Bogdan-Cristian Tătăroiu

The BDFS Distributed File System

Computer Science Tripos Part II

Churchill College

May 13, 2014

Proforma

Name:	**Bogdan-Cristian Tătăroiu**
College:	**Churchill College**
Project Title:	**The BDFS Distributed File System**
Examination:	**Computer Science Tripos, 2014**
Word Count:	**11984**
Project Originator:	Ionel Gog and Malte Schwarzkopf
Supervisor:	Ionel Gog and Malte Schwarzkopf

Original Aims of the Project

The aim of the project is to design and demonstrate a working fault tolerant file system which distributes both file data as well as file system structure and metadata across multiple machines. Within the context of growing popularity of frameworks for shared-nothing distributed data processing running on large clusters of machines it becomes important to design systems which continue to operate properly in failure scenarios. Distributed file systems constitute the storage layer that lies at the foundation of such data processing networks. The main objective of this project was to eliminate a single point of failure which exists in the file system structure and metadata layer of the commonly used HDFS [1] file system and of the original GFS [2] file system on which HDFS is based on.

Work Completed

All core project goals have been completed successfully. A distributed file system providing replication-based fault tolerance support for file contents, metadata and directory structure was implemented. The file system is able to gracefully handle single or multiple node failures and redistribute workload over other machines. It is able to continue partial operation under network partition and recovers gracefully once the other machines have rejoined the group. The file system's performance was evaluated in both homogeneous and heterogenous environments running on physical machines part of a high speed home network and as virtual machines in

Amazon’s EC2 cloud service. The file system was shown to make use of network capacity well and, as desired, scales linearly in terms of performance with the number of machines dedicated to the cluster.

Special Difficulties

None.

Declaration of Originality

I, Bogdan-Cristian Tătăroiu of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In recent years we have seen an increasing interest in distributed shared-nothing data processing frameworks. This is motivated both by changes in the hardware market and in the software market. Reduced computer hardware costs and the advent of on-demand computing as a service platforms such as Amazon EC2[1] or Google Compute Engine[2] allow both individuals and companies to store and process larger and larger amounts of data over more and more machines. Additionally, there is an inherent shift towards parallel applications as a result of single threaded performance hardware limits being reached and of the subsequent need to move to increasingly more multi-core CPUs. Shared-nothing frameworks provide inherently parallelisable abstractions which means that one can increase overall performance and scale up the amount of data one can process simply by allocating more resources to the task rather than requiring any rethinking of the underlying algorithms.

One such framework, Hadoop, started off as a result of scientific papers published by Google detailing its internal systems. The first of these papers describes the Google File System [2], a distributed, fault-tolerant file system designed to run on large clusters of commodity hardware and to provide a solid foundation upon which data-intensive applications can be built. The next paper describes the MapReduce programming model [3], which enables processing data sets in a highly parallel manner by dividing the input data set into smaller subsets and modeling execution as a sequence of functional map and reduce steps that can be run in isolation on each subset of data. Finally, the BigTable [4] paper describes a distributed storage system built on top of GFS with support for structured data and high performance database queries.

Hadoop was developed internally at Yahoo, who at the time were seeing similar data processing needs to Google and sought to provide equivalent frameworks to its own developers in the form of HDFS [1], Hadoop MapReduce and HBase respec-

[1] `http://aws.amazon.com/ec2/`

[2] `https://cloud.google.com/products/compute-engine/`

tively. It has since been open sourced and adopted by many companies around the world as the canonical framework for data processing.

Through this wide adoption, a number of problems with each of the three components have been identified, originating both from lack of feature parity, with the Hadoop implementation trailing behind the Google systems, and from inefficiencies present in the implementation, poor architectural choices or changing use cases.

Since GFS/HDFS sits at the very lowest layer of these frameworks, it makes sense to try to address issues with it first. The major complaints I will be addressing are the presence of a single point of failure in the file metadata and directory structure layer and poor performance when dealing with a very large number of files. Both of these problems stem from the decision made when designing GFS to not distribute the metadata layer across multiple machines. A secondary complaint with the open source HDFS that will be ameliorated is that it achieves relatively low throughput compared to machine limits, which seems to stem from inefficient implementation rather than poor architectural design.

1.2 Project aims

The aim of the project is to design and implement a distributed file system from scratch, using the GFS/HDFS architecture as a reference starting point and providing a proof-of-concept solution to some of the limitations identified above.

A distributed file system allows one to access and store an order of magnitude more files and data than what would be possible on a single machine. This is achieved by dividing up the information stored within it and spreading it over a large set of machines, while providing users with an interface that makes it seem as if all of their data is stored in one place. It provides the familiar abstraction of a hierarchical directory structure rooted at one point, with files containing data being placed within directories.

The GFS and HDFS papers distinguish between their data layer, which deals with actual file data, and their metadata layer, which deals with the directory hierarchy and additional information about the directories and files, such as, for example, permission bits or modification times.

In terms of the file data layer, both GFS and HDFS partition files into independent fixed sized chunks. A file's contents is then represented in the metadata layer simply as a list of file chunk identifiers. The chunks are independently stored and replicated on a number of different machines.

Thus, GFS and HDFS provide a fault-tolerant solution for file data. However, information about the directory structure and file metadata is stored and managed by a single host within the cluster. This means that whenever that host goes down, the file system becomes completely inoperable. Additionally, all of this information is stored within the host's RAM for performance reasons which means that there is a fundamental ceiling on the number of directories, files and individual pieces of

data that can be managed by the file system. Finally, while GFS guarantees no data loss by replicating changes to other machines and thus allowing a replacement host to assume the responsibility of the master eventually in case of failure, HDFS only provides weak guarantees and can lose most recent changes as a result of failure.

BDFS aims to provide fault-tolerance for **all** layers of the file system without data loss and without sufferring a large penalty in terms of throughput on a single host failure while raising the limit on the number of objects that can be stored in the file system as well.

Clients connecting to a BDFS cluster may interact with it either via a client library or via an interactive prompt through which one can browse and perform changes to the file system. Figure 1.1 shows an interactive client connecting to a BDFS cluster and performing file system operations such as directory and file creation as well as file transfer to and from the local machine.

1.3 Related work

With the open sourcing of the Hadoop platform under the Apache Foundation and subsequent wide industry adoption there have been several companies (for example Cloudera [3] or MapR [4]) providing slightly or radically different implementations of the platform. These were either designed to make setup and management easier or they were optimised for specific use cases by implementing different MapReduce job schedulers, different HDFS data replication strategies or providing complete backwards compatible rewrites.

One strategy employed to fix the single point of failure within the metadata layer of HDFS (known as the NameNode) is to run two NameNodes in sync, with one of them standing by ready to jump in when the other one fails. This approach was employed by Facebook in 2012 under the name AvatarNode [5] and a similar approach also found its way into the second version of the Hadoop platform released shortly after the start of this project as HDFS High Availability over NFS [6].

MapR provided a solution in 2011 [7] to raising the limit on the maximum number of objects by effectively partitioning the file system into multiple independent regions that are managed using the same architecture as a normal HDFS cluster and providing a means of viewing all of the partitions as one consistent system. A similar approach which partitioned the file system into separate namespaces sharing the same data storage layer was implemented in the second version of the Hadoop platform in 2014 under the name HDFS Federation [8].

Google has provided some very limited information about the second iteration of its Google File System known as “Collossus” in a 2010 talk [9] stating support for

[3] `http://www.cloudera.com/`

[4] `http://www.mapr.com/`

a sharded metadata layer and for space efficient replication using Reed-Solomon error correction. [5]

Microsoft Research published a paper in 2011 on TidyFS [10], a simple and small distributed file system designed from scratch, which exploits the properties of the typical workloads observed when using distributed computation frameworks such as MapReduce to produce a clean, robust and simple design. It provides a fault tolerant metadata layer by employing a replication scheme based on the Paxos [11] consensus protocol.

I will further discuss the influence that this work has had on the design of BDFS in the following chapter.

[5] http://en.wikipedia.org/wiki/Reed-Solomon_error_correction

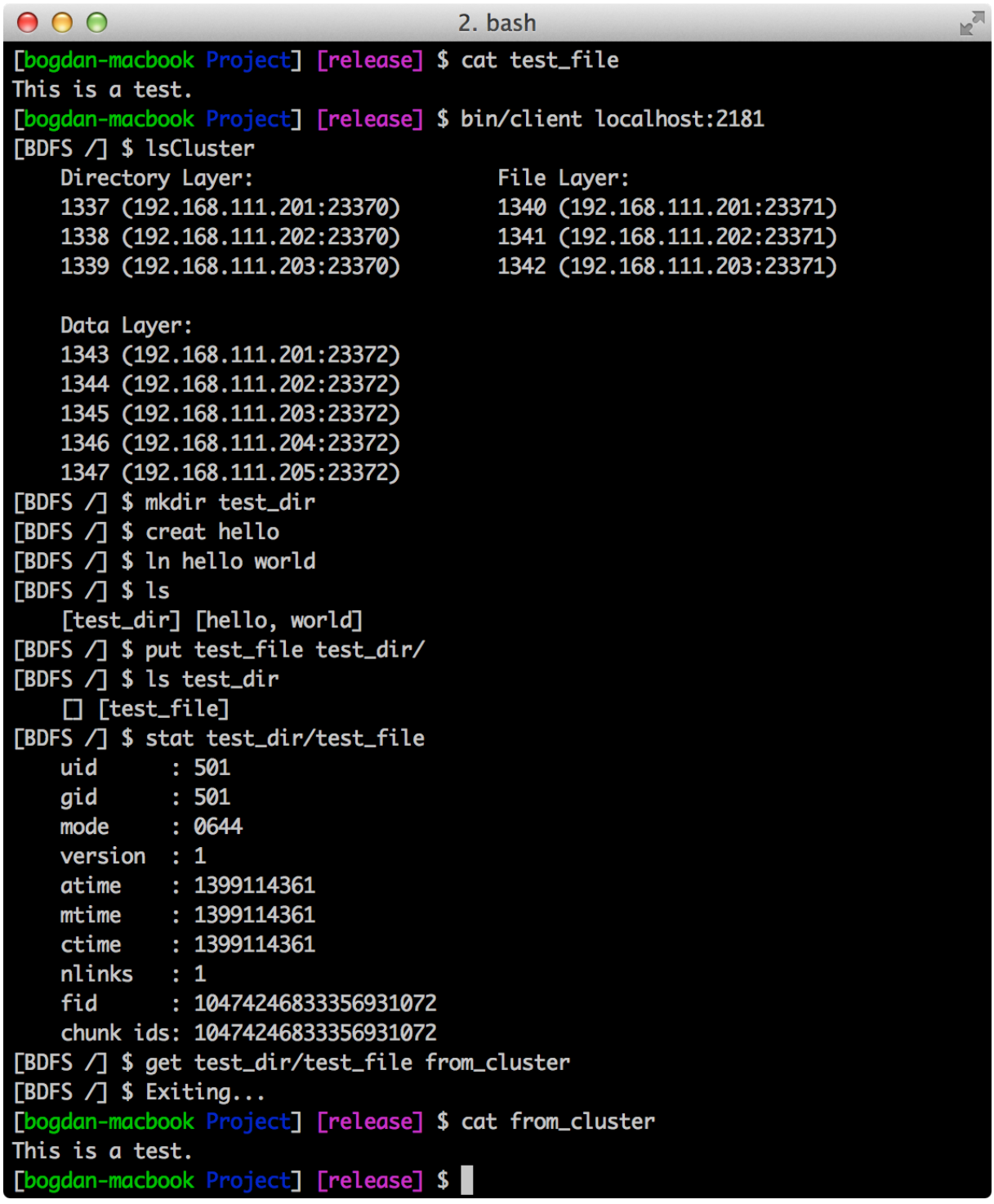

Figure 1.1: Interactive BDFS client in operation.
Figure shows a client connecting to a BDFS cluster composed of 5 machines, the first three of which store both file and directory metadata as well as data, with the last two storing only file data.
The figure shows a client creating a directory and a file, performing a hard link, listing directories, transferring a file from local storage to BDFS, fetching information about the new file and transferring it back successfully.

