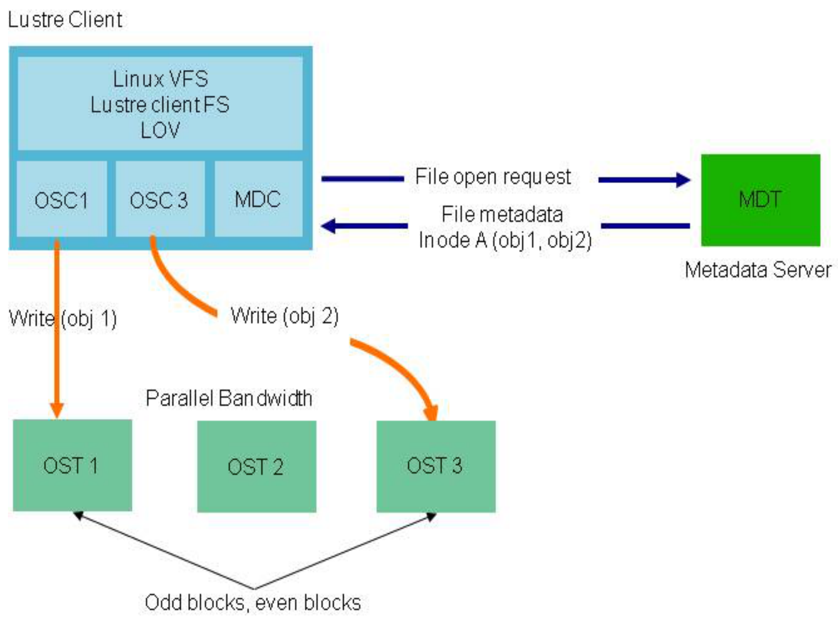

Figure 8.3: File open and file I/O in Lustre [22]

When a client wants to read from or write to a file, it first fetches the inode from the MDT object for the file [21]. The client then uses this information to perform I/O on the file, directly interacting with the OSS nodes where the objects representing the file data are stored.

The scalability of metadata operations on Lustre is enhanced through the use of an intent-based locking scheme [24]. One way of using this is as follows - when a client wants to create a file, it requests a lock from the MDS to enable a lookup operation on the parent directory, and also tags this request with the intended operation (in this case file creation). If the lock request is granted, the MDS uses the intention specified in the tag to modify the parent directory, creating a new file and returning a lock on the new file instead of the directory.

8.3 Recovery mechanism in Lustre

Lustre provides a powerful recovery mechanism which is deployed when any communication or storage failure occurs. The client is normally subjected to a timeout constraint when accessing data [24]. In the event of a server or network failure, the timeout can occur without the client having accessed what it was looking for. It can then query a Lightweight Directory Access Protocol (LDAP) server to obtain information about a replacement server and, upon obtaining a reply detailing the identity of the replacement server, immediately directs queries to that server. An 'epoch' number on every storage controller, an incarnation number on the metadata server/cluster, and a generation number between clients and other systems are the main components of the recovery system in Lustre. These numbers enable clients and servers to detect restarts and select appropriate servers.

The ability of Lustre to allow data storage in various Linux file systems such as ext3 allows utilisation of these file systems' journaling mechanisms, as mentioned earlier. This serves to increase further the recoverability of Lustre [24].

The overall file system availability is improved by having a single backup MDS, and by using distributed OSTs [24]. This helps eliminate any one MDS or OST as a single point of failure. If widespread hardware or network outages occur, the transactional nature of the metadata stored in the MDSs significantly reduces the time needed to restore file system consistency. This is because this type of metadata minimizes the chance of losing control information such as object storage locations and actual object attribute information.

8.4 Possible use cases of Lustre

While the features of Lustre make it suitable for many applications, it is not the best choice in all cases [21]. It is ideally suited for cases in which the capacity of a single server is exceeded. There are certain scenarios, however, in which Lustre can perform better with a single server than other file systems due to its strong locking and data coherency.

Lustre is not well-suited for 'peer-to-peer' usage models where clients and servers are running on the same node, each sharing a small amount of storage [21]. This is due to the lack of data replication at the Lustre software level. In such cases, if one client/server fails, the data stored on that node will not be accessible until the node is restarted.

Chapter 9

Spark and Tachyon

This chapter reviews two in-memory technologies, Apache Spark and Tachyon. A recent development in the field of in-memory computing is Apache Ignite, which processes data in-memory, and has its own in-memory file system. It was created as In-Memory Data Fabric by GridGain Systems, and found its way into the Apache Software Foundation through Incubator, the entry point of open-source projects to be part of Apache [2].

9.1 Apache Spark

Apache Spark is an open-source cluster computing framework originally developed at UC Berkeley's AMPLab. Spark was designed to support applications which reuse a working set of data across multiple parallel operations [49]. While MapReduce and its variants have been highly successful in processing large amounts of data on commodity clusters, they are mainly suitable for acyclic (or non-cyclical) data flow models. Use cases where they are not suitable include:

- Iterative jobs - Many machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter. With MapReduce, each iteration would represent a separate job, and in each job the data would be loaded from the disk, lowering the performance significantly.
- Interactive analytics - Hadoop is often used for ad-hoc exploratory queries on large datasets, through SQL interfaces like Hive and Pig. Ideally, the dataset should be loaded into memory across

a number of machines once, and then queried repeatedly. With Hadoop, each query incurs significant latency as it is a separate MapReduce job, and has to read data from the disk each time.

Spark overcomes these limitations by using an abstraction called a resilient distributed dataset (RDD), which is a collection of objects partitioned across the nodes of the cluster [49]. The collection of objects is read-only that can be rebuilt if a partition is lost. Users can explicitly cache the RDDs in memory and reuse them in parallel operations. RDDs achieve fault-tolerance through the concept of *lineage* - if one partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to rebuild just that partition. Thus, the partitions do not have to be replicated; the transformations which generated them are stored, and replayed to regenerate a partition if it is lost.

Spark is implemented in Scala. Spark can also be used interactively through the interpreter, which allows users to define RDDs, functions, variable and classes and use them in parallel operations in a cluster. In iterative machine learning jobs, Spark can outperform Hadoop by ten times, and can interactively query a 39 GB dataset with sub-second response time.

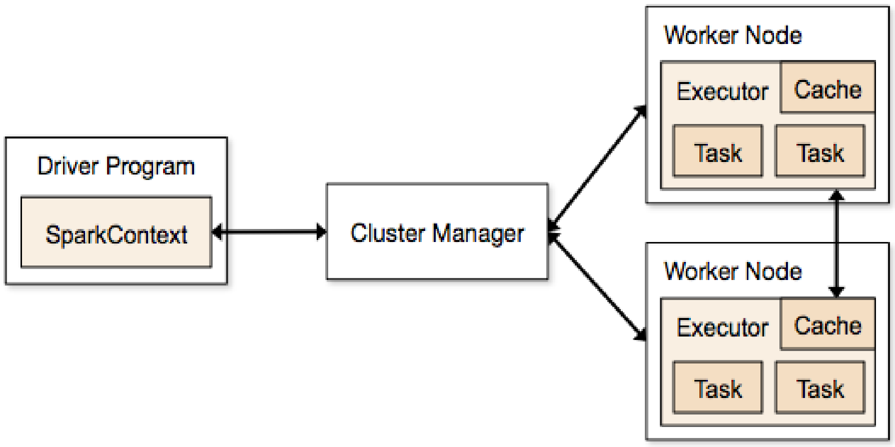

Figure 9.1: Spark cluster components [39]

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in the main program (also known as the *driver program*) [39]. The SparkContext object can connect to different types of *cluster managers* (such as YARN, or Spark's own standalone cluster manager), which allocate resources across applications. Once connected, Spark acquires *executors* on the cluster nodes. Executors are processes that run computations and store the data for the application. The SparkContext object then sends the application code to the executors, and finally send *tasks* for the executors to run.

