UnityFS: A File System for the Unity Block Store

by

Wei Huang

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

UnityFS: A File System for the Unity Block Store

Wei Huang
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2013

A large number of personal cloud storage systems have emerged in recent years, such as Dropbox, iCloud, Google Drive *etc.* A common limitation of these system is that the users have to trust the cloud provider not to be malicious. Now we have a Unity block store, which can solve the problem and provide a secure and durable cloud-based block store. However, the base Unity system does not have the concept of file on top of its block device, thus the concurrent operations to different files can cause false sharing problem. In this thesis, we propose UnityFS, a file system built on top of the base Unity system. We design and implement the file system that maintains a mapping between files and a group of data blocks, such that the whole Unity system can support concurrent file operations to different files from multiple user devices in the personal cloud.

Acknowledgements

The work of this Master thesis would not have been possible without the help and support of many people.

First and foremost, I would like to express my deep gratitude to my supervisor, Professor David Lie, for his patience, kindness, invaluable guidance and strong support to me since I started my academic life in the University of Toronto. I sincerely appreciate his precious time, advice and training in all aspects of my academic development.

I would also like to thank my group members in the project of Unity system, Beomheyn Kim and Afshar Ganjali, my fellow graduate students James Huang, Billy Zhou, Zheng Wei and Michelle Wong for their help. I am very grateful to my family for their moral support that maintains my sanity.

Finally, I would like to thank University of Toronto and the Department of Electrical and Computer Engineering for their financial support.

Acronyms

AFS Andrew File System

API Application Programming Interface

App Application

CIFS Common Internet File Systems

DE Data Entity

DHT Distributed Hash Table

EXT Extended File System

FAT File Allocation Table

FUSE File system in USEr space

LAN Local Area Network

MAC Modification, Access and Change times

NBD Network Block Device

NFS Network File System

RPC Remote Procedure Call

SMB Server Message Block

UBD Unity Block Device

UFS Unity File System

VFS Virtual File System

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

There are many good reasons for building a distributed file system. Among those, building a cloud-based personal secure file system which provides confidentiality, integrity and durability is certainly worth of serious consideration. On the current market of storage services, a large number of personal cloud storage solutions have emerged in these years, such as Dropbox, iCloud, Google Drive, Skydrive and SpiderOak. These services consist of an online storage server and automatic synchronization software that mirrors some or all of the files the user has stored in the cloud to the user's clients. However, a common problem with all those personal cloud storage system is that users have to trust the cloud provider. Even a user may encrypt sensitive data before uploading it to the cloud, it is still possible that a malicious cloud corrupted the user's data, then the bad files propagate from the cloud to all the personal devices the user owns.

We now have a built secure and durable distributed block store without having to trust the cloud provider: Unity block store. The Unity system encrypts data before storing on the cloud to protect confidentiality and replicates blocks with versions over the user's clients to ensure durability like recent work [1–3]. However, there are some remaining problems in Unity: most important ones among them are performance and flexibility of managing files. The Unity system only has the concept of data blocks and the group of certain data blocks called DE, which means that as a block device on the client side, it is one DE from Unity's point of view, and it is not aware of files in the file system mounted on top of it. Because of this, when a user accesses two files on more than one clients, the problem of false sharing occurs: under the assumption of personal cloud storage, there is only one client can be accessed by a single user at the same time. In this way, although they tend to get access to different files, one of the clients has to wait in line.

In this thesis, we describe the design and implementation of a file system built on top of the Unity block store. We design the architecture of the file system to meet four basic requirements: (1) It should serve as a normal file system at client side to make the block device, the existence of cloud and other clients almost transparent to the end user; (2) It should guarantee the consistency of files when other clients want to access one file at the same time: blocking read/write to the file until the cloud decided which client gets the lease of the corrensponding DE; (3) It should be able to flush and invalidate all the caches not stored inside the file system, for consistency of a file when its lease switched to/from other clients; (4) It should maintain mappings between the DE IDs, block IDs and the file names, and provide translations of the mappinsg upon requests by the Unity block store.

We demonstrate that our implemented prototype UnityFS can full-fill all the four requirements. The UnityFS consists of three parts: a UFS kernel module based on Minix $v3$ file system, a UBD block device driver, and a service thread of Unityd daemon process running in user space. We evaluate the system from different aspects using different workloads.

1.1 Overview of this Thesis

This thesis comprises seven chapters. This chapter has introduced the problem and given a brief overview of the base block store and the file system. Chapter 2 will summarize some related work, including traditional networked file systems and secure distributed file systems.

Chapter 3 will describe the base Unity block store by summarizing what each component does and how the cloud and clients interact with each other. The chapter prepares a context in which the UnityFS is built. We then proceed to discuss what the architecture UnityFS should use in Chapter 4. We first give four requirements for the file system, and then give three candidate architectures, all of them can meet the requirements. After discussions on the options, we will choose one architecture and then discuss about the implementation details in Chapter 5. The chapter of implementation explains in details the functionalities of the modules of UnityFS, and how they are implemented. Examples will be given to illustrate the execution flows of several typical file system operations.

Then follows with evaluations of UnityFS in Chapter 6. Three sets of evaluations show that UnityFS has the advantage of supporting concurrent file operations comparing to the base Unity system, that the main source of system overheads is the lease-switching time, and that the extra cost of network bandwidth consumption that the Unity system has in comparison with NFS, UBD and Dropbox.

In the end, we conclude the thesis in Chapter 7, and propose some future work that may be of interest.

Chapter 2

Related Work

The Unity file system is by no means the first nor the last securing distributed file system. In this chapter, we will address some representative systems that have been implemented in the past. We will first discuss about general distributed file systems, some of which were significant in the file system history. Then we summarize several recent secure distributed file systems, which can run in the presence of untrusted server. We will also raise instances of distributed storage and their usage scenario, and techniques invented for measuring those systems.

2.1 Distributed File System

Network File System (NFS) [4, 5], developed by former Sun Microsystems in 1980s, is one of the first and now the most widely-deployed distributed file systems providing transparent and remote access for clients on top of Remote Procedure Call (RPC). NFS follows a client-server architecture that a centralized server stores and maintains data that can be shared by multiple clients. The shared data is exported by servers in a file system tree structure. Clients can import and mount it onto their local file system mounting points such that system users could access the files through standard Unix-like file system interface.

Other early distributed file systems were developed for similar purposes but with different features. For example, Alpine [6] by Xerox in 1980s had the primary goal of storing database files than other ordinary files. Communications are also via RPC, which is provided by Cedar interface. Alpine has a log-based method for atomic file update that provides atomic transactions as well. The transaction-based file system of Alpine has low performance since it has to set lock on files.

Andrew File System (AFS) [7, 8], a part of a distributed computing environment developed by Carnegie Mellon University tried to solve the scalability issues at the beginning, for supporting thousands of campus-wide workstations connected by network. The Andrew file system presents location-transparent file name space to all clients used by students and staffs in CMU. The AFS uses local persistent storage for client side caching, which could contain a whole file on the local disk and help reduce the frequency of accesses to the servers. The AFS provides multi-readers single-writer synchronization, in this way, no explicit locking is require by application programs, but by caching management programs implicitly.

Descended from AFS directly, Coda file system [9] aims at solving the problems for distributed file

system under weakly connected networking environments or server failures. Coda supports transparent server replications to provide high availability by serving clients from an alternative server in the event of the server crash. It also provides trickle reintegration that allows clients to commit changes to the server steadily and gradually without significantly compromising the performance of the client. Coda assumes low probability of having conflicts [10], and does not use too restrictive and expensive cache consistency strategy, while instead, it allows all servers to receive updates at the same time, and detects and reports when conflicts taking place, and sends them to a handler, and could be solved with either manual or automatic method.

Remote File Sharing (RFS) [11, 12] developed by Bell Laboratories is a part of AT&T Unix System V3. Different from NFS, the RFS is not based on RPC and not reliable on server failures, but it can support file locking and append write, which NFS does not have of UNIX file system's semantics.

The Server Message Block (SMB) [13] protocol was first developed by IBM in 1980s, for programmers to write programs interacting via IBM PC Network accessing resources transparently. It was the first protocal allowing PC users under DOS 3.1 to use features of distributed file system, then, it was also implemented on BSDs and VMS. When a remote access call is made, the call is taken by the Netbios who establishes a VC connection with a remote server. In early implementations, SMB has no support for user authentications, transactions or failure recovery, but later Microsoft improved the protocol in the 1990s and renamed it to Common Internet File Systems (CIFS) [14] and introduced SMB2 [15] in 2006.

As variants to the traditional client-server architecture distributed file systems, many of them are designed for server clusters, often used for parallel applications. As examples, Google File System (GFS) [16] and Ceph [17] allow clients parallel accessing to a pool of persistent storage devices. For performance improvement on target applications, they use file-stripping techniques in the cluster-based distributed file systems. They distribute a single file across multiple servers, and enable fetching different part of the file in parallel. For large server clusters, clients read and write files across thousands of machines, which could be as large as mutiple gigabytes. Files contain lots of smaller objects and most frequent operations of updates to files are appending rather than overwriting. For a large scale server cluster like Google's infrastructure, it is relatively frequent that machines are crashed. GFS constructs clusters of servers that consist of a single master and multiple chunk servers. The master is only contacted for the location information of the required part of file, which is chunked and distributed across chunk servers. Chunks are replicated across different chunk servers for high availability for the similar purpose as the Coda. As the master node's load is only metadata management and inquiry, it can scale much better than traditional client-server architectures.

2.2 Securing Distributed File System

The above mentioned distributed file systems and usage infrastructures have little consideration on security and privacy, and based on the same assumption, that they trust the servers in any case, are not malicious. There are many existing works for securing distributed file system in the presence of the untrusted server component. The summarization of these projects follows.

2.2.1 Cryptographic File Systems

The SUNDR system [18, 19] by NYU is one of the first group of distributed systems make untrusted storage system work to satisfy desired properties and guarantee an expected linearized order of operations by a series of clients. It means that clients in the system can fail, or miss updates from other clients. The fork consistency can guarantee that clients who share their observations of the operation orders can discover misbehaviour as soon as they did enough exchanges. The fork consistency has a total overhead costs to the system with n clients proportional to n^2 bits bits per operation. There are improvements to this result [20] showing that if we do not consider the situation that malicious clients may collude with the bad server and concurrent updates from clients blocking each other, the communication overhead can be lowered to only $O(n)$.

