Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Προσαρμογή Παράλληλου Συστήματος Αρχείων για την Παροχή Υπηρεσίας Αποθήκευσης Αντικειμένων πάνω από Δίκτυο Περιοχής Αποθήκευσης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΙΩΑΝΝΗΣ ΧΑΤΖΗΜΙΧΟΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2015

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Προσαρμογή Παράλληλου Συστήματος Αρχείων για την Παροχή Υπηρεσίας Αποθήκευσης Αντικειμένων πάνω από Δίκτυο Περιοχής Αποθήκευσης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΙΩΑΝΝΗΣ ΧΑΤΖΗΜΙΧΟΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Μαρτίου 2015.

Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Αναπ. Καθηγητής Ε.Μ.Π.	Δημήτριος Φωτάκης Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2015

Ιωάννης Χατζημίχος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η αποθήκευση αντικειμένων (object storage) είναι μία πολύ δημοφιλής αρχιτεκτονική αποθήκευσης, καθώς επιτρέπει την οργάνωση τεράστιας ποσότητας δεδομένων, υποστηρίζοντας την εύκολη επεκτασιμότητα της υποδομής με σχετικά χαμηλό κόστος. Τα παραπάνω πλεονεκτήματα ώθησαν πολλές υπηρεσίες να επιλέξουν να υιοθετήσουν αυτήν την τεχνολογία για τις αποθηκευτικές ανάγκες τους.

Παρόλα αυτά, πολλοί οργανισμοί έχουν ήδη εγκαταστήσει Δίκτυα Περιοχής Αποθήκευσης (Storage Area Networks), οι οποίες είναι λύσεις αποθήκευσης που βασίζονται σε αποθήκευση σε συσκευές μπλοκ (block storage). Αν και οι Περιοχές Δικτύων Αποθήκευσης έχουν γενικά μεγαλύτερο κόστος και δεν είναι το ίδιο εύκολα επεκτάσιμες, ταυτόχρονα το υλικό τους είναι πιο αξιόπιστο και ανεκτικό σε σφάλματα (fault-tolerant). Επιπλέον, το κόστος και η πολυπλοκότητα των SANs έχει μειωθεί σημαντικά τα τελευταία χρόνια.

Η παρούσα διπλωματική εργασία παρουσιάζει τον σχεδιασμό και την υλοποίηση μιας αρχιτεκτονικής για την παροχή υπηρεσίας αποθήκευσης σε αντικείμενα πάνω από Δίκτυο Περιοχής Αποθήκευσης. Ειδικότερα, επικεντρώνεται στην παροχή εικονικών δίσκων για Εικονικές Μηχανές (VMs) σε υποδομές υπολογιστικού νέφους (cloud computing).

Για το σκοπό αυτό, χρησιμοποιήσαμε το παράλληλο σύστημα αρχείων OCFS2, τροποποιώντας το σε μικρό βαθμό για να αυξήσουμε την επίδοσή του στο σενάριο χρήσης μας. Οι πρώτες μετρήσεις δείχνουν ότι η προσαρμογή μας είναι ταχύτερη έως και κατά 30% στις μεμονωμένες εντολές E/E.

Λέξεις κλειδιά

OCFS2, Synnefo, okeanos, Archipelago, synapsed, εικονική μηχανή, υπολογιστικό νέφος, παράλληλο σύστημα αρχείων, Δίκτυο Περιοχής Αποθήκευσης, object storage, block storage, SCSI, cloud computing, QEMU, KVM

Abstract

Object storage is a very popular storage architecture, since it enables the storage of massive amounts of data, allowing the infrastructure to scale accordingly in a relatively inexpensive manner. These advantages have led many services to adopt this storage infrastructure.

Nevertheless, many organisations have already chosen to set up Storage Area Networks, which are storage solutions based on block storage. Even though Storage Area Networks are generally more expensive and harder to scale, they consist of reliable and fault tolerant-hardware. This, combined with the fact that the cost and complexity of SANs has dropped in the last years has led to wider market adoption.

This diploma thesis presents the design and implementation of an object storage service over a Storage Area Network, focusing on providing virtual volumes which are then used by VMs in a cloud computing infrastructure.

To this end, we used the OCFS2 parallel filesystem, modifying it slightly to improve its performance for our use case. Our early performance evalution shows that the modifications render the individual I/O requests faster by up to 30%.

Key words

OCFS2, Synnefo, okeanos, Archipelago, synapsed, virtual machine, cloud, parallel filesystem, Storage Area Network, object storage, block storage, SCSI, cloud compting, QEMU, KVM,

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή Νεκτάριο Κοζύρη, που μου έδωσε την ευκαιρία να ασχοληθώ με το πολύ ενδιαφέρον αντικείμενο του cloud computing.

Ακόμη, θα ήθελα να ευχαριστήσω ιδιαίτερα τον Δρα. Βαγγέλη Κούκη ο οποίος με τον ενθουσιασμό και τη μεταδοτικότητα του με έκανε να αγαπήσω τον προγραμματισμό συστημάτων και με βοήθησε καθοριστικά με την εκπόνηση αυτής της διπλωματικής εργασίας.

Επίσης, οφείλω ένα ευχαριστώ στην ομάδα του Synnefo Software και ιδιαίτερα στους Κωνσταντίνο Βενετσανόπουλο, Αλέξη Πυργιώτη και Στράτο Ψωμαδάκη για τη συμβολή τους τόσο μέσω των συζητήσεων που είχα μαζί τους όσο και μέσω της τεχνικής υποστήριξης που μου προσέφεραν.

Τέλος, ευχαριστώ την οικογένεια και τους φίλους μου για την αγάπη τους και την στήριξή τους σε όλη την διάρκεια της φοιτητικής μου διαδρομής.

Ιωάννης Χατζημίχος,

Αθήνα, 23η Μαρτίου 2015

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The purpose of this thesis is to investigate existing solutions and propose a new design for providing an object storage service over a block storage architecture, according to the principles that we define in Chapter 6.

Object storage provides the manipulation of underlying storage in terms of addressable units called objects. Unlike file storage, there is no hierarchy of objects and each object is addressed by its unique identifier, called the object's *key*, in a flat address space. Object storage systems are less structured compared to file systems, which leads to easier data distribution as there are little to no dependencies between distinct objects. Therefore, distributed object storage systems are a popular choice for storing massive amounts of data.

Block storage, on the other hand, is a storage architecture which manages data as named linear spaces (”volumes”) of numbered blocks. It is prominent in Storage Area Networks (SANs), where storage arrays are connected with servers and workstations through the use of Fibre Channel technology. That way, block volumes are visible to the nodes as locally-attached storage.

Although block storage is the de facto type of architecture at the device level, it is not convenient for applications to manage their data this way. Thus, it is common to implement some other type of storage architecture at the system and interface levels which exposes more convenient semantics. For example, filesystems that are typically implemented at the system level expose file semantics while essentially implementing a mapping from file storage to block storage. The pros and cons of the different types of storage technology are explained more thoroughly in Section 2.3.

To this end, we designed and implemented an architecture for object storage over an underlying block storage architecture such as a Storage Area Network. SANs consist of reliable and fault-tolerant hardware which are well-tested in production. They are actively supported by vendors and play a significant role in the infrastructure of many companies. Note, however, that such a topology has the disadvantage of being hard to scale, since it is not distributed but rather based on an expensive SAN infrastructure.

Due to the reasons stated above we believe that is useful to examine solutions for deploying object storage architectures over an existing Storage Area Network, targeting companies and individuals who already possess such infrastructure.

1.1 Thesis motivation and background

The motivation behind this thesis emerged from the need to deploy the Synnefo [1] cloud software over a Storage Area Network. Synnefo powers the ~**okeanos** [2] public cloud service [8]. We will briefly explain what ~**okeanos** and Synnefo are in the following paragraphs.

~**okeanos** is an IaaS (Infrastructure as a Service) that provides Virtual Machines, Virtual Networks and Storage services to the Greek Academic and Research community. It is an open-source service that has been running on production servers since 2011 by GRNET S.A. [3]

Synnefo [7] is the cloud software stack, also created by GRNET S.A., that implements the following services which are used by ~**okeanos** :

- *Compute Service*, which is the service that enables the creation and management of Virtual Machines.
- *Network Service*, which is the service that provides network management, creation and transparent support of various network configurations.
- *Storage Service*, which is the service responsible for provisioning the VM volumes and storing user data.
- *Image Service*, which is the service that handles the customization and the deployment of OS images.
- *Identity Service*, which is the service that is responsible for user authentication and management, as well as for managing the various quota and projects of the users.

This thesis will deal exclusively with the Storage Service of Synnefo. More specifically we will dive into the core internal part that handles the VMs' data as stored inside virtual volumes, which is called Archipelago and is presented in Chapter 3.

The Storage Service has dual responsibility: except for providing virtual block devices for the Virtual Machines, it can be used as a simple Object Storage Service by exposing an API compatible with OpenStack Swift. Accordingly, we aim to design a system that can support both interfaces.

1.2 Existing solutions and limitations

RADOS is a distributed object storage system that can provide an efficient solution to our problem, since it provides both block storage and object storage semantics. Interacting with the system by storing and retrieving objects directly is done via the librados library. Block-based storage interfaces are also provided via virtual raw block devices.

[1] www.synnefo.org/
[2] https://okeanos.grnet.gr/
[3] Greek Research and Technology Network, https://www.grnet.gr/

However, RADOS is distributed in nature meaning that it is meant to be deployed on a cluster of servers with commodity hardware and local storage. It does not directly support a block storage backend.

The only way to deploy RADOS over a Storage Area Network is to split the storage array into LUNs, one for each node. This setup prompts a dilemma: if we choose not to use the RADOS object replication mechanism for redundancy, then we lose fault tolerance since when a node goes down no other node is meant to access its data even though it has access to the same storage pool. Conversely, if we choose to enable replication then we keep the fault-tolerance property but we decrease our storage capacity substantially (to 50% or less).

It becomes apparent that using a distributed system with a non-distributed storage architecture does not make much sense. In accordance, we explore solutions that consider the SAN as a single logical block-addressable space which can be accessed and modified by each node in its whole.

1.3 Thesis structure

The thesis is organized as follows:

Chapter 2:
We provide the necessary theoretical background for the concepts and entities that are being discussed throughout the thesis.

Chapter 3:
We present the architecture of Archipelago and explain how Archipelago handles I/O requests. Moreover, we provide information about RADOS, one of the storage backends of Archipelago, as well as sosd, an Archipelago component that has been created to communicate with RADOS and has been the subject of a previous CSLab thesis [16].

Chapter 4:
We introduce OCFS2, a shared-disk filesystem that is generally mounted by many nodes simultaneously on top of shared block storage. We dive into its internals, focusing on its Distributed Lock Manager and its caching mechanisms.

Chapter 5:
We present synapsed, a secondary component of Archipelago, whose purpose is to transfer Archipelago requests over the network effectively allowing the off-load of device I/O requests to be executed in other nodes than the host.

Chapter 6
This chapter provides a comprehensive explanation of our design proposal. We explain step-by-step the challenges we faced and the compromises we made to reach the final design.

Chapter 7:
We describe the steps we took in implementing the design that we presented in the previous chapter. The first part describes how we improved the performance of OCFS2 and the second describes our modifications in the synapsed codebase.

Chapter 8

We evaluate the performance of our design and implementation. We also run the tests on the existing RADOS solution that we described previously and comment on how the two compare.

Chapter 9:

We provide some concluding remarks about our thesis and assess in what extend has it managed to achieve the goals that were set. Finally, we discuss some paths for future work that were out of the scope of this thesis.

Chapter 2

Theoretical Background

2.1 Block Devices

One of the primary needs for computers is the ability to persistently store and retrieve data. To achieve this, they use a variety of storage devices that are attached to them, like hard disks, USB flash drives and optical disks. Contrary to RAM access, these storage media do not allow access to individual bytes. Instead they are addressed by whole chunks of bytes, called **physical blocks**. The size of the physical block depends on the hardware geometry. Common block sizes for hard drives are 512 and 4096 bytes. However, the kernel uses a different unit for read/write operations whose size may or may not be equal to the physical block size. This unit is called the **logical block** of the device.

Different types of storage devices require different device drivers to communicate. The operating systems abstracts the low-level details of the drivers by providing a unified way to access all the storage media that are addressable in units of blocks. This abstraction is called a **block device** or a **block special file**.

Most operating systems have similar mechanisms for abstracting access to devices with special files, however we will focus on the naming conventions of Unix-like operating systems. Device files for on Unix-like systems are placed in the `/dev` directory. The name of block devices corresponding to hard drives has the prefix `hd` or `sd` (depending on whether it is an IDE or a SCSI device), followed by a letter of the latin alphabet. If the hard drive has many partitions, a separate device file will be created for each partition with a number at the end of the name that identifies the partition. For example, the file `/dev/hdb` would correspond to the second IDE driver, while the file `/dev/sdc4` would correspond to the fourth partition of the third SCSI driver.

Since performing I/O on a disk is much slower than performing it on main memory, Linux caches disk blocks in the **page cache**. To bypass the buffer cache Linux offers support for **direct I/O**, a mechanism whereby reads and writes go directly to the storage medium. An application invokes direct I/O by setting the `O_DIRECT` flag in the `open` system call. Direct I/O performs Direct Memory Access (DMA) to the storage device, so the length of the I/O request must be a multiple of the logical block size. Except for the request length, DMA also requires the memory that is written to and read from to be aligned to the logical block size.

2.2 File Systems

A file system is a part of the operating system that controls how data is stored and retrieved. File systems make storage friendlier by allowing users to store data into files. Files are organized into directories. A directory is a collection of files and other directories. Since directories can contain other directories, a tree-like hierarchy is formed where each file can be addressed by the concatenation of its name with the names of the directories of its path in the tree.

2.2.1 The Unix directory tree

In Unix-like operating systems the directory tree is global in the sense that it is shared by all the file systems. To make a file system appear on the tree, one must first inform the file system about the type of the file system, the block device on which it resides and the place in the directory tree under which the file system's data will appear. This process is called *mounting* and is usually done with the `mount` command-line tool. Most file systems allow the user to enable and disable some of their features when mounting them by adding certain flags to the `mount` command.