9.1.1 Resilient distributed datasets

A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines which can be rebuilt if a partition is lost [49]. In Spark, each RDD is represented by a Scala object. Programmers can construct RDDs in four ways:

- From a file in a shared file system, like HDFS.
- By *parallelizing* a Scala collection like an array in the program - the collection is divided into a number of slices that are sent to multiple nodes.
- By *transforming* an existing RDD - using an operation called *flatMap*, a dataset with elements of one type can be converted to a dataset of another type.
- By changing the *persistence* of an existing RDD. By default, RDDs are lazy and ephemeral, i.e. partitions of an RDD are materialized on demand when they are used in a parallel operation and are removed from memory after use. The user can, however, alter the persistence of an RDD through two actions - 1) the *cache* action makes the dataset lazy, but indicates that it should be kept in memory after the first time it is computed, because it will be reused, and 2) the *save* action evaluates the dataset and writes it to a distributed storage system like HDFS. This saved version is used in future operations on it.

9.1.2 Parallel operations

The parallel operations supported on Spark RDDs include [49]:

- *reduce* - It uses an associative function to combine dataset elements to generate a result.

- *collect* - It sends all elements of the RDD to the program. For example, on eway to update an array is to parallelize, map and collect the array.
- *foreach* - This passes all elements of the RDD through a user-provided function.

9.1.3 Shared variables

Operations like *map*, *reduce* and *filter* are invoked by passing functions (called *closures*) to Spark. These closures refer to variables in the scope in which the closures are created [49]. When Spark runs a closure on a worker node, the variables are copied to the worker. However, Spark does allow programmers to create two restricted types of *shared variables* to support two usage patterns:

- *Broadcast variables* - If a large read-only piece of data is used in multiple parallel operations, it is better to distribute it to the workers only once instead of packaging it with every closure. The broadcast variable is copied to each worker only once.
- *Accumulators* - Workers can only 'add' to these variables using an associative operation, and only the driver program can read the values. These could be used to implement counters. Accumulators can be defined for any type that has an add operation and a 'zero' value. Due to their 'add-only' semantics, they can be made fault-tolerant easily.

9.2 Tachyon

Tachyon is an open-source, distributed storage system which stores data in-memory, (i.e. in RAM) while providing fault-tolerance. It enables file sharing at memory speeds across cluster frameworks like Spark and MapReduce [41]. Tachyon was developed in UC Berkeley's AMPLab and currently has over 60 contributors from more than 20 institutions like Intel and Yahoo. It is a part of the Fedora distribution and the storage layer of the Berkeley Data Analytics Stack (BDAS).

While caching can significantly improve the speed of reads, writes are slowed down by replication on disk, making them either network or disk-bound [20]. Like Spark, instead of replication, Tachyon provides fault-tolerance by leveraging the concept of lineage, in which lost output is recovered by re-executing the operations that generated the output.

The main challenge in making a long-running lineage-based storage system is timely data recovery (or recomputation) in case of failures [20]. For a single computing job using MapReduce or Spark, the recomputation cost (i.e. time) is bounded by the job's execution time. Being a storage system, however, Tachyon runs indefinitely, which means the recomputation time can become unbounded. Tachyon bounds the data recomputation cost by asynchronously checkpointing the files in the background. It uses the Edge algoritm to select which files to checkpoint and when.

Another challenge is allocating resources for recomputations. For example, if jobs have priorities, Tachyon has to make sure that recomputation tasks have adequate resources (even if the cluster is fully utilized) and also ensure that recomputation tasks do not severely impact the performance of currently running jobs with higher priorities [20]. To address this challenge, Tachyon uses resource allocation schemes which respect job priorities under two common cluster allocation models - strict priority and weighted fair sharing. For example, in a cluster using a strict priority scheduler, if missing input is requested by a low-priority job, the recomputation minimizes its impact on high-priority jobs. If this same input is later requested by a high-priority job, Tachyon automatically increases the amount of resources allocated for recomputation to avoid priority inversion.

A job P which reads file set A and writes file set B would typically be executed by Tachyon thus - before the job produces the output B, it must submit the lineage information L to Tachyon. This information describes how to run P, i.e. command line arguments, configuration parameters etc. Tachyon records L reliably using a persistent storage layer (described later). L guarantees that if the output B is lost, Tachyon can recompute it.

Like HDFS, Tachyon is an append-only filesystem, that supports standard file operations, such as create, write, read, close etc [20]. It also provides an API to capture the lineage across different jobs and frameworks. While the lineage API adds complexity to Tachyon over replication-based file systems such as HDFS, only framework programmers need to understand the API. As long as a framework like Spark integrates with Tachyon, applications on top of the framework take addvantage of lineage-based fault tolerance transparently. A user can also choose to use Tachyon as a traditional file system if he/she does not wish to use the lineage API. In that case, the application will not have the benefit of memory writes, but the performance will be no worse than that of a replicated file system.

9.2.1 Architecture

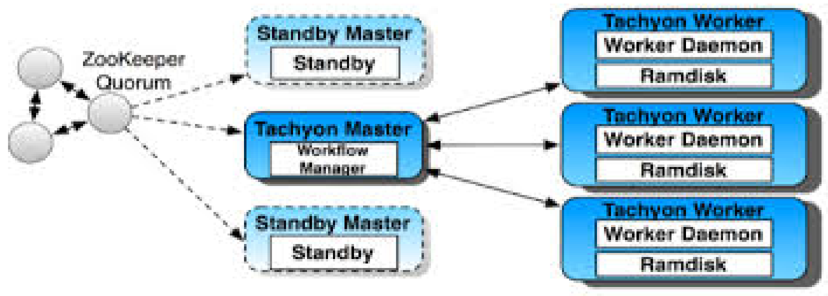

Figure 9.2: Tachyon architecture [20]

Tachyon consists of two layers - lineage and persistence [20]. The lineage layer provides high I/O throughput and tracks the sequence of jobs that created the output. The persistence layer persists data onto an underlying storage system without the lineage concept. This is done for the asynchronous checkpointing. Currently, the underlying storage systems supported are HDFS, S3, GlusterFS and the local filesystem.

Tachyon uses a master-slave architecture similar to HDFS. The master manages the metadata, and contains a *workflow manager*. This manager tracks lineage information, computes checkpoint order, and interacts with the cluster resource manager to allocate resources for recomputation. Each worker runs a daemon that manages local resources, and periodically reports the status to the master. Each worker uses a RAMdisk for storing memory-mapped files. A user application can bypass the daemon and interact directly with the RAMdisk, thereby increasing the speed of data access.

As seen in Figure 9.2, Tachyon uses a 'passive standby' approach to achieve fault-tolerance of the master. Every operation is logged synchronously to the underlying file system by the master. If the master fails, a new master is selected from the standby nodes [20]. The new master recovers the state by reading the logs. As the metadata size is significantly smaller than the output data, the overhead of storing and replicating is negligible.

9.2.2 Checkpointing

Tachyon uses asynchronous checkpointing to limit the amount of time needed to retrieve a distributed file lost to failures [20]. This file may be produced from a MapReduce/Spark job. The checkpointing is termed asynchronous because it happens in the background without stalling writes, which happen at memory speeds. It is a low priority process to avoid interference with existing jobs. A checkpointing algorithm would ideally have the following properties:

- Bounded recomputation time - In a long-running file system like Tachyon, lineage chains can grow very long. The algorithm has to limit the time needed to recompute the data in case of failures.
- Checkpointing hot files - More popular files, which are being used in jobs, should be checkpointed.
- Avoid checkpointing temporary files

Files are asynchronously checkpointed in the order in which they are created [20]. For example, consider a lineage chain, where file A_1 is used to generate A_2, which in turn generates A_3 and so on. By the time A_6 is generated, maybe only A_1 and A_2 would have been saved to persistent storage in the underlying file system. If a failure occurs, only A_3 through A_6 would be recomputed. The longer the lineage chain, greater is the recomputation time. Spreading the checkpoints throughout the chain would reduce the number of files to be recomputed, and thereby reduce the recomputation time.

The developers of Tachyon have designed an algorithm, called Edge, for the checkpointing. Firstly, the algorithm is called so because it checkpoints the edges (or *leaves*) of the lineage graph. Secondly, prioritizes files, favoring high-priority files over low-priority ones when it comes to checkpointing. Lastly, the algorithm caches only those datasets which can fit in memory to avoid synchronous checkpointing, which would slow down writes to disk speed.

The algorithm checkpoints leaves of the lineage graph by modeling the relationship of files with a directed acyclic graph (DAG) - a directed graph with no directed cycles [20]. Vertices of the graph are files, and there is an edge from A to B if B was generated by a job that read A. When there are two jobs in the cluster which generate files A_1 and B_1, the algorithm checkpoints both of them. After they have been checkpointed, files A_3, B_4, B_5, B_6 become leaves. After checkpointing these, files A_6 and B_9 become leaves. Thus, in a pipeline consisting of files A_1 to A_6 in which the first three are checkpointed, if a failure occurs when A_6 is being checkpointed, only A_4 through A_6 have to be recomputed by Tachyon to get the final result. The edges, rather than the earliest files, are checkpointed to reduce the recomputation time.