Sirius [21] is a distributed file system made on user-level targeting at secure file sharing on an existing file system without enforcing any changes to the original file servers. It can work as a layer over network file system such as NFS and SMB/CIFS. The system is implemented over NFS and the file storage strategy is to separate encrypted content data and file metadata that contains access control information. The content data is encrypted by a writer-signed symmetric key, and metadata is associated with metadata freshness file stored in each directory and associated with sub-directories. The freshness file is constantly updated to gurantee the file tree is fresh by checking hash. In this way, the strengths of integrity provided to content data and metadata are different. The details are hidden from users by the Sirius system so that the file system can run on top of any storage server given that the semantics of the server interactions with clients are known.

Plutus [22] by HP Labs also assumes minimal trust over storage servers. Some of ideas like file-group sharing and file-block encryption are based on the earlier work of Cepheus [23]. In Plutus file system, blocks of files are encrypted when created, as well as the inodes' file name entries of the directory. The files further grouped with same sharing atribute are seen as equivalent and associated with the same symmetric key. The key management and distribution is essential to the whole file system's performance. The encryption overheads are minimized by using lazy revocation that delays the crypto process until a file is actual updated.

2.2.2 Fault-tolerant Distributed Storage Systems

The above works can protect data security over files stored distributed on different nodes, however, in the presence of server failure, the system-wide data availability shall be challenged. For example, if the file owner on a client of Plutus is off-line, the key cannot be distributed to other revoked users. Other works on peer-to-peer-like file systems can deal with the problem, such as Farsite [24] and OceanStore [25].

Farsite is a project by Microsoft. It has the motivation of building a scalable distributed system running on desktop computers as clients that could be unreliable and of limited security. The system is generally designed for a network environment with high bandwidth and low latency and under such assumptions, strong consistency is provided. The files are randomly replicated in the storage systems in order to provide availability and reliability, encrypted with file contents to provide confidentiality and maintained with Byzantine-fault-tolerant protocol to grant integrity. The directory server of Farsite uses tree-structured file identifiers, which allows the metadata of files to be dynamically partitioned at arbitrary granularity.

OceanStore by Berkeley is an infrastructure for large-scale peer-to-peer storage service using encryp-

tion, erasure coding and Byzantine agreement to guarantee availability, security and durability. The file system is fully-decentralized and uses distributed hash table (DHT) for placing and locating files, which reduces the storage consumption and management costs. The protocol used by OceanStore generally assumes there are a large number of servers alive in the network overlay, and files can always be located at run-time using their multiple-hop routing protocol.

The two P2P-like file systems can tolerate Byzantine fault, using quorum-based protocols flooding network with broadcast messages. They could not avoid expensive costs over the communication network, and their use cases all have the implicit assumptions of multiple users and under this scenario, strong consistency is a goal too hard to achieve.

There are works also provide security and durability guarantees even with untrusted servers, and can be considered closely related to our project:

Depot [2] is a cloud storage system developed by University of Texas at Austin. It aims at achieving minimum trust on either client or server, which can be tolerated with buggy or malicious behaviours. The data is stored in the cloud storage system in a form of key/value and the consistency of file updates from clients are guaranteed by using version history and version vector hashes. Multiple versions of concurrent updates are stored, and conflicts can be solved when they occur: by the clients either manually or automatically. Similar to Depot, The Sporc project [1] by Princeton University can guarantee a variation of fork-consistency. It is capable of recovering storage faults after detecting them, by using conflict resolution mechanism of Operational Transformation. Venus [3] is built with Amazon S3 to secure user interactions on cloud storage where the cloud server is not trusted. It supports weak-fork consistency in general and when server is correct, client failures will not affect other clients. The client-client communications in Venus are using signed emails to prevent network attacks. Venus has the limitations that applications have to use key-value store, and a certain set of clients have to be online to achieve consistency guarantees. Moreover, under all scenarios the above mentioned systems consider, they do not regard bandwidth efficiency as an important measure.

2.3 Use Scenarios for Distributed Storage

Related to the distributed file system, to provide an infrastructure based mobility support, the Internet Suspend/Resume (ISR) [26, 27] can get a desktop environment perceived at any location and time. As users of ISR, they could continue working on their own customized desktop environments by checking out VMs from the Internet storage service. For consistency issues, the ISR uses locking mechanism upon keeping data blocks. The locking is as simple as holding a lock on a running VM and releasing it after executing the suspension command.

While realizing VM location transparency, ISR still requires images of the running OS including all working environments stored on a centralized server, or at least a part of the image being carried by users on a portable storage device such as portable hard drive or USB flash disk. This brings manageability problems of configuring and maintaining users' working environment, which is one of targets the Collective project [28–30] aims at.

The Collective mainly focuses on the issues of migrating the state of running machines across different physical devices. The starting point is from an individual person's perspective that she might encounter with difficult problems when administrating the desktop environment all by herself. Such burdens can be eased by the Collective by keeping a golden VM image containing consistent software environment,

well-configured settings and applicable security patches. The golden VM image is kept on the Network File System (NFS) server, but the idea of usage is different from the distributed file system we mentioned in previous sections.

2.4 Measuring Cloud Storage and Services

Recent years many companies have been developing all kinds of cloud storage services. They tend to satisfy different needs from users and provide different features while having their own advantages as well as shortcomings. Works have been done to provide evaluations and measurements to the cloud services.

Large-scale network file system workload has been measured [31] on CIFS protocol. Through this study, we can see that the identified workloads over the past 5-10 years have changed greatly, which gives us useful insights into how to design future networked file system and cloud storage services for new file access patterns from modern industrial world.

Garfinkel [32] has done some experiments on several cloud service providers. His measurements includes data throughput, transaction-per-second for ead/write operations. He found out that for different locations the data throughput varies significantly, most of time much lower than the data transfer rates capability between those sites. Garfinkel was one of the first researchers who studied cloud providers as a black box by conducting large amount of experiments.

A UIUC technical report [33] demonstrates how to use Open Cirrus [34], an open cloud-computing research testbed to evaluate the performance of Illinois Cloud, and shows the performance mainly depends on network characteristics between the cloud facility and its users.

Cloud Metric [35] provides a series of metrics to different cloud services categories including SaaS (software as a service), PaaS (platform as a service) and IaaS (Infrastructure as a service). It measures a cloud service from following aspects: (1) Availability, (2) Concurrency, (3) Dynamic load Balancing, (4) Intensiveness, (5) Security, (6) Independent running applications.

For personal cloud storage systems, measurement has been done [36] on the most popular cloud storage application DropBox [37] over the real Internet workloads. They found out what is critical to the performance and identified improvements for the current protocal. As of general cloud storage system performance measurement methods, people have identified the key point is to set up a baseline for performance and then charaterize how techniques deployed can gain. Benchmarking tools for comparing between service providers of remote storage system are starting to take place, for example, CloudSleuth [38] and CloudHarmony [39].

Chapter 3

Overview of the Base Unity System

Before beginning to describe the UnityFS, it is necessary to have an overview of the base Unity system: a cloud based block store, base on which UnityFS is built. The base Unity system uses a cloud provider's availability and network bandwidth to efficiently provide security and durability for personal block data stored in the cloud. Moreover, by using cryptographic methods, the users can enjoy these advantages of this system so that they do not have to fully trust the cloud provider.

A Unity cloud consists of several user-controlled devices and a cloud storage provider. Here we start by introducing some basic concepts, components and the protocol of the base Unity system. Then, we will briefly summarize how each component is implemented.

3.1 Data Entity

In the perspective of the Unity block store, all data is organized as data blocks and DE (there is no concept of file). Data stored in the cloud consists of fixed-sized blocks of data and is grouped in units. We call each group a *Data Entity* (DE), which is the basic unit of consistency in the base Unity system. A DE is an array of blocks and each data block has a unique index within the DE. A DE could be any unit of data in the cloud storage. A DE is version-ed upon writes to the blocks it contains. Multiple versions of DEs can be stored in cloud storage provider or the user-controlled devices to provide consistency, fail-safe and durability.

Noting that the terminology "Data Entity" is also used in the area of database, where it enables the ability of a group of data to be transparently used by different databases. However, the concept of Data Entity in this thesis is different from the area of database: a DE directly maps to a group of data blocks stored on a block device.

3.2 Cloud Storage Provider

The cloud storage service is provided by a cloud provider. It consists of two sub-components – a *central coordinator* and a *cloud storage node*, which we will refer to simply as the *coordinator* and the *cloud node* hereinafter for brevity.

The coordinator receives heartbeat messages that the user-controlled clients, according to the protocol, regularly send, such that the coordinator can detect if a client has failed or lost network connection.

At the same time, only one user client may read/write from/to a certain unit of data (DE, as we are using here) stored in the cloud. To do so, the client must first obtain a lease from the coordinator. Once obtained, a correctly working user client does not actively give up a lease – it only gives up when another user client requests a lease for the same unit of data or it may lose the lease if it fails to send the coordinator a heartbeat message. The coordinator is also responsible for keeping records of all write updates that user clients ever make, providing those records, and deciding who gets a lease of a DE, which we will elaborate later in details in this chapter.

The cloud node – assuming it is correctly functioning – maintains storage for every block-version of every DE necessary to create a consistent image of each DE, while user devices may each store subsets of these block-versions to enable recovery against a malicious or failed cloud provider.

3.3 User-Controlled Clients and Lease-Switch Protocol

The user-controlled clients follow protocols to communicate with the cloud storage provider and with other clients. The protocols decide who gets the lease of a certain DE. With wide availability of cellular data networks nowadays, we expect user clients to be connected to the network most of the time, so the protocol can be followed under the scene that everyone is having healthy connection with each other.

When a user client realizes that it does not have a certain data block or needs an update to a new version of a block, it can request the data block stored on the cloud by sending a *block request* that specifies the DE, block index and version of the desired block. After taking the lease of that DE, a user client becomes a lease-holder. The lease-holder can write a block and accordingly create a *write update* message, which contains the DE number, block index, a version number and a hash of the block contents. The lease-holder buffers write updates and the new block contents and periodically flushes the write updates to the coordinator and the block contents to the cloud node. The coordinator keeps all such write updates in an append-only log for each DE, which we call the *state* of each DE. Clients can request a copy of this log at any time from the coordinator using a *state fetch* request. Both the cloud node and user clients store block contents in a key-value store that is indexed by DEs, block IDs and their version numbers.