The file system that is mounted on the root node of the directory tree is called the *root file system* and it generally follows the structure specified in the *Filesystem Hierarchy Standard* [10]. There may exist small deviations from the standard, depending on the operating system or even the distribution of it.

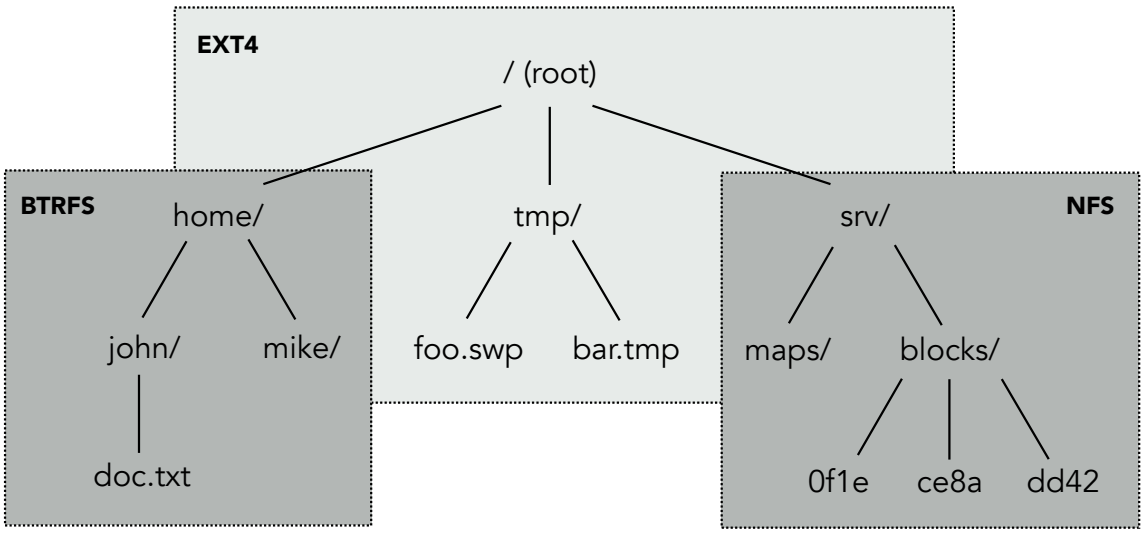

Figure 2.1: An example snapshot of a directory tree

Mount point	File system type
`/`	ext4
`/home`	btrfs
`/srv`	nfs

Table 2.1: The mount table of the directory tree shown in Fig. 2.1

2.2.2 File system data structures

Except for the actual file contents, file systems also store several file metadata:

- The size of the file in bytes.
- The ID of the device
- The User ID of the file's owner.
- The Group ID of the file.
- The file mode, which is a bitset that determines the file type and how the file's owner, its group, and others can access the file. The file has read, write and execute permissions.
- Additional system and user flags to further protect the file (limit its use and modification).
- Timestamps telling when the metadata structure itself was last modified (`ctime`), the file content last modified (`mtime`), and last accessed (`atime`).
- A link count telling how many hard links point to the inode.

These attributes are stored in the file's *inode*, a data structure used to represent a filesystem object. Inodes are important for another reason: they contain pointers to the disk blocks where the file's actual data are stored. The structure of those pointers depends on the type of the file system. The inodes are stored in a flat array in the disk that is generally pre-allocated during the creation of the file system. Inodes are addressed by their index in this array, called the *inumber* or the *inode number*.

To keep track of its contents each directory consists of a list of dentries, where a *dentry* is a structure that maps an inode number to a file name. Note that the file name is not stored in the inode. The reason for this is that a file can have multiple names in the same or in different paths of the directory tree.

In order to minimize disk I/O requests, the kernel caches commonly used inodes and dentries in the *inode cache* and *dentry cache* respectively.

2.2.3 Shared-disk File Systems

With certain topologies, we can have many nodes with parallel shared access on the same permanent storage devices. Here are some examples of network types that achieve shared storage:

- FireWire [1], a standard for a serial bus for real-time data transfer. FireWire is a peer-to-peer network, allowing multiple devices to be connected to a single bus.
- The iSCSI Protocol, a mapping of SCSI over TCP/IP. This technology is part of our test-bed and is described in detail in Section 2.4.

- Fibre Channel Protocol, a mapping of SCSI over Fibre Channel. This is the most prominent method for setting up a Storage Area Network to provide shared storage.
- Fibre Channel over Ethernet (FCoE), a computer technology that encapsulates Fibre Channel frames over Ethernet networks.

There are quite a few reasons why a typical local-storage filesystem cannot be used with a shared storage backend:

1. Simple filesystems rely heavily on the inode and dentry caches to improve performance. Since the filesystem data structures can't be modified externally, the filesystem code is certain that the contents of the cache are always valid. Therefore, a write-back policy is used.
2. Likewise, local-storage filesystems heavily rely on the operating system's page cache and only periodically commit file data to permanent storage.
3. Even if the filesystem didn't use the caches at all, there is also the problem of concurrency control. If no synchronization takes place between the filesystem instances, if different nodes try to perform operations on the same files at the same time they will cause corruption on the data.

Therefore, to be able to provide safe file-level operations over shared block storage we need a special filesystem that deals with the problems mentioned above. A *shared-disk filesystem* uses a distributed lock manager to add mechanisms for concurrency control on filesystem resources and maintain cache coherency. Thus, it provides a consistent and serializable view of the filesystem from multiple operating system instances, avoiding corruption and unintended data loss.

It is a common practice for shared-disk filesystems to employ some sort of a fencing mechanism to prevent data corruption in case of node failures, because an unfenced device can cause data corruption if it loses communication with its sister nodes, and tries to access the same information other nodes are accessing.

This thesis deals extensively with the OCFS2, one of the best open-source shared-disk file systems. A detailed explanation of the internals of OCFS2 is given in Chapter 4.

2.2.4 The Network File System Protocol

NFS is a distributed filesystem protocol that allows users to access files that are stored on a remote server. It provides remote access to filesystems across networks while being machine, operating system, network architecture independent [2].

In a typical scenario, a server exports one or more of its filesystems to one or more clients (typically using the `/etc/exports` configuration file and the `exportfs` command). Afterwards, each client can access the exported filesystems it is interested in and use them as if they were local, by mounting them on its local tree. The client does not need to know the type of the remote filesystem it wants to mount. It rather reports the type of the mounted filesystem as `nfs`.

To avoid confusion, we stress that NFS itself is not a filesystem. It is merely the mount protocol that allows clients to attach remote filesystems to the local directory tree. Since the clients are not aware of the type of the remote filesystem, any operation to its files essentially issues network requests on the server.

Not all types of file systems can be exported. To be able to be exported, a file system must support and implement NFS exports. As en example, in Chapter 4 we describe how the OCFS2 parallel filesystem added support for NFS exports.

Finally, notice that the data in an NFS architecture are stored locally on a single machine (the server). The rest of the machines (clients) can only access the data through the network. In other words, the NFS server is a single point of failure.

2.3 Types of storage technology

There are three main different types of storage technology, depending on what is exposed by the storage architecture.

2.3.1 File Storage

File storage is a storage architecture that stores data as files. The files are organized into directories, where each directory may contain a number of files as well as other sub-directories. This form of data organization creates the tree hierarchy that we described in Section 2.2. Regarding the file operations, file systems try to conform to the semantics defined in the POSIX Standards [6].

File storage is not limited to locally attached storage media, like hard drives. A file server can be used to share files with clients over the network. This topology is called **Network-Attached Storage**.

A well-known example of such an architecture is the use of an NFS server (see Section 2.2.4) to provide file access to clients via a network protocol. Other known network file sharing protocols include:

- The Server Message Block / Common Internet File System Protocol (SMB/CIFS), which is prominent in Windows systems.
- The Apple Filing Protocol (AFP), formerly AppleTalk Filing Protocol, which is a proprietary protocol mostly used in the Macintosh world.

File storage over the network is simple in terms of usability and implementation but provides limited scalability and reliability. The NFS server is a single point of failure. It turns out that keeping a tree structure consistent over the network, while conforming to POSIX semantics for file operations, is a hard problem. Accordingly, modern distributed systems have turned to other types of storage technology for storing massive amounts of data.

2.3.2 Block storage

Most storage media provide block-level abstractions. Their storage space is exposed to the operating system as a large array of blocks where each block is addressed by its index in the array. Each block is a sequence of bytes and operations can only be performed on whole chunks of blocks (see also 2.1). Although block-level storage is the most common storage abstraction for interacting directly with hardware, it is not a very convenient way for storing data by application. Therefore, higher level abstractions — like file storage and object storage — are usually implemented on top of block storage.

Block Storage is prominent in Storage Area Networks (SANs). In a SAN, storage arrays are connected with servers and workstations through the use of fibre channel technology. That way, block volumes (LUNs [1]) are visible to the nodes as locally attached storage.

Contrary to Network Attached Storage (NAS), Storage Area Networks do not provide file-level abstractions. The nodes operate on storage media by issuing SCSI commands which are then sent over Fibre Channel to the actual devices.

A relevant technology is iSCSI which is described in Section 2.4.

2.3.3 Object Storage

Object storage provides the manipulation of underlying storage in terms of addressable units, called objects. Unlike file storage, there is no hierarchy of objects and each object is addressed by its unique identifier, called the object's *key*, in a flat address space.

Although object-based storage implementations are generally realized on the system level or on the interface level, it can also be implemented on the device level with Object Storage Devices (OSDs) [12].

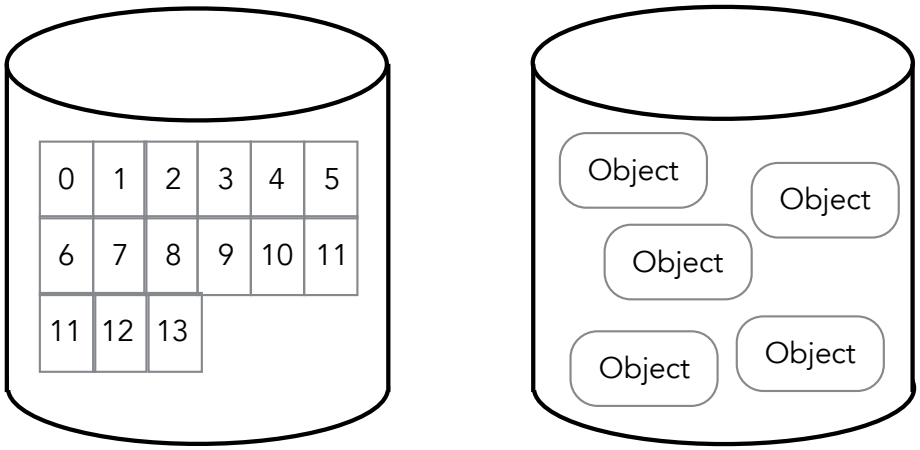

Figure 2.2: Representation of block-based storage and object-based storage on the device level

[1] Logical Unit Numbers

In the following paragraph we present the base concepts behind widely used object storage systems.

Amazon S3

Amazon Simple Storage Service [2] (Amazon S3) is a highly-scalable object storage service by Amazon that is reported to store more than 2 trillion objects as of April 2013. Since Amazon S3 is used by millions of companies, we believe it represents the state of the art in scalable object storage implementations and we will briefly describe its features.

In S3, objects are stored in a one-level hierarchy of buckets. Buckets cannot be nested (each bucket can only hold objects, not buckets) and are used as a way of simplifying administration rather than as a means to enforce structure on objects. Specifically, administrators can select different ACLs, pricing models as well as choose the geographical region where the objects will be stored depending on the bucket. Each object is uniquely identified by its bucket name and its key.

Once can also mimic a hierarchical structure on the objects by having the object keys contain the full the path in the tree. As an example, an object called `history.txt` in the `logs` folder will have the key `logs/history.txt`. The service supports this concept natively to an extent (for example, renaming folders is not possible).

Except for the key and its data, an object can also have a number of metadata. There are two types of metadata: the service metadata supplied by S3 (`last-modified`, `content-length`...) and arbitrary metadata supplied by the user.

The service provides a REST API, so interacting with it is done via standard HTTP requests. The most important operations provided by the service are:

- Creating, listing, retrieving and deleting buckets
- Uploading, retrieving and deleting an object. Modifying parts of an existing object is not possible.
- Editing the object permissions and metadata.
- Listing all objects with the same prefix. This operation is particularly useful when objects are named with their full path as described above.

Renaming an object is generally not supported but can be achieved by copying the object and deleting the original.

2.4 The SCSI and iSCSI Protocols

The *Small Computer Systems Interface (SCSI)* is a family of protocols for communicating with storage devices (most commonly hard disks and tape drivers). It is a client-server architecture, where peripheral devices act as servers, called *targets*. The clients of a SCSI interface, called *initiators*, issue SCSI commands to targets to request

[2] http://aws.amazon.com/s3/

The **iSCSI protocol** [11] allows two hosts to negotiate and transport SCSI packets over an IP network and thus enables accessing I/O devices over long distances using the existing network infrastructure. It uses the TCP protocol to transmit data. Implementations of iSCSI targets and initiators exist in software as well as hardware. Dedicated hardware, which takes the form of a host bus adapter (HBA) in the case of the initiator and a storage array in the case of a target, improve the performance by mitigating the overhead of Ethernet and TCP processing.

2.4.1 Discovery and Login Phase

The purpose of iSCSI discovery is to allow the initiators to find the targets and storage resources that they have access to. This can be done either with static configuration when the initiator knows the IP address and TCP port of the target, or with zero configuration where the initiator sends multicast discovery messages. After finding the appropriate targets, the initiators authenticate their identity using the *Challenge-Handshake Authentication Protocol (CHAP)*.

2.4.2 Full Feature Phase

Once the initiator successfully logs in, the iSCSI session is in Full Feature Phase. Although an iSCSI session may consist of a group of TCP connections, completing the Login Phase on the first (leading) connection is sufficient to set the whole session in Full Feature Phase. In this phase, the initiator may send SCSI commands to the LUNs on the target through the iSCSI session. For any iSCSI request issued over a TCP connection, the reply must be sent over the same connection.

2.4.3 Termination

An initiator can issue a logout request to either remove a connection from the session or close the entire session. When the target receives the request, it issues the response half-closes the the TCP stream by sending a FIN. The initiator, then, receives the response and closes the other end.