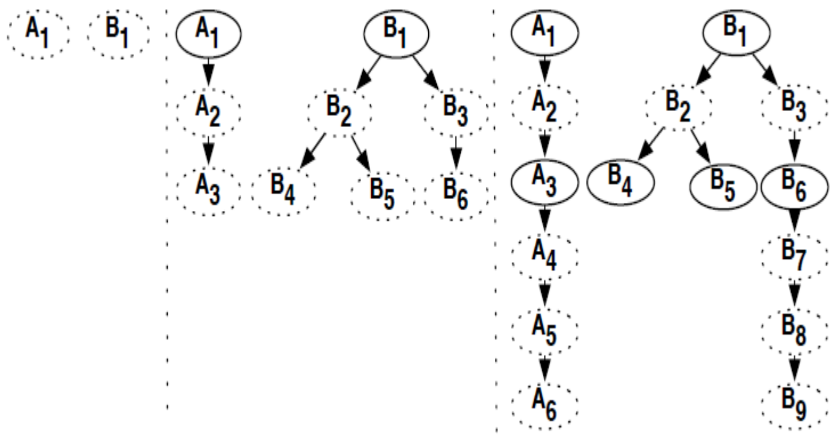

Figure 9.3: Edge checkpoint example [20]

Chapter 10

Sheepdog

Sheepdog is a distributed object storage system in user space for QEMU, iSCSI and RESTful services [36]. It provides block level storage volumes which can be attached to QEMU-based virtual machines. These volumes can be attached to non QEMU-based virtual machines or operating systems running directly on the host hardware (called *baremetal*, due to absence of a host OS) if they support the iSCSI protocol. It also supports storage for Openstack Cinder, Glance and Nova.

Sheepdog scales to several hundreds of nodes, and supports advanced volume management operations such as snapshots, cloning and thin provisioning. Volumes, snapshots, ISO images etc can be stored in the Sheepdog cluster. The capacity and power (IOPS + throughput) are aggregated, and hardware failures suitably hidden. The number of nodes can be dynamically increased or reduced.

10.1 Background information on QEMU and iSCSI

QEMU (full name is Quick EMUlator) is a free and open-source hypervisor (i.e. creates and runs virtual machines) [32]. It emulates CPU architectures by using binary translation create one instruction set of a computer architecture from another, thereby allowing it to run guest operating systems, like VirtualBox or VMWare.

iSCSI is an acronym for Internet Small Computer System Interface [16]. It is an IP-based networking standard for linking data storage facilities.

Executing SCSI commands over IP networks allows data transfer over local area networks (LANs) and wide area networks (WANs), as well enables data storage in different independent locations. The protocol allows clients (called *initiators*) to send SCSI commands to SCSI storage devices (called *targets*) on remote servers. iSCSI is a storage-area network, as it provides access to block level data storage.

10.2 Sheepdog features

Some attractive features of Sheepdog include:

(a) It is minimally dependent on the underlying kernel and file system. It requires a Linux kernel version which is equal to or greater than 2.6.32, and can work with any type of file systems which support extended attribute (xattr).

(b) It has plenty of features. Some of them include the ability to create snapshots, clones, cluster-wide snapshots etc. It supports user-defined replication/erasure code scheme on VDI(Virtual Disk Image) basis. It also has auto disk/node management features.

(c) It is easy to set up the cluster with thousands of nodes. A single daemon can manage an unlimited number of disks in one node as efficient as RAID0. There can be as many as 6000+ nodes in a single cluster. It is linearly scalable in performance and capacity.

(d) It is small in size. It is fast and has a very small memory usage (less than 50 MB even when busy). The code is easy to modify and maintain, with around 35000 lines of code in C as of now.

(e) Object storage via an HTTP interface is supported. It has an account/container/object style interface like Amazon S3 and OpenStack Swift. This makes it suitable for storing static blob contents such as photos and videos.

(f) The Sheepdog design is such that there is no single point of failure. Even if a machine fails, the data is still accessible through other machines.

(g) It is easy to administer. When administrators launch the Sheepdog daemon at a newly added machine, Sheepdog automatically detects the added machine and begins to configure it as a member of the storage system.

10.3 Sheepdog architecture

Sheepdog has a symmetric, scale-out architecture - it has no special nodes e.g. metadata server, and the difficulty of adding a new node does not depend on the number of existing nodes in the cluster. Virtual disks in Sheepdog are stored as objects [35]. The disks are divided into fixed-size objects. The objects are written to the sheepdog cluster and replicated onto multiple nodes.

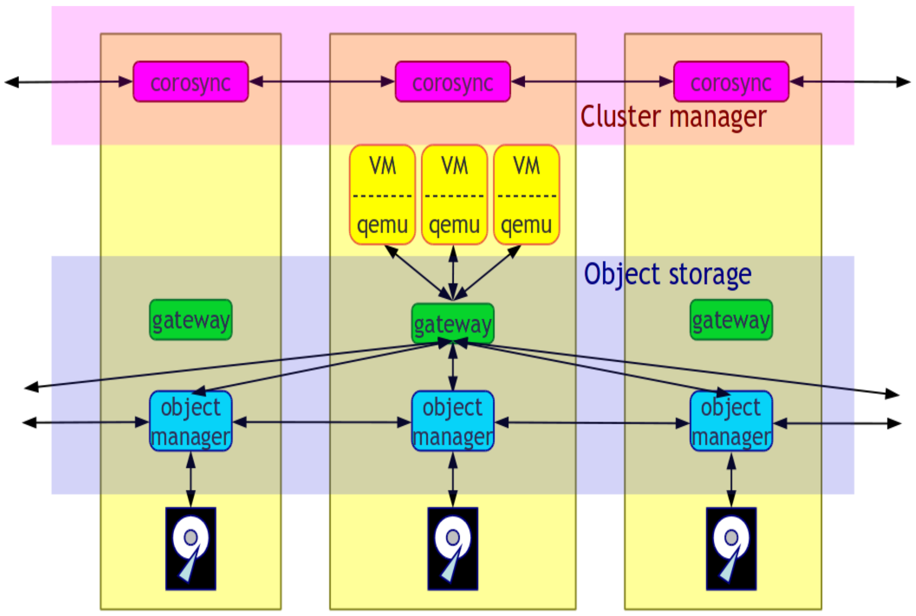

Figure 10.1: Sheepdog architecture [37]

The main components of the Sheepdog architecture are [35]:

(a) Object storage - An object in Sheepdog is flexible-sized data and has a globally unique identifier. Read/write/create/delete operations to objects may be performed by specifying this identifier. Sheepdog is not a general file system. The Sheepdog daemons create a distributed object storage system for QEMU. Object storage in Sheepdog comprises of a gateway and an object manager.

(b) Gateway - The gateway receives I/O requests (consisting of the object ID, offset, length and operation type) from the QEMU block driver, calculates the target nodes based on the consistent hashing algorithm, and forwards I/O requests to the target nodes.

(c) Object manager - The object manager receives the forwarded I/O requests from the gateway, and executes read/write operations to its local disk.

(d) Cluster manager - The cluster manager manages node membership (detection of failed/added nodes and notification of node membership changes) and some operations which requires consensus between all nodes (VDI creation, snapshots of VDI etc). Currently, corosync cluster engine is used as the cluster manager.

(e) QEMU block driver - The QEMU block driver divides a virtual machine (VM) image into fixed-size objects (4 MB by default) and stores them in the object storage system via the gateway.

10.3.1 Object storage

Each object has a globally unique 64-bit integer as its identifier, and is replicated to multiple nodes [35]. The object storage system is responsible for managing where to store the objects. Objects in Sheepdog are of 4 types:

- data object - This contains the actual data of virtual disk images. The virtual disk images are divided into fixed-size data objects. Clients access these objects.
- VDI object - This contains the metadata of virtual disk images, such as image name, disk size, creation time, data object IDs belonging to the VDIs etc.
- vmstate object - This stores the VM state of a running virtual machine, which is used when the administrator takes a live snapshot.
- VDI attr object - Attributes for each VDI can be stored in this type of an object. The attribute is key-value style.