Consider the case that a user client who is not currently the lease-holder to a DE wants to get access to it, the client should send a *lease-holder switch* request to the coordinator, which then sends a *lease-holder revocation* message to the current lease-holder. The current lease-holder sends any pending updates to the directory and then appends to the log with a *lease-holder update* message, which contains the identity of the new lease-holder to indicate that it has finished flushing all the state to the coordinator and is now ready to give up the lease. Before taking on the lease, the new lease-holder will request the state of the DE from the coordinator and thus become aware of the latest versions of all blocks. In case that a lease-holder was disconnected before it had sent all pending writes, the client should check the DE state on the coordinator after it reconnects to the cloud. If there were other writes to the DE while it was disconnected, it will have to resolve the conflicting writes, either manually or automatically by other separate tools. Otherwise, it can simply make a lease-holder switch request, acquire the lease and send its outstanding writes to the cloud provider.

3.4 Implementation of Unity System

The prototype of Unity system is implemented in two components: a standalone coordinator, which runs as a simple event-driven server, and a Unity daemon, called Unityd, which runs on user clients and the cloud node. Unityd can be specialized by a *client*, which is linked against a library component of Unityd. We have implemented the client UBD, which emulates a Linux block device, on top of which one can mount any standard file system. The coordinator consists of about 4K lines of C code and Unityd consists of about 15K lines of C code. Our clients vary between 2K-4K lines of C code.

Here we briefly go through the main components as below with the illustration of Figure. 3.1.

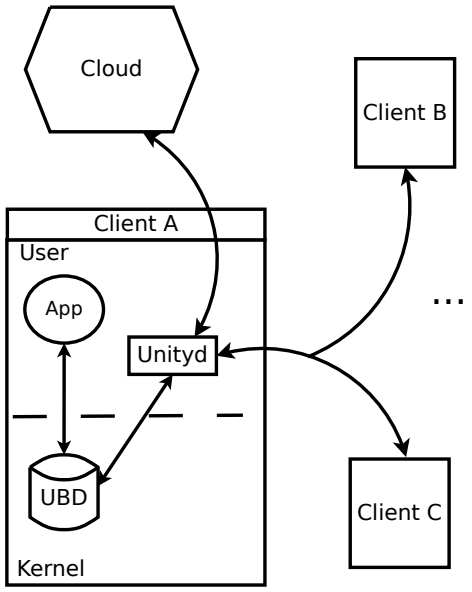

Figure 3.1: Unity block store system structure

Unityd : The daemon process Unityd consists of four major components:

- *Library Interface.* The library of API calls and callbacks is linked with the client. A client uses API calls it starts up and shuts down, creating, reading from and writing to DEs. The two call back functions are used by Unityd to notify the client lease-holder switch events when the client is about to lose the lease, such as flushing the cache. Full story on how the clients use the library interfaces will be told in the following chapters.

- *State Buffer.* The state buffer buffers DE state messages to the coordinator. It also buffers new

write, lease-holder and replication updates generated by the user client that have not been sent to the coordinator yet. To store this information, an in-memory state structure is implemented that stores the write and lease-holder updates as a per-DE append-only log. Each write update contains the block ID, version number sequence numbers, array of client IDs replicating the block, the block's SHA1 hash and a 64-bit RSA signature of the entire update. Any replication updates are stored with the associate write update along with their respective signatures as well. Lease-holder updates store the sequence number and the IDs of the old and new lease-holders, along with a signature.

- *Block Store.* The block stores the block contents that make up the DEs in the user's personal cloud. It is used both to store block contents downloaded from the cloud node, including old block versions to guarantee durability and current block versions to service a read requests from the client. The block store in our prototype is implemented using a combination of a LevelDB key-value store and a large linear buffer on an ext4 file system. The key used for LevelDB is a concatenation of the 64-bit DE ID, 64-bit block ID and then 32-bit version number and the value is the offset into the buffer where the contents of the corresponding block-version are stored.

- *Threads.* There are three threads handle events and operations in Unityd process: a replication thread that replicates blocks for durability, a client thread that handles requests from the client and lease-holder switching, and a controller thread that handles block requests from other devices and updates to the coordinator.

Coordinator: The coordinator is a simple, single-threaded server that serves requests to fetch and update state information for DEs. It uses essentially the same code used to manage the state buffer in Unityd except that it is unable to sign or verify any of the signatures. As a result, a slight modification is made to allow it to store the signatures along with the updates. Devices fetch all updates for a DE greater than some sequence number. To service these requests quickly, the coordinator uses the sequence number index to find the update from which to start reading the update with which to compose the response.

The other function of the coordinator is to detect client failure and inform the other devices. To do this, it maintains a table for all connected clients, which contains their client ID, IP address, their heartbeat period and time since it last received a message from that client. This table can also be fetched by clients so they can find the IP addresses and IDs of other user clients.

Clients: The Unity Block Device (UBD) is implemented as a Unity client, which maps a block device onto a single large DE. Clients can mount local file systems on UBD. It is built using NBD module in Linux Kernel 3.2.14. We run nbd-client and nbd-server of version 2.9.15 on the same machine and use nbd-client to capture block requests from a local file system and send them to the user-space. We use a modified nbd-server to translate block requests into library calls to Unityd's client library. The nbd-server can receive variable length requests, which UBD pads or partitions as needed into 4KB block-sized requests to Unityd daemon.

One of the benefits to use Unity block store is that it allows the clients to gain confidentiality, integrity and durability for data stored in the cloud without having to trust the cloud node or coordinator. This is done using cryptographic methods in the implementation of Unityd. The details are elided for space and not relevant to UnityFS, except for introducing performance overhead.

3.4.1 Unity Consistency and Fail-Safe Guarantee

The base Unity system replicates data blocks across the user's clients to provide fail-safe guarantees in the event of node-failure. Unity also guarantees that each DE can be replicated to a consistent snapshot of its contents, similar to the prefix semantics guaranteed by Salus [40]. So Unity must replicate in the order they were written and must also replicate older versions of blocks even if a newer version exists. It is worth mentioning that all write updates are signed, so a malicious coordinator cannot tamper with write updates or forge new ones.

The coordinator on the cloud keeps the DE states, each DE state contains a list of clients that have replicated the blocks. A newly written block on a client will be replicated to other clients through the coordination of the cloud, and the client will update the coordinator by sending a replication update that specifies the DE, block index and version that the client replicated, so that the coordinator can update the corresponding DE state.

We give some examples of how the Unity system deals with failures that happens when cloud provider or user client is unavailable or crashed.

1. **Cloud provider failure**. When a user client detects that the cloud provider is unavailable, the clients will associate them with a new cloud provider. They can combine locally cached DE states and get the replicated blocks together on any of the client who has sufficient storage space. Since every data block has already been replicated for at least twice, no data is lost in this case. The lease-holder for each DE has the newest state of that DE and could send to the new cloud assigned.

2. **User client failure**. When a user client fails, if it is not a lease-holder, other clients just need to remove the failed client's replication updates, and re-replicate the blocks for durability. If the failed client is a lease-holder to a DE, the other clients can switch to using the previous lease-holder to that DE as the current lease-holder.

In this way, the Unity system guarantees the fail-safe property given that the cloud and clients are not failed at the same time. Multiple block replications are stored in different clients, so the file system on top of it is benefited by the Unity system features that files are also durable in the system.

3.4.2 Unity Block Device Shortcomings

Besides the advantages of using UBD, there are obvious shortcomings of it as well. One of them is what we have mentioned at the beginning of this thesis, the false sharing issue. We give a specific example and demonstrate what would happen when two user clients access different files, if we used UBD alone.

Suppose that a user on Client-1 accesses file $F1$, and concurrently Client-2 accesses $F2$. Both of the files are stored on UBD, while $F1$ contains Block 5 and $F2$ contains Block 9, as we show in Figure 3.2. Because the whole block device is treated as one DE, after Client-1 taking the lease of the DE, Client-2 has to obtain the lease though it tends to access a different file. In this way, false sharing problem occurs, which compromises the system performance because the lease-switching costs time when one client is waiting for the lease.

To alleviate this false sharing issue, we will demonstrate our second client with support for multiple DEs, the UnityFS where each file in a file system is mapped to its own DE. The details of the client architecture and implementation will be further elaborated in the following chapters.

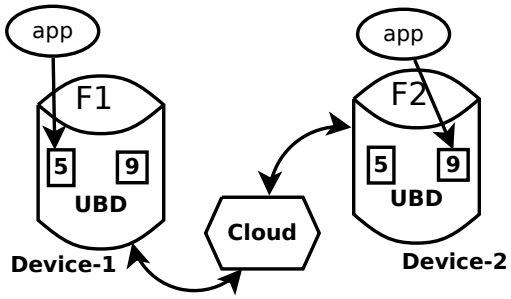

Figure 3.2: Illustration of false sharing problem when using UBD

Chapter 4

Unity File System Architecture

The base Unity block store has only the view of data blocks without being aware of files. This can be a problem when there are many concurrent accesses to the block device. A file system presents an abstraction of data blocks to help users with managing files, and naturally solves the problem of concurrent operations on the same block device. So we propose UnityFS, a file system that provides an operational interface for users to read, write and modify the grouped fixed-sized blocks of data as the basic units of consistency in the base Unity system, namely the Data Entities. In the following sections of this chapter, we will discuss about the requirements for UnityFS, give several options to the design of the file system and then justify the reasons that we choose the candidate architectures.

4.1 Requirements for the Unity File System

Unlike other distributed file systems that have a self-contained module on each client to communicate with a central server or other clients directly, UnityFS is designed to be an extension to Unity block device, which can interact with the daemon process (Unityd) through library interface. Unityd is supposed to communicate with the coordinator and with other clients. In this way, the consistency of user's data is not the responsibility of the file system running on clients, and the file system itself can be efficiently operate on files and translations between DEs and block IDs. The main purpose of doing the translations is to provide concurrent accesses to multiple DEs, without triggering lease-holder switches too frequently.

In this section we list four main requirements for the file system, then try to explain and justify the needs from perspectives of both the system and the end users:

(1) Basic file system features

(2) Blocking read/update

(3) Cache flushing and invalidation

(4) DE book-keeping

We further elaborate these requirements in details, before we describe the candidate architectures in the next section.