2.4.4 Linux Support

There are three main implementations of the iSCSI target for Linux servers:

- Linux SCSI target framework (tgt)[3]
- The iSCSI Enterprise Target (iscsitarget)[4]

[3] http://stgt.sourceforge.net/
[4] http://iscsitarget.sourceforge.net/

- Linux-IO (LIO) Target[5], which is the standard since Linux 3.1 - when the iSCSI target fabric was merged in the kernel. The management of the Linux-IO Target is done with the `targetcli` shell.

On the client side the open-ISCSI project[6] is the preferred open-source solution to get an iSCSI Software Initiator in a Linux machine.

[5] http://linux-iscsi.org/
[6] http://www.open-iscsi.org/

Chapter 3

Archipelago

Archipelago is a distributed storage layer that is part of the Synnefo cloud software, serving the purpose of decoupling Volume and File operations from the underlying storage [16]. Archipelago is responsible for the creation and management of the virtual machines' volumes, as well as for carrying out the write/read requests of the volumes by interacting directly with the underlying storage technology. Apart from decoupling storage logic from the actual data store, Archipelago aims to solve problems that arise on large scale cloud environments by supporting data deduplication, thin cloning and snapshotting.

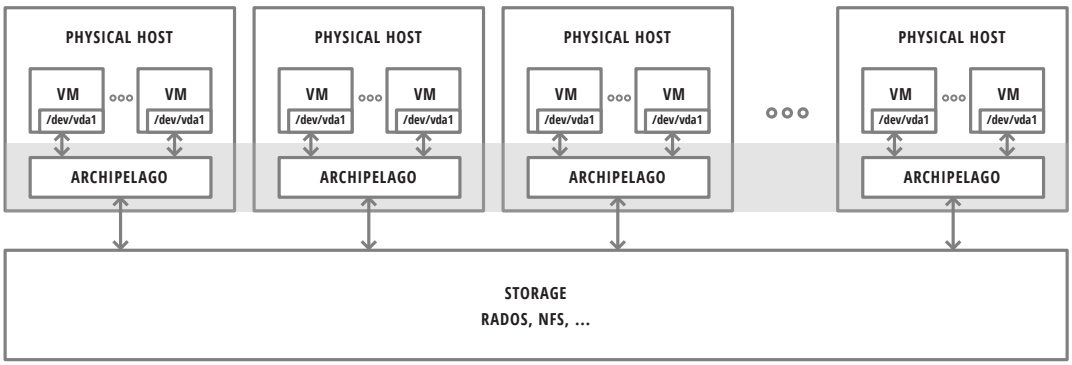

Figure 3.1: Archipelago overview

3.1 Archipelago topology

Archipelago is divided into three layers: the northbound interface, the Archipelago core and the southbound interface.

The *northbound interface* is a collection of endpoints (called *northbound drivers*) that provide access to the Archipelago volumes and objects for the user or upper layers to interact. Currently, these are:

- the block device driver
- the QEMU driver

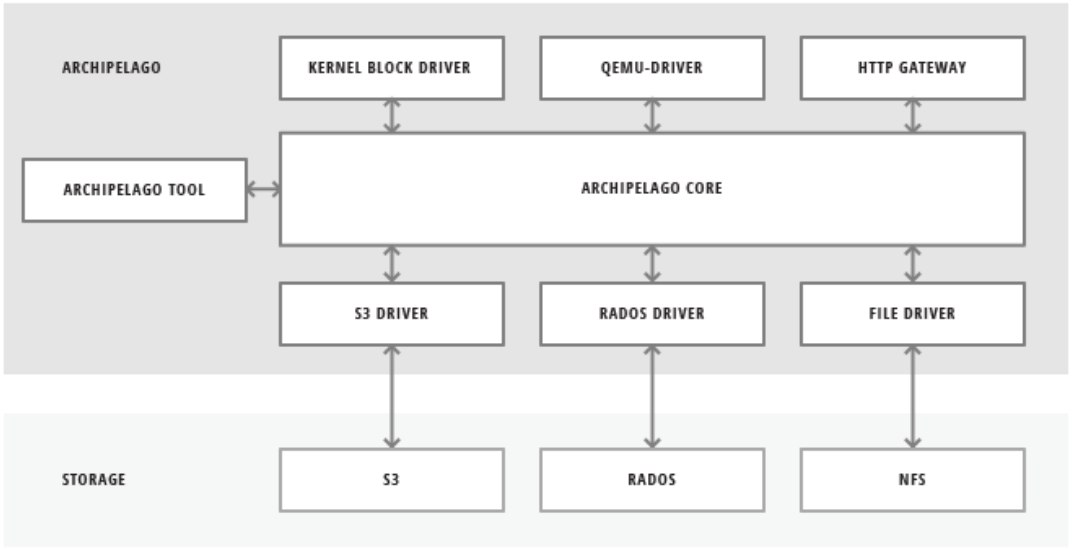

Figure 3.2: Archipelago topology

- HTTP gateway for files

The *core* stands in the middle in the Archipelago topology and implements the actual composition, mapping and deduplication logic.

On the backend, Archipelago employs its *southbound drivers* to communicate with the underlying technology. Archipelago currently has two drivers on the southbound interface, each for a different type of storage backend.

- The shared file driver is used when the underlying storage technology exposes file semantics. This driver is generally used when a distributed file system is present, like NFS.
- The RADOS driver is used by backends that expose object semantics via a Ceph/RADOS cluster. It uses librados to communicate with the cluster.

Currently there is no backend driver for interacting with storage systems that lack any of the previous high-level semantics, exposing only a physical device that provides raw block-level operations.

3.2 Internal architecture

Archipelago consists of several components, both user-space and kernel-space, that communicate via a custom-built shared memory segment mechanism called XSEG. Thus, each Archipelago component is said to be an *XSEG peer*. The components are described below:

The Volume composer daemon (vlmcd)

The volume composer receives I/O requests to a volume, translates them to object requests and then sends those requests to the blocker. The translation of the requests is done with the help of the mapper.

The Block devices (blktap)

Each of the Archipelago's volumes is esentially a peer that is exposed to the userspace as a block device in the `/dev/xen/blktap-2/` directory. I/O requests that are performed on those special device files are retrieved by the corresponding peer who then directs them to the volume composer for completion.

The Mapper daemon (mapperd)

The mapper is responsible for keeping the mapping between the volume blocks and the objects that are stored on the storage backend. It is also responsible for creating new volumes, cloning and snapshotting. The mappings themselves are stored inside objects on the storage backend so the mapper communicates with the blocker to perform I/O requests on them.

The Blocker daemons (blockerd)

The blockers constitute the southbound interface that we described in Section 3.1. In other words, blockerd is not a specific entity but a family of peers, each of which can communicate with a specific underlying storage type. Every Archipelago instance normally has two blocker daemons: one communicates with the mapper to serve the map objects while the other serves block objects by responding to the volume composer's requests.

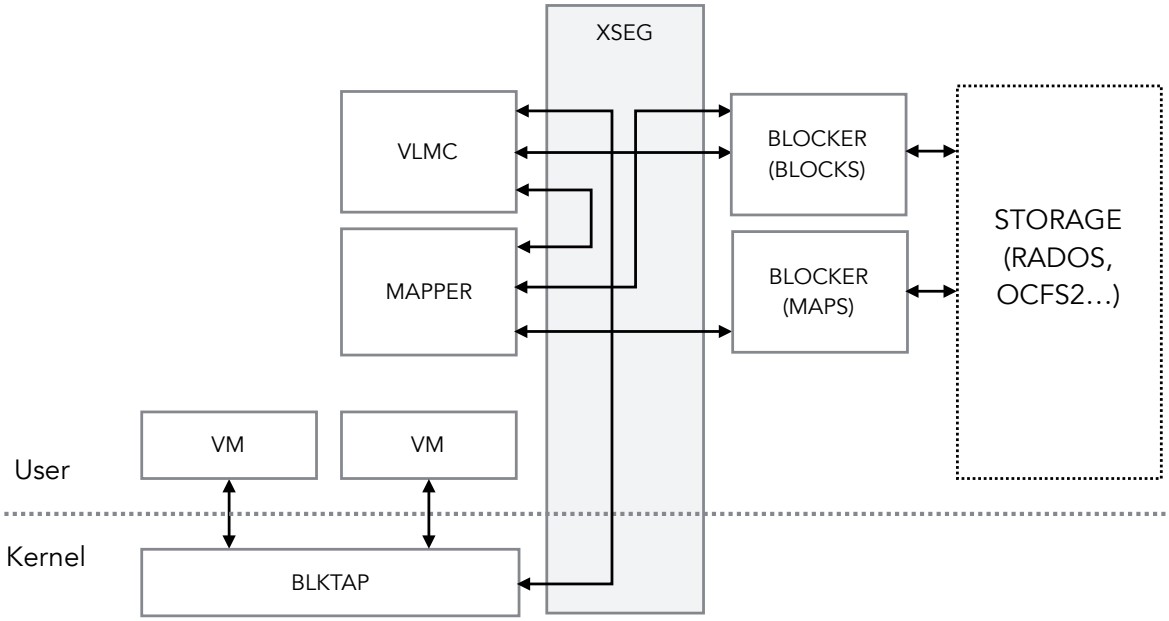

Figure 3.3: Interaction between the internal components of Archipelago

3.3 The RADOS peer

Archipelago is not restricted to a storage backend and can work with any backend for which a blocker can be created. There is one backend however that provides many important features that Archipelago exploits for increased scalability and reliability.

This backend is RADOS [14], which is the object store component of the Ceph[1] system. Ceph is a free distributed object store and file system that has been created by Sage Weil for his doctoral dissertation [13] and has been supported by his company, Inktank, ever since.

The architecture of Ceph can be seen in Figure 3.4. [2]

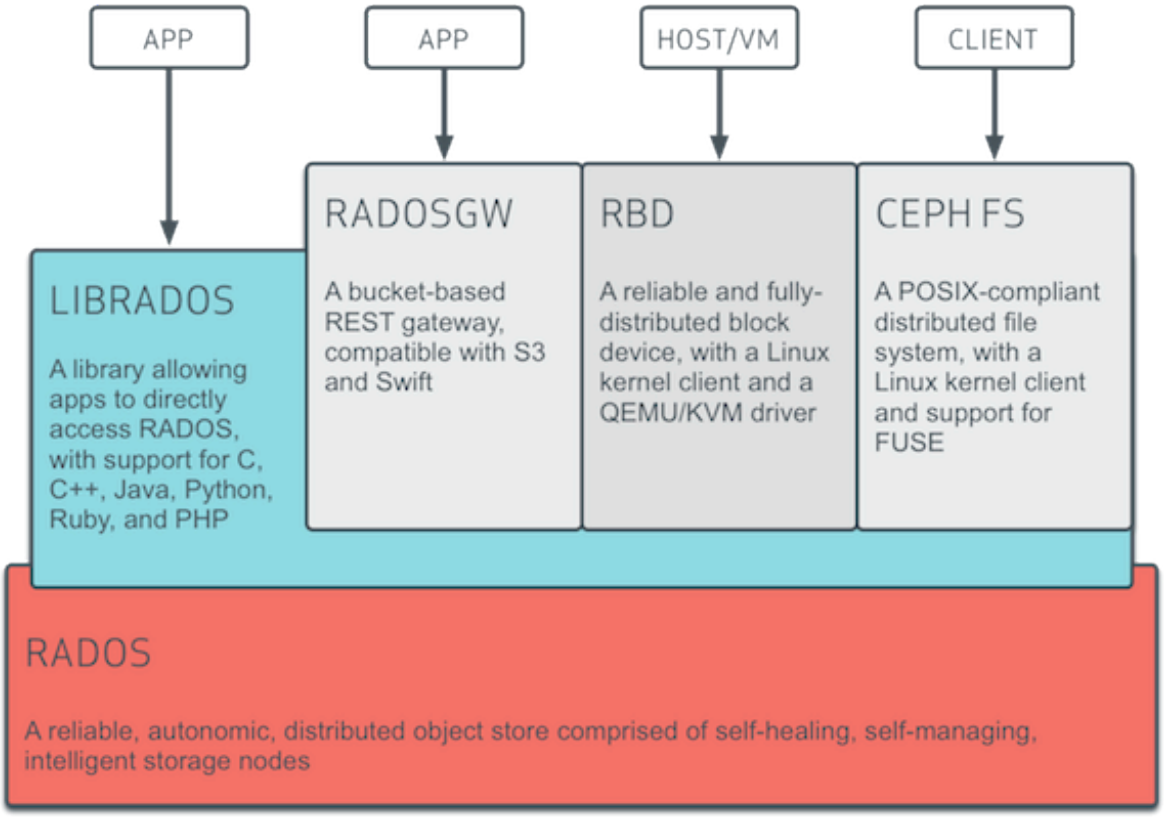

Figure 3.4: Ceph entry points

Ceph utilizes RADOS to achieve the following:

- *Replication*, which means that there can be many copies of the same object so that the object is always accessible, even when a node experiences a failure.
- *Fault tolerance*, which is achieved by not having a single point of failure. Instead, RADOS uses elected servers called **monitors**, each of which have mappings of the storage nodes where the objects and their replicas are stored.
- *Self-management*, which is possible since monitors know at any time the status of the storage nodes and, for example, can command to create new object replicas if a node experiences a failure.

[1] ceph.com
[2] Picture retrieved from the official website on September 24, 2013. All rights go to the respective owners.

- *Scalability*, which is aided by the fact that there is no point of failure, which means that adding new nodes theoretically does not add any communication overhead.

In a nutshell, RADOS consists of the following components:

- *object storage daemons*, which are userspace processes that run in the storage backend and are responsible for storing the data.
- *monitor daemons*, which are monitoring userspace processes that run in an odd number of servers and form a Paxos part-time parliament[9]. Their main responsibility is holding and reliably updating the mapping of objects to object storage daemons, as well as self-healing when an object storage daemon or monitor daemon has crashed.

The Ceph Storage Cluster has a messaging layer protocol that enables clients to interact both with monitors and storage nodes, called **OSD daemons**. This messaging functionality is provided in the form of a library, librados, which has bindings for C, C++ and Python. Archipelago uses a blocker called sosd [16] for I/O requests which has been created to facilitate the communication between RADOS and Archipelago. The sosd blocker uses the librados API.