In terms of how they are accessed, Sheepdog objects can be categorized into 2 groups -

- writable - One VM can read and write the objects but other VMs cannot.
- read-only object - No VM can write the object but any VM can read it, e.g. data objects of a snapshot VDI.

This means that virtual machines cannot share the same volume at the same time. This restriction avoids write conflicts and significantly simplifies the implementation of the storage system.

Sheepdog object storage accepts copy-on-write requests - when a client sends a create and write request, they can also specify the base object (the source of the copy-on-write request). This is used for snapshot and clone requests.

In the 64-bit identifier, the first 32 bits (0-31) is the object-type-specific space. Bits 32-55 are used as the VDI identifier, bits 56-59 are reserved, and the most significant 4 bits (60-63) represent the object type identifier.

10.3.2 Gateway

The node in which an object will be stored is determined by consistent hashing. Consistent hashing is a way to distribute the contents of a hash table over a distributed hash table (DHT). If a hash table contains K keys distributed over n servers, adding or removing a server will require the relocation of O(K/n) keys. For example, let the key space be 128 bits. Each compute node selects a random unique node identifier n_i between 0 and 2^{128} - 1.All compute nodes are ordered clockwise in a virtual ring of servers in increasing node identifier order. To store a data item with key k_j, the node with the smallest node i with node ID n_i such that $k_j \leq n_i$ is selected as the server to host it. If a new node k needs to join the virtual ring with ID n_k, it will be allocated all data items k_l for which n_k is the smallest identifier such that $k_l \leq n_k$.

Objects in a Sheepdog cluster may be encoded to data stripes and parity stripes, as well as be replicated. The advantage of erasure coding is that write performance and space efficiency are improved without degrading durability. While read performance may be improved in some cases, in others it may be degraded, although this can be covered by caching. Another disadvantage of erasure coding is that encoding and decoding of stripes is a CPU-intensive process. Coexistence of replication schemes, i.e. replication and erasure coding, is also allowed.

Write I/O flow - It is assumed that there is only one writer, so write collisions cannot happen. Clients send write requests to all the target nodes [35]. The write request is successful only when all the replicas can be successfully updated. It is because if one of the replicated objects is not updated, the gateway would read the old data from the not-updated object the next time.

Read I/O flow - The gateway calculates the target nodes with consistent hashing, and sends a read request to one of the target nodes. However, the consistency of the replication could be broken if a node breaks down while forwarding write I/O requests. Hence the gateway tries to fix the consistency when it reads the object for the first time - read object data from one of the target nodes and overwrite all replicas with it.

Sheepdog stores all node membership histories. The version number of the node membership is termed *epoch*. When the gateway forwards I/O requests to the target node and the latest epoch number does not match between the gateway and the target node, the I/O requests fail and the gateway tries the requests until the epoch numbers match. This is done to keep a strong consistency of replicated objects. I/O retry can also happen when the target nodes are down and fail to complete I/O operations.

10.3.3 Object manager

The object manager stores objects to the local disk [35]. Currently, it stores one object as one file. Objects are stored in a path having the following format:

/store_dir/obj/[epoch number][object ID]

All object files also have an extended attribute *sheepdog.copies*, which specifies the number of redundant objects.

write journaling - When the sheepdog daemon fails during write operations, objects could be partially updated. This is not a problem if the VM does not receive the success signal, there is no guarantee about the content of the written sectors. However, with regards to VDI objects, the update must happen atomically (i.e. all-or-nothing way). This is because if the VDI objects are updated partially, their metadata could be broken. To avoid this problem, the write-journaling operation is used for VDI objects. The stages of this operation are:

(a) create a journal file "/store_dir/journal/[epoch]/[vdi object id]"

(b) write data to the journal file first

(c) write data to the VDI object

(d) remove the journal file.

10.3.4 Cluster manager

In most cases, Sheepdog clients can access their images independently since they are not allowed to access the same images at the same time [35]. However, some VDI operations, such as cloning and creating VDIs, must be done exclusively because the operations update global information. To implement this without central servers, earlier Corosync cluster engine was used. However, Corosync is not recommended for production use as the method used for synchronization is susceptible to packet loss under reload. Nowadays, for reliable operations, Zookeeper is highly recommended.

10.3.5 QEMU block driver

Sheepdog volumes are divided into 4 MB data objects [35]. The objects of newly created volumes are not allocated at all. Only written objects are allocated. The operations performed by the QEMU block driver are explained below:

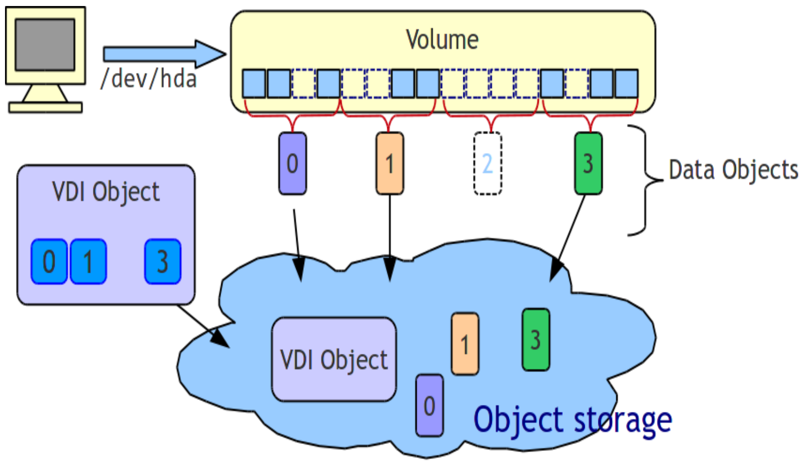

Figure 10.2: Block driver operations [35]

(a) open - The QEMU block driver reads a VDI object from the object storage system through the gateway in brdv_open().

(b) read/write - The block driver calculates the data object ID from the requested sector offset and size, and sends requests to the gateway. When the block driver sends write requests to the data object which does not belong to the current VDI, the block driver sends a copy-on-write request to allocate a new object.

(c) write to snapshot VDI - A snapshot VDI can be attached to the QEMU. When the block driver sends the write request to the snapshot VDI for the first time, the block driver creates a new writable VDI as a child of the snapshot, and sends requests against the new VDI.

Chapter 11

Experiments

The motivation of this work is to create an open-source implementation of Amazon's Elastic Map Reduce (EMR) service, and determine its performance. For big-data processing, EMR uses its own implementation of the Hadoop framework on EC2 instances, in conjunction with S3 object storage. The open-source equivalent of this is running Apache Hadoop on virtual instances in a cloud, and saving the data to Ceph object storage. Such a cluster will consist of both storage and compute nodes. The storage nodes will have several hard disks, while one hard disk would be sufficient for the compute nodes. The compute nodes should be used for running jobs with different types of frameworks, such as Hadoop and MPI, depending on requirements.

Jobs processing large datasets frequently have many intermediate stages which produce a lot of files. Writing these files to disk in storage systems which rely on replication for fault tolerance is undesirable due to two reasons - 1) writing the files to disk reduces to overall data-processing speed to disk speed, and 2) frequently such files can be easily regenerated, so replicating them can become unnecessary. Therefore, to optimize the intermediate stages of jobs, an in-memory storage system which does not use replication is ideal. Tachyon fits this description.

To determine the performance of the aforementioned open-source implementation, the read-write performances of Ceph, HDFS and Tachyon have been benchmarked.

The experiments were performed using the virtual machines available in Pouta, a production infrastructure-as-a-service (IaaS) cloud service offered by CSC - IT Center for Science Ltd. CSC is a Finnish state-owned company which provides IT infrastructure for information, education and research management. Pouta allows CSC customers to run virtual machines connected to the Internet. It runs the Grizzly version of the OpenStack cloud software [31].

Storage in Pouta is of two types :

- Ephemeral storage - In addition to the root disk space of 10 GB, most virtual machine types (or *flavors*) have an additional *ephemeral storage*. The stored data persists as long as the instance is running. It is ephemeral because the data does not persist if the instance is shut down. The stored data is also not saved when creating a snapshot of the image.
- Persistent volumes - As the name suggests, data in persistent volumes are retained even when the instances are removed. They can be attached or detached from the instances while the instances are running.