4.1.1 Basic File System Features

From the perspective of an ordinary user, the normal usage of file name service, directory and file metadata should be the same as other commonly used file systems. It is supposed to run locally with a coordinator process on the same machine as well, and the experience of using it with a remote cloud provider should be fairly the same. Depending on the networking environment settings, a user might sense delay, which mostly comes from the network. Similar to all other existing distributed file systems, the network delay is considered to be inevitable given the current realistic Internet communication environment. Unity system optimizes the use of network bandwidth, which is an advantage for user experience.

The configurations on a user's client should be as simple as possible. When the coordinator is running a remote cloud server, no extra configuration needs to be changed for the file system except that the local Unityd daemon process may need to update the coordinator's IP address. The information of other nodes that a client may need to be acknowledged has to be automatically updated from the cloud provider to the client. The whole process is transparent to the file system level, so that the file system can focus on interactions with Unityd process in user space.

4.1.2 Blocking Read/Updates

Technically, concurrent accesses to the same file on multiple user clients could still cause consistency issues. To avoid conflicts, when there is a need for the switch, the file system should block all read/write operations to the DE. This is because before lease-holder switch happens, the requester sent out a *lease-holder switch request* to the coordinator, and then when the current lease-holder gets the *lease-holder revocation* message it needs some time to send pending updates. During the switching time, any new updates to the DE can cause inconsistency between the two user clients, which must be avoided.

The only inconvenience a user may have to experience is that during the switching of DE lease-holder that the user performs, the procedure of lease-holder switch could trigger a short pause of the file system for a few seconds or less. During the pausing time, the reads and writes to the DE lease-holder on switch will be blocked, and resumed after that lease completes its transfer.

We argue that the short pause time is tolerable on the user side. Considering the case that when a user is actively switching between her own computing devices, then she is probably aware of the fact that some files need to be synchronized and updated to the newest version she wants to use, so the pause time is probably expected by the user. The block time to access to the DEs could be even less than device starting/activation time. Therefore in most cases, we believe that if the pause time to DEs on lease-switch can be limited to 3 seconds, there should not be an uncomfortable issue for ordinary users.

4.1.3 Cache Flushing and Invalidation

In Linux systems, Virtual File System (VFS) involves with management of caches for repeatedly-accessed file and directory nodes. A user space application sends requests through VFS and looks for cache first and then the actual file system and block device. For a typical VFS in Linux, it keeps inode cache, directory cache and buffer cache. These caches are important because at the point of time that the user device switches the lease, the original lease-holder has to flush the cached data to file system and block device, and the lease requester should invalidate its hold cache. Without cache flushing and invalidation, the user process has the chance of accessing the out-dated data instead of the synchronized contents.

4.1.4 Data Entity Information Book-Keeping

The UnityFS needs to keep a mapping between each DE and its blocks on whatever block device the file system is residing on. The reason why this is necessary is because from the view of Unity block store, there is no concept of "file", only DEs and block IDs. When we are building a file system, there needs to be a module that translates between block IDs and files/DEs, keep their relationships in a book somewhere for reference. The requirements of the functionalities of this DE book-keeping module should at least include the following:

- When there is an update to an existing DE, the file system should keep a record of which block(s) corresponding to the DE has been modified. The record can be either centralized gathered as an index, or distributed stored as attached to each file or inode. Whichever option is selected for keeping DE-to-blocks mappings, there needs to be an algorithm in the file system to search inquired values of the mappings efficiently.
- Given a block ID, the file system has to be capable of providing a traceback to a specific DE that corresponds to it.
- The DE book-keeping information should be able to be synchronized among the Unity system, including other client devices and the cloud node.

To give an example how the DE book-keeping is used, we suppose that there are two client devices A and B, each has the UnityFS running, and cloud node C, which has the *coordinator* on it. At first, A is holding the lease for a DE that corresponds with file F . Now B wants to open file F, it should first check with cloud node C to obtain the lease of that file. However, the coordinator process does not have an idea what the file is, so the DE book-keeping module on B has to translate from file F to a tuple {DE ID, Block ID} and send the coordinator on C. Then C asks A to release the lease the DE, and the book-keeping module on A translates the request from C to a the file F it has to release.

We could use several different ways to implement the DE book-keeping functionalities. We will discuss them in the next section where we introduce the candidate architectures.

4.2 Candidate Architectures

According to the above four requirements that UnityFS needs satisfy, one could design many different system architectures. Here we choose three representative architectures, describe how they are structured and work, then compare their pros and cons.

4.2.1 User Space File System

We can make use of the FUSE (File System in User Space) mechanism that allows a non-privileged user to create a file systems without having to implement a file system kernel module. The architecture depends on a FUSE kernel module in the kernel and a FUSE library in the user space. The user space file system communicate with the kernel module through glibc. The whole system architecture can be summarized as in Figure 4.1.

It is a natural and intuitive way to put the main functions of the UnityFS in user space. With the help of FUSE kernel module and user space library, the call to a file in the file system will be delivered to

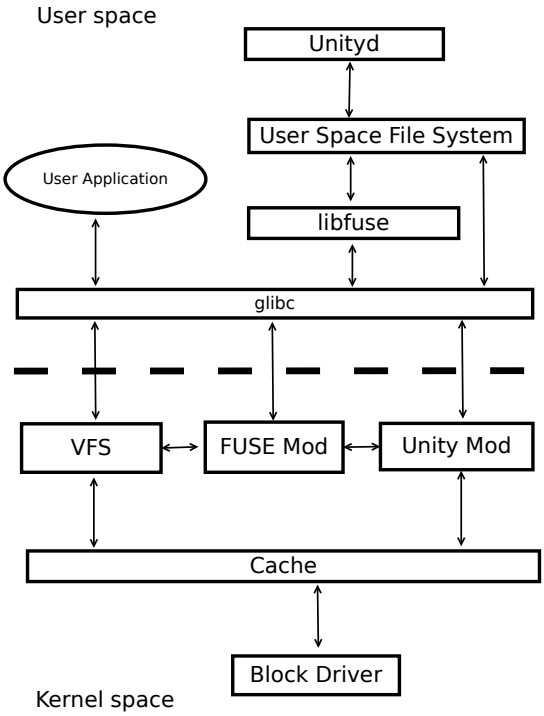

Figure 4.1: Candidate architecture 1: User space file system

the interface of the virtual file system (VFS) in kernel. The kernel module will direct the corresponding system call to VFS to the user space file system functions, executing what they have to do, and follow the coming way back through the kernel module and VFS to the caller.

This architecture can satisfy the four requirements we have previously mentioned in 4.1:

(1) `Basic file system features`: According to the basic functions provided by FUSE, the user space and FUSE kernel module can process all file system related calls.

(2) `Blocking read/update`: All read/write related operations get to the FUSE user space file system. If the action of blocking is requested, the functions that takes cares of these calls can hold the request and push them into a job queue, then decide what to do with them.

(3) `Cache flushing and invalidation`: When primary switch happens, the message can be passed through FUSE kernel module to the Unity kernel module. With the help of the Unity kernel module, it can access the buffer cache to force it to be flushed and invalidated.

(4) `DE book-keeping`: The FUSE file system in user space can keep the DE mapping file, and do the translation as needed.

Most of the implementation for this architecture can be done in the user space. However, in order to deal with the cache problem for lease-switching case, an additional kernel module is also needed to flush and invalidate the inode cache, directory cache and buffer cache. When lease-switch happens, the user space file system needs to notify the Unity kernel module to flush or invalidate the caches.

4.2.2 Kernel File System

The other architecture is to push the main functions of the Unity file system to the kernel, making it an independent kernel module as in in Figure 4.2.

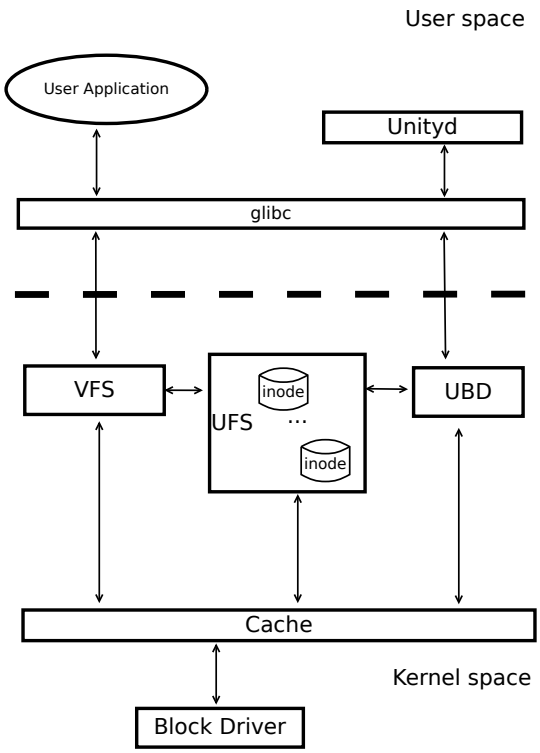

Figure 4.2: Candidate architecture 2: Kernel file system

The architecture of kernel file system can meet the four requirements as well:

(1) `Basic file system features`: With a file system kernel module that interacts with VFS, all basic file system calls can be supported like any other file system like FAT and EXT4.

(2) `Blocking read/update`: The blocked file system calls can be handled inside the kernel file system, with the help of wait queue in kernel.

(3) `Cache flushing and invalidation`: When primary switch happens, we could modify the Unity block device in the kernel to support buffer cache flushing.

(4) `DE book-keeping`: We have not discussed about how to record the DE mapping information, which can be done either differently or similarly as in a user space file system. Here we introduce two solutions:

- `A`: keeping the DE mappings to the extended file attributions (xattr) in each file.
- `B`: making the book-keeping functions in user space and keeping one mapping file.

Option A

A data entity is by default corresponded to a file or a directory in the file system. A file or a directory stores all their information in an inode data structure. Along with all the other file meta-data stored in inodes such as file name, file size and access time, the DE-blocks mapping information can be stored in the extended attributes in the form of name-value pairs.

The kernel module of the file system also needs to communicate with user space daemon process. It could be done in the same way as how UBD module sends to its user space server.

Option B

We provide another way to keep the file system as a kernel module, but this time we are not using extended file attributions. Instead, all DE mapping information is recorded in special file which is automatically updated.

In this architecture, the file system kernel module takes user's calls from user space, and passes the data block information to user space daemon process via UBD module. All book-keeping work is done in the Unityd process. .