Chapter 2

Preparation

This chapter underlines the principal design decisions that were made before implementation occurred and illustrates some of the trade-offs made in design and implementation.

First, I recount the goals that the project tries to achieve in the form of a requirements analysis in section 2.1. Next, I look at a high level description of the design of the distributed file system and go over the principal division into modules in section 2.2. Following that, in sections 2.3 and 2.4, I look at the mechanism used for sharding and replication of data within each component of the file system and some of the difficulties involved. Next, in section 2.5, I look at ensuring locality of data that is inter-connected, but managed by different components. I then look over distributed consensus protocols and distributed transaction handling in sections 2.6 and 2.7. Finally, in section 2.8 I look over the software engineering tools, libraries and techniques employed to ensure smooth implementation of BDFS.

2.1 Project goals – requirements analysis

As discussed in section 1.2, the aim of the project is to design and implement a distributed file system providing a proof-of-concept solution to some of the limitations of the metadata layer of the GFS and HDFS file systems.

This dissertation's primary contribution is thus the design of a fault tolerant sharded metadata storage layer that eliminates the single point of failure present in GFS/HDFS and that raises the limit on the maximum number of objects (be they files, directories or data chunks) that can be handled by the file system.

From the core success criterion mentioned in the original project proposal, I extract the following must-have behavioural requirements:

1. The file system must distribute file contents, metadata and directory structure over multiple machines.

2. The file system must provide replication for file contents, resulting in fault tolerance of single node failures.

3. The file system must be able to read existing files or directories, create new files or directories and append to existing files.

In addition to these, I add the following strongly-want behavioural requirements related to the metadata layer of the file system as **extensions**:

1. The file system must provide replication for file metadata and directory structure, resulting in fault tolerance against single node failures.
2. The file system must provide strong consistency guarantees for metadata and directory structure operations, meaning that successfully executed operations cannot be lost due to node failures.
3. The file system's design must enable full or partial operation under multiple node failures and gracefully recover once nodes rejoin.
4. The file system's design must enable partial operation under network partition conditions and gracefully recover once the network is restored.

The **CAP theorem** states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees: consistency – all nodes see the same data at the same time, availability – a guarantee that every request receives a response and partition tolerance – the system continues to operate despite arbitrary message loss or failure of part of the system. BDFS aims to ensure **consistency** and **partition tolerance** sometimes at the expense of **availability**.

2.2 High level design overview

When designing the distributed file system, I decided to go for a modularised approach with each component being as loosely coupled to the others as possible. I identified four major components, as follows:

- **The cluster coordination service** is responsible for monitoring the status of all machines within the cluster, detecting any host failures, managing joins of new machines and managing rejoins of machines that previously failed. Its primary role is to provide an overview of what machines are present within the cluster, which areas of the file system they are responsible for and to ensure that all machines within the cluster share this consistent view. It provides both a means through which new machines or new clients may learn the cluster composition and a means through which existing machines in the cluster are notified of any changes to the cluster.
- **The file data layer** sits at the very bottom of the file system stack. It stores and manages file data chunks that are of bounded size and provides a means of allocating unique identifiers to newly created chunks. It manages mutual exclusion of clients issueing write requests and provides support for GFS-style concurrent writers by serialising concurrent append requests.

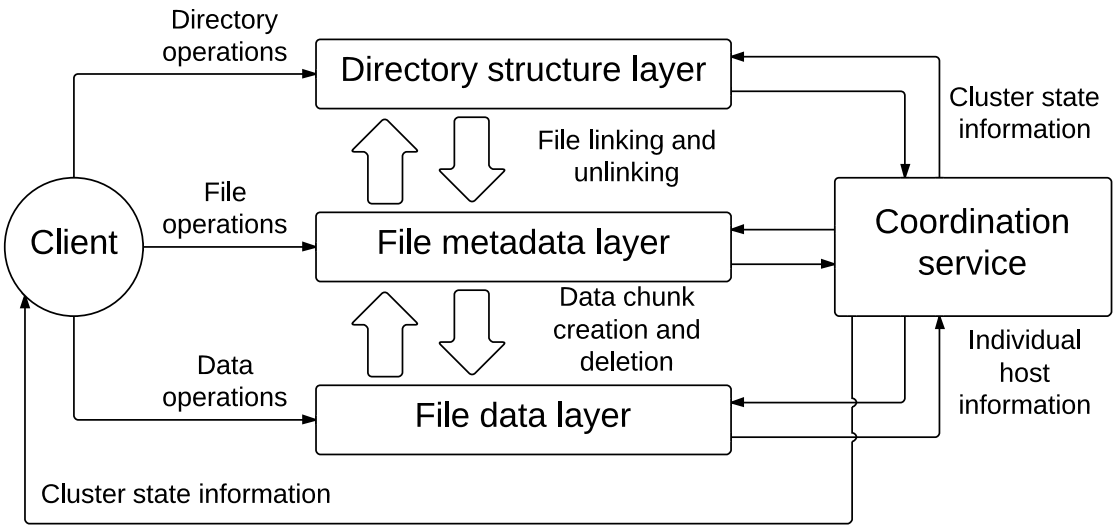

Figure 2.1: High level overview of the file system's architecture highlighting a client interacting with each of the four major components of the file system - the directory structure layer, the file metadata layer, the file data layer and the cluster coordination service.

- **The file metadata layer** sits in the middle of the file system stack. It keeps track of all the files present in the file system and provides a means of allocating unique identifiers to newly created files. Each file entry stores metadata such as the owner and group owner of the directory, UNIX permission bits and access and modification timestamps. Additionally, each file entry stores an ordered list of data chunks that, when concatenated, form the contents of the file. Finally, since an individual file can be linked to from multiple directories, a count of the number of links is stored for garbage collection purposes.

- **The directory structure layer** sits at the top of the three-layer file system stack. Its role is to maintain the hierarchical directory structure of the file system. Each directory entry stores metadata about the directory as well as a list of its child directories and a list of the files that are present within it.

Clients interact directly with each of the four components independently via the use of a client library by obtaining an overview of the cluster configuration from the cluster coordination service and then routing requests to the correct machine within the correct layer based on the type of operation that needs to be performed. In other words, there is no single bottleneck on the data path of BDFS.

Figure 2.1 provides an overview of the design that was just described.

2.3 Sharding and replication

As discussed in section 1.2, one of the goals of BDFS is to raise the limit on the maximum number of objects the distributed file system can store by distributing directory structure and file metadata across multiple machines.

In section 1.3, I mention two solutions to this problem, both of which involved partitioning the file system into independent regions which are unaware of each other and providing a mechanism through which these regions can be viewed by clients as a unified file system. With BDFS, I took the opportunity to instead provide a unified system where each layer of the file system is aware of the entire file system. Table 2.1 describes the three approaches and table 2.2 provides an overview of their relative merits.

In BDFS, each layer of the file system stack has a means through which a particular object stored within the layer is uniquely identified via a key. By partitioning the key space into smaller subregions and distributing each subregion to independent machines, we can lower the amount of data that needs to be handled by each machine and thus overall be able to support more data. This technique is known as **sharding**. Each subregion is referred to as a **shard**.

One possible problem with sharding arises with skewed data access patterns where one portion of the key space is used disproportionatily more often than others. Another problem with sharding is that sometimes a large amount of data movement between different shards is required. I look into these problems as they affect the directory layer and how to ameliorate them in sections 2.4 and 2.5.

	Description
MapR	The file system is divided up into separate regions, each storing its own directory structure, file metadata and file data. Client queries are issued to all regions with the responses being aggregated together.
HDFS Federation	The file system directory structure and file metadata is divided up into separate namespaces. Each namespace is independent from each other, but file data is stored in a common data storage layer. Namespaces are mounted similarly to a UNIX file system and clients are given a mounting table describing which portions of the file system correspond to which namespace.
BDFS	Directory structure, file metadata and file data is partitioned into shards according to object unique identifiers. Each machine within each layer is aware of all objects stored in that layer.

Table 2.1: Description of different solutions to metadata layer scaling

Additionally, we are interested in providing a fault tolerant system that gracefully handles single node failures. This means that no single machine should hold the single copy to any item of data. Therefore, rather than having each shard be allocated to a single machine, we allocate it to multiple machines and employ a distributed consensus protocol amongst these to ensure that operations are performed with strong consistency guarantees. This technique is known as **replication** and I investigate it further in section 2.6.

Figure 2.2 gives a visual representation of a file system layer sharded into 16 distinct regions and distributed over 6 cluster machines.

Option	Advantages	Disadvantages
MapR	• Can reuse the single metadata server - multiple data server design of GFS / HDFS for each region • Relatively low risk and low engineering effort	• Need to ocassionaly move data between different regions • Can not share files between regions easily, data duplication
HDFS Federation	• Stores file data in a single shared place – no need to move data between regions • Can reuse existing implementation of metadata server since each namespace is independent of each other • Relatively low risk and low engineering effort	• Manual setup of namespace mount points • Need to ocassionaly move data between different namespaces • Can not share files between namespaces easily, metadata duplication
BDFS	• All information stored in a single system • No need to move data between different regions or different federation namespaces • Support for operations requiring sharing of data, such as hard links	• Partitions are not necessarily location aware

Table 2.2: Comparison of different solutions to metadata layer scaling

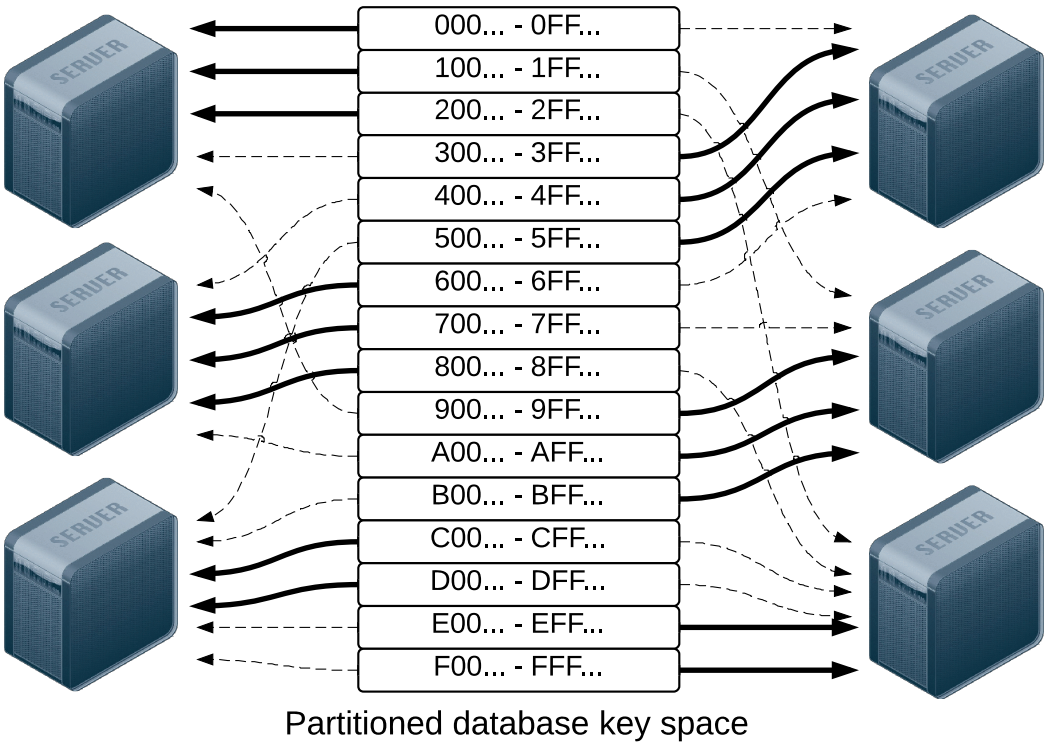

Figure 2.2: Example 16-way sharded database distributed across 6 servers with one level of replication. Thick solid lines connecting servers to shards show that a server is responsible for the shard and accepts client requests for it. Thin dotted lines show that a server replicates the shard without serving client traffic.
In this example, we partition the space into equally sized ranges assuming equal load across each of the shards and distribute the shards evenly across machines. The assignment of shards to machines is managed by the cluster coordination service as discussed in section 3.2.