3.4 Communication between peers

As mentioned above, the archipelago peers use IPC[3] to send and receive requests on a shared-memory segment called XSEG. This segment provides custom ports where peers can bind them and listen for requests in a fashion similar to network ports. A peer can either bind a single port or a range of ports. The ports are not addressed in any way other than their number. Thus, when a peer wants to send a request it must know the destination peer's port beforehand.

Each custom port on XSEG has two queues where a peer listens for requests:

1. a **request queue** where the peer accepts new requests.
2. a **reply queue** where the peer gets the replies to the requests that it sent to another peer previously.

When a Peer A wants to send a request to a Peer B, it enqueues it to Peer B's request queue. After processing the request, Peer B enqueues the reply to Peer A's reply queue.

After binding itself to a port, the peer will sleep and wait for notifications about new received requests via signals (specifically `SIGIO`). A signal wakes up the peer and runs its signal handler. The default signal handler checks all ports, accepts any pending requests and then serves them. After serving all the requests the peer does not sleep immediately, because context switching is expensive. Instead, it keeps checking for requests for a few more cycles.

[3] Interprocess Communication

This behavior can be altered by using a custom signal handler, although commonly there is no need for this.

The structure of an XSEG request is shown in Listing 3.1. It contains the port numbers, pointers to the `data` and `target` buffers along with their length, the `size` in bytes that the peer is requested to fill, the number of `serviced` bytes. The rest of the fields are irrelevant to our purpose. The buffers are also shared, so that both peers can access them.

```
struct xseg_request {
    xserial serial;
    uint64_t offset;
    uint64_t size;
    uint64_t serviced;
    uint64_t v0_size;
    xptr data;
    uint64_t datalen;
    xptr target;
    uint32_t targetlen;
    uint32_t op;
    volatile uint32_t state;
    uint32_t flags;
    xport src_portno;
    xport transit_portno;
    xport dst_portno;
    xport effective_dst_portno;
    struct xq path;
    xqindex path_bufs[MAX_PATH_LEN];
    /* pad */
    xptr buffer;
    uint64_t bufferlen;
    xqindex task;
    uint64_t priv;
    struct timeval timestamp;
    uint64_t elapsed;
};
```

Listing 3.1: The structure of XSEG requests

In addition, XSEG also allows a peer to attach some private data to a request. These data must be provided in the form of a generic pointer.

Chapter 4

OCFS2

The Oracle Cluster File System [5] is an open-source shared disk filesystem developed by Oracle Corporation and integrated in the Linux Kernel. It uses its own distributed lock manager which is based on the OpenDLM. A separate suite of userspace tools, called *OCFS2 Tools*, is developed to aid in the management and debug process of the OCFS2 filesystem. Part of this suite are the programs *mkfs.ocfs2*, *mount.ocfs2*, *tunefs.ocfs2* and *debug.ocfs2*.

4.1 The Distributed Lock Manager

A lockable entity in the lock manager is called a **lock resource** [3]. It can correspond to an object like a file in the filesystem, a dentry or a process that requires synchronization like mounting/unmounting and orphan scanning. Section 4.3 provides a full list of the different types of lock resources that exist on OCFS2.

In the OCFS2 DLM, a lock resource consists mainly of:

- The *Resource Name*, which is a 32 byte string divided in four parts as depicted in Figure 4.1. As we will explain later, some type of locks, like cluster-wide locks, may not use the block and/or the generation number.

Figure 4.1: The lock resource name analyzed in its parts

- The *Lock Value Block (LVB)* which is a small block of data associated with the lock resource. The type of data stored depends on the lock type. As an example, metadata locks use the LVB as a cache for the inode's metadata. Most of the lock types do not use the LVB at all.

- The *Generation Number* which is a number that is incremented every time an inode number is reused. By incrementing it whenever an inode number is reused, the server ensures that a client with an old file handle can't accidentally access the newly-allocated file. Referencing the inode just with its number is not enough, so filesystem uses the tuple (inode number, generation number) to uniquely refer to an inode at any time. This generation number is also part of the lock resource name for the same reasons.

Each lock resource also has three queues associated with it:

Grant Queue
The grant queue holds all the locks that are currently granted.

Convert Queue
The convert queue contains all the locks that are currently granted but have attempted to convert to a mode that is currently incompatible with the mode of some other currently granted lock.

Wait Queue
The wait queue holds all the new lock requests that are not yet granded because they are incompatible with some other lock that is already granted.

4.2 Lock Modes

There are only three lock modes (also called *lock levels*) used by the Distributed Lock Manager of OCFS2:

- Exclusive lock (EX): The requesting node has full write and read access to the lock resource, while preventing any other nodes from accessing it.
- Protected Read (PR): The nodes in this mode can read the lock resource concurrently, but no node is allowed to write.
- No Lock (NL): The node has no access to the resource.

Each lock has a lock mode which must be compatible with the lock mode of all the other locks on the same resource (Table 4.1).

Name	Access Type	Compatible Modes		
		NLMODE	PRMODE	EXMODE
NLMODE	No Lock	✓	✓	✓
PRMODE	Read Only	✓	✓	
EXMODE	Exclusive	✓		

Table 4.1: Compatibility of lock modes in OCFS2

A node can request the upconvert or the downconvert of the mode of a lock that it already posseses. Upconverting asks the DLM to update the mode of the lock to level greater than

the current one. It is granted if there are no other locks with incompatible modes, otherwise it fails. Downconverting is the opposite procedure of the upconvert and is done when the currently granted mode on a lock is incompatible with the mode another process requests on the lock.

4.3 Lock Types

Lock Type	Identifier
Superblock Lock	S
Open Lock	O
Orphan Scan Lock	P
Read/Write Lock	W
Meta Lock	M
Dentry Lock	N
Flock Lock	F
Rename Lock	R
Data Lock	D
Refcount Lock	T
NFS Sync Lock	Y
Qinfo Lock	Q

Table 4.2: The full list of OCFS2 locks and their identifying characters

4.3.1 Superblock Lock

One of the fields that the superblock contains is the slot map, which holds which slot was chosen by each node. Consequenctly, the node holds the superblock lock to choose an appropriate slot in a race-free manner during volume mounting and unmounting.

The lock is also used during filesystem recovery to guarantee that no node tries to mount the volume during the recovery process and that only one node attempts to recover the dead node. What actually happens is that the first node that gets the superblock lock will look in the slot map to determine which journal the dead node was using. After finding the journal, it replays it, marks it clean and then the dead node is taken out of the slot map. Now the rest of the nodes will take the superblock lock one by one but they will not find the dead node's number in the slot map, so they will assume that the recovery was completed by another node.

4.3.2 Open Lock

This lock resource is acquired in protected-read mode when a file opens and dropped as soon as the file closes. When a node wants to delete a file, it first tries to convert its lock to an exclusive. If it succeeds, then it can safely delete the file as it is not open in any other node. If the file is being used, the node makes it an orphan by marking it with a `OCFS2_ORPHANED_FL` flag and moving it to the orphan directory.

4.3.3 Orphan Scan Lock

The orphan scan is a procedure in OCFS2 that scans the orphan directory to check for files not being accessed by any of the nodes in order to delete them permanently. It runs every five minutes in a dedicated kernel thread. It may be run from any node, in fact all nodes will eventually try to run the orphan scan. To prevent congestion, OCFS2 employs the following techniques:

- A random value of up to five seconds is added to the initial five minute timeout of the orphan scan timer, to minimize multiple nodes firing the timer at the same time.
- When the node gets the orphan scan lock it checks the sequence number that is stored in the LVB. If the sequence number has changed it means that some other node has performed the scan. This node skips the scan and tracks the sequence number. If the number didn't change, it means a scan hasn't happened. The node queues a scan and increments the sequence number in the LVB.

4.3.4 Read/Write Lock

The read/write lock is a very expensive lock, since it is generally acquired before each write operation. Some of the cases that it is used are:

- To prevent Asynchronous I/O and Direct I/O while some node is truncating the file.
- To serialize intra-node Direct I/O requests. OCFS2 has the `coherency` mount flag which allows the administrators to control the behavior in this case. More specifically, the write lock is requested in Protected Read mode when `coherency` is set to `buffered` and in Exclusive mode when `coherency` is set to `full`.

4.3.5 Meta Lock

Generally, an OCFS2 node never trusts an inode's contents in the inode cache, unless it holds the meta lock for this inode. For metadata reads, it takes the lock in PR mode. For metadata changes, it takes it in EX mode. It becomes evident that requiring a cluster lock for any inode access adds a big overhead to shared-disk filesystems. One technique that OCFS2 uses to reduce this cost is to store the most important inode's fields in the lock resource's LVB to avoid reading them from the disk in some cases.

The terms "meta lock" and "inode lock" are used interchangebly.

4.3.6 Dentry Lock

Whenever a node performs a lookup on a name, it reads its dentry and also puts it in the dentry cache. While the dentry is in the dentry cache, the node also holds its cluster Dentry Lock in PR mode. If a user requests an unlink or a rename, the protected read is upgraded to an exclusive lock. This way, other nodes who have the dentry in their cache will be informed that they need to downgrade their lock, which will involve releasing the dentry from the cache.

4.3.7 Flock Lock

The life of an flock lock starts when a node opens a file and ends when the file closes. Its only purpose is to support cluster-aware flock locking. This is the only type of lock that does not support lock caching: when OCFS2 receives an unlock request via the `flock()` system call, it will destroy the corresponding flock cluster lock. Disabling this feature is possible via the `localflocks` mount option.

4.3.8 Rename Lock

This is a cluster-wide lock that is only used for serializing the renamings of directories. As an example, consider the following scenario. Assume two nodes have mounted the same ocfs2 filesystem with the following directory hierarchy:

`a/b/c`

`a/d`

where a,b,c, and d are all directories. The nodes run the following commands at the same time:

node1: `mv b/c d`

node2: `mv d b/c`

Without the rename lock, this will result in a deadlock.

4.3.9 Other Lock Types

- *Data locks* are not used anymore as the meta lock now covers both meta-data and data.
- *Refcount locks* are used for synchronization by refcount trees, which is the data structure used by ocfs2 to handle reflinks. This lock type does not concert the archipelago file driver as it never creates reflinks.
- OCFS2 uses one *NFS Sync lock* per superblock to synchronize between NFS export operations (specifically `get_dentry`) and inode deletion.
- *Qinfo lock* is another type of lock that does not directly concern archipelago. Its purpose is to serialize modifications to the global quota file.

4.4 Lock caching

Lock caching is a very important property of the OCFS2 that played a big role in the design of our architecture.

In order to understand the lock caching mechanism, we need to provide a brief interview of the Distributed Lock Manager's calling model. In the OCFS2 DLM, all the locking operations are asynchronous. Status notification is done via a set of callback functions, the two

most important of which are *Asynchronous System Traps (ASTs)* and *Blocking Asynchronous System Traps (BASTs)*.

The DLM will call an AST function after a lock request has completed (whether it has succeeded or not). BAST functions are called to notify the caller that a lock it is currently holding is blocking the upconversion of the lock of another process due to lock mode incompability. When a process receives a BAST it must generally downconvert its lock as soon as possible to prevent deadlocks.

To ensure sure that it can safely downconvert a lock when needed, each node keeps local counters associated with each lock resource that store how many intra-node processes use the lock currently. Therefore, if a lock resource's counter is zero, then its lock can safely be downconverted.

However, even if the counter reaches zero for a lock resource, but there is no other competing lock waiting to be upconverted, the node will *not* release lock. This mechanism of keeping the locks even if the resources that they are protecting are not used anymore is called **lock caching**.

As an example, consider a Node A that opens a file *foo* twice (from two different processes). It will aquire an open lock in PR mode on the file and its counter will be set to two. After the two processes close the file, the counter will be set to zero, but the lock will not be released. Now, if a process on Node B wants to delete the file *foo*, it will try to acquire the open lock in EX mode. This will trigger a BAST on Node A, which will only then request a downconvert on the open lock. Finally, both nodes will receive an AST (Node A will receive it due to successful downconversion and Node B will receive it due to the acquisition of the EX mode lock).

More importantly, if two nodes perform many write operations to the same file at the same time, then the write and metadata locks are sent back-and-forth, creating a bottleneck in the Distributed Lock Manager. This is the main reason that concurrent operations on the same files (e.g., concurrent writes) are very slow and should be avoided.

Chapter 5

The synapse peer

As we have already mentioned, Archipelago consists of a number of peers that communicate by sending and receiving packets with a pre-specified format over XSEG. This architecture imposes the limitation that all peers must run on the same node, as they must all have access to the same shared block of RAM in order to communicate.

The inconvenience of this limitation became apparent for the first time while designing a caching entity for Archipelago which led to the development of a new peer, called cached [15]. The results of the performance evaluation showed that it was impractical to run cached on the same node that hosts the virtual machines, as it would drain the CPU.

To overcome this obstacle, a new peer was created called *synapsed*, which connects two arbitrary XSEG ports through the network.

5.1 Synapsed design

Synapsed is a network peer that was designed to be a channel between two peers that run on different nodes. To achieve that, one needs to run a synapsed peer on the first node that listens for requests on an XSEG port (or a range of XSEG ports) and sends them through a TCP stream to a synapsed peer running on a different node. After receiving an XSEG request from the network, the second synapsed peer will place it in a preconfigured XSEG port and eventually it will be received by the final peer. After completing the processing of the request, the reply is eventually pushed back to the peer that made the initial request following the reverse procedure through the two synapsed peers.

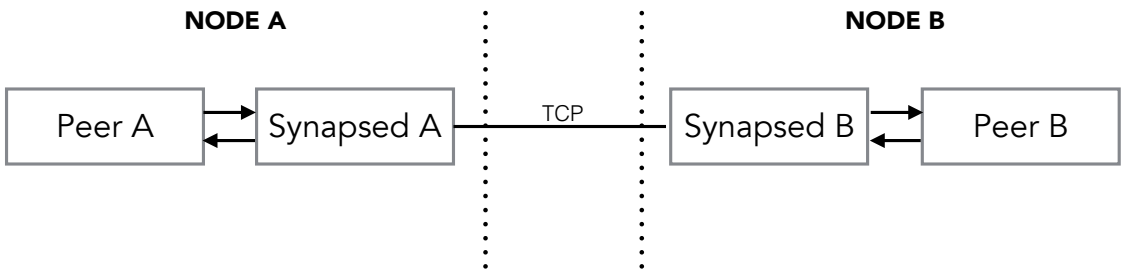

Figure 5.1: Synapsed design

Below we present the detailed steps of what happens when a Peer A that runs in a Node A wants to send a request to a Peer B that runs in another node B.