4.3 Architectures Comparison and Implementation Plans

Considering the three candidate architectures, each from which has its own benefits and drawbacks. The architecture based on FUSE support has the advantages of portability over platforms, feasibility for deployment and all the other benefits including easiness to debug, though Linus Torvalds insists it cannot be more than a toy [41]. The additional kernel user space switching can unfavorably downgrade the performance because the block information still needs to be retrieved from the Unity block device, especially when the frequency of accesses to the file system is high. Another reason is that during lease-holder switch, in user space file system, it is hard to block all file system calls right away when they are made to the kernel. Moreover, developing a file system with FUSE cannot reduce the work in kernel space for buffer cache flushing and invalidation, which means we have to bear with performance loss for moving file system to user space without enjoying the benefits of having easier development.

For the two kernel file system architectures, the only difference is where to store the DE mapping information. It might be a good approach to keep such information distributed to each file's extended attribution because no extra space and structure for book-keeping is needed, all mappings are naturally attached to the file itself. However, without a centralized index, it can be too time-consuming to find a DE from a certain block ID– all the files might need to be accessed for the traverse algorithm over the file system.

From the reasons above, we finally choose the file system architecture of Kernel File System B: make the file system in the kernel and keep the DE information in a centralized fashion in user space.

4.3.1 Road-Map for New Features Adding to the Base Unity System

Reviewing what we have got in the base Unity system by referring to Figure. 3.1, we can draw a plan to add new features on top of that. There are four major features we need to insert from the architectural point of view:

1. In the base Unity system there is only one DE mapped for the whole block device. We need to conceptually split the single DE to multiple DEs, by making each DE mapping to a file in the file system.

2. The original UBD user space server thread, which is in Unityd daemon process, needs to handle multiple DEs. We need to add a series of functions to it in order to book-keep all DEs and their mappings.

3. The UBD kernel module needs to be aware of the existence of multiple DEs, when the UBD thread in user space pass the DE ID to it.

4. Besides the basic file system functionalities, the kernel file system also needs to pass file information to the user space via the new UBD kernel module.

Most file systems have the feature of journaling, which means all actions performed on the file system data are tracked and saved in a journal. The operations on the metadata of files are treated as transactions, that are first saved in a journal, then actually performed. The journaling feature mainly helps with retaining file system consistency and executing consistency checks in case of the events like system crash. The UnityFS does not have the journaling designed within the file system module, and itself cannot provide more consistency guarantees than other non-journaling file systems. However, because the base Unity system has got the feature of the fail-safe guarantee as we mentioned in 3.4.1 by keeping old versions of data blocks distributively on multiple clients, it is capable of retaining the consistency on block level. The coordinator on cloud node keeps DE replication status information, and it is aware of where the old versions of block replications are stored in case of a failure happens, so that the data blocks can be reverted to a same old version consistently.

Once the above four major changes are made, it would be possible for the UnityFS to support concurrent file operations. Here we give a specific example of concurrent file operations:
Client A want to read from file `F1`, while Client B wants to write to a different file: file `F2` concurrently. The VFS on A gets a `read` request to `F1` from user application, and sends to the kernel file system module. The file system passes `F1`'s information through UBD kernel module to the user space DE book-keeping thread, which later translates the file name to a DE ID=1 and requests lease for this DE from the coordinator in cloud. After getting the lease, the UBD user space server thread sends the contents to be read via UBD kernel module to the kernel file system. Now comes the concurrent file operation: the user application on B sends the `write` request to `F2`. The request goes through the same pathway on B to the DE book-keeping thread in user space. The translation result is that `F2` maps to the DE ID=2, and no other client is holding that DE, lease granted immediately. In this way, while A is still reading contents from `F1`, B could write to `F2` concurrently.

The four major changes will be discussed with other detailed issues and implementation techniques in the next chapter.

Chapter 5

Implementation of Unity File System

Based on the candidate architecture: Kernel file system A that we have chosen in the last chapter, our prototype of UnityFS is implemented in three components:

1. A UFS file system kernel module that takes calls from user applications and interacts with kernel VFS. Minix is one of the earliest implemented file system that works with Linux. We choose Minix $v3$ on Linux as our base kernel file system module in our prototype because it is compatible with POSIX and is light-weighted, only 5K lines of code. Comparing to EXT3/4 journaling file systems, the Minix $v3$ is more straight-forward in the way of processing VFS calls. We have modified/added 2K lines of the source code to make it work with Unity system.

2. A UBD as a kernel module that emulates a Linux block device on which the UFS is mounted. UBD is built on top of Linux 3.2.14 kernel's `nbd` module. We have added a series of data structures and functions to make it support flushing the buffer cache. We also make the kernel module capable of passing DE information between UFS kernel module and Unityd, in order to support the DE book-keeping functions in user space. About 150 lines of code have been modified/added for the new functionalities for UBD.

3. A Unity daemon process running in user space that communicates with the kernel file system and the coordinator process. The DE book-keeping module is implemented inside the Unityd by modifying the original `nbd-server` to add related functions for managing DE mappings and calling Unity library interface functions. We have modified/added 1K lines of code to the original nbd-server code, which is about 3K lines.

We illustrate the structure of the implemented system in Figure 5.1, based on the original Unity block store system in Figure 3.1. The changes made to the base Unity system is marked in the figure according to the road-map section in Chapter 4.3.1: the shaded components with solid lines contains works added to the base Unity system and is contributed by this thesis. The following sections will describe implementations of each components in details first, and then give examples for how typical file system operations work.

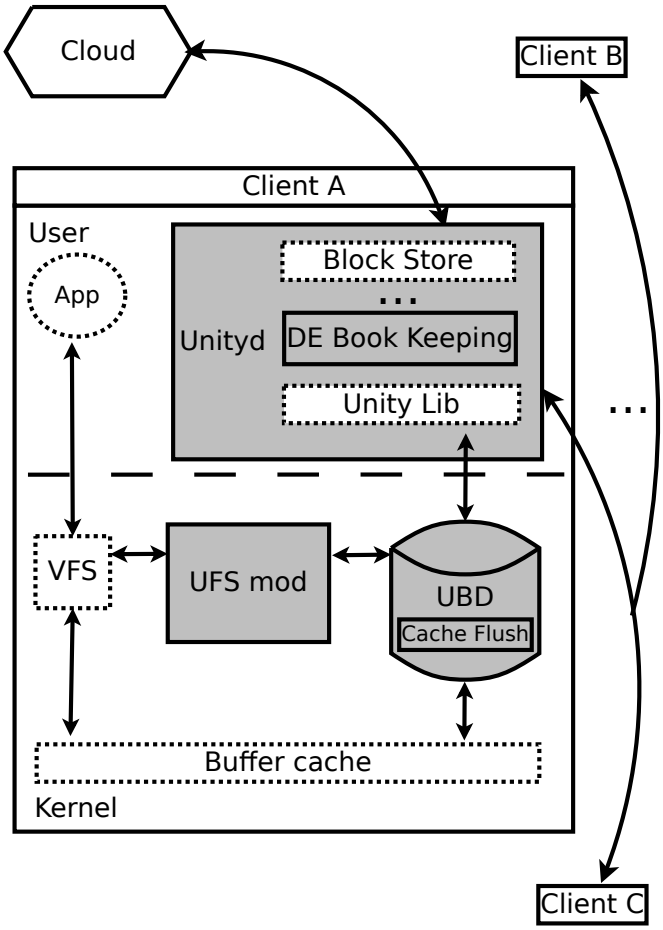

Figure 5.1: Unity file system structure

5.1 Unity Block Device

The Unity block device is built on top of the nbd kernel component of Network Block Device in Linux kernel 3.2.14, which is the upstream kernel source of Ubuntu 12.04 LTS. It runs as a kernel module that prepares block requests in the form of 4KB block to the Unity daemon process in user space. These requests are directly taken from the local file system on the same machine, which is the UFS kernel module. Another important job that Unity block device does is to allow the Unity daemon process to send a message for flushing kernel's buffer cache before a lease-holder switch. The flush of the buffer cache has to be done to guarantee consistency over the swap of devices because the above layer would never have known if the buffer cache is still the newest.

The "`ubd.c`" ("`ubd.o`" as a module) kernel driver is given a socket file descriptor to Unity daemon

Name	Type	Location	Description
__ubd_ioctl()	func	ubd.c	driver function that processes ioctl calls, called by ubd_ioctl() after error checks
blk_queue_flush()	func	blk−settings.c	block layer cache flush
sock_xmit()	func	ubd.c	sending or receiving message via socket
UBD_DISCONNECT	cmd	ubd.h	disconnect with client
UBD_CLEAR_SOCK	cmd	ubd.h	clear socket to user space
UBD_SET_SOCK	cmd	ubd.h	set socket to user space
UBD_SET_BLKSIZE	cmd	ubd.h	set size of block
UBD_SET_SIZE	cmd	ubd.h	set byte size
UBD_SET_TIMEOUT	cmd	ubd.h	set connection timeout
UBD_SET_FLAGS	cmd	ubd.h	set flags
UBD_SET_SIZE_BLOCKS	cmd	ubd.h	set byte and inode size
UBD_DO_IT	cmd	ubd.h	to start driver thread
UBD_CMD_FLUSH	cmd	ubd.h	to translate from REQ_FLUSH
UBD_FLAG_SEND_FLUSH	flag	ubd.h	flag to send flush
REQ_FLUSH	flag	blk_types.h	block level flag to flush
ubd_request	struct	nbd.h	structure of nbd request
ubd_send_req()	func	ubd.c	send ubd request to user space
do_ubd_req()	func	ubd.c	handle ubd requests from queue
ubd_forward()	func	ubd.c	pass info from FS via socket to userspace

Table 5.1: Key functions and structures in Unity block device source

process via an ioctl call. The driver is started by UBD_DO_IT ioctl call that only returns upon socket connection to the user space terminates, instead of returning immediately like most UNIX ioctl calls. The modifications we have done to the original code are:

(1) Adding UBD_CMD_FLUSH command to "linux/ubd.h" and related source code to "ubd.c" in order to support flushing the buffer cache. Because the NBD kernel driver does not support the linux writeback cache control, when the file system needs to force data out to a non-volatile storage by functions such as sync(), fsync() and umount(), there is no guarantee that the cache is flushed. In this case we have to add a new command to be translated from REQ_FLUSH flag. The translation works as a chain: In the function __ubd_ioctl(), the ioctl UBD_SET_FLAGS will receive the flags. In the command UBD_DO_IT, we have linked UBD_FLAG_SEND_FLUSH to the function blk_queue_flush() which has the second argument of REQ_FLUSH. In this way, the REQ_FLUSH is translated to UBD_CMD_FLUSH when it is an active flag.