For simplicity, I chose to reuse the same sharding and replication approach for all three layers of the file system. This differs from the replication mechanism present in GFS or HDFS which operates on individual blocks rather than shards.

2.4 Sharding the directory structure layer

I considered two distinct approaches to distributing the directory structure layer, whose relative merits I discuss in the following section.

2.4.1 Option 1: Unique directory identifiers

This approach assigns a unique identifier to each directory once it is created, without taking into account any information about the directory such as its name or

path. This would require choosing a scheme which evenly distributes the unique identifiers across the key space to avoid skewed data access patterns.

In order to access a file in directory `/a/b/c/d/e/f/g/` we would have to in turn retrieve directories `/`, `/a/`, `/a/b/`, `/a/b/c/` etc. and, unfortunately, this can only be done in an iterative fashion. Therefore, if we choose to distribute these directories evenly across machines, we end up having to jump around from machine to machine fetching one directory at a time and incurring network latency at each step. On the other hand, if we choose to make it so that child directories are likely to be on the same machine as parent directories, we run the risk of creating skewed access patterns.

This approach therefore does not seem ideally suited for a distributed file system. However, it does, have the advantage that checking access permissions comes for free as we walk down the directory structure and that operations such as moving a directory are simple to implement in constant cost.

2.4.2 Option 2: Path-derived directory identifiers

This approach – somewhat unintuitively – discards the hierarchical nature of the directory structure and identifies directories by a hash of their full path. Since we want directories to be evenly distributed across the key space, we can use the 160-bit long cryptographically secure SHA1 function to uniquely identify directories based on their full path.

This approach has the advantage that we are immediately able to access a particular directory's information since we know exactly which subregion it is in. In order to properly implement access control, however, we still need to obtain information from all the directories along the path from the root, but the difference here is that we are able to query all relevant machines in parallel and thus only incur a relatively small network latency penalty. This is ilustrated in figure 2.3.

The disadvantage of this approach is that it means an atomic operation such as `move` or `rename` needs to potentially touch a very large number of directories in the subtree. All directories that are rooted at the directory being moved now have their full paths altered and thus need to be redistributed to other subregions of the key space, which is evidently an expensive operation.

One way of implementing such an operation is illustrated in figure 2.4 and involves performing a distributed transaction. First, the entire subtree rooted at the directory being moved is frozen via a locking mechanism. Then, all information is transferred to the new location. Finally, all of the locks are freed.

Considering real world use cases, it seems like **Option 2** is preferrable to the first, since it optimises performance of very common operations like lookups at the expense of fairly rare operations like recursive move, hence, I chose this approach for BDFS. Furthermore, the move operation could be sped up significantly using a more complicated, lazy-move method described in appendix D.

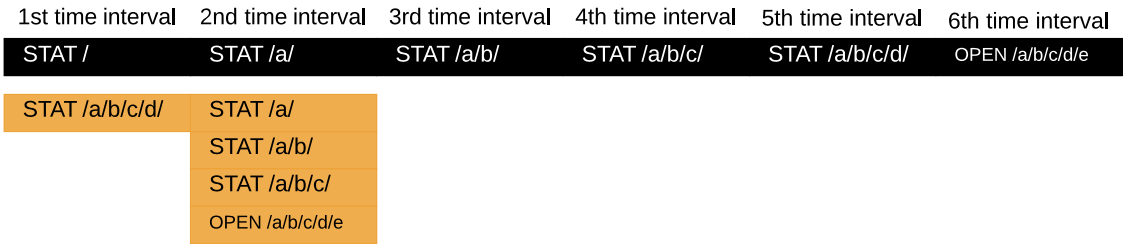

Figure 2.3: Timing diagram comparing the two proposed options of sharding the directory structure layer when performing an open operation on file `/a/b/c/d/e/`. Dark blocks illustrate Option 1, while light blocks illustrate Option 2. Actions that are performed serially are laid out side by side, while actions that are performed in parallel are laid out one on top of the other.
Option 1 requires us to serially traverse the directory hierarchy before being able to open the file, while Option 2 allows us to directly access the parent directory and thus open the file and check permissions all along the directory hierarchy in parallel in only 2 steps.

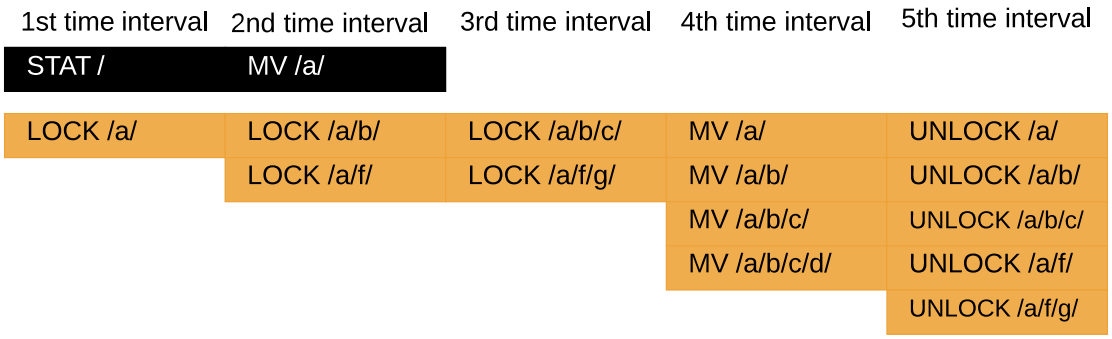

Figure 2.4: Timing diagram comparing the two approaches when performing a move of directory `/a/`. Option 1 allows us to move a directory in one operation once we have reached its parent directory. Option 2 optimises for the more common file and directory access operations at the expense of a complex and slow recursive move operation. In this example implementation, one needs to first recursively freeze the subtree rooted at `/a/`, an operation which depends on the depth of the tree, then perform all the moves in parallel and then finally unlock everything that was locked.

2.5 Spatial locality of information

One thing to note when examining the proposed layered view against the architecture of GFS is the added separation of the directory structure from the file layer. The motivation behind this is to allow us to avoid having to copy file information when performing directory moves and to support features such as file hard links.

It rapidly becomes apparent that if we do not intelligently distribute directory structure and file information among the metadata machines, we can end up having

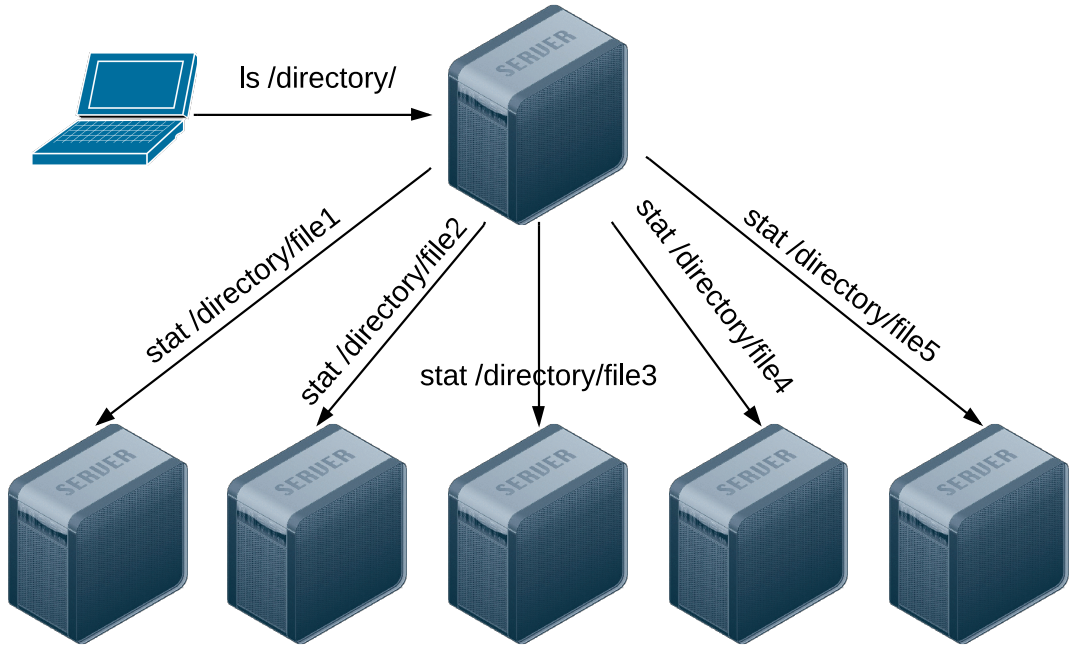

Figure 2.5: Example showing the network traffic that ensues when performing a detailed list operation on a directory in a situation where spatial locality of file information is not enforced. Contacting multiple machines in order to retrieve metadata about the directory's children incurs a network overhead that can be avoided by trying to group together files that appear within the same directory.

to touch a very large number of machines for something as simple as listing the files in a directory. For example, assume that we have n machines in total and we store directory A on machine 1, but store files evenly across machines ignoring any information regarding which directories the files are part of. Then, listing information about all files within a directory containing at least n files means that we can end up touching all metadata machines at once.

Since the network traffic involved in contacting so many machines can be quite substantial, it makes sense to try to group files from within the same directory onto as few machines as possible. One way of achieving this, as ilustrated in figure 2.6, is to share a number of bits between the file identifier and its parent directory's identifier. Assuming a fixed size 64-bit unique identifier for files, encoding this information about the parent directory becomes a tradeoff between having too many files allocated to the same machine resulting in skewed access patterns and not supporting a large enough number of files.

At this point, it is useful to point out that the exact scenario described here applies when considering file chunk identifiers. It makes sense therefore to implement the same scheme for both of these, while taking into account the fact that the number of chunks will always be at least as large as the number of files.