1. A *Peer* A initiates an XSEG request and puts it in the accept queue of the XSEG port that the *Synapsed A* peer listens to.

2. *Synapsed A* receives the request. Afterwards it packs it in a network packet and sends it via TCP to Synapsed B.

3. *Synapsed B* unpacks it and puts in the accept queue of a port that *Peer B* listens to. *Synapsed B* knows this port because it has it in its configuration.

4. *Peer B* receives the request, processes it and then puts the reply in the reply queue of the same port.

5. *Synapsed B* gets the reply, packs it and sends it to *Synapsed A* through their TCP session.

6. *Synapsed A* receives the reply XSEG packet from the network, unpacks it and puts it in its reply queue.

Contrary to most of the archipelago peers, synapsed has been designed as a single-threaded peer, because it never makes blocking calls.

5.2 Implementation of synapsed

5.2.1 Initialization

As is the case with every Archipelago peer, synapsed gets its configuration options as command-line arguments during initialization. Next, we describe the most important options:

- `portno_start` and `portno_end` provide the range of XSEG ports which will be used by synapsed to accept XSEG requests.
- `host_net_port` is the network port where the peer will listen for connections from remote synapsed peers.
- `remote_net_port` and `remote_net_address` are the port and address where the peer will connect to forward the XSEG packets.
- `target_xseg_port` is the XSEG port where the synapsed peer will place the requests that it received from the network.

Synapsed can also be started by the `archipelago` command line tool. In this case, the above options are set in the `synapsed` section of the `archipelago.conf` file.

After reading the configuration options, synapsed creates a socket and listens for connections at `host_net_port`. The connection to the remote peer is delayed until it is actually required (i.e. it needs to send a packet for the first time).

5.2.2 Polling for requests

There are three different ways in which a synapsed peer can receive requests:

1. New XSEG requests: These are the requests that are received on the request queue of the XSEG port that the synaped peer listens to. After accepting the request, the peer establishes a connection with the remote synapsed peer and then marshals (see Sec. 5.2.3) and transmits the request.

2. XSEG reply requests: These requests are received on the reply queue of the XSEG port. In this case, the synapsed peer will once again establish a connection with the node that made the original request and then follow the same procedure of marshalling and transmitting it.

3. Requests received over TCP/IP: These can be either new or reply requests depending on which node made the request. If it is a new request, the synapsed peer will propagate the request to another peer for completion. If it is a reply request, then the synapsed peer will put it into the reply queue of the XSEG port of the peer that made the original request.

To summarize, a request may be received either from a socket or from an XSEG port where the peer is notified via a SIGIO signal. Thus, we use the `ppoll()` system call which polls on a given set of file descriptors allowing an application to safely wait until either a file descriptor becomes ready or until a signal is caught. After the peer wakes up and processes the request, it does not immediately go back to sleep since the caller may be sending more than one requests. Therefore, before going back to sleep, the peer does a brief busy-wait loop where it checks the XSEG queues and the sockets for new requests. In this case the sockets are checked with the `poll()` request with a zero timeout.

5.2.3 The structure of a request

The XSEG requests cannot be transmitted as they are, since its data and target name are stored in two different buffers of variable length. Besides, the `xseg_request` contains pointers which will be invalid since it is destined to be put on the shared segment of a different node.

At the start of every transaction, we send a fixed-length header (Listing 5.1) which contains the information about the length of the two buffers, namely the fields `targetlen` and `datalen`. The contents of the buffers follow. The receiver, first reads the header and allocates enough memory to store the two buffers and then reads their contents from the socket.

There are a few more interesting fields in the `synapsed_header` structure:

- The `original_request` structure contains some fields that identify the request in the node that created it and must be sent back intact in the reply. Its contents are mainly pointers which are invalid in the context of the receiving node and are thus not used. The structure is bound to the recreated request, by making the `priv` field of the `xseg_request` (Listing 3.1) point to it.

```
struct synapsed_header {
    struct original_request orig_req;
    uint32_t op;
    uint32_t state;
    uint32_t flags;
    uint32_t targetlen;
    uint64_t datalen;
    uint64_t offset;
    uint64_t serviced;
    uint64_t size;
};
```

Listing 5.1: Synapsed header

```
struct original_request {
    struct peer_req *pr;
    struct xseg_request *req;
    uint32_t sh_flags;
}
```

Listing 5.2: The `original_request` structure

- The other fields are the mandatory information needed to recreate the XSEG request on the receiving node.

The `original_request` structure contains fields that are not useful to the receiving node, but must be sent back to the sender with the reply. The sender needs these data to be able to identify the reply and route it to the appropriate peer.

Finally, it is worth mentioning that we have employed the `readv()`/`scatterv()` function calls which allow us to do scatter/gather I/O.

Chapter 6

Design

This chapter describes our design for adding support for block storage backends to Archipelago. First, we begin with defining the problem that we aim to solve and setting the goals that we aim to achieve. Then, in Section 6.1 we present some of the solutions that we studied but eventually rejected. These led to the final design proposal which we explain in Section 6.2. Next, in Sections 6.3 to 6.6 we compare our proposal against the goals we initially set, seeing to what extent it achieves them and explaining various fine details in the process. Finally, Section 6.7 provides a summary of the chapter.

Our primary design goals are:

High Availability
: To provide a fault-tolerant system architecture, we must take great effort to ensure that there are no single points of failure. While designing the architecture, we take for granted that the Storage Area Network itself is already redundant without examining it further.

Performance
: A challenge we have to solve is maximizing throughput and minimizing latency while keeping the total cost of the equipment as low as possible.

Scalability
: Scalability in our case means the ability to increase the volume of hosted virtual machines with the lowest cost possible and without degrading performance. Once again, we do not concern ourselves with the scalability of the underlying storage array.

Simplicity
: Wherever possible we follow the *"Don't reinvent the wheel"* mantra, taking care not to sacrifice performance.

Apart from these goals, it is also worth mentioning that we cannot accept anything less than strong consistency in our final design. Since the objects can be hot blocks of the virtual machine volumes, any other consistency model can cause filesystem corruption, resulting in a very unreliable system. Accordingly, we only investigate designs that provide strong consistency guarantees. This includes the section of rejected designs as well.

6.1 Rejected Designs

First, we present some solutions that were examined during the preparation of this thesis, but were eventually rejected. Presenting these designs and their drawbacks is useful as it aids to a deeper understanding of the difficulties we faced and the decisions we made that led to our final design proposal.

6.1.1 Block storage driver

An obvious way to add support for block storage in Archipelago is to write a block storage southbound driver to complement the already existing file and RADOS drivers. This driver must be able to store and retrieve objects by communicating directly with a raw block device. To achieve this, an accompanying external tool must be written that builds a structure on the block device, similar to `mkfs` for filesystems. This structure must include at least:

- One or more superblocks that are placed on predefined physical blocks. Those contain important metadata such as the index boundaries (see below).
- A key-value data structure that indexes the objects on the block device. Candidates are B+ trees and hash tables.
- A *free space* bitmap.
- A range of blocks that contain the actual object data.
- Possibly another data structure to store and manage the object metadata. This data structure needs to be growable as object metadata are unbounded (contrary to file metadata).

Coming up with the right parameters for these data structures is not an easy task. For example, having a hash table of big size as an index results in less collisions and, consequently, faster lookup times. On the other hand, if the index is too big it will reserve a notable percentage of storage capacity, leaving less space for storing actual objects. To make matters worse, sometimes it is impossible to precompute the optimal value of the parameters because they may depend on the use case. In our previous example, if we have a large number of small objects then a bigger index is better. But if, instead, we need to store only a few large objects then a small index would be fine.

A sensible solution is to expose those parameters to the administrators, allowing them to configure them as they see best while monitoring the status of the system. This creates the need for another supporting tool, similar to `tunefs`, which will allow tuning the structure's parameters when the driver is not in use.

Since the data structures can be accessed and modified concurrently by multiple nodes we need to be able to serialize concurrent I/O to prevent corruption. This can be done in the following two ways:

- The SCSI protocol supports hardware-assisted locking via the `COMPARE_AND_WRITE` command, which is logically equivalent to atomic test-and-set. This allows the filesystem to lock a region of sectors without having to obtain a traditional SCSI reservation to reserve exclusive access to the whole volume.

- The aforementioned SCSI command is part of the relatively new SBC-3 specification, thus it is not widely supported by hardware yet. To circumvent this, it has to be combined with some software-based solution: a distributed lock manager.

We figured out it would require an enormous amount of work to build something usable. A lot of work would go towards replicating technology that is already built by shared storage filesystems and it would increase the maintenance costs on our end. For this reason, after creating a rough prototype we decided not to invest further time in pursuing our own implementation. Instead, we focused on designing an architecture that works with existing shared storage technology. Specifically, we decided to explore the possibility of modifying an existing shared-disk filesystem to fit our needs.

6.1.2 Centralized NFS Server

Consider the topology where the storage array is connected to a single machine that exposes it to rest of the nodes via NFS (Fig 6.1).

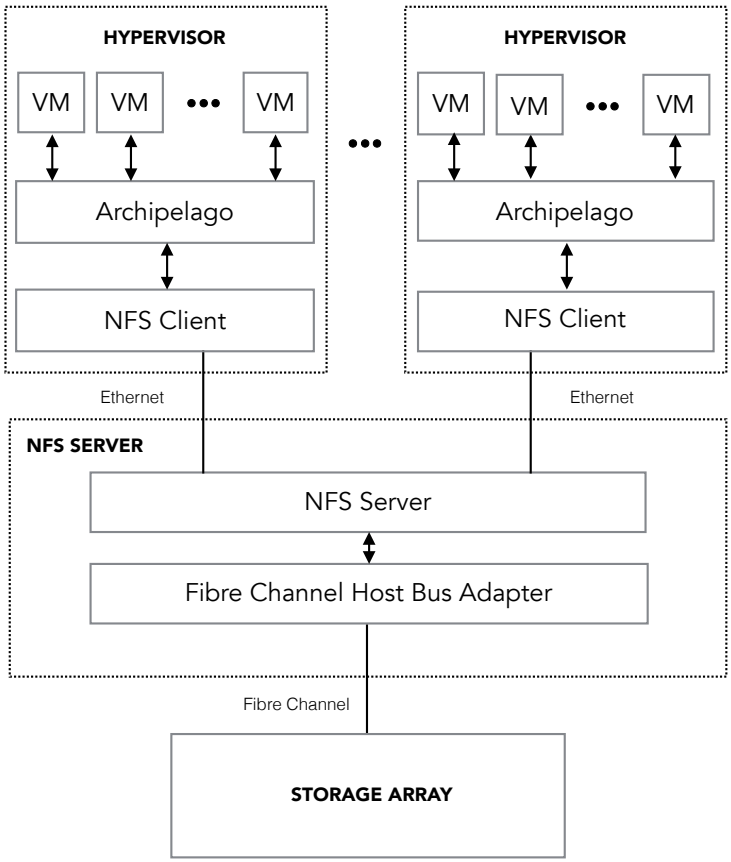

Figure 6.1: The topology of the architecture based on a centralized NFS server

While this architecture is correct it does not meet our design goals due to two serious limitations.

- Having only one node with direct access to storage means that all requests have to go through it. This creates bottlenecks on the CPU of this node and/or the bandwidth of the

network. Such an infrastructure can host a very limited amount of virtual machines, not being able to scale after some point.

- If the NFS server fails, all the hypervisor nodes immediately lose their access to the storage array. NFS does not have automatic fail-over capability due to its statefulness. Therefore, in case the server fails we cannot have another server replace it automatically because it cannot know the failed server's internal state.

6.2 Design Overview

We present a topology where the nodes are divided in two groups, depending on whether they have direct access to the storage array. As we have already explained, the nodes that are directly connected to the storage array must access it through a shared disk filesystem. The only open source shared disk filesystems as of the time of writing of this report are OCFS2 and GFS, both of which are in the Linux Kernel. After researching their pros and cons, we decided to experiment with the OCFS2 filesystem mainly for performance reasons.

We choose not to run any virtual machines on the OCFS2 nodes for two main reasons:

- As shown in Figure 6.3 each OCFS2 node is directly connected to the SAN via Fibre Channel, which means that fiber-optic cables and Fibre Channel Host Bus Adapters (HBAs) are needed for each node. The cost of this technology limits scalability.
- More importantly, OCFS2 is designed so that a node will fence itself when it realizes that it has lost connection to the rest of the cluster unexpectedly. This process is explained thoroughly in Section 6.3 where we talk about high availability. An immediate reboot on the hypervisor node is obviously unacceptable, as there is no time to evacuate the node by migrating the virtual machines to other nodes.

Instead, virtual machines are run on different nodes, each of which is connected to an OCFS2 node via Ethernet. These machines, called **hypervisor nodes**, are also connected in a separate cluster and managed by virtual server management software, such as Google Ganeti. Essentially each OCFS2 node *owns* some hypervisor nodes and is responsible for issuing requests to the storage backend on behalf of them. The hypervisor nodes do not have direct access to storage, but rather send the objects to the OCFS2 node that they are connected to.

Using NFS is one way to achieve this, where the OCFS2 node will act as an NFS server (export via NFS version 3 and above is supported by OCFS2), exposing the OCFS2 file system to the hypervisors who will act as NFS clients. However, in our implementation we decided not to follow this solution in order to avoid the overhead introduced by NFS. Instead, we chose to go with a more integrated approach, running the Archipelago blockers on the OCFS2 node. We also use the synapsed Archipelago network peer to transfer xseg read/write requests from the Archipelago instances running in the hypervisors to the instance running in the OCFS2 node (Fig. 6.2).

Since we now have a filesystem to store the objects, we can use the Archipelago file drivers without modifications, since synchronization on concurrent access is handled by the filesystem itself. The semantics are investigated in Chapter 7.