(2) Changing the list of types of UBD_REQUEST. The UBD can be made to send update request to the user space, so that the Unity daemon process can learn that it is the time to update DE book keeping information to the file. This is useful when the Unityd keeps previous DE information changes in memory and now needs to write it to the DE configuration file. Accordingly modifications are made to related functions in UBD kernel module functions such as ubd_send_req() and do_ubd_req().

(3) Forwarding DE information from UFS kernel module to Unityd in user space. We add a function ubd_forward() which can be called in the file system kernel module for passing information such as DE and block ID to the user space daemon process via the same socket as the one for communications between UBD and Unityd.

In the end, we provide a list of core functions and structures in "ubd.c" and "linux/ubd.h" in Table 5.1 for reference.

5.2 UFS Kernel Module

The UFS kernel module is built by modifying Minix $v3$ file system on Linux. The main job of this module is to maintain a mapping between file names and DE IDs and a mapping between disk blocks and offsets into those DEs. When the UFS receives an I/O request, it translates the request into a request for a DE and block ID, and passes this information to the UBD kernel module. The UBD will then forward the information to the Unity daemon process on user space.

All file system operations can be divided into two categories: reading from a device, or writing to a device. For operations that only read data but not modify contents (except MAC time of a file), they just tell the Unityd to obtain the leases to according DEs and read what they have to read. For one operation that has to do some writing to a device, it should also make sure through Unityd that it has obtained the lease first, then send content with all arguments for writing. The difference is that if a writing failed, it should not let Unityd update the DE book keeping file. Since the Minix $v3$ file system has already set up a framework for a basic usable file system, here we only list the essential functions in the file system we modified in order to make the file system work for processing all DE information.

- `ufs_llseek()` [read]. Given a file descriptor, it repositions the offset of an open file to SEEK_SET (beginning of file), SEEK_CUR (current position of file), or SEEK_END (end of file). It returns the resulting file position. Considering the function is never called before a file is opened, we do not make a specific change for Unity on it, only re-wrap it with `generic_file_llseek()` for logging.

- `ufs_read()` [read]. Read from a file.

- `ufs_write()` [write]. Write to a file.

- `ufs_fsync()` [write]. The function calls `generic_file_fsync()` with a pointer of file, offset of start and end.

- `ufs_splice_read()` [read]. The function originally aims at reading data from pipe and avoiding a round-trip to the user space. However, for the UnityFS it needs to obtain the lease and goes to user space anyway, so we call an ordinary `ufs_read()` function instead.

- `ufs_setattr()` [write]. It modifies the `dentry` pointed to an inode structure. The file metadata changes will also send through UBD to Unityd and update the DE book keeping file. The inode can be found by the `dentry` via function `ufs_inode_by_name()`.

- `ufs_getattr()` [read]. Similar to `ufs_setattr()`.

- `ufs_new_inode()` [write]. Create a new inode, and update the DE book keeping file by adding a new entry in user space via Unityd.

- `ufs_set_inode()` [write]. Update to an existing inode, similar to `ufs_new_inode()` but it does not create any new entry in DE book keeping file.

- `ufs_mknod()` [write]. Make a new file. A new inode is created by `ufs_new_inode()` function. The `ufs_mknod()` function will not pass anything to Unityd because either `ufs_new_inode()` or `ufs_set_inode()` will be called inside this function. Other file system functions will also call `ufs_mknod()`, for example, `ufs_create()`.

- `ufs_create()` [write]. Call `ufs_mknod()` directly, create a new file. This function is wrapped for kernel logging purpose.

- `ufs_mkdir()` [write]. Call `ufs_new_inode()` to create a new inode. This function is wrapped for kernel logging purpose.

- `ufs_rmdir()` [write]. Removes a directory by calling `ufs_unlink()`.

- `ufs_symlink()` [write]. Create symbolic link, which is essentially an inode. The function calls `ufs_new_inode()` and `ufs_set_inode()`.

- `ufs_readdir()` [read]. Call `fill_dir()` at the end, given inode number and offset as arguments.

- `ufs_delete_entry()` [write]. Remove an inode, and update to the DE mapping.

- `ufs_minix_unlink()` [write]. This function does a `ufs_find_entry()` first, if exists, calls `ufs_delete_entry()` to remove the DE entry. Otherwise does nothing.

- `ufs_rename()` [write]. The rename function combines creating a new inode and deleting an old one.

5.3 Unity Daemon Process

The Unityd has several components as described in Chapter 3. The components that are directly related to the UFS kernel module and Unity block device are two: the library interface and the modified nbd-server thread that takes message from Unity block device in kernel space. The modified nbd-server thread uses API functions provided by the library interface to communicate with the controller thread in Unityd process.

5.3.1 Unity library

The Unityd library exposes a simple API to the client consisting of 7 library calls and 1 callback function that the client must provide as described in Table 5.2. The first 4 API calls are only used during startup and shutdown of the device and the remaining 3 calls are used for creating DEs, reading from DEs and writing to DEs. The two call back functions are used by Unityd to notify the client lease-holder switch events. The `callback_revoke_lease` call back is called when the client is about to lose the lease. In general, this should cause the client to flush any writes it is caching to Unityd so that they can be sent to the cloud provider. The client should complete these flushes before returning from this callback function, which is guaranteed by the UBD kernel driver. If the client calls `read` or `write` on a DE it doesn't have the lease for, Unityd library will automatically acquire the lease before returning from the API call.

5.3.2 DE book-keeping thread

The modified nbd-server thread is the one that manages all the DE information in a file. Referring to the definition of DE in 3.1, a DE book-keeping file records the information of each DE as the structure in Table 5.3. Each entry of the table has a unique DE number, under which there is information about the DE in the form of file pathname, inode number and a list of block IDs.

The thread reads/writes through API calls provided by the library. We put it at the client-side and the client itself should be responsible for assigning unique IDs to DEs when they are created. This is

API Call Prototype	Description
init (ul_config_t conf)	Initialization function taking various parameters. Called before starting Unityd. Client may choose to create DEs after initialization and before starting Unityd.
cleanup (void)	Clean up function before disconnecting device from the Unity cloud.
controller_start (pthread_t *thread)	Starts up Unityd.
controller_stop (pthread_t thread)	Stops the Unityd. Should be used with clean to terminate
create_entity (u_int64_t ID, u_int64_t entity_size)	Used to create a new DE of specified size and ID. The client must ensure that it is called with a unique ID every time.
read (u_int64_t ID, u_int64_t blockID, int offset, char *buf, int size)	Reads data of specified size into buffer from offset within a particular block. Will always read the latest version block for a particular block ID.
write (u_int64_5 ID, u_int64_t blockID, int offset, char *buf, int size)	Writes data of specified size from buffer to offset within a particular block. This will create a new version of the block.
callback_revoke_lease (u_int64_t ID)	Callback function used to notify client to prepare for the lease-holder switch for the DE specified by ID.

Table 5.2: API calls in Unity Library. There are two types of API calls, library calls made by the client and callback functions called by Unityd, which the client must supply.

DE ID 0
file pathname inode number block ID(s)
DE ID 1
file pathname inode number block ID(s)
...
DE ID 500
file pathname inode number block ID(s)
...

Table 5.3: The DE book keeping file structure

because it is not safe to trust the coordinator to do this since it could return the same ID for two different requests from different devices and in this way, obtain two different signed updates for the same DE and block ID. To ensure that clients always pick a globally unique ID, we leverage Unity itself. A special *bootstrap* DE (DE ID 0 in our prototype) is used to maintain the largest used DE ID. When allocating a new DE, clients atomically read and increment the value stored in the bootstrap DE and use this value is the ID for the new DE. The consistency and fail-safe property of the DE ID 0 like all other DEs, is guaranteed by the underlying base Unity system.

5.4 File System Operation Examples

Having the implemented prototype UnityFS with functionalities of each component, it should be necessary to have a flow of procedures explaining how the major operations in the UnityFS work. Under this purpose, in the following part of this chapter we give a few examples on how typical file system works in the system.

We pick three representative system calls to elaborate the work flows of UnityFS: `read` (Algorithm 1), `write` (Algorithm 2) and `unlink` (Algorithm 3). The file system operations `read` and `write` are a pair of VFS calls to access file contents given `fd` and `offsets`. To be more specific, `read` gets a part of file content, does no changing (except MAC time of inode metadata) to the file system, and nothing should be changed to the DE mappings after the operation of `read` is done. On the other hand, `write` adds/changes contents to a specific file stored in Unity block store, it also needs to update the DE book-keeping records. The operation `unlink` is more complex comparing to the previous two: it might need to deal with `dentry`, it might have to calculate/change link count to an inode, and remove the inode. According changes also needs to be made to the DE book-keeping record if there is a DE to be removed.

The three system calls can cover all types of behaviors in the interactions between each component of UnityFS: UFS kernel module passes DE information to Unityd via UBD for inquiry/update the DE mapping records, UBD sending messages to Unityd asking for lease, Unityd tells UBD to flush buffer cache, *etc.* With illustrations of the flows of the three file system operations, it is confident enough to

derive a conclusion that all other file system operations can be completed in similar ways.

```
1  UFS gets read system call from VFS on client A;
2  if fd exists then
3      if client A holds the lease then
4          read directly;
5      else
6          call UBD to send message to Unityd for lease, block read/update to the file;
7          Unityd looks up fd to get file name, translates to DE ID, sends to cloud;
8          if clouds finds DE lease is taken by client B then
9              cloud sends message to client B to release the lease;
10             UBD flushes buffer cache on client B;
11         else
12             obtain lease directly;
13         end
14         get contents from Unityd, passed through UBD;
15         UFS gets return, unblocks read/update to the file;
16     end
17 else
18     go to error handler;
19 end
```

Algorithm 1: Example for file system operation: `read`

```
1  UFS gets write system call from VFS on client A;
2  if fd exists then
3      if client A holds the lease then
4          write to UBD;
5          UBD sends contents and fd, size and offset to Unityd;
6          Unityd look up book-keeping file to get corresponding DE and block ID, update content;
7          UFS gets return;
8      else
9          call UBD to send message to Unityd for lease, block read/update to the file;
10         Unityd looks up fd to get file name, translates to DE ID, sends to cloud;
11         if clouds finds DE lease is taken by client B then
12             cloud sends message to client B to release the lease;
13             UBD flushes buffer cache on client B;
14         else
15             obtain lease directly;
16         end
17         send contents, fd, size, offset to Unityd, passed through UBD;
18         Unityd looks up book-keeping file to get corresponding DE and block ID, writes to block
           store;
19         UFS gets return, unblocks read/update to the file;
20     end
21 else
22     go to error handler;
23 end
```