When deciding what number of bits out of the 64 to allocate to directory information, it is useful to consider limits seen with the open source implementation of

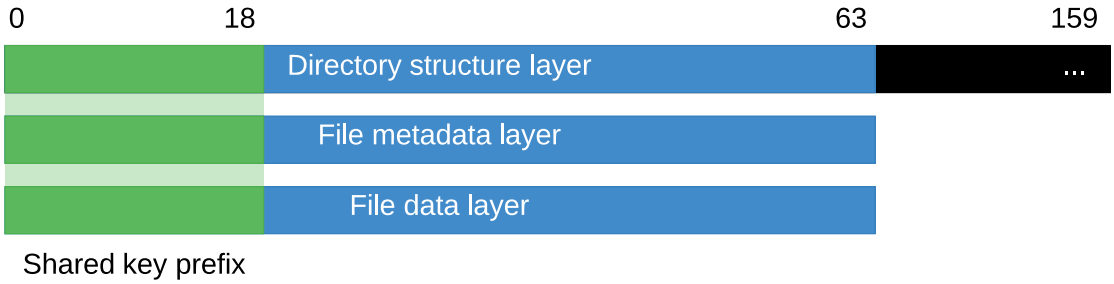

Figure 2.6: Diagram illustrating the shared key prefix scheme used to improve spatial locality of information. The first 19 bits of the 160-bit long directory key are copied to the first 19 bits of the 64-bit long file identifier of files created in this directory. Likewise, all 64-bit long data chunk identifier share the first 19 bits with those of their parent files.

HDFS. As of 2014, HDFS stores file and block IDs as 32-bit integers, which means it is able to store roughly 10^9 files. If we take the 64-bit space described and allocate 19 bits for the directory information and 45 bits for the unique file or chunk identifier, this theoretically allows us to partition data across roughly 500.000 machines without needing to ever distribute file information for files in the same directory to more than one machine, while allowing 10^{14} chunks to be managed by a single machine. Both of these limits are quite far beyond what is achievable practically for single cluster systems.

The above key prefix sharing is of course only enforced to some degree. Files that are hard linked to multiple directories can only share the key prefix with at most one of its parents. Furthermore, hard linked files can not change their identifiers without having to simultaneously update multiple directory entries. However, when moving files with a single link, it is possible to update the file identifier accordingly in constant time. For directory moves, we would like to avoid changing all child file IDs as that might be very costly. Instead, we rely on the fact that those files still share a prefix amongst themselves and as such are still local to a single machine. Any newly created files in that directory can be added to the same shard as well.

2.6 Distributed consensus protocols

As stated in the section 2.3, we are interested in having multiple machines within each layer handle the same subregion of the key space such that if any of one of the machines fail, the file system can seamlessly continue its operation without clients noticing. In particular, this means that a failure must not result in any data loss. Once an operation has been reported back to a client as being successful, it has been performed on enough machines to guarantee that, regardless of host failures, any other client attempting to read back the data will receive this latest version. This is known as a **Strong Consistency** guarantee.

Algorithms that implement such a guarantee are known as **distributed consensus protocols**. Protocols such as **Two-Phase Commit** or **Three-Phase Commit** seem simple to implement at first sight, but their original incarnation requires the network to support atomically broadcasting a message to all interested parties such that either all or none of the endpoints see the message. Without this guarantee, they require complicated additional mechanisms which result in degraded performance. I therefore looked at two different consensus protocols: **Paxos** [11], which is an industry standard protocol, and the currently under-development **RAFT** [12] protocol, which was designed to be an easier to understand and implement alternative to Paxos.

While I initially wanted to use RAFT, it quickly became apparent that open source implementations of the protocol made the assumption that one machine is only ever part of one RAFT consensus group, which is not the case for BDFS where one machine can handle multiple shards. Modifying an open source implementation would have likely been non-trivial, as would implementing the entire RAFT protocol from scratch.

Hence, using the ideas underlying RAFT as inspiration, I chose to design a simpler protocol. Each shard has a single machine, the **leader**, that accepts client traffic. It maintains a **log of operations** that it has committed, as illustrated in figure 2.7. Any new log entries are sent to all other machines in the group and once a sufficiently high number of these have acknowledged that they have committed the modifications themselves, the client is notified of success. The protocol relies upon the cluster coordination service to elect the leader machine, to detect failures and to resolve conflicts once a machine rejoins the cluster. This approach trades performance for a greatly simplified design as it moves fault tolerance from the ma-

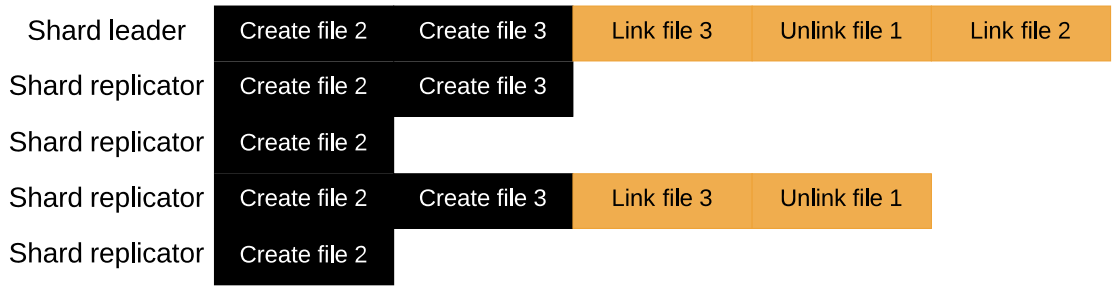

Figure 2.7: Example log of operations being performed in one file layer shard. The dark entries are those which have been replicated to a sufficient number of machines that they are are considered committed. The first log entry shown is fully replicated to all five hosts, while the second log entry is only replicated onto three hosts, which is enough to form a majority within the shard. The following entries are not yet replicated on enough hosts for them to be considered committed. The last entry is recent enough that it has not yet been replicated to any other host other than the leader.

chines handling the shards to a dedicated service. I discuss this in more detail in section 3.2.

Strong consistency is also a requirement within the cluster coordination service itself, since we want all machines within the cluster to have a consistent view of what each of the other machines is responsible for. I discuss the implementation of the cluster coordination service in section 3.2.

2.7 Distributed transaction handling

The distributed nature of the file system means that we are partitioning data that is inherently inter-dependent both across the different layers of the file system and across different shards within the same layer. Therefore, it is often the case that we require atomic execution of a set of particular operations across multiple machines. This is known as a **distributed transaction**.

For example, when creating a new directory, we need to create a new directory node in one subregion of the directory structure key space and link the new node to its parent in another subregion. When creating a file, we need to create a new file node in the file metadata layer and link it in the directory structure layer.

In order to guarantee that different transactions do not interfere with each other BDFS employs a locking scheme where it first locks all relevant identifiers across all machines. It can then proceed to perform the operations and then finally release all the locks. This approach is known in literature as **two-phase locking** or **2PL**.

The obvious problem with this approach is that if the machine that is running the transaction fails, then we end up in an inconsistent state with only some of the changes being committed and with all locks being held. Thus, we need a way of resuming transactions upon recovery.

We notice that any file system operation can be split up into a sequence of simpler operations that only require changes to single, individual shards. Thus, these simpler operations do not require a distributed transaction to be performed.

Note that each file system operation is issued to a particular shard leader, which might then in turn issue a sequence of these simpler operations. Furthermore, in order to be able to resume a transaction, it is sufficient to know which operation in this sequence was the last one to be executed. Therefore, it makes sense to explicitly keep track of how far along a transaction the leader is and replicate this information to the other machines in the shard. For this purpose, we reuse the log of operations introduced in section 2.6. Henceforth, I will refer to this log as the **transaction log** as it contains information about both local and distributed transactions that have been committed on the shard.

Whenever a request is made to a different machine, the leader adds an entry into its log, which is then replicated to the other machines. When a response is received, the leader resumes execution from where it left off. If another request needs to be made then it commits another log entry and the process repeats itself. These log

entries are called **transaction checkpoints** and are used to mark progress within the execution of a distributed transaction. Once the transaction is completed, a final log entry is added, marking the transaction as **committed**. In case a transaction needs to be rolled back, a log entry is added, marking the transaction as **aborted**. Local transactions do not require any checkpointing and thus will only have a single entry of type **committed** in the log.

An example of a shard's transaction log can be seen in figure 2.8.

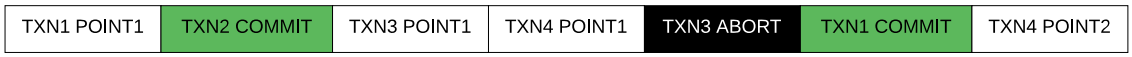

Figure 2.8: Example distributed transaction log showing four transactions being performed on a database shard. The first transaction checkpoints and eventually commits. The second transaction is a local transaction, so it commits directly without any checkpoint. The third transaction checkpoints once, but eventually aborts due to a remote shard objecting to the transaction. Finally, the fourth transaction starts, performs two checkpoints and is still running. If the host were to fail at this point, transaction four would have to be resumed from its second checkpoint.

Since logs are replicated, if the transaction coordinating machine goes down, we can simply resume from where we left off once a new shard leader is elected. Upon recovery, the transaction log is traversed in order and a set of running transactions is maintained, along with the last checkpoint that each transaction reached. Checkpoint entries add or update transactions in the set, while commit and abort entries remove transactions from the set. Once the entire log has been traversed, the set will contain only those transactions which need to be resumed.

I discuss the distributed transactions system in significantly more detail in appendix B.

2.8 Software Engineering

2.8.1 Development process

Once the planning phase was over and development work on the project began, an iterative **Agile Development** [13] strategy was adopted. Typically, I would maintain a list of possible features or modules that were ready to be implemented and run one-week long **Sprint** cycles where I would set one of these features as the main work item for the week. The objective was then to implement both the feature itself as well as unit tests that cover as much of the code as possible for that particular feature. Time permitting, I would start implementation on any other features that were in the backlog, planned for future weeks.

The very core of the project was written in low-level **C** with the vast majority of the project being written in **C++**. Appropriate encapsulation strategies were

employed to ensure loose coupling of the various components. Both abstract base classes and inheritance as well as C++ templating was used to allow significant code reuse, particularly among the database code within the three different file system layers and in the actual request processing code. Some external open source libraries were used to ease implementation time. Documentation was added inline in the code in a standard format which allows an automated documentation generator to parse and present it in a user-friendly format.

2.8.2 Revision control tools

I chose to use **Git**[1] as a version control system both due to previous experience and comfort working with it and due to its strong support for branches, its powerful merge resolution tools as well as its support for local change management and for history editing.

Two separate repositories containing respectively the project code and the dissertation were maintained and hosted externally on **GitHub**[2], both as a means of backup and as a means of sharing progress with my supervisors. Additionally, as an extra safety precaution, the repositories were automatically and continuously backed up using the cloud storage service **Dropbox**.[3]

2.8.3 Development tools

The main development environments I used were a customised **Vim**[4] text editor with basic IDE features such as fuzzy project file search and autocompletion as well as the recently increasingly popular **Sublime Text**[5] editor that came with said features out of the box.

Project compilation was handled by **GNU make**[6] which allowed me to only recompile the bare minimum necessary whenever changes were performed. Compilation times were reduced using the **ccache**[7] compiler cache.

I made sure that the project was designed to be cross-platform as development occurred both on Arch Linux using version 4.8 of the **gcc**[8] compiler and on Mac OS X Mavericks using version 508.0.38 of the **clang**[9] compiler running on version 5.1 of Apple's **Low-Level Virtual Machine**.[10]

I performed debugging using **printf-style debugging** where logging calls are introduced at various points in the program whenever unexpected behaviour oc-

[1] `http://git-scm.com/`
[2] `http://github.com/`
[4] `http://www.vim.org/`
[5] `http://www.sublimetext.com/`
[6] `https://www.gnu.org/software/make/`
[7] `http://ccache.samba.org/`
[8] `http://gcc.gnu.org/`
[9] `http://clang.llvm.org/`
[10] `http://llvm.org/`

curred in order to diagnose where the issue occurs. This was done since using a step-by-step debugger such as **gdb** would have been unwieldy for multiple independent server applications running at the same time.