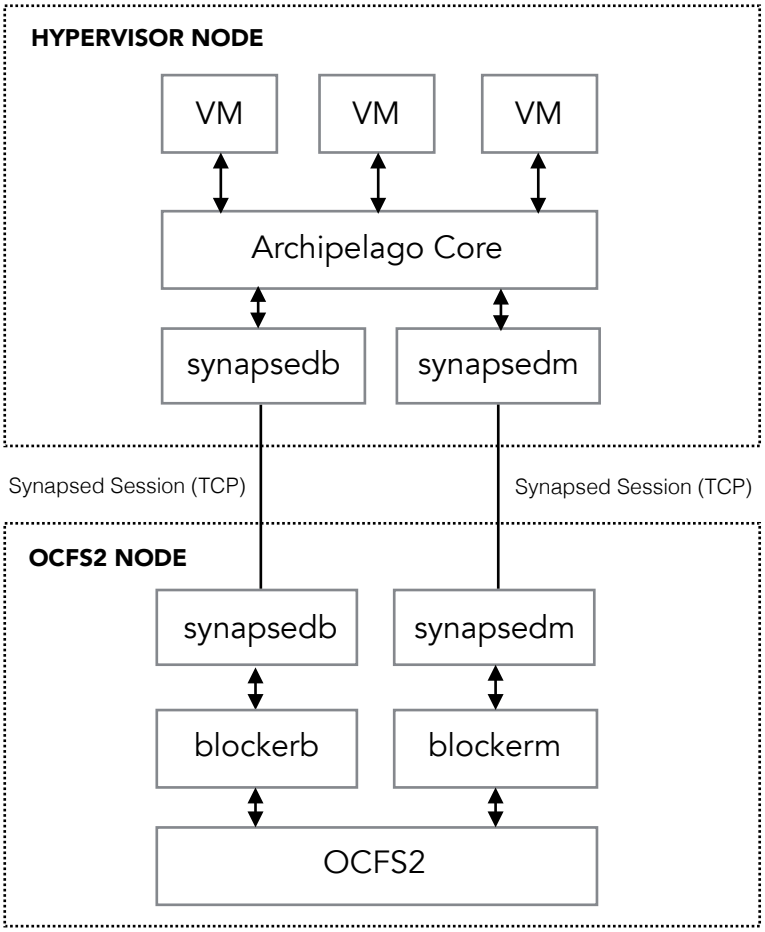

Figure 6.2: The connection of the synapsed peers running on the Hypervisor and OCFS2 nodes

6.3 High Availability

In this section we explain in detail how high availability can be achieved by the proposed architecture. More specifically, we list the various types of components that may fail and describe for each one how the system will respond to ensure availability.

The most obvious type of failures is node failures. In our design there are two types of them: hypervisor node failures and OCFS2 node failures. First of all, in the case that a hypervisor node fails, its virtual machines can be migrated to any other live hypervisor regardless of the OCFS2 node it is assigned to. After the migration is complete, the first request of each object that belongs to the migrated VM's volume will generally have increased latency, because OCFS2 will need to take the object's locks before performing any I/O. However, we can avoid the increased latency penalty if the migration occurs between hypervisors that communicate with the same OCFS2 node. One can achieve this on Ganeti by creating a custom IAllocator[1].

[1] A framework for using external (user-provided) scripts to compute the placement of instances on the cluster nodes.

The other type of nodes that can fail also are the nodes in the OCFS2 cluster. Note that even in workflows that require a small number of active virtual machines at a time (and thus a small number of hypervisor nodes), at least two OCFS2 nodes are needed two avoid single points of failure. One possible solution would be to let each hypervisor node send every I/O request to two or more OCFS2 nodes, so that if an OCFS2 node fails it can still be serviced by the other(s). However, we do not recommend this solution since concurrent writes to the same file by different OCFS2 nodes are very slow as explained in Sec. 4.4. A better solution is to simply move all the hypervisor nodes that belong to the failed OCFS2 node, to a new node. This is possible since the filesystem state is shared between all the nodes in the cluster. Furthermore, it be done automatically by assigning the IP of the failed node to the new one. A related technology that provides automic IP failover is UCARP.

6.4 Performance

The performance is largely affected by the quality of the underlying hardware, including the network infrastructure and protocol.

On the software side, we did minimally intrusive changes to the OCFS2 source to improve its performance based on certain guarantees of our use case. The modifications are presented in Chapter 7.

A thorough performance evaluation was carried out and the results are presented in Chapter 8. It includes comparisons with the RADOS-based architecture that we described in Introduction (Sec. 1.2).

6.5 Scalability

Even though the hardware needed by the proposed design are more costly compared to a distributed architecture that uses RADOS, there are technically no restrictions to the size of the two clusters. The performance will never be degraded as long as the ratio between the number of the OCFS2 nodes and the number of hypervisor nodes is kept constant.

However, there is one ceveat. OCFS2 has a separate journal for each machine, each of which is stored on the disk. When creating a new OCFS2 filesystem with the `mkfs.ocfs2` tool, one has to specify the number of the nodes that can concurrently mount the partition so that the appropriate storage chunk is reserved for node journals. It is a common practice to set the maximum number to a value higher than the current number of OCFS2 node to provide for future infrastructure expansion. Nevertheless, if one needs to add new nodes to the cluster and the total number now exceeds that value, the number has to be reset with the `tunefs.ocfs2` tool. The problem here is that `tunefs.ocfs2` expects the cluster to be online, but deactivated. This limits the availability of the system, but at least is is a procedure than can be expected and scheduled.

6.6 Simplicity

We managed to create an architecture by combining existing building blocks (namely OCFS2, Archipelago and its synapse peer) with small modifications, thus minimizing the development maintenance cost.

6.7 Summary

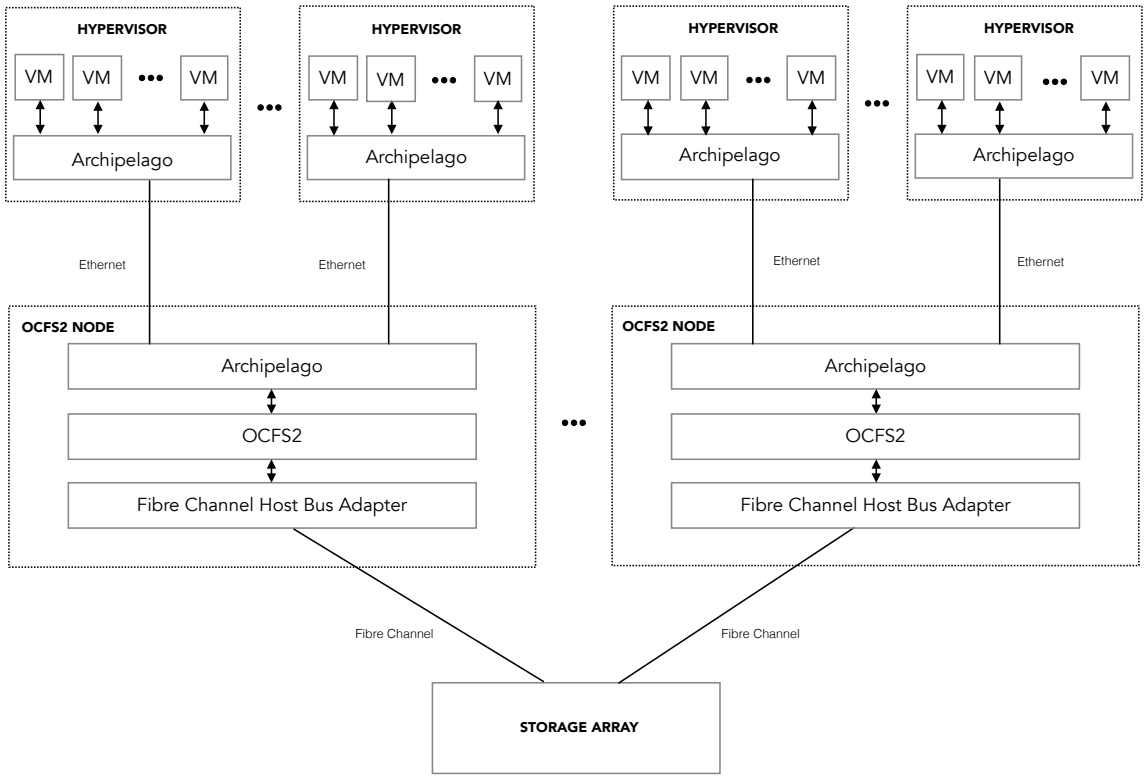

Figure 6.3: Overall architecture of block storage for archipelago

In conclusion, our architecture consists of two layers of nodes:

- The **hypervisor nodes**, in a ganeti cluster, that host the virtual machines. Each of these nodes also runs an archipelago instance that provides the volumes for its VMs.
- The **OCFS2 nodes** that run the OCFS2 filesystem in a cluster. These nodes form the middle layer of the architecture since they essentially connect the hypervisor nodes to the storage array. They also run an archipelago instance that basically consists only of two blockers and two synapsed peers (one for each blocker).

The blockers on the OCFS2 nodes are simple file drivers. The hypervisor nodes run instances of the volume composer and the mapper peers but they do not run any blockers. The communication between the two layers is done via the synapsed peer, an instance of which is run on every node. A synapsed peer on a hypervisor node has a single TCP connection with the OCFS2 node that it belongs to.

Chapter 7

Implementation

In this section we describe the modifications that we implemented in the components that compose our architecture. The extension of the synapsed peer, described in Section 7.2, was a necessary modification. The rest of the modifications were done on the subsystems that we identified as bottlenecks with the purpose of improving the overall performance.

7.1 Removing Locks From OCFS2

7.1.1 Disabling Write Locks

Write type locks add a big overhead to I/O operations in an OCFS2 filesystem. Surprisingly, Archipelago guarantees allow us to completely disable this type of locks.

Pithos objects are immutable because they are content-addressable. This means that their contents are are written only once (during their creation). Afterwards, they are read-only; changing their content would not make sense as it would ruin their content-addressable property.

Objects that belong to volumes of Virtual Machines are of two types:

- Objects that are shared by multiple volumes can exist. However, these objects are part of an Image and have Copy-on-Write semantics.
- The rest are live objects that belong to a single volume.

Only one OCFS2 node performs writes to a volume since a volume can only belong to one VM at a time and a VM runs on a single hypervisor at a time.

As we have already seen, concurrent writes are never performed in Archipelago currently by different VMs. However, suppose that Archipelago adds support for concurrent writes in the future. As Archipelago is designed to be agnostic to the storage backend, it cannot trust the storage backend to serialize concurrent writes, so it will have to add locks on its own layer and use its own lock manager to synchronize them. In any case, we can disable the Write locks safely.

It is also worth mentioning that concurrent writes due to shared VM volumes are still directly supported as write locks can still be enabled with mount options (see Sec. 7.1.4), if need be.

It must also be noted that, contrary to regular files, concurrent access on directories does happen when using the Archipelago file driver. Since the object files in Archipelago are content-addressable, their name is the checksum of their content. The files are grouped into directories by a prefix of their name, so in this way files that belong to different OCFS2 nodes may end up in the same directory. Thus, we take care to disable the write locks of regular files only, which we achieve with the `S_ISREG` macro.

7.1.2 Disabling Open Locks

A file's Open Lock prevents a node from deleting its inode while it is still open by another node. As we showed in Section 4.3.2, the open lock is taken every time the file it belongs to is opened. Similarly, when the file is closed its Open Lock is released as well. It is not hard to see that this process adds a high overhead and it is only natural to make an effort on disabling Open Locks when trying to improve the performance of OCFS2.

Once again, we can prove that Archipelago's semantics render open locks useless and we can safely disable them to remove their overhead. The reasoning course is as follows: Objects are only deleted through garbage collection when they are no longer used. This statement is also valid for volumes, as the `vlmc remove` tool does not directly delete the objects but instead just marks the volume inaccessible for usage.

7.1.3 Removing the Orphan Scan

Since we decided to disable the open lock, we are guaranteed that requests for file deletion will be performed directly, there will be no orphan files and, therefore, the orphan directory will always be empty. Consequently, disabling the open locks makes the orphan scan irrelevant and we can take the extra step to remove it completely to boost performance even more.

7.1.4 Adding Mount Options

Having decided that those locks are not useful when OCFS2 is used as the Archipelago backend, we now want to be able to disable them with mount options. The first place to look, when wanting to add new mount options to the OCFS2 filesystem, is obviously the `mount.ocfs2` command line tool. The mount options are given to `mount.ocfs2` with the `-o` flag.

However, what happens internally is that the `mount.ocfs2` tool only parses a small subset of the options that are of interest to it. The rest of the options are concatenated into a canonical string and passed in the `data` argument of the `mount` system call.

On the kernel side filesystems call the `register_filesystem()` function to register themselves to the kernel, passing a `file_system_type` structure. This struct contains a pointer field, called `mount`, that points to a function which is called by the `mount()` system call every time a mount is request for this type of filesystem. The `mount()` system call also passes its `data` argument to that function.

In OCFS2, the name of this function is `ocfs2_mount()`. All it does is call the kernel's `mount_bdev()` method that mounts a filesystem residing on a block device. This method takes a `fill_super` callback function that initializes the superblock structure.

One of the first things that `ocfs2_fill_super` does is call the `ocfs2_parse_options()` method to parse the mount options as given by the `data` argument. While parsing the options, this method creates a bitset which is eventually saved to the `s_mount_opt` field of the superblock structure. With the help of this bitset, one can check whether a specific option is checked almost anywhere in the OCFS2 codebase.

So to add a new option all one has to do is:

- Add a token that matches the option's name in the `ocfs2_fill_super` method.
- Implement code in the parser that checks whether the token we added is present and, if it is, sets a new bit in the bitset.

Following this procedure we added two options, `noopenlocks` and `nowritelocks`, which set the `OCFS2_MOUNT_NOOPENLOCKS` and `OCFS2_MOUNT_NOWRITELOCKS` bits in the bitset accordingly. To check whether the options are actually set during mount we use statements of the form:

```
1 if (osb->s_mount_opt & OCFS2_MOUNT_NOOPENLOCKS) {
2     ...
3 }
4
5 if (osb->s_mount_opt & OCFS2_MOUNT_NOWRITELOCKS) {
6     ...
7 }
```

Listing 7.1: Mount option checks

7.2 Synapsed

As we already described in Chapter 5, synapsed is a peer that acts as a channel connecting two arbitrary XSEG ports that are located on different physical nodes through TCP/IP. However, in our topology the requirements are more complex:

- Each hypervisor machine runs all the basic Archipelago components except for the blockers.
- We need to run the blockers on the nodes that are part of the OCFS2 cluster. Each such node is connected to more than one hypervisor and must be able to accept and serve XSEG packets from each one of them.