Algorithm 2: Example for file system operation: `write`

```
1  UFS gets unlink system call from VFS on client A;
2  UFS gets inode number from dentry argument;
3  if client A holds the lease then
4  |  execute ufs_find_entry(); passes inode number to Unityd via UBD;
5  |  Unityd looks up book-keeping file to get corresponding DE, gets update to the DE entry;
6  |  UFS gets return, execute ufs_delete_entry(), decrease link count;
7  else
8  |  call UBD to send message to Unityd for lease, block read/update to the dentry;
9  |  Unityd looks up the DE ID, sends to cloud;
10 |  if clouds finds DE lease is taken by client B then
11 |  |  cloud sends message to client B to release the lease;
12 |  |  UBD flushes buffer cache on client B;
13 |  else
14 |  |  obtain lease directly;
15 |  end
16 |  execute ufs_find_entry(); passes inode number to Unityd via UBD;
17 |  Unityd looks up book-keeping file to get corresponding DE, gets update to the DE entry;
18 |  UFS gets return, execute ufs_delete_entry(), decrease link count, unblocks read/update to
   |  the dentry;
19 end
```

Algorithm 3: Example for file system operation: `unlink`

Chapter 6

Evaluation

We evaluate three aspects of the UnityFS. First, we show the advantage of supporting multiple DEs in the file system by executing concurrent file system operations. Second, we evaluate the performance overheads introduced by the UnityFS. Finally, we measure Unity's use of bandwidth against NFS, NBD and Dropbox.

In the evaluations, we use three machines to represent three user clients: Client-1 with 3.4GHz Intel i7-2600 CPU and 16GB memory, Client-2 with 2.83GHz Intel Core2 Quad CPU and 4GB memory and Client-3 with 2.13GHz Intel Core2 Duo CPU with 2GB memory. The coordinator is always running on Client-1. Several workloads including micro-benchmarks and real-world jobs are selected to evaluate different aspects of the UnityFS.

6.1 Advantage of Supporting Concurrent Operations

The most important reason that we build UnityFS is to support concurrent operations to an underlying Unity block device. In this section of evaluation, we aim at showing the advantage of having a file system supporting multiple DEs by comparing the performance between using a single-DE UBD with base Unity block store, and running the same workload on UnityFS. The workload used is compressing the Linux kernel 2.2 with `gzip` at the same time on different folder on between 1-3 clients with the base Unity system only and the same settings but the clients running UnityFS.

We expect that under concurrent accesses to multiple files, running UFS will out-perform the same clients with UBD as a single DE. And due to the time consumptions on lease-switch, when the number of user clients increases, we expect the performance of UFS would be even better in comparison to UBD only.

From Figure 6.1, we can see that UBD performs extremely poorly as the number of user clients simultaneously accessing the DE increases. In many cases, we observed that the contention was so bad that a client would barely get an I/O operation done in between lease-holder switches. One of the clients runs slightly faster than the others in the single DE case. We found that this was the client that the workload was started on, and it signals the other clients to start their workloads with a call to ssh. As a result, this client likely was able to run exclusively on the DE for a little bit of time at startup and thus finished faster.

In comparison, because UFS maps each file to a different DE, this eliminates the false sharing and

virtually all lease-holder switches except to the single DE that stores the DE to file mappings. As a result, the performance of UFS remains relatively unchanged with the number of user clients. We also compare UFS with vanilla ext4 on a single node to get the baseline overhead of our file system. We can see that UFS itself imposes very little overhead over a standard local file system.

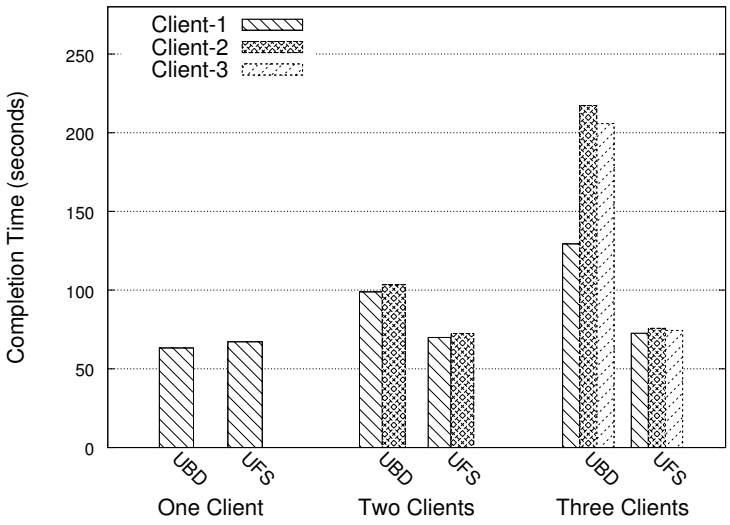

Figure 6.1: Compare Single DE with Unity file system

6.2 Performance Overheads

Any distributed file system would add performance overheads due to more components with interactions between them, network delays and extra operations for consistency. It is questionable that how significant is the overheads introduced to the original local file system, and what overhead contributes the most. So we measure these performance overheads by running the workloads on three scenarios we have chosen:

1. {Baseline FS} We run all the workloads on machines with unmodified Minix $v3$ Linux file system. In this setting, no Unity system is running in background, no network connection is enabled, so we only have this local file system running for getting a set of baseline numbers for further comparisons with other scenarios.

2. {Single DE} In this scenario, we treat the whole file system as a single DE. It means that when mounting the file system on Unity block device, the full block device is registered as one data entity. When a client accesses to any file in the file system, the Unity system regards it as accessing to the same DE.

3. {Multiple DEs} Each single file in the file system mounted on Unity block device is registered as a DE, so there can be multiple data entities in the file system. When two clients accesses two different files, they are accessing to different DEs, so both of the two clients can obtain the lease of the file they want simultaneously.

In the experiments, we instrument in each components of system, such as UFS kernel module, Unity block device and Unity Daemon process, to get the execution time of different operations. Four typical

times are calculated here: (1) New DE registration time, (2) Primary switch time, (3) DE handling time, and (4) Total run-time. Noting that for the baseline FS case we only count the total run-time, and for the scenario that Unity running on a single node, there is no lease-switch time calculated.

We expect that the lease-switch time be the main source of performance overhead, because when it happens, a client has to contact the coordinator on cloud for obtaining the lease. Moreover, there is chance that the client has to wait until the former holder of the lease to release, not to mention all incoming accesses to the same DE needs to be blocked. Even if all those operations can be completed within tens of milliseconds, when the frequency of concurrent operations is high, the overheads accumulate. The overhead of new DE registration should not be heavy if there is not a large amount of new files created. The DE mapping file handling happens very fast locally in user space, we expect the time consumption not to be significant. At last, as we hold our evaluations in a LAN environment, network delay is not considered.

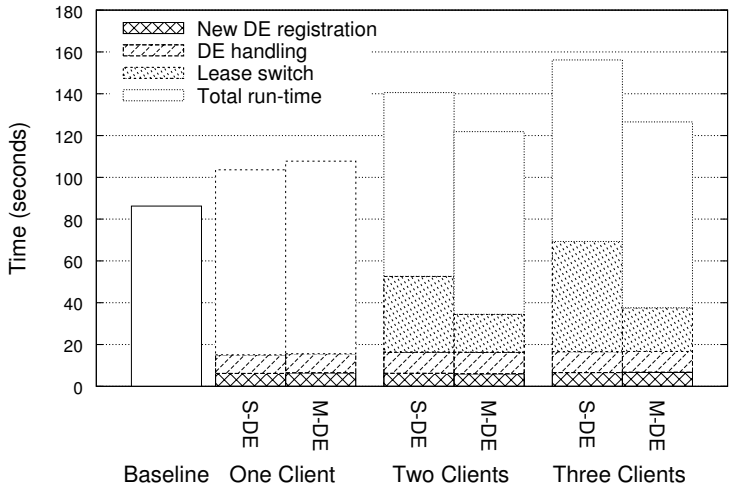

Figure 6.2: Measurement of performance overheads – Workload "Writing"

We collect results from workload 2 (Writing) (Figure 6.2) and workload 3 (Reading) (Figure 6.3). The workload 2 writes a 1024KB new file to the client file system, and then delete the file after the write is completed. After a short pause each time, the write-delete loop repeats 100 times. From the result, it shows that the "New DE registration" and "DE handling" time stay almost the same as the number of user clients grows, and most of the overheads comparing with baseline file system is contributed by "lease-switch". The workload 3 is reading repeatedly from a large file (256MB) after creating a new file at the first time. The results reflect the similar phenomenon as in workload 2, *i.e.*, "lease-switch" is the primary source of the overheads in the UnityFS. Noting that in both workloads, in the single client case the UFS is a little bit slower than single-DE UBD, this is because when having multiple DEs to manage, UnityFS has to spend more handling the DE book-keeping records. But this cost is much less than the lease-switching cost if the latter happens a few more times.

In realistic use scenarios, we can imagine that it is rare that a user is using two of her clients accessing to the same file at same time. Under this assumption, we believe that when the user is switching her client from one to another, she would be prepared for a short period of pause time allowing the process of lease-switch to happen.

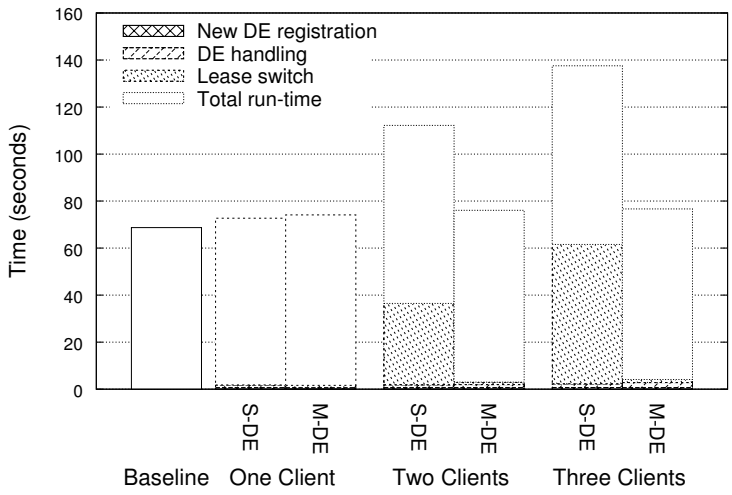

Figure 6.3: Measurement of performance overheads – Workload "Reading"

6.3 Use of Bandwidth

Existing distributed storage systems like NFS, NBD and Dropbox all consume network bandwidths. Unity system could not possibly provide security and durability protections without any extra costs, which means that for those added features, the whole Unity system including UFS may have to consume more bandwidth than other popular distributed system without these features. We want to show that the extra costs on network bandwidth of Unity system is tolerable.