Different flavors of virtual machines/images are available in Pouta - namely tiny, tiny-noephemeral, mini, small, medium, large and fullnode. They differ in features such as the available RAM, size of ephemeral disk space, and the number of virtual CPUs (vCPU).

For the experiments, the softwares used, along with the versions are listed below:

(a) Hadoop 2.2.0,
(b) Spark 1.1.0,
(c) Tachyon 0.5.0,
(d) Ceph 0.94.1, and
(e) Simple build tool(SBT) 0.13.8

11.1 Experiment set-up

For all three types of experiments (read-only, write-only, and read-write), two separate clusters were used - one on which Tachyon and Hadoop were installed, and another on which Ceph was installed. Both clusters had three nodes each. Ubuntu 14.04 was the operating system on all the nodes.

The Tachyon-Hadoop cluster was created using three 'fullnodes'. A fullnode in Pouta has 60 GB RAM, 900 GB ephemeral disk, and 16 vCPUs. As Tachyon stores files in memory (RAM) instead of disk, a large amount of the available RAM in each fullnode (40 GB) was allocated to the storage system. 15 GB of memory was assigned to each Spark executor. The total available RAM for Tachyon was thus 120 GB (40*3).

Ceph was installed on a separate three-node cluster. For Ceph, nodes of the 'mini' flavor were used. A mini node in Pouta has 3.5 GB RAM, 110 GB ephemeral disk space and uses 1 vCPU. Each node functioned as an object storage device (OSD). One node, along with being an OSD, also functioned as the metadata server and admin nodes.

The Ceph nodes were added to an 'anti-affinity' server-group. A server-group with anti-affinity setting in OpenStack has each virtual machine in the group installed on a different physical host. That way, it was ensured that the VMs in the same server group were not competing for the same ephemeral disk I/O. To ensure that there were no variations in the installation process, the installations of Hadoop, Tachyon and Ceph were automated by the use of scripts. The creation of a bucket - named 'test' - in the Ceph cluster was also automated by running a script. The relevant scripts for installing Hadoop, Tachyon and Ceph will be found in the following Github link:

`https://github.com/Alapan/Thesis-files`

Spark, instead of Hadoop, was used for the data-processing in the experiments, for the greater speed it offers. The root partition of all virtual machines in Pouta have 10 GB space, out of which 3-4 GB is used by the existing softwares. Since the remaining 6 GB (approximately) was not sufficient for storing the files produced by the workers during processing, they were saved to the ephemeral space instead, of size 900 GB in fullnodes. The SPARK_WORKER_DIR configuration option in the nodes was set to `/mnt/work`, `/mnt` being the location where the ephemeral storage was mounted.

For HDFS, the data was written to the ephemeral storage space as well, due to the large amount of space available. For each slave node, the datanode location was `/mnt/datanode`, while for the master node, the namenode location was `/mnt/namenode`. One node in the 3-node cluster functioned as both the master and slave.

11.2 Tachyon modifications

Tachyon 0.5.0 was used for benchmarking. To provide fault-tolerance, it checkpoints in-memory data (i.e. stores intermediate files produced in a pipeline) to an underlayer file system. This file system is configurable, and currently HDFS, S3, GlusterFS, and local filesystems in single nodes are supported.

For the experiments, Ceph was configured to be the underlayer file system. To do this, the Simple Storage Service (S3) interface of Hadoop was modified to point to the Ceph cluster set up. More specifically, the jets3t library used by Hadoop to access S3 storage was modified by setting the endpoint to the admin node of the Ceph cluster used in the experiments.

The source code of Tachyon was also altered to make it accept S3N URLs. The format of an S3N URL is given below:

```
s3n://bucket_name/folder_name
```

For the 0.5.0 version of Tachyon, the existing code does not recognise the bucket name, and instead regards it as a server name, i.e. similar to HDFS. Thus, the code had to be altered slightly to accomodate the bucket name in the URL.

To reduce the task completion times, the checkpointed files were instead saved to the RAM of the nodes in the Hadoop-Tachyon cluster itself. This effectively rendered the underlayer file system in the experiments, Ceph, redundant. The configurations made to Tachyon for saving the checkpointed files to RAM are given below:

- tachyon.data.folder - `/mnt/ramdisk/data`
  (default - `$TACHYON_UNDERFS_ADDRESS/tmp/tachyon/data`)
- tachyon.workers.folder - `/mnt/ramdisk/workers`
  (default - `$TACHYON_UNDERFS_ADDRESS/tmp/tachyon/workers`)

Here, too, the source code of Tachyon was changed to allow the metadata to be saved to RAM instead of the configured underlayer file system.

11.3 Raw disk write and network speeds

In addition to the benchmarks, the raw disk write speeds in both the Ceph and Tachyon-HDFS clusters were measured, along with the network speed within each cluster individually, and between the two clusters. This was done to identify any bottlenecks in writes due to the disks and/or the network.

The commands used to determine the raw disk write speed were:

```
cd /mnt
sudo dd if=/dev/zero of=here bs=32M count=30 oflag=direct
```

`/mnt` was the directory in which the ephemeral storage was mounted, i.e. location of the datanodes and OSDs. The command indicates the writing of 30 files, each of 32 MB size. For network speeds, the following commands were used:

Server: `iperf -s`

Client: `iperf -c 192.168.50.107 -i1 -t 10 -s`

The client and server were two separate nodes in a cluster when the speed within a cluster was being determined. For the network speed between two different clusters, one node from each cluster was used.

11.4 RADOS benchmarking

Writes with RADOS were also benchmarked using the internal benchmarker of RADOS. The commands used, in the admin node of the Ceph cluster were:

```
rados mkpool my_pool

rados bench -p my_pool 300 write
```

The command indicates writing to a pool 'my_pool' of placement groups for 300 seconds. Parallel writes are performed by 16 threads, which is the default number. The data is written in objects of size 4 MB.

11.5 Read-only benchmarks

The read-only benchmark involved reading a file on the storage system being benchmarked, and performing some tasks on the data contained in the file to generate an output, without creating any output files. This would ensure no writing was done.

The first part of the experiment involved generating the input data. To do this, the `teragen` program was run to create an input folder of size 24.90 GB. The input parameters to the program were the number of partitions to be created and the range of numbers to be generated. The `teragen` source code was altered slightly, to produce readable text files as output, instead of the compressed (bzip2) format used in the terasort benchmark. This is because compression reduces the overall read/write performance from CPU to disk speed. The modified `teragen` program was used for the write-only and read-write benchmarks as well.

The task involved counting the total number of numeric values in the input partitions/files, and displaying the calculated value on the output console. Thus no output files were created. The experiment was repeated 10 times for each storage system - Tachyon, Ceph and HDFS - and the average read throughput calculated. A piece of sample Scala code, for HDFS is given below:

```
val file = sc.textFile("hdfs://master-full:54310/tera-output")
val splits = file.map(word => word.toLong)
println(splits.count())
```

The numbers in the output from the `teragen` code (named `tera-output` in the code) were in string format. They were converted to long, to make the read process easier. The read-only benchmark thus used a separate script to process the `teragen` output.

For HDFS, the experiment had two variants - one in which caching of the input files was enabled, and another in which it was disabled. As the input files in HDFS were written on disk, the files were being stored in the Linux buffer cache by default, to speed up the reading process. For the experiment where caching was disabled, the cache was cleared after every run, to ensure the file was read fully from the disk. This was not necessary in the case of Tachyon as the input file was already on RAM. Ceph is similar to HDFS, in that the input file is stored on disk as well. However, the caching could not be completely eliminated for Ceph, due to an extra layer of virtualization present for the external cluster.

11.6 Write-only benchmarks

The write-only experiment involved generating data files and writing them directly to the output storage system. Thus, no input files were present. The `teragen` program was run and the time taken to write the files to the output storage systems measured. As with the read-only benchmark, the input folder was of size 24.90 GB. For each storage system, the experiment was run five times, and the average write throughput calculated. No other scripts were necessary for this experiments, other than the existing `teragen` code.

11.7 Read-write benchmarks

Both input and output files were created for the read-write benchmark. To benchmark the read-write performance of each storage system, the input and output files were read from and written to the same storage system.