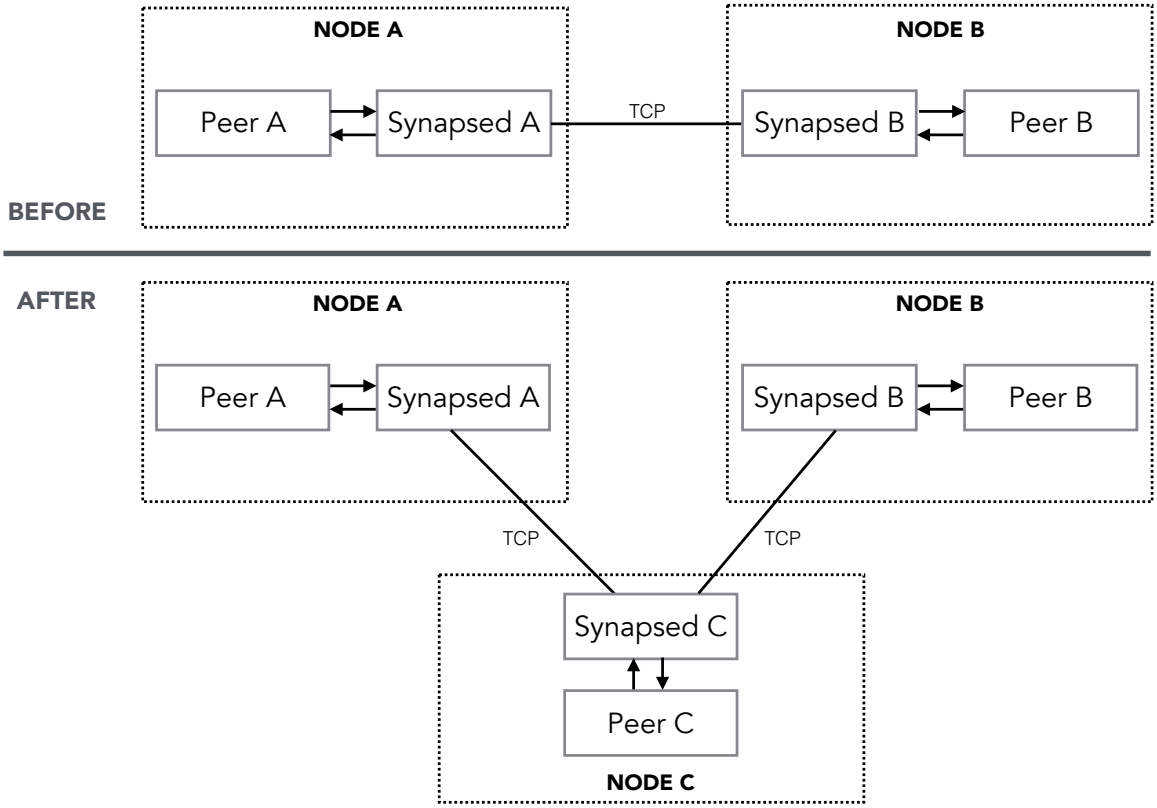

Figure 7.1: Synapsed design with three nodes

Mandated by our design, we had to extend the synapsed peer to allow it to accept requests from more than one node simultaneously and serve them accordingly. This is depicted clearly in Figure 7.1.

Although this looks like a typical client-server architecture, we decided to keep the peer-to-peer model. This requires fewer changes to the codebase, while at the same time retaining the benefits of allowing a single synapsed peer to serve original requests (requests originating from its host node) and reply requests (requests originating from a remote node targeting a peer at its host node) at the same time. Additionally, this decision does not alter the semantics of the initial implementation meaning that existing configurations (mainly infrastructures that use the cached peer) can deploy the new version without a single change.

The first modification involves the routing of the reply packets. A synapsed peer needs to ”label” the requests that it receives from the network so that it knows where to forward the reply after its host peer finishes processing it. The mechanism that Archipelago uses for labeling XSEG requests is to attach a generic pointer to arbitrary data. Synapsed uses a `struct original_request` to store the extra data, where we add and populate a new `struct sockaddr_in` field.

Note that only the new original XSEG requests are transmitted via the connection made to `remote_net_address`. The XSEG reply packets are always transmitted through the same connection that they were first received. This exception allows a synapsed peer to communicate with more than one nodes simultaneously.

```
1 struct original_request {
2     struct peer_req *pr;
3     struct xseg_request *req;
4     uint32_t sh_flags;
5     struct sockaddr_in raddr_in; // where the original request
      belongs
6 };
```

Listing 7.2: The modified `original_request` structure

Chapter 8

Benchmarks

8.1 Testbed description

Our testbed consists of five nodes which have identical software and hardware configurations as shown in Tables 8.1 and 8.2. The nodes are set up in an iSCSI SAN topology (Fig. 8.1).

Component	Description
CPU	2 x Intel(R) Xeon(R) CPU E5645 @ 2.40GHz [4] Each CPU has six cores with Hyper-Threading enabled, which equals to 24 hardware threads.
RAM	2 banks x 6 DIMMs PC3-10600 Peak transfer rate: 10660 MB/s

Table 8.1: Testbed hardware specs

Software	Version
OS	Debian Squeeze
Linux kernel	3.2.0-0 (backported)
GCC	Debian 4.4.5-8

Table 8.2: Testbed software specs

Nodes A and B are the host machines. They also run an instance of archipelago, with synapse daemons instead of blockers, that provides the volumes to the virtual machines.

Nodes C and D form the OCFS2 cluster. They are connected and frequently transmit OCFS2 heartbeat and DLM packets. They also run a software iSCSI Initiator and an archipelago instance with blocker and synapse daemons.

One node (Node E) has local access to six SSD hard disks which are combined in one RAID0 volume[1] via a hardware RAID controller. Each SSD has a capacity of 256GB for a total of 1.5TB. The node exports the array via iSCSI with the Linux-IO target. Only the two OCFS2 nodes are authorized to access the storage resource.

Nodes A to D have 3 Ethernet NICs[2], each with a 1Gbps bandwidth. Node E has 10 Ethernet

[1] RAID (redundant array of independent disks) is a technology that combines multiple hard disk drives into one logical unit. RAID0 consists of striping, which improves the throughput of read and write operations.

[2] Network Interface Controller

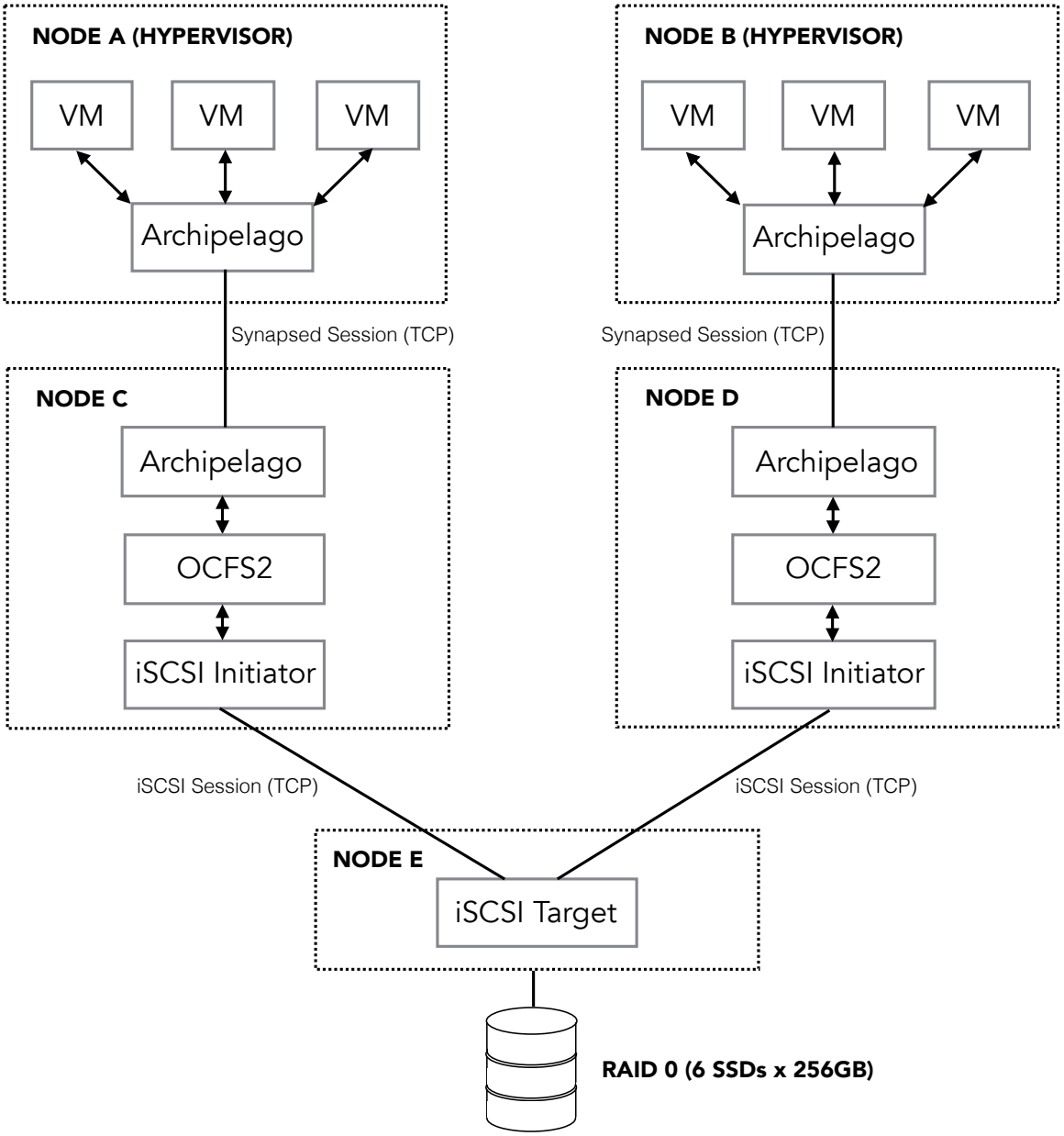

Figure 8.1: Testbed topology

NICs with the same bandwidth. The machines are connected together with a switch. The Maximum Transmission Unit (MTU) of every path is 1500.

To combine the NICs of each node to increase our throughput we use a form of link aggregation, called NIC bonding. With bonding, each TCP connection gets a hash value according to its source and destination IP and port number and then mapped to a single NIC. Bonding also has the additional benefit of acting as a high availability solution since connections can use another NIC in case a NIC fails.

By default, iSCSI creates a single TCP stream to transmit the data. One can create multiple iSCSI session, but each session will spawn a new block device on the initiator node. To solve this problem, we use the Multipath functionality of the Linux Device Mapper (DM) component which creates a new virtual block device that consolidates the underlying block devices for the same volume.

8.2 Evaluation of OCFS2 modifications

In this section, we evaluate our modifications to the OCFS2 file system. We evaluate file operations where the locks needed to complete them are not in cache. These are:

- Creating a file. This operation leads to the creation of the file's lock resources as well.
- Writing to or reading from a file from a node different than the one that holds its locks resources. This happens when an object is accessed for the first time after a migration.

8.2.1 Create operations

noopenlocks flag	nowritelocks flag	avg. latency (ms)	performance boost
		2.77	-
✓		1.97	28.8%
	✓	2.72	1.8%
✓	✓	1.90	**31.4%**

Table 8.3: Latency of a single create operation

We notice that turning off the write locks causes almost no improvement in the execution time. OCFS2 already performs an optimization where it does not wait for write lock acquisition when creating a new file, since no other node can hold this lock resource.

8.2.2 Read operations

We measure the latency of a single read operation where the appropriate lock resources are not in the node's cache.

noopenlocks flag	nowritelocks flag	avg. latency (ms)	performance boost
		5.09	-
✓		4.35	14.5%
	✓	4.03	20.8%
✓	✓	3.50	**31.2%**

Table 8.4: Latency of a single read operation

8.2.3 Write operations

Again we measure the latency of a single *synchronous* write operation where the file's lock resources are not in the node's cache.

noopenlocks flag	nowritelocks flag	avg. latency (ms)	performance boost
		4.45	-
✓		3.79	14.8%
	✓	3.76	15.5%
✓	✓	3.15	**29.2%**

Table 8.5: Latency of a single write operation

8.3 Comparison of the two architectures

Next we see how the proposed architecture compares to the RADOS-based architecture (Sec. 1.2). We expect the RADOS-based architecture to perform better since it avoids the use of a Distributed Lock Manager with a technique called **sharding**: the objects are partitioned according to their key and each partition is stored into a different OSD. Therefore, we consider the performance of the RADOS-based architecture as the goal which the proposed architecture aims to achieve, but see Section 1.2 on why deploying RADOs is not an optimal choice for an existing SAN.