We study how the Unity system uses network bandwidth in our two workloads, and compare the bandwidth consumption of Unity with NFS, NBD and Dropbox. For NFS and NBD, all data was placed on the server and only one client was used to run the workload. In Dropbox, data was uploaded to the Dropbox server from one client and when the upload was complete, a second client where the workload would be run was connected to the Dropbox service. Because Dropbox does not fetch block data as needed like NFS, NBD or Unity , we then waited for Dropbox to finish synchronizing data to the new client before starting the workload. We enabled Dropbox's LAN Sync feature, which allows clients to transfer data directly to each other if they are on the same LAN (which is true for our clients and network environment) without having to relay the data via Dropbox server.

With the help of D-Link DGS-1100 Gigabit EasySmart Switch, which supports bandwidth control individually limit upstream and downstream bandwidth on each port, we evaluate Unity on two network configurations:

- The "Home" configuration has unconstrained Gigabit network connected between all the three user clients.
- The "Mobile" configuration emulates a more typical network setup. Client-1 is a non-battery machine having high-end cable modem-like connections with 32Mbps upload bandwidth and 128Mbp download bandwidth. For two mobile nodes Client-2 and Client-3, each of them has a LTE-like connection with 16Mbp download and 8Mbps upload.

Table 6.1 gives the bandwidth measurements of workloads across the different systems and network

	Home		Mobile	
	WL "Writing"	**WL "Reading"**	**WL "Writing"**	**WL "Reading"**
NFS	12 / 245	772 / 25	6 / 233	695 / 12
NBD	10 / 237	734 / 20	4 / 209	688 / 11
Dropbox	12 / 241	709 / 32	6 / 229	673 / 12
Unity	292 / 266	804 / 34	298 / 260	812 / 15

Table 6.1: Bandwidth consumption for Unity, NFS, NBD and Dropbox in MBytes (Download/Upload)

types. We can see that from the results that the UnityFS is not as efficient in bandwidth as in other distributed file systems, largely due to the large control data overhead. However, we can note that the upload bandwidth consumption is not that bad – a bit worse than other file systems but comparable. We can conclude that the UnityFS pays some download bandwidth as price for added durability and security, and we believe it is acceptable.

Chapter 7

Conclusions and Future Work

This thesis presents Unity file system, a distributed file system which is built using Unity block store as base system. The contribution made by this thesis can be summarized as follows.

1. We design and implement a file system on top of Unity block store. The file system has a series of features that ensure it cooperates with the base Unity system, which include: (1) It has a kernel file system module mounting on top of UBD; (2) It can block read/write to the file until the cloud decided which client gets the lease of the file; (3) It has the UBD block device to flush and invalidate all the caches, for consistency of a file when its lease is switched to/from other clients; (4) It maintains a mapping between the DE ID, block ID and the file name, records it in a file which is managed by a user space thread in Unityd.

2. We demonstrate that having a file system supporting mappings between DE IDs, blocks and file names for the base Unity system can reduce the damage of false sharing problem happens when the whole block device is regarded as one single DE.

3. We study the performance overhead of distributed file system, give some insights which part may contribute the most to the total system overheads added to an original local file system. We also evaluate the network bandwidth usage of the full Unity system and compare it with NFS, NBD and Dropbox.

We have learned something during the work for this thesis. One thing is that it is important to plan a reasonable architecture carefully before rushing to implementation. There is an old saying that "A beard well lathered is half shaved." – An architecture that makes sense could save us a lot of time wasted in coding. Another technical learning is that we have broken a block device into many DEs (files), but we still have to book keep the DE information locally in a centrally organized file, which becomes the new scalability bottleneck.

There are also a few interesting problems related to the thesis. We put them in the future work and hopefully they can be solved (by me or by someone else) in the future:

- Now that each DE is fixed mapping with one file, but it could be a waste of resource of DEs and bring unnecessary overheads. Because a user could have a group of files she does not usually separately access on different devices, *e.g.*, a music file with its lyrics file, or a dvi file with its eps

figures. Those files could be more efficiently grouped and mapped to one DE. In other words, the DE mapping with files can be assigned with more flexible choices, or even dynamically.

- The current UnityFS does not support MAC time very well in terms of efficiency. Because each `read` operation also changes the file meta data such as the last access time. The meta data change cause according change to the DE book-keeping as well, which greatly increase the frequency of changes to the DE book-keeping records. A finer solution to deal with the MAC time stamp needs to be well considered.

- Since the book-keeping file could become a new scalability bottleneck, more work needs to be done to make arrangement for the DE mapping information. Possible solutions include making part of the DE mapping information distributed with inode.

- We have not included the user studies in this work. The behaviors of human beings when they are using the file system need to be studied to give developers some insights. For example, if users tend to access to a type of a group of files together exclusively on one client, we probably should make that group of files map to one DE to improve the efficiency – this could be possible after the dynamic DE mapping implemented.

Bibliography

[1] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *The 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[2] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "DEPOT: cloud storage with minimal trust," in *The 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[3] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: verification for untrusted cloud storage," in *CCSW*, 2010.

[4] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the sun network filesystem," in *USENIX*, 1985.

[5] R. Sandberg, "The Sun network file system: Design, implementation and experience," in *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.

[6] M. Brown, K. Kolling, and E. Taft, "The Alpine file system," *ACM Transactions on Computer Systems*, vol. 3, no. 4, pp. 261–293, 1985.

[7] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and performance in a distributed file system," in *Proceedings of 11th Symposium on Operating Systems Principles*, 1987.

[8] J. Howard, M. Kazar, S. Menees, and D. Nichols, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, 1988.

[9] M. Satyanarayanan, J. J. Kistler, and P. Kumar, "CODA: a highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, 1990.

[10] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the CODA file system," *ACM SIGOPS Operating System Review*, vol. 25, pp. 213–225, 1991.

[11] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, "RFS architectural overview," *Australian UNIX systems User Group Newsletter*, vol. 7, pp. 4–5, 1987.

[12] M. J. Bach, M. W. Luppi, A. S. Melamed, and K. Yueh, "Remote file cache for RFS," in *Proceedings of the Summer 1987 USENIX Technical Conference and Exhibition*, 1987.

[13] M. Hurwicz, "MS-DOS 3.1 makes it easy to use IBM PCs on a network," *Data Communications*, November 1985.

[14] P. J. Leach and D. C. Naik, "A common Internet file system (CIFS/1.0) protocol," Microsoft, Tech. Rep., December 1997, network Working Group Internet-draft.

[15] Microsoft Corporation, "Server message block (smb) protocol versions 2 and 3," Microsoft, Tech. Rep., Feburary 2013.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *The 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[17] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "CEPH: A scalable, high-performance distributed file system," in *The 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[18] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *The 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[19] D. Mazières and D. Shasha, "Building secure file systems out of byzantine storage," 2002.

[20] C. Cachin, A. Shelat, and A. Shraer, "Efficient fork-linearizable access to untrusted shared memory," 2007.

[21] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "Sirius: Securing remote untrusted storage," in *The 10th Symposium on Network and Distributed System Security (NDSS)*, Feburary 2003.

[22] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *The 2nd USENIX Conference on File and Storage Technologies (FAST)*, April 2003.

[23] K. Fu, "Group sharing and random access in cryptographic storage file systems," June 1999.

[24] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *The 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[25] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," in *The 2nd USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

[26] M. Kozuch and M. Satyanarayanan, "Internet ssuspend/resume," in *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, August 2002.

[27] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, D. O'Hallaron, A. Surie, A. Wolbach, J. Harkes, A. Perrig, D. Farber, M. Kozuch, C. Helfrich, P. Nath, and H. Lagar-Cavilla, "Pervasive personal computing in an internet suspend/resume system," *IEEE Internet Computing*, vol. 11, no. 2, pp. 16–25, 2007.

[28] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam, "The Collective: A cache-based system management architecture," in *The 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[29] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual appliances for deploying and maintaining software," in *The 16th Large Installation Systems Administration Conference (LISA)*, Oct. 2003, pp. 81–194.

[30] C. Sapuntzakis and M. S. Lam, "Virtual appliances in the collective: A road to hassle-free computing," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems*, 2003.

[31] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads," in *In Proceedings of the 2008 USENIX Annual Technical Conference*, 2008.

[32] S. L. Garnkel, "An evaluation of Amazon's grid computing services: EC2, S3 and SQS," Harvard University, Tech. Rep., August 2007.

[33] A. Khurshid, A. Al-Nayeem, and I. Gupta, "Performance evaluation of the illinois cloud computing testbed," University of Illinois at Urbana-Champaign, Tech. Rep., June 2009.

[34] A. Avetisyan, R. Campbell, I. Gupta, M. Heath, S. Ko, G. Ganger, M. Kozuch, D. O'Hallaron, M. Kunze, T. Kwan, K. Lai, M. Lyons, D. Milojicic, H. Y. Lee, Y. C. Soh, N. K. Ming, J.-Y. Luke, and H. Namgoong, "Open Cirrus: A global cloud computing testbed," *Computer*, vol. 43, no. 4, pp. 35–43, 2010.

[35] G. Singh, S. Sood, and A. Sharma, "CM – measurement facets for cloud performance," *International Journal of Computer Applications*, vol. 23, no. 3, pp. 37–42, 2011.

[36] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding personal cloud storage services," in *Proceedings of the 2012 ACM conference on Internet measurement conference*, ser. IMC '12, 2012.

[37] DropBox, `http://www.dropbox.com`.

[38] CloudSleuth, `http://cloudsleuth.net`.

[39] CloudHarmony, `http://www.cloudharmony.com`.

[40] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, "Robustness in the Salus scalable block store," in *NSDI*, 2013.

[41] L. Torvalds, "Re: [PATCH 0/7] overlay filesystem: request for inclusion," `http://lkml.org/lkml/2011/6/9/462`.