I used **Valgrind**[11] whenever a major feature was implemented in order to detect any memory leaks that I might have introduced or to detect any usage of uninitialised memory or other types of memory-related errors. This saved a lot of debugging time in the long term by detecting subtle bugs early.

2.8.4 Open source libraries used

A few cross platform open source libraries were used to ease development. The reasons for choosing each of these libraries will become clear in the implementation chapter. The most significant ones include:

- **libev**[12] – an event loop library used at the core of the server to process incoming requests from clients;
- **libpthread**[13] – a popular cross-platform threading library;
- **libdb_cxx**[14] – a library providing C++ bindings to Oracle's BerkeleyDB;
- **libzookeeper_mt**[15] – a library providing a C/C++ client interface to Apache's Zookeeper;
- **libcrypto**[16] - the OpenSSL cryptographic library used for hashing purposes.

2.9 Summary

In this chapter, I looked at the principal design decisions that were made prior to the beginning of the actual development work:

- In section 2.1, I gave an overview of the goals that the project set out to achieve in the form of a requirements analysis.
- In section 2.2, I gave a high level overview of the distributed file system and its division into four major components: the cluster coordination service, the directory structure layer, the file metadata layer and the file data layer.

[11] `http://valgrind.org/`
[12] `http://libev.schmorp.de/`
[13] `http://en.wikipedia.org/wiki/POSIX_Threads`
[14] `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html`
[15] `http://zookeeper.apache.org/releases.html`
[16] `http://www.openssl.org/docs/crypto/crypto.html`

- In section 2.3, I looked at sharding and replication in general for all three layers and, in section 2.4, I looked in particular at the two different ways of sharding the directory structure layer with their respective advantages and disadvantages.
- In section 2.5, I looked at how to best distribute file and data chunk information so as to increase locality of information by encoding parent directory information within the unique identifiers of files and data chunks.
- In section 2.6, I looked at possible distributed consensus protocols and gave an overview of the simplified protocol that was in the end designed to fit the needs of this project.
- In section 2.7, I looked at how distributed transactions spanning multiple machines are implemented by reusing the distributed consensus protocol to perform transaction checkpointing.
- Finally, in section 2.8, I gave an overview of the software engineering techniques employed and the tools used for the BDFS' development.

Chapter 3

Implementation

In this chapter, I look more specifically at how I approached implementation of the components described in the Preparation chapter.

In section 3.1, I give an overview of the functionality BDFS provides and how users interact with it.

In section 3.2, I describe the most complex part of BDFS, the cluster coordination service. I look into its role in monitoring machine status, coordinator election, fault tolerance and management of sharding and replication.

In section 3.3, I look at how information is persistently stored on disk within each layer of the file system.

In section 3.4, I describe the server's non-blocking event-based design allowing for a high number of concurrent connections to individual servers. In section 3.5, I look at the `RequestProcessor` abstraction used to ease development of non-blocking code by providing support for basic semi-coroutines (also known as generators).

Afterwards, in section 3.6, I give an overview of the protocol used to transmit messages over the wire and describe the request and response APIs for the various file system operations. Finally, in section 3.7, I briefly look over the implementation of the BDFS client.

3.1 BDFS overview

To recap, BDFS provides its users with the familiar abstraction of a single file system for storing all of their data that one would find on any local machine, while actually partitioning this data and distributing it among a set of machines that form the BDFS cluster. This process, discussed in the preparation chapter, in sections 2.3 to 2.7, is invisible to the client, which merely observes that the file system is able to store many more files than what would be possible on a single host.

Clients connect to BDFS either via an interactive client, as shown in figure 1.1, which allows them to issue commands to the cluster and receive back responses or via a client library that they can integrate within their applications.

The API BDFS provides is similar to that of any file system allowing users to create and delete directories and files, list directories, retrieve metadata information about files and directories, check access permissions, change owners or permission bits and create file hard-links. Additionally, the BDFS interactive client allows one to transfer whole files to and from the cluster.

Creating hard-links is an example of an operation which is supported by BDFS, but not by HDFS or GFS.

Similarly to HDFS, BDFS only implements support for appending data to existing files, however random writes are possible using the current design.

3.2 Cluster coordination service

As discussed in section 2.2, the cluster coordination service is responsible for monitoring the status of all machines within the cluster, detecting any host failures, managing joins of new machines and managing rejoins of previously failed machines.

Additionally, it provides a means through which any interested party, such as a client or a machine that is part of the cluster, can obtain information about the cluster and which machines are responsible for which shards. Finally, it ensures that any updates to this information are pushed consistently across all cluster machines.

At a high level, monitoring machine status can be achieved by periodically exchanging keep-alive packets between cluster machines. Additionally, propagation of information to machines in the cluster can be achieved via the use of some distributed consensus protocol as described in section 2.6. Both of these superficial descriptions actually hide many subtle corner cases that one runs into under real-world conditions. As such, it makes sense to use a service which already solves these problems.

3.2.1 Apache ZooKeeper

ZooKeeper [14] is a unified solution built by engineers at Yahoo to these and many other core distributed system problems.

At a high level, ZooKeeper provides clients with an in-memory, hierarchical, versioned key-value store. The abstraction provided is itself similar to a file system namespace, with the primary difference being that each node in the namespace can both store data and have children nodes at the same time. Nodes can then be accessed using UNIX path name semantics, with the root node being addressed as / and subsequent children nodes forming a path through the hierarchy separated by the / character.

Clients connecting to ZooKeeper are able to perform operations such as creating new nodes as children of existing nodes, deleting existing nodes and those rooted at a node recursively or updating the values stored within nodes. Another distinction from traditional file systems is the possibility of creating ephemeral nodes. These

are nodes that are created by clients and that cease to exist once the client's connection to ZooKeeper is interrupted either gracefully or by failure. This makes them useful, for example, for detecting failures within a cluster. Disconnection is detected by implementing a heartbeat protocol similar to the one described before.

Figure 3.1 gives an overview of the ZooKeeper nodes BDFS uses for its coordination service. These are described in the following sections.

As mentioned earlier, ZooKeeper maintains version information about each existing node in the form of version numbers which are incremented upon every update and exposed to clients retrieving information about nodes. These can be used as a means of determining whether two hosts agree upon a piece of information that is shared amongst them or if one of the hosts is lagging behind the other.

Whenever a client performs an operation on a node in ZooKeeper, it has the option of adding a watch on it. If a client is watching a node, it will receive notifications from ZooKeeper whenever changes are made to that particular node, whether they be changes in data or in its children lists. One also has the possibility of adding a watch to a not-yet-existing node to be notified of its creation.

3.2.2 ZooKeeper consistency guarantees

In terms of implementation, ZooKeeper is itself a distributed system that replicates its key-value data fully onto each machine in the system. It implements write operations via a "sloppy quorum" approach where the majority of machines vote on

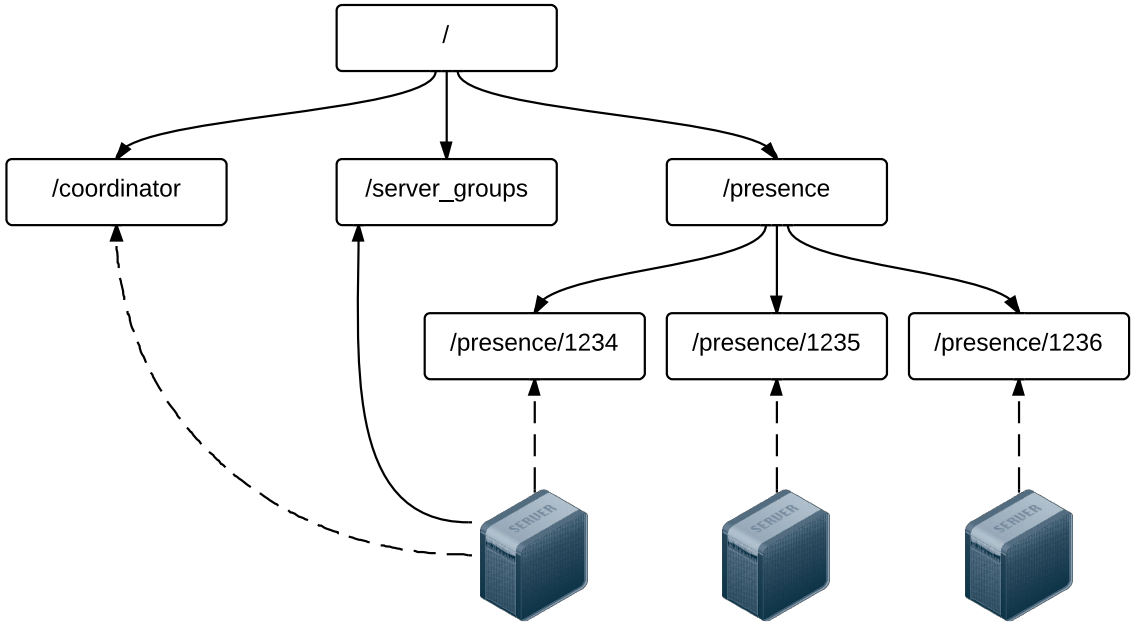

Figure 3.1: Structure of coordination data stored in ZooKeeper. Lines connecting servers to ZooKeeper nodes show which machines created or perform writes to the nodes. Dotted lines signify that the node is ephemeral and thus will be deleted if the machine goes offline.

whether to execute operations and in what order to serialise conflicting operations. Changes are considered committed once the update is performed on at least a majority of the machines, but reads can go to any one single machine in the cluster.

This means that ZooKeeper provides “almost-strong” consistency in the sense that clients may read stale data ocassionally. This is acceptable due to the presence of the watch mechanism described above which means that clients eventually get notified of the fact that they read a stale value once the update propagates to all nodes. ZooKeeper also guarantees that any updates received as part of a watch occur in the same order in which they are serialised, so while it is possible that some updates might be suppressed, clients can always trust that the last value received is the most up to date that the ZooKeeper host is aware of.

Since writes require at least a majority of the machines in the cluster to agree in order to be accepted, ZooKeeper is resilient to a network partition in the sense that it will never have more than one partition capable of performing writes and thus is not subject to the “split-brain” problem.

3.2.3 Election of coordinator

A number of actions need to be taken whenever changes to the composition of the BDFS cluster occur, whether it be a new machine joining or an existing machine leaving. As such, it makes sense to have a mechanism by which one machine in the cluster takes on the responsibility of performing these actions. This machine is known as the **cluster coordinator**.

Relative to file system workload, we do not expect the computational resources required to coordinate the cluster to be significant and as such we allow any machine in the cluster to assume this role.

Upon a cluster boot up, each individual host contacts ZooKeeper and tries to read the contents of `/coordinator` in order to obtain information about the cluster coordinator. If this node does not exist, all hosts within the cluster race to create the node as an ephemeral node, thereby proposing themselves as coordinators. Out of all concurrent requests, ZooKeeper only allows one to proceed and informs the others of failure. At this point, every machine in the cluster has a watch on `/coordinator`, which means that if the coordinator were ever to fail, all hosts in the cluster will simply re-run the same race for power described above. Figure 3.2 gives a diagram showing the events that occur during coordinator election.