The benchmark tests were conducted with FIO. We wrote a custom shell script that passes an FIO job file (Listing 8.1) to the benchmark tool, after setting the `IODEPTH` and `BLOCKSIZE` environment variables.

```
[random-write]
rw=randwrite
size=3g
bs=${BLOCKSIZE}
filename=/dev/rbd0
ioengine=libaio
iodepth=${IODEPTH}
direct=1
invalidate=1

[random-read]
rw=randread
size=3g
bs=${BLOCKSIZE}
filename=/dev/rbd0
ioengine=libaio
iodepth=${IODEPTH}
direct=1
invalidate=1
```

Listing 8.1: FIO job file

8.3.1 Random Writes

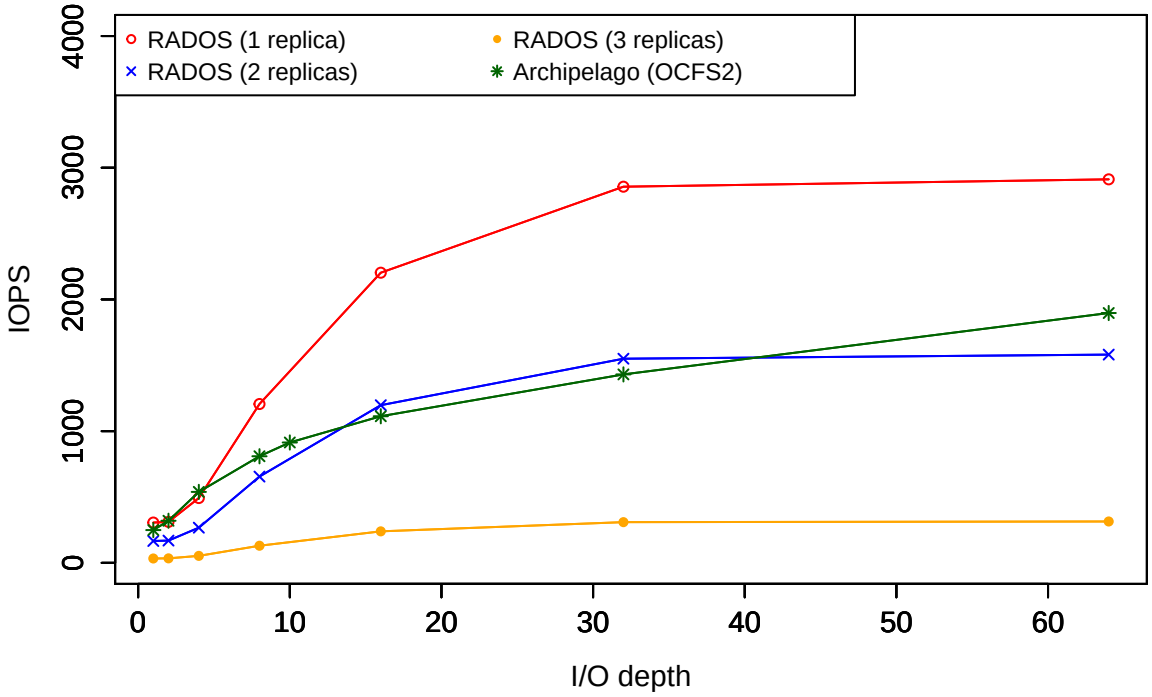

Figure 8.2: Random 4KB writes on virtual block devices

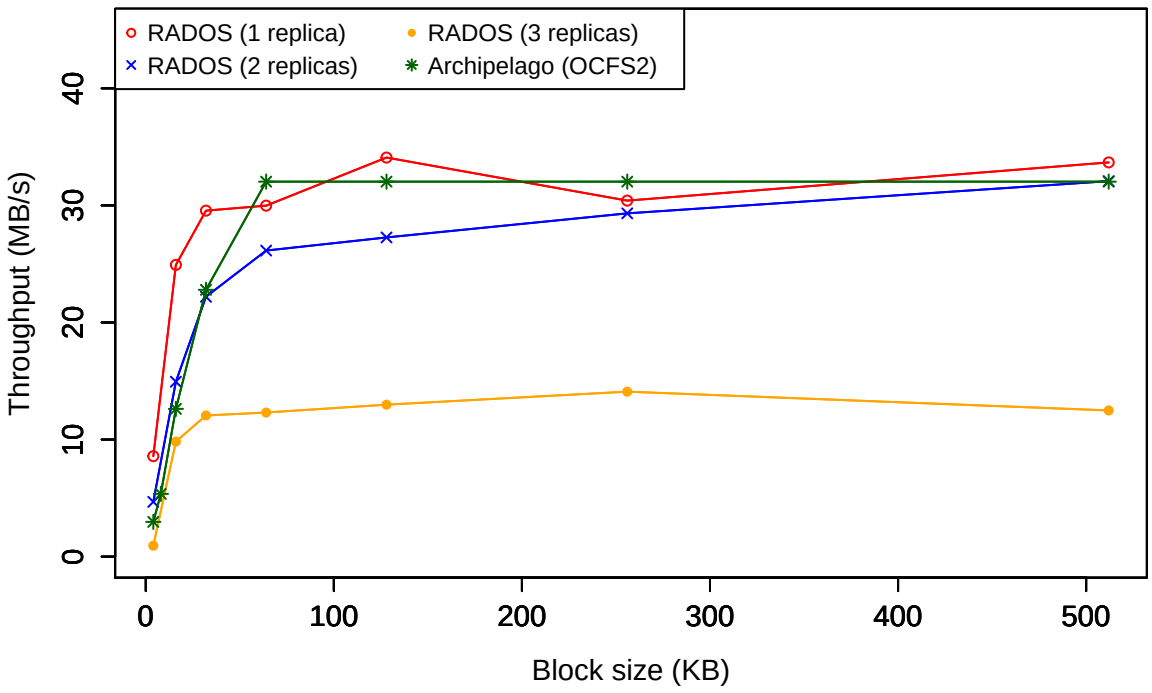

Figure 8.3: Random writes on virtual block devices with I/O depth set to 16

8.3.2 Random Reads

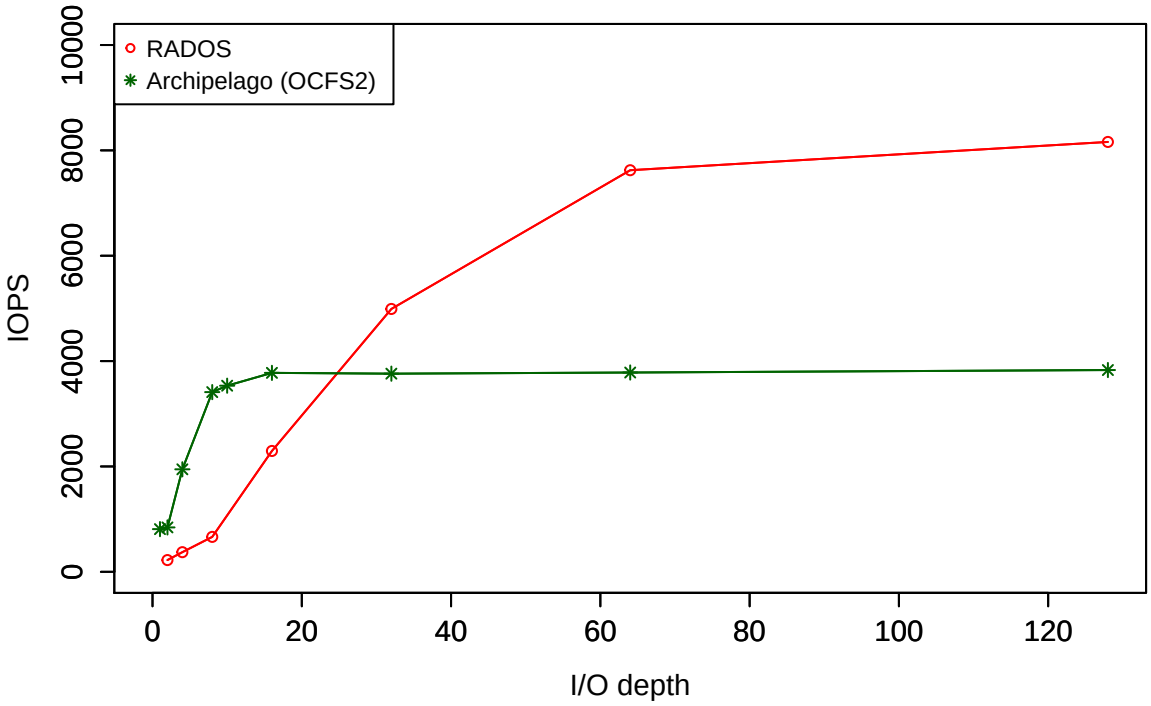

Figure 8.4: Random 4KB reads on virtual block devices

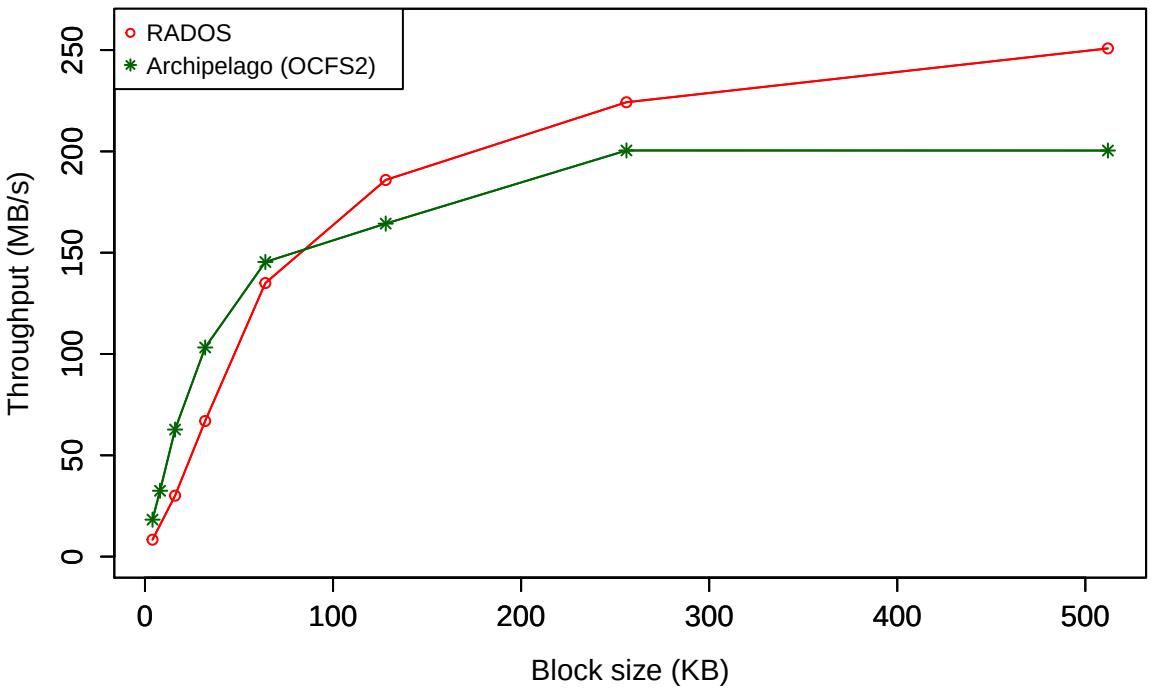

Figure 8.5: Random writes on virtual block devices with I/O depth set to 16

8.3.3 Comments

We notice that our proposed architecture offers comparable write performance with the RADOS-based architecture. Its IOPS and throughput are aproximately equal with the RADOS 2-way replication architecture. Our benchmarks show that when we configure RADOS to keep two extra replicas for each object, its performance gets poor.

Regarding the read operations, we see in Figure 8.4 that the RADOS-based architecture is two times as fast as the Archipelago-based one when many I/O requests are sent on the fly. Finally, RADOS peaks at 250MB/s, while Archipelago peaks at 200MB/s. Finally, we must mention that the limited network bandwidth is a bottleneck in our testbed and plays a big role in these benchmark results.

Chapter 9

Conclusion

9.1 Concluding remarks

This thesis is an effort to tackle the problem of providing an efficient object storage service over a Storage Area Network. We considered two fitting architectures:

- Setting up a RADOS cluster on top of the Storage Area Network.
- A proposed alternative architecture that involves Archipelago and a parallel filesystem.

Early performance evaluations show that our offered alternative design is comparable to the simpler RADOS-based design, being about as fast on write operations and slightly slower on read operations. It also has the additional benefit of utilizing the internal Storage Area Network redundancy mechanisms. On the other hand, RADOS implements data replication on its own layer sacrificing storage capacity.

Moreover, there is still room for performance improvement. As a proof-of-concept, we modified the OCFS2 parallel filesystem so that some locks that are irrelevant in our system architecture can be turned off. This resulted in a 30% performance boost in certain occasions.

9.2 Future work

We intend to deploy the proposed architecture on an actual Storage Area Network infrastructure with Fibre Channel (as opposed to an iSCSI topology) to measure its performance. We also aim to set it up on a production environment to draw conclusions about its scalability.

Additionally, to improve the performance of the system we need to profile it and identify the bottlenecks in the various subsystems. Afterwards, we can redesign the appropriate parts of Archipelago, we can continue carefully disabling OCFS2 features according to our guarantees etc.

Finally, another important feature will be the support of multiple TCP sessions on each peer-to-peer connection of the synapsed peer. By having more multiple simultaneous TCP sessions, we can route the XSEG requests in a round-robin fashion. This results in a redundant connection model, but more importantly adds support for link aggregation.

Bibliography

[1] Rudolf HJ Bloks. The IEEE-1394 high speed serial bus. Philips Journal of Research, 50(1):209–216, 1996.

[2] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS version 3 Protocol Specification. Technical report, RFC 1813, Network Working Group, 1995.

[3] Christine Caulfield. Programming Locking Applications. Red Hat Inc, 2007.

[4] Intel(r) xeon(r) cpu e5645 specifications. http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.

[5] Mark Fasheh. OCFS2: The Oracle Clustered File System, version 2. In Proceedings of the 2006 Linux Symposium, pages 289–302. Citeseer, 2006.

[6] IEEE. Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]. IEEE, New York, NY, USA, 1996.

[7] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. Synnefo: A Complete Cloud Stack over Ganeti. ;login, 38(5):6–10, October 2013.

[8] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. ~okeanos: Building a Cloud, Cluster by Cluster. IEEE Internet Computing, 17(3):67–71, May 2013.

[9] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[10] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. Filesystem hierarchy standard. V2, 3:29, 2004.

[11] Julian Satran and Kalman Meth. Internet small computer systems interface (iSCSI). 2004.

[12] R Weber. SCSI object-based storage device commands-2 (OSD-2), 2009.

[13] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[14] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07, PDSW '07, pages 35–44, New York, NY, USA, 2007. ACM.

[15] Πυργιώτης Αλέξιος. Σχεδίαση και Υλοποίηση Μηχανισμού Κρυφής Μνήμης για Κατανεμημένο Σύστημα Αποθήκευσης σε Περιβάλλον Υπολογιστικού Νέφους. Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, 1 2014.

[16] Φίλιππος Γιαννάκος. Ανάπτυξη Οδηγού Στο Λ/Σ linux Για Την Υποστήριξη Εικονικών Δίσκων Πάνω Από Κατανεμημένο Σύστημα Αποθήκευσης Αντικειμένων. Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, 7 2012.