The input files were generated using the aforementioned `teragen` program. The task itself was copying the numbers in the input files/partitions to the files in the output folder. This meant that the input and output folders produced would be of equal size. The size of the input folder would therefore have to be such that both the input and the output data would fit in the available space. The limiting factor here, then, was Tachyon. The 120 GB RAM allocation for the Tachyon cluster meant that, theoretically at least, a maximum input folder of size 60 GB could be accomodated (60*2=120). For the experiments, the input file size was the same as in the other experiments - 24.90 GB.A sample of the relevant source code, for HDFS, is given below:

```
val file = sc.textFile("hdfs://master-full:54310/tera-output")
val splits = file.map(word => word.toLong)
splits.map (row => (NullWritable.get(),new LongWritable(row)))
.saveAsNewAPIHadoopFile("hdfs://master-full:54310/output/",
classOf[NullWritable], classOf[LongWritable],
classOf[TextOutputFormat[NullWritable, LongWritable]])
```

11.8 Results

The results obtained are presented below:

11.8.1 Disk/network speeds

The raw disk write speeds were similar for nodes in the Ceph and Tachyon-HDFS cluster. The optimal speed was close to 90 MB/s, and the slowest writes were in the 50-60 MB/s range. The network speeds (within Ceph cluster, within Tachyon-HDFS cluster, and between the two clusters) are tabulated below:

Interval (sec)	Transfer (MB/s)	Bandwidth (GBits/sec)
0.0-1.0	766	6.42
1.0-2.0	893	7.49
2.0-3.0	893	7.49
3.0-4.0	890	7.47
4.0-5.0	893	7.49
5.0-6.0	890	7.47
6.0-7.0	900	7.55
7.0-8.0	902	7.56
8.0-9.0	898	7.53
9.0-10.0	896	7.52

Table 11.1: Network connection speed within Ceph cluster

Total data transferred in 10s in Ceph cluster - 8.61 GB

Bandwidth - 7.40 Gbits/sec

Interval (sec)	Transfer (MB/s)	Bandwidth (GBits/sec)
0.0-1.0	694	5.82
1.0-2.0	843	7.07
2.0-3.0	851	7.14
3.0-4.0	971	8.14
4.0-5.0	959	8.05
5.0-6.0	965	8.10
6.0-7.0	965	8.10
7.0-8.0	810	6.79
8.0-9.0	803	6.74

Table 11.2: Network connection speed within Tachyon-HDFS cluster

Total data transferred in 10s in Tachyon-HDFS cluster - 8.43 GB

Bandwidth - 7.24 Gbits/sec

Interval (sec)	Transfer (MB/s)	Bandwidth (GBits/sec)
0.0-1.0	773	6.36
1.0-2.0	842	7.02
2.0-3.0	836	6.96
3.0-4.0	825	6.27
4.0-5.0	826	7.13
5.0-6.0	844	7.82
6.0-7.0	926	7.32
7.0-8.0	891	6.87
8.0-9.0	855	6.82
9.0-10.0	851	6.38

Table 11.3: Network connection speed between Ceph and Tachyon-HDFS clusters

Total data transferred in 10s between Ceph and Tachyon-HDFS clusters- 8.27 GB

Bandwidth - 7.10 Gbits/sec

11.8.2 RADOS benchmarks

The results of the RADOS write benchmark are given below:

Total time run: 302.543860

Total writes made: 1348

Write size: 4194304

Bandwidth (MB/sec): 17.822

Stddev Bandwidth: 13.7206

Max bandwidth (MB/sec): 60

Min bandwidth (MB/sec): 0

Average Latency: 3.5878

Stddev Latency: 1.09165

Max latency: 6.89335

Min latency: 0.56164

11.8.3 Tables and graphs

Job type	Average time (seconds)	Throughput(MB/s)[(24.90 GB * 1000)/average time]
Number count with Spark - Input files on Tachyon	28.54	872.46
Number count with Spark - Input files on HDFS (with caching)	32.07	776.43
Number count with Spark - Input files on HDFS (without caching)	157.27	158.33
Number count with Spark - Input files on Ceph	189.21	131.60

Table 11.4: Read-only benchmarks

Job type	Average time (seconds)	Throughput(MB/s)[(24.90 GB * 1000)/average time]
Teragen on Spark - Output files on Tachyon	37.33	667
Teragen on Spark - Output files on HDFS	555.68	44.81
Teragen on Spark - Output files on Ceph	1665.55	14.95

Table 11.5: Write-only benchmarks

Job type	Average time (seconds)	Throughput(MB/s)[(24.90 GB * 1000)/average time]
Number copy with Spark - Output files on Tachyon	226.10	110.13
Number copy with Spark - Output files on HDFS	1542.75	16.14
Number copy with Spark - Output files on Ceph	4353.15	5.72

Table 11.6: Read-write benchmarks

The tabulated data are represented in the following bar graphs:

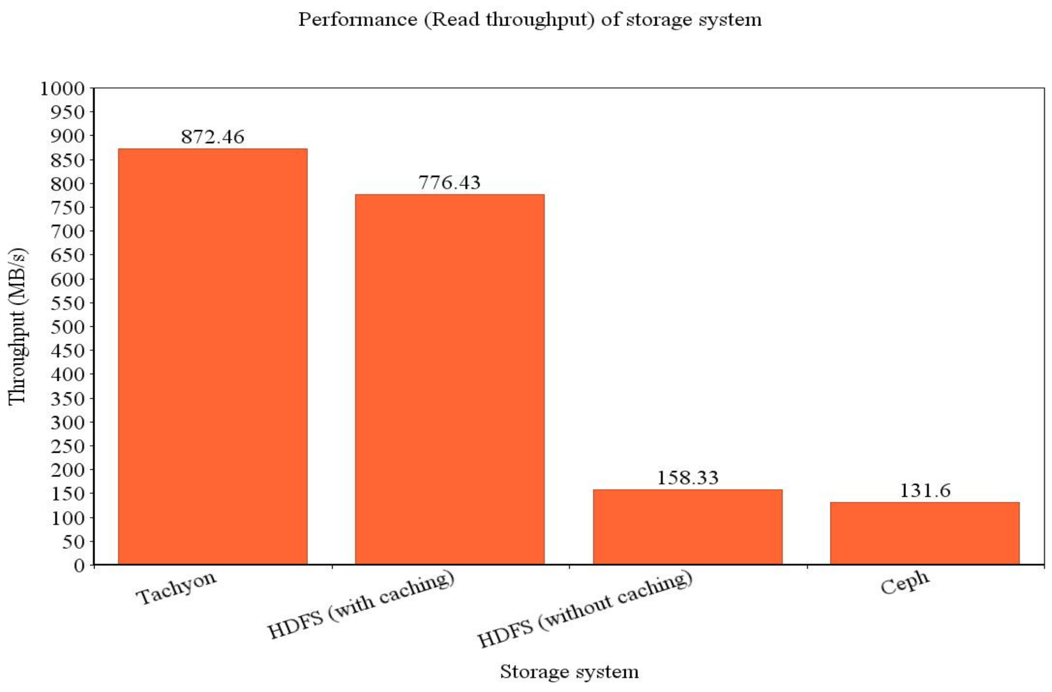

Figure 11.1: Read-only performance of storage system

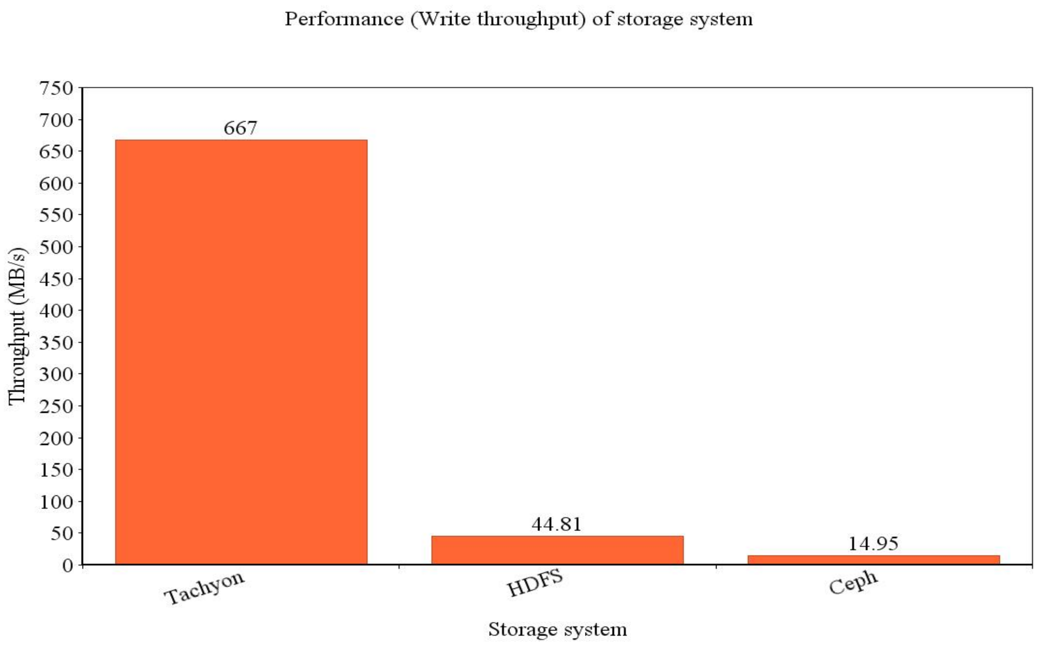

Figure 11.2: Write-only performance of storage system

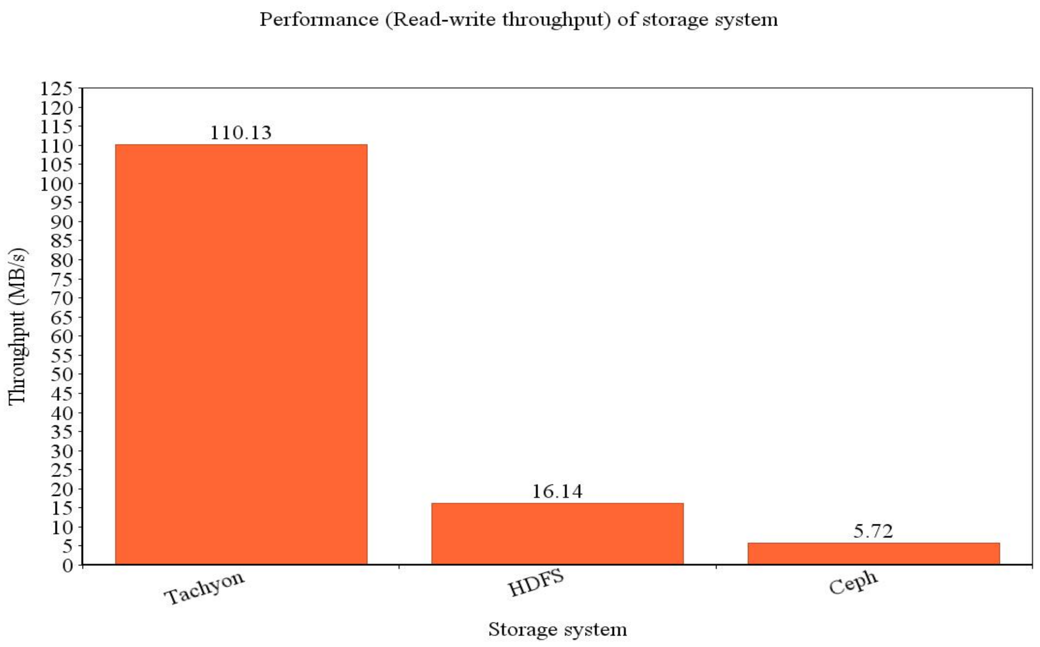

Figure 11.3: Read-write performance of storage system

11.8.4 Discussion

In general, being an in-memory storage system, Tachyon's performance, especially in writes, was far superior to that of HDFS and Ceph. Of the three, Ceph showed the least performance. The raw disk-write (∼55 MB/s in the worst case) and network (700-1000 MB/s) speeds are too high to be responsible for the low Ceph speeds. The Ceph write speed (∼15 MB/s) is closer to the RADOS write benchmark (17.8 MB/s), indicating that the Ceph speeds are determined by RADOS, and not the network as originally thought.

The write speed to a single disk (55-90 MB/s) is considerably more than writes through the RADOS gateway (17.8 MB/s). This could be explained by RADOS's replication strategy - the data is written first to log files and then the local storage. On top of that, three-way replication among the OSDs is being performed, which means 6* input/output operations per second (IOPS) as a single write is taking place. This could slow down the write bandwidth of RADOS.

The virtual instances of Pouta run on the physical machines of the Taito supercluster provided by CSC. A variety of jobs run on this supercluster, and depending on the number of jobs hitting the disks at any time, the read/write speed for the disks can be affected. This could explain the variation in the raw disk-write speeds. Consecutive runs of the raw disk-speed test would yield significantly differing write speeds, from 90 MB/s to 55 MB/s for example.

Another factor to consider is the anti-affinity setting for the server group to which the instances are being added. While the setting is used to assign each virtual machine to a different physical host (so that the machines do not compete for the same ephemeral storage), creating a virtual machine takes time and if the commands are applied very fast to create the instances (e.g. through a script), the setting may not work. That could result in the instances being created on the same physical host, regardless of the anti-affinity setting. For the experiments, the commands for creating the instances were applied manually.

11.8.4.1 Read-only and write-only performance

Tachyon's read performance was only slightly better than HDFS when the latter had caching enabled. It was, however, significantly faster than HDFS when caching was disabled in the latter. This was expected as the input files in HDFS were then being read from the disk each time. When caching was disabled, the read speed of HDFS was nearly the same as Ceph.

The write speed of Tachyon was about 15X that of HDFS and 48X that of Ceph. This is in keeping with the RADOS write bandwidth determined earlier.

11.8.4.2 Read-write performance

Initially, the read-write experiment for Ceph was performed using a single bucket, i.e. the same bucket was to store the input and output files. With this, however, the job was starting, but not progressing beyond some initial reads. The job ran successfully when 2 buckets were created - one for the input files, and the other for the output.

While performing the read-write experiment, it was seen that with a larger input file, say 40 GB or more, tasks were failing in the Spark executors. This was not happening with HDFS or Ceph. For a total RAM size of 120 GB for the Tachyon cluster, theoretically the maximum input file that could be stored is 60 GB (60*2 = 120, 60 GB for the input and output data sets). To fail with files of size 50 GB or thereabouts was not expected. This is a potential problem in a pipeline - if 50 GB of input data are being read from the disk, and an equal amount of output data being written to disk, then, the intermediate stages (where Tachyon is ideal) should also work with 50 GB data. Thus, although Tachyon showed very high read, write and read-write speeds compared to the other two storage systems, its read-write performance with large datasets (in relation to the total amount of RAMdisk assigned to the Tachyon cluster) is a potential area for further study.

Chapter 12

Summary

The main contribution of this work has been the integration of Ceph as an underlying storage system with Tachyon, and the subsequent benchmarking. With its in-memory storage system, Tachyon speeds up read-only, write-only, and read-write processes considerably. It is thus ideal for optimising the intermediate stages of a pipeline process, by having the generated files stored to memory, rather than the disk. To these ends, the work has been successful. The benchmarking process also revealed an important point about Ceph - the RADOS gateway to the object storage cluster is a main factor in determining the cluster's performance.

However, there were difficulties faced during the work. These included the following:

(a) It took a fairly long time to set up Tachyon with Ceph as the underlying storage system. The version of Tachyon used for the experiments, 0.5.0, had a code defect in the way S3N URLs were handled. For S3N, the name following the `‘s3n://’` part of the URL (e.g. `‘test’` in `‘s3n://test/tera-output’`) refers to the bucket in which the files are to be stored, not the server. However, in version 0.5.0 of Tachyon, the code regarded the name to be that of the server. The code was appropriately modified.

(b) Tachyon does not support saving of checkpointed files to memory. They are meant to be written to more persistent storage systems. For our experiments, saving these files to memory was necessary to save time and space. Hence, the source of Tachyon had to be changed to allow in-memory storage of checkpointed files.

(c) Being an in-memory storage system, Tachyon gave high performances for read-only, write-only and read-write tasks. However, it was difficult to run read-write jobs on Tachyon with larger input data sets, as tasks assigned to Spark executors were being lost with even 40 GB input data.

(d) When installing Ceph, there were errors using the ephemeral storage space, instead of the root partition, for the object storage devices (OSDs). Again, using ephemeral storage was necessary as the root partition had only 10 GB of space. Thus, changes had to be made in the Ceph installation process.