3.2.4 BDFS cluster status

Upon a machine start-up, it allocates itself a random 32-bit identifier which will become the unique identifier of that machine within the cluster. Before joining the cluster, the host issues a request to ZooKeeper to create an ephemeral node at `/presence/{machine ID}`. If this is unsuccessful, this means that the ID is already being used by some other machine within the cluster and the host tries again with

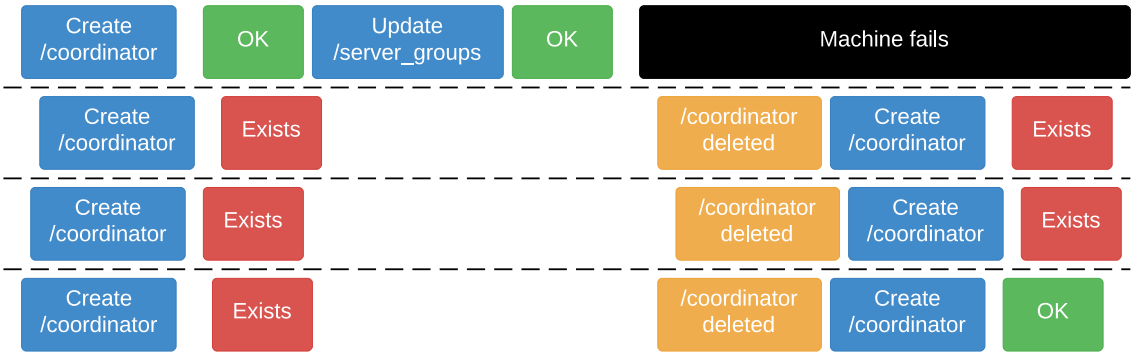

Figure 3.2: Cluster coordinator election process in a 4 machine environment. All machines boot up at roughly the same time and issue a request to be the cluster coordinator. Only one of these succeeds and that is the machine that creates or updates the `/server_groups` node.
The machine that was elected coordinator suddenly fails, which cause the other machines to be notified and the election process is re-run.

a newly generated random ID. The coordinator monitors `/presence` for updates to its children list, thus allowing it to detect both cluster joins and machine failures.

Each host maintains information about what shards it serves and replicates in a persistent, on-disk format. This information is itself stored in the `/presence/{machine ID}` ZooKeeper node, so that the coordinator can keep track of the cluster's state.

Finally, the coordinator is responsible for generating a unified view of the cluster and of which machines are responsible for which shards. The data found in the `/presence/` nodes is aggregated and the result is stored under the `/server_groups` ZooKeeper node. Each machine in the cluster monitors this node for changes and updates its internal records accordingly. The aggregation is done to provide serialisation of all cluster changes, rather than merely the per-host serialisation given by the `/presence` nodes.

Figure 3.3 shows a diagram of the operations a new machine goes through in order to register itself to the cluster.

When a host joins the cluster, it is possible that it already has data for some shards from before a failure. These shards will most likely no longer be up to date and the machine must wait for the coordinator to decide how to proceed.

Each individual shard enters an initial `SERVING_JOIN_WAIT` or `REPLICATING_JOIN_WAIT` state. When a shard is in these states, a host only accepts requests touching those shards from other machines within the cluster. The coordinator makes sure that the shard is brought up to speed with what changes have occurred since the new host's last activity if necessary and then orders the host to switch to either the `SERVING` or the `REPLICATING` state.

When a host requests to join a shard, the other machines part of the shard are ordered to switch to either the `SERVING_PAUSED` or the `REPLICATING_PAUSED` state in

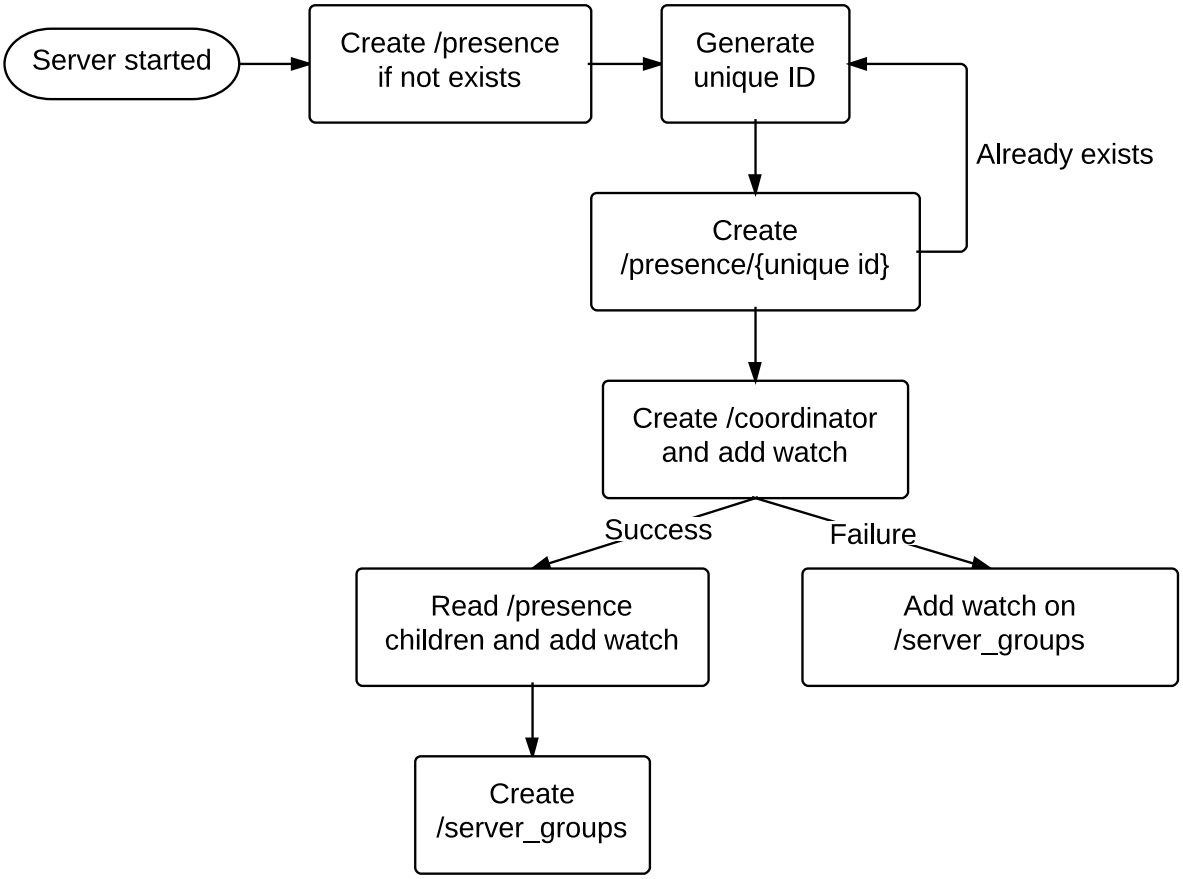

Figure 3.3: Sequence of operations a new machine goes through to register itself to the cluster.

order to allow for the new machine to be brought up to speed. These states are similar to the `JOIN_WAIT` states and machines only accept requests to that shard from other machines in the cluster. The distinction comes from the fact that, for implementation reasons, a running `SERVING` machine may never be turned into a `REPLICATING` machine, but one that is waiting to join as a serving machine may be asked to simply provide replication.

Figure 3.4 shows a detailed diagram of the possible state transitions a shard may go through.

I discuss how shards are allocated to machines and redistributed among them in a process called **shard balancing** in appendix A.1.

3.2.5 Race conditions

One concurrency problem resulted from the use of `/server_groups` as an authority on which machines are part of the cluster.

The machines within the directory structure and file metadata layers maintain open pair-wise connections between themselves. These connections are initiated once hosts obtain a copy of the cluster configuration. Machines within the cluster

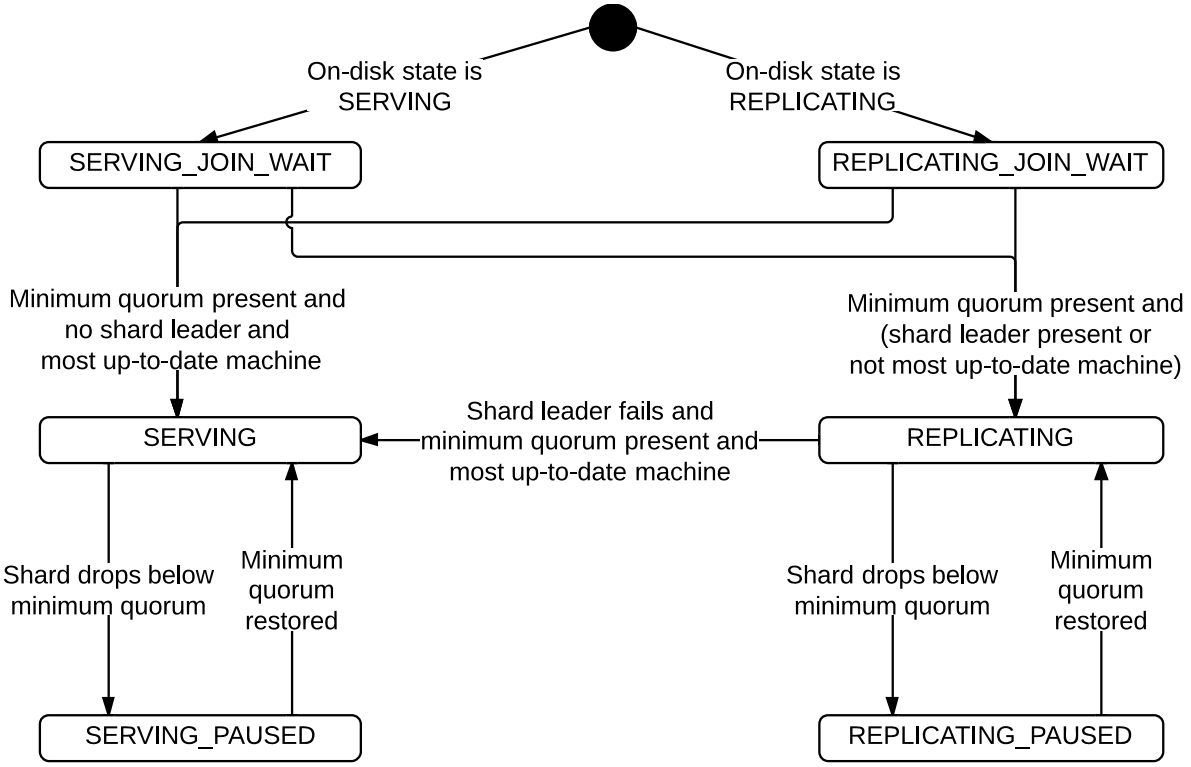

Figure 3.4: Diagram showing the state transitions a shard may undertake

authenticate themselves to each other in order to be able to perform operations that normal clients are not allowed to. When a new machine joins, it will attempt to authenticate itself with all other machines in the cluster, but these may occasionally not recognize it due to not having yet received the updated cluster view from ZooKeeper.

The solution employed for this problem was to send the server group node version number along with the authentication request and have the receiving end wait if its internal version number was less than the one in the authentication request rather than rejecting the connection directly.

3.2.6 Single host failures

Since moving shards is quite an expensive operation requiring a lot of network traffic, I chose to treat failures optimistically and allow the failing host enough time to recover before deciding to reallocate the shards to new machines.

Upon failure, any shards that were served by the machine are reallocated to other machines that already replicate them. This is done by iterating through the shards and choosing the least loaded machine out of those that replicate the shard. This approach would not necessarily be optimal if the load weights were dependent on shard activity, but it works well for the simple weights. A general solution would involve binary searching for a upper bound on machine loads and modeling the problem as a maximum flow problem.

Additionally, a 10 minute timer is started after a host's failure. Once this period has elapsed without the host resuming operation, we must decide on a set of machines that will replicate the shards that were previously handled by the failed host. These are chosen using the balancing shards process described in appendix A.1.

If at any point, a shard becomes under-replicated to the point where continued operation becomes unsafe, a rebalancing is forced early.

If the host does eventually resume operation, it is caught back up to speed with the changes that have occurred in the meantime and it joins in the operation of its shards. Shard balancing might need to be performed again as a result of shards becoming over-replicated.

3.2.7 Network partition

BDFS relies upon ZooKeeper's network parition tolerance in order to provide the same guarantee. It ensures that only a single group within the partition may end up processing client requests thus avoiding the "split-brain" problem. By enforcing a lower bound on the number of machines that replicate any shard, it guarantees that the cluster can continue partial operation safely. I give a detailed explanation of how this is achieved in appendix A.2. The CAP theorem states that it is impossible to design a distributed system which provides consistency, availability and partition tolerance guarantees at once. BDFS provides consistency and partition tolerance guarantees at the expense of availability, which is why it only supports partial operation under heavy failure scenarios.

3.3 Persistent storage

As described in section 2.3, each layer of the file system stack has a means through which a particular object stored within the layer is uniquely identified via a key. Therefore, in terms of storing the data persistently on disk, we are only ever interested in being able to quickly retrieve or update entries via their unique identifiers. As such, a simple key-value store database is sufficient for our purposes. The implementation of the BDFS file system makes use of Oracle's Berkeley DB open source embedded database. Each shard that a host handles is treated completely independently and is stored using a separate BerkeleyDB instance.

3.3.1 BerkeleyDB

Berkeley DB is a software library that provides a high-performance embedded database for key-value data. It provides multiple alternative data stores optimised for different use cases, which are presented in Appendix C.

In addition, Berkeley DB provides mechanisms through which we can group together multiple different database or potentially different types into a single

database environment. This allows us to then perform atomic transactions that perform updates to multiple databases within the environment at the same time.

For the directory structure layer, I use a database environment formed of three databases. First, I store metadata about directory entries. This includes information such as the owner and owning group of the directory, UNIX permission bits, the access, modification and change times and others. Second, I store a sorted list of each directory's children. Third, I store a log of the transactions that have been performed on this shard.

The file metadata layer uses only two databases. The first one stores, for each file, its metadata and a list of data chunks forming its content, while the second one stores the transaction log.

The file data layer uses only two databases as well, one which initially contained data chunks and one for the transaction log. During evaluation, it became obvious that the performance of this scheme is very poor and I developed a custom solution where the actual data chunks are stored within simple files on disk and the database merely stores identifiers to these chunks. I discuss this in more detail in section 4.2.

3.4 Architecture of a server

The traditional approach to building a networked server has been to maintain a pool of separate execution threads or processes for handling incoming requests. When a client would connect, it would simply get picked up by one of the idle threads.

The disadvantage of this approach is that once a thread is handling a client's request it can not do anything else until the request is processed. This means that if the thread is ever waiting for a resource it must block. Thus, we might end up wasting computation power if too many threads are blocking. Additionally, blocking on network resources and subsequent unblocking results in **kernel-mode** context switches which are themselves expensive.

An asynchronous event based approach was developed and successfully demonstrated by the nginx[1] web server that showed successful processing of 10.000 concurrent requests at a time when the most popular alternative, apache, based on the pre-fork model, would struggle with significantly fewer. As such, this is the approach that BDFS employs.

The event loop model spawns a single process or thread for each computational core available on the host machine. Each thread is at any point in time responsible for a number of requests rather than a single one. Instead of blocking whenever a request is waiting on an external resource, the thread checks whether any other requests it is handling can resume execution. This means that we eliminate the problem where we have too many blocked threads and we can't make full use of the computational power available to us. Additionally, switching between processing of

[1] `http://nginx.org/`

different requests is done in **user-mode** without requiring the kernel scheduler as often and thus overhead is reduced slightly.

Figure 3.5 gives a high level overview of what the event loop infrastructure looks like and what are the types of events that BDFS processes.

Implementation-wise, we first open a server port and then spawn one **worker** thread per CPU core. Each thread has a set of sockets on which it is listening for events. The server socket is the only socket that is shared between all of the worker threads. When a request comes in, the threads race to accept the connection and

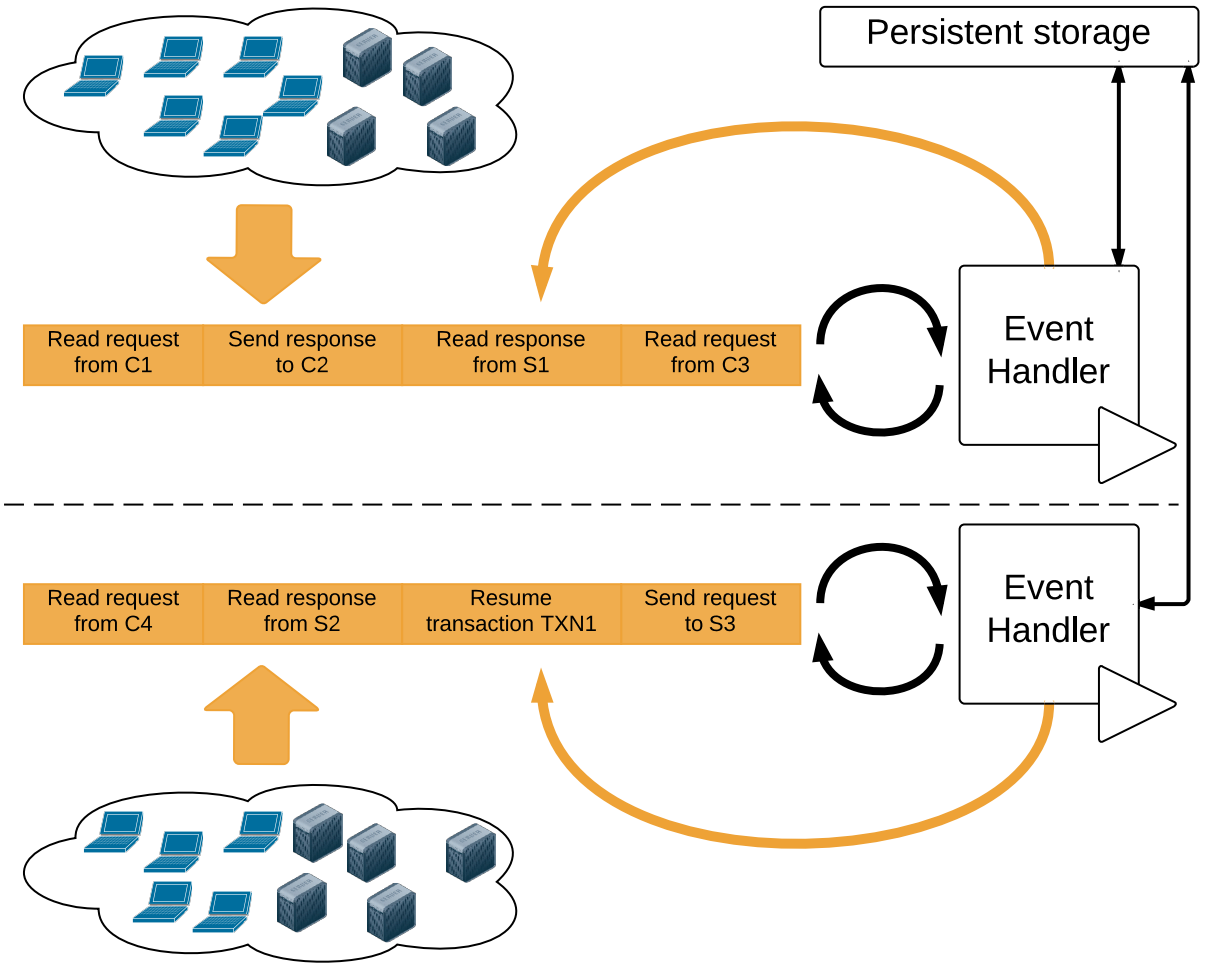

Figure 3.5: Overview of non-blocking event loop server architecture. Diagram shows two different working threads, each handling their respective set of external clients and servers from within the cluster. Each worker thread effectively maintains a queue of events that are processed one by one by the event handler. New events can join the queue both as a result of a message coming in from a client or server in the pool or as a result of event processing. For example, a request to create directory request can cause a 'Send link directory request to BDFS server' event to be enqueued. Upon receival of a response from another machine in the BDFS cluster, a 'Resume transaction' event is enqueued.

add the resulting socket to their set. Note that the accept system call guarantees that only a single thread will be able to accept the connection. Within each set, we can either listen for read events on sockets, in which case the thread is notified when data is received on the socket, or for write events, in which case the thread is notified when there is space available in the output buffers of the socket.

Each thread is effectively a loop. On each iteration, it **polls** all of the sockets in its set, blocking if necessary until at least one of them has events. The loop then iterates through all sockets that have events and processes them accordingly. In BDFS, the socket polling is abstracted by the use of **libev**. Under the hood, libev can either choose to use the POSIX **select** system call which accepts a set of read sockets and a set of write sockets and returns a list of available sockets or use more efficient, yet platform-specific alternatives, such as **epoll** on Linux or **kqueue** on Mac OS X. Note that performing the polling does require a kernel-mode change since we are using a system call, but even so, we are able to receive multiple events and thus do progress on multiple requests with a single system call.

3.5 Request processing using semi-coroutines

`RequestProcessor` is an abstraction designed to make it easier to write code that is non-blocking, as required for an event loop architecture. At its core, it simply allows us to divide up a linear sequence of operations into multiple steps and yield execution after any individual step whenever we would otherwise block waiting for a resource. The ability to yield execution effectively implements one half of the coroutines paradigm, referred to as a generator or semi-coroutine.

A `RequestProcessor` provides the programmer with an interface for performing blocking operations such as acquiring locks, issuing requests to other machines in the cluster, waiting for replication to be performed or returning responses to clients.

The server is then responsible for waking up the processor whenever it becomes unblocked. Note that, within the event loop architecture, we rely on incoming network traffic from the client in order to start a request processor. However, resuming execution of a processor needs to occur without any activity on the sockets on which the worker threads are listening. This is done by creating an anonymous in-process pipe for each worker thread and adding it to the set of sockets which are polled by that thread. When a processor is unblocked, we write a pointer to it in the pipe, which is then detected by the worker thread and the processor's execution is resumed on that worker thread. This is referred to as an **asynchronous** event.

3.5.1 End-to-end example

We now look at the implementation of the create directory request processor as a representative example of how metadata operations are implemented within BDFS.

When a client issues a **create directory** request, also known as a **mkdir** request, we need to perform two main operations as part of a transaction: create a directory node at the desired destination and add the new directory to the list of children of its parent directory. As such, the shard performing the mkdir request needs to issue a **link directory** request to the shard holding the parent directory. The link directory operation is an example of a **system-triggered** operation. These operations are internal and are only executed if they originate from other BDFS cluster machines.

Table 3.1 illustrates the steps involved in processing a **mkdir** request.

Step	Locks held	Waiting on	Operation performed	Checkpoint
0	None	-	Issue access control requests to shards holding all directories on the path from the root to the new directory	No
1	None	Access control responses	Acquire database lock on the new directory's ID	No
2	Directory	Lock	Create directory node	Yes
3	Directory	Replication	Issue link directory request	No
4	Directory	Link directory response	Commit transaction if the link was created or if the link already exists. The latter may occur if the machine running the transaction failed after issueing the link request, but before processing its reply. Abort transaction otherwise.	Yes
5	None	Replication	Issue response back to client	No

*Table 3.1: Steps involved in processing an **mkdir** request*

3.6 On-wire protocol

Communication among different machines within the cluster and between clients and cluster machines is done via a messaging protocol. A message is composed of 4 different fields describing the message's type, its size, its contents and a checksum of the contents. The contents of a message will generally be a seriailsed datastructure such as a Request or a Response object. The message sending infrastructure allows for the content to be sent over the wire in chunks, so that the server does not need to store the entire serialised message in memory until it is done sending.

The BDFS implementation uses TCP as its transport layer protocol and relies upon its in-order packet transmission. It does not however rely upon its error detection or retransmission capabilities. Message retransmission is implemented in the application layer since BDFS needs to support running in an environment where machines may fail and the destination of a request may change as a result.

Table 3.2 provides an overview of message protocol which BDFS implements. It's worthy to note that servers may batch together multiple file system operations into a single message for increased performance in high latency scenarios. As such, we distinguish between messages, requests and responses, with the former encapsulating the latter two. Table 3.3 provides an overview of the file system operations that BDFS supports and table 3.4 provides an overview of the internal-only operations that are used to implement the former.

3.7 Client implementation

The BDFS client reuses much of the same infrastructure that is put in place for the server code. It gets information about the cluster configuration from the cluster coordination service and then spawns multiple worker threads running asynchronous event loops, just like a BDFS server. These worker threads are then used to issue requests to and process responses from machines within the BDFS cluster.

A simple method call API is provided for each file system operation that one might wish to execute. These could be provided in the form of an external library to applications wishing to use BDFS as a storage layer.

Under the hood, these API methods start executing corresponding request processors which issue one or multiple requests to the cluster, process the responses and return the information to the user. Request processors are used since they already plug well into the event loop infrastructure and they provide a good abstraction for operations that require multiple different requests to be issued.

A simple operation, such as listing a directory, requires a single `OP_LIST` request to be issued. A slightly more complex operation, such as linking a file, requires first determining the unique identifier of the source file by issueing a `OP_STAT` request and only then can a `OP_LINK` request be issued to perform the actual linking.

Implementing the `put` operation, which transfers a file from local disk to the distributed file system, requires many different requests to be performed in order to create the new file, allocate chunks for it and transfer each individual chunk. Similarly, the `get` operation used to transfer a file from the distributed file system to the local machine, first needs to obtain the list of chunks via a `OP_STAT` request and then read each individual chunk.

Figure 1.1 shows the interactive client connecting to a BDFS cluster and performing file system operations such as directory and file creation as well as file transfer to and from the local machine.

Name	Description
`MSG_AUTH_SERVER`	Internally used to authenticate a server into the distributed system.
`MSG_AUTH_USER`	Used by a client when first connecting to a server to authenticate its user.
`MSG_COORDINATE_ORDER`	Contains orders from the cluster coordinator. Can be used to instruct machines to join a new shard, drop an existing shard or perform a state transition in an existing shard. See section 3.2.
`MSG_REQUEST`	Encapsulates a single file system operation request, identified by a unique numeric ID.
`MSG_RESPONSE`	Encapsulates a response to a previously received request, identified by its unique ID.
`MSG_REQUEST_BATCH`	Encapsulates multiple file system operation requests, identified by a single unique numeric ID.
`MSG_RESPONSE_BATCH`	Encapsulates responses to a previously received request batch identified by its unique ID.
`MSG_REPLICATE`	Used for replication purposes. Contains the portion of the transaction log of a single shard which has not yet been committed by the replicating machine.
`MSG_REPLICATE_ACK`	Used to signal back the last entry that a replicating machine has committed for a particular shard. The shard leader may send an empty `MSG_REPLICATE` message to a machine joining a shard to determine if it needs to be brought up to speed.
`MSG_SNAPSHOT`	Used by a shard leader to provide a machine with a complete snapshot of the shard.

Table 3.2: Overview of the message protocol used by BDFS. The five shaded messages are only for internal use within the cluster, while the others may originate from either clients or other machines.

Name	Description
OP_ACCESS	Check access permissions for a path.
OP_LIST	List directory by path
OP_MKDIR	Create a directory
OP_RMDIR	Remove a directory
OP_CREAT	Create a file in a directory
OP_LINK	Link a file to a directory
OP_UNLINK	Unlink a file from a directory
OP_CHOWN	Change ownership of a file or directory
OP_CHMOD	Change a file's or a directory's permission bits
OP_STAT	Get metadata information about a file or directory
OP_ALLOC_CHUNK	Allocate a number of chunks for a file
OP_CHUNK_READ	Read data from a chunk
OP_CHUNK_APPEND	Append data to a chunk
OP_MOVE	Move or rename a file or directory

Table 3.3: Overview of the file system operations supported by BDFS. Operations that receive as argument a directory, such as `OP_LIST`, and those that receive an argument which can be either a directory or a file, such as `OP_CHOWN`, identify the argument via its full path. Operations that receive a file as argument, such as `OP_ALLOC_CHUNK`, or those that receive a data chunk as argument, such as `OP_CHUNK_READ`, identify the argument via its unique ID.

Name	Description
`OP_ACCESS_DIR_NODE`	Check access permissions for a directory node
`OP_ACCESS_FILE_NODE`	Check access permissions for a file node
`OP_LINK_DIR`	Link directory to its parent
`OP_UNLINK_DIR`	Unlink directory from its parent
`OP_UPDATE_NLINKS`	Update a file's link count
`OP_CHOWN_DIR`	Change a directory's owner
`OP_CHMOD_DIR`	Change a directory's permission bits
`OP_CHOWN_FILE`	Change a file's owner
`OP_CHMOD_FILE`	Change a file's permission bits
`OP_STAT_DIR`	Get metadata information about a directory
`OP_STAT_FILE`	Get metadata information about a file
`OP_LINK_CHUNK`	Link chunks to file
`OP_CHUNK_DELETE`	Delete chunks corresponding to deleted files

Table 3.4: Overview of internal file system operations used by BDFS. Directory, file and data chunk arguments are identified by their key. As opposed to the public operations, the type of all arguments is unambiguous.

3.8 Summary

In this chapter, I provided an in-depth analysis of the implementation of the components of BDFS:

- In section 3.1, I gave an overview of BDFS's functionality and the ways in which users interact with it.
- In section 3.2, I described the cluster coordination service, its role in fault tolerance and distribution of workload across machines and the process through which new hosts go through when joining the cluster.
- In section 3.3, I described how information is persistently stored on disk within each layer of the file system.
- In sections 3.4 and 3.5, I described the core asynchronous event loop architecture and the `RequestProcessor` abstraction used to ease development of non-blocking code.
- In section 3.6, I gave an overview of the protocol used to transmit messages between clients and BDFS machines and between the cluster machines themselves.
- Finally, in section 3.7, I briefly described the implementation of the BDFS client.

Chapter 4

Evaluation

The purpose of this chapter is to demonstrate succesful operation of BDFS in multiple environments and configuration setups, evaluate its performance in both data and metadata operations, demonstrate its ability to scale as more resources are allocated to the BDFS cluster and demonstrate its fault tolerance.

In section 4.2, I look at the file system's data layer performance by measuring the time it takes to upload a file to the cluster and to download it back. I show how performance was significantly improved from the first BDFS iteration through a number of optimisations. Finally, I look at performance observed for a server running on a local machine and for a small heterogenous cluster running over a home network and show it to be close to hardware limits.

In section 4.3, I look at a cluster of machines running on Amazon's EC2 cloud service and measure throughput for metadata read and write operations and how it varies with the number of machines, number of shards per machine and replication factor. I show that BDFS scales linearly with the number of machines allocated to the cluster, as expected.

Finally, in section 4.4, I show BDFS' fault tolerance in action by measuring metadata throughput while simulating machine failures.

4.1 Overall results

The success criteria as mentioned in the original project proposal and in section 2.1 are as follows:

1. **The file system must distribute file contents, metadata and directory structure over multiple machines.**

   This goal was achieved via key space sharding as described in section 2.3 with implementation details surrounding distributed transactions, persistent storage and transaction isolation being described in sections 2.7, 3.3 and in appendix B.

2. **The file system must provide replication for file contents, resulting in fault tolerance of single node failures.**

   Replication was achieved by the use of a per-shard transaction log as described in sections 2.6 and 3.3. Fault tolerance was achieved by the implementation of a cluster coordination service as described in sections 2.2 and 3.2 and in appendix A.1.

3. **The file system must be able to read existing files or directories, create new files or directories and append to existing files.**

   This goal was achieved by providing an implementation of the BDFS protocol as described in section 3.6 through which clients may issue file system operations to the cluster and by implementation of a BDFS client as described in section 3.7 which may run as an interactive prompt or may operate as an external library.

The additional behavioural requirements as described in section 2.1 that were implemented as **extensions** to the core success criterion are as follows:

1. **The file system must provide replication for file metadata and directory structure, resulting in fault tolerance against single node failures.**

   This goal was achieved using the same principles mentioned earlier as they applied to the data layer.

2. **The file system must provide strong consistency guarantees for metadata and directory structure operations, meaning that successfully executed operations cannot be lost due to node failures.**

   Strong consistency guarantees were achieved by the use of replication, where operations are only considered committed once they have been replicated to a minimum number of machines. This is described in section 2.6 with details about how the cluster coordination service manages this being presented in section 3.2 and in appendix A.2.

3. **The file system's design must enable full or partial operation under multiple node failures and gracefully recover once nodes rejoin.**

   and

4. **The file system's design must enable partial operation under network partition conditions and gracefully recover once the network is restored.**

   These two goals were achieved by the use of network-partition tolerant cluster coordination service as described in section 3.2 and in appendix A.2.

4.2 Data transfer performance

I'll first take a look at the performance observed with data operations. Specifically, I'll look at the performance of transferring a file from local storage to the distributed file system and vice versa. These are henceforth known as **write** and **read** operations respectively.

For the purposes of this evaluation, I generated a random 512MB file and used it throughout all of my testing. The fact that the file was randomly generated meant that no meaningful compression could be performed and, as such, I would be measuring raw storage and network transfer performance.

4.2.1 Local host performance

The first testing environment used for this consisted of 15 BDFS processes running on the same machine – a MacBook Pro running Mac OS X Mavericks on a 2.5GHz Intel Core i5 processor with 8GB RAM and 128GB of solid state drive storage. 6 of these processes were dedicated to the data layer and each data shard was replicated 3 times. This meant that every write operation actually wrote 1.5GB worth of data on disk.

This test eliminates network traffic and is meant to ensure that BDFS properly manages its disk operations without introducing significant overhead.

It quickly became apparent that the initial iteration performed very poorly, with a write taking around 5 minutes. This was partially due to how replication was being implemented where we would end up writing the contents of a chunk in both the transaction log and in the chunk database.

I changed the replication mechanism such that the transaction log would only store the fact that an operation was performed and replicating machines would instead issue a request to read the modified chunk from the shard leader