(e) It was difficult to use a Ceph cluster for a long time. After some experiments, a few placement groups in the cluster would become inconsistent, due to scrubbing (comparing objects and their replicas in placement groups to ensure that there are no missing or mismatched objects) problems. Despite running repair operations of the placement groups, after a few more experiments, the placement groups would become inactive/stale. The cluster would have to be set up again.

(f) For read-write tasks using a single bucket in Ceph, the job was starting, but not progressing beyond a few reads. The problem was rectified was using separate buckets for the input and output file sets. This solution may not scale well, i.e. having separate input and output buckets for every user may create data management problems.

Keeping these points in mind, the following represent areas for further study:

(a) In the experiments performed, one node doubled as the admin node and an object storage device (OSD). It could be worthwhile to separate the two of them, i.e. have one node as the admin only and others as OSDs.

(b) Read-write jobs with Tachyon could be run on clusters of varying RAMdisk sizes, and varying instance configurations.

The long-term goal of this work is to set up an HPC (high-performance computing) cluster with both storage nodes providing Ceph storage, and compute nodes providing support for a wide range of frameworks such as Hadoop and MPI (Message Passing Interface) for users to run their jobs. In particular, the compute nodes could be used elastically, i.e. users can run various kinds of jobs with different frameworks when needed. This would allow the nodes to be used optimally. This represents an open-source alternative to Amazon's EMR (Elastic Map Reduce) service, which uses its own Hadoop framework with S3 object storage.

Along with the HPC cluster, it would be ideal to run jobs involving bioinformatics pipelines with a large number of intermediate stages, if an in-memory storage system like Tachyon would be present in the compute nodes to store the intermediate files. In-memory data processing has gathered a lot of interest in recent times, due to the significantly increased job speeds. This increased speed can be used to optimise the intermediate stages of the aforementioned pipeline. Lower RAM costs, coupled with the larger RAM space in current 64-bit systems, make in-memory processing an attractive option.

This work identified two major issues in the relevant technologies which could serve as roadblocks for the scenario described earlier - 1) the need to create separate Ceph buckets for input and output file sets for each user, and 2) the problems with Tachyon in running read-write jobs with large files when the reads and writes are of equal/comparable amounts. Along with these, there are smaller, but important, issues with Ceph and Tachyon (described earlier) which need to be resolved before they can be deployed in production.

Bibliography

[1] Data striping. http://www.webopedia.com/TERM/D/disk_striping.html.

[2] GridGain Becomes Apache Ignite. http://www.infoq.com/news/2014/12/gridgain-ignite.

[3] OpenStack Swift Architecture. http://thetrendythings.com/read/4816.

[4] What is File Level Storage vs. Block Level Storage? http://www.iscsi.com/resources/File-Level-Storage-vs-Block-Level-Storage.asp.

[5] Borthakur, D. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[6] Boudjnah, C. Ceph and Swift: Why we are not fighting. http://techs.enovance.com/6427/ceph-and-swift-why-we-are-not-fighting.

[7] Brewer, E. A. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2000), PODC '00, ACM, p. 7.

[8] Cade Metz. Google Remakes Online Empire with 'Colossus'). http://www.wired.com/2012/07/google-colossus/.

[9] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M. F. u., Haq, M. I. u., Bhardwaj, D.,

Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 143–157.

[10] Ceph: Architecture. `http://ceph.com/docs/master/architecture/`.

[11] Chandrasekar, S., Dakshinamurthy, R., Seshakumar, P., Prabavathy, B., and Babu, C. A novel indexing scheme for efficient handling of small files in Hadoop Distributed File System. In *Computer Communication and Informatics (ICCCI), 2013 International Conference on* (Jan 2013), pp. 1–8.

[12] Factor, M., Meth, K., Naor, D., Rodeh, O., and Satran, J. Object storage: the future building block for storage systems. *International Symposium on Mass Storage Systems and Technology 0* (2005), 119–123.

[13] Lustre FAQ: Fundamentals. `http://ceph.com/docs/master/architecture/`.

[14] Florent Flament. OpenStack Swift made understandable. `http://www.florentflament.com/blog/openstack-swift-ring-made-understandable.html`.

[15] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[16] iSCSI. `http://en.wikipedia.org/wiki/ISCSI`.

[17] Jones, M. Ceph:a Linux petabyte-scale distributed file system. `http://www.ibm.com/developerworks/library/l-ceph/`.

[18] Windows Azure Storage: A highly available cloud storage service with strong consistency. `http://yongchengwangcsci8980.blogspot.fi/2012/10/critique_29.html`.

[19] Layton, J. B. Ceph: The Distributed File System Creature from the Object Lagoon. `http://www.linux-mag.com/id/7744/`.

[20] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 6:1–6:15.

[21] Lustre Software Release 2.x -Operations Manual. `https://build.hpdd.intel.com/job/lustre-manual/lastSuccessfulBuild/artifact/lustre_manual.pdf`.

[22] Introduction to Lustre. `http://wiki.lustre.org/manual/LustreManual18_HTML/IntroductionToLustre.html`.

[23] Lustre Storage System. `http://www.nor-tech.com/solutions/hpc/lustre-storage-system/`.

[24] Lustre: A Scalable, High-Performance File System. `http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf`.

[25] Maltzahn, C., Molina-Estolano, E., Khurana, A., Nelson, A., Brandt, S., and Weil, S. Ceph as a scalable alternative to the Hadoop Distributed File System. *USENIX; login 35*, 4 (2010), 38–49.

[26] McKusick, M. K., and Quinlan, S. GFS: Evolution on Fast-forward. *Queue 7*, 7 (Aug. 2009), 10:10–10:20.

[27] Mesnier, M., Ganger, G. R., and Riedel, E. Object-based storage. *Comm. Mag. 41*, 8 (Aug. 2003), 84–90.

[28] Monitor Config Reference. `http://ceph.com/docs/master/rados/configuration/mon-config-ref/`.

[29] OpenStack Architecture. `https://www.swiftstack.com/openstack-swift/architecture/`.

[30] Placement Groups. `http://ceph.com/docs/master/rados/operations/placement-groups/`.

[31] Pouta User Guide. `https://research.csc.fi/pouta-user-guide`.

[32] QEMU. `http://en.wikipedia.org/wiki/QEMU`.

[33] SDS - software-defined storage. `http://www.webopedia.com/TERM/S/software-defined_storage_sds.html`.

[34] OpenStack Swift. `https://developers.seagate.com/display/KV/OpenStack+Swift`.

[35] Sheepdog design. `https://github.com/sheepdog/sheepdog/wiki/Sheepdog-Design`.

[36] Sheepdog wiki. `https://github.com/sheepdog/sheepdog/wiki`.

[37] Sheepdog: Architecture. `https://github.com/sheepdog/sheepdog/wiki/Sheepdog-Design`.

[38] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[39] Cluster Mode Overview. `https://spark.apache.org/docs/1.0.1/cluster-overview.html`.

[40] Windows azure storage: a highly available cloud storage service with strong consistency. `http://csci8980.blogspot.fi/`.

[41] Tachyon Overview. `http://tachyon-project.org/`.

[42] Taneja, A. How an object store differs from file and block storage). `http://searchcloudstorage.techtarget.com/feature/How-an-object-store-differs-from-file-and-block-storage`.

[43] Ukov, D. (Object Storage approaches for Swift and Ceph: Understanding Swift and Ceph). `https://www.mirantis.com/blog/object-storage-openstack-cloud-swift-ceph/`.

[44] Wang, F., Nelson, M., Oral, S., Atchley, S., Weil, S., Settlemyer, B. W., Caldwell, B., and Hill, J. Performance and Scalability Evaluation of the Ceph Parallel File System. In *Proceedings of the 8th Parallel Data Storage Workshop* (New York, NY, USA, 2013), PDSW '13, ACM, pp. 14–19.

[45] Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T., and Huang, I. Understanding Lustre filesystem internals.

[46] Windows Azure Storage: A highly available cloud storage service with strong consistency. `http://blogs.msdn.com/b/windowsazurestorage/archive/2010/12/30/windows-azure-storage-architecture-overview.aspx`.

[47] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. Crush: Controlled, scalable, decentralized placement of replicated data. In *IN PROCEEDINGS OF THE 2006 ACM/IEEE CONFERENCE ON SUPERCOMPUTING (SC â06* (2006), ACM, p. 2006.

[48] YoyoClouds. HDFS Architecture. `http://yoyoclouds.wordpress.com/2011/12/15/hdfsarchitecture/`.

[49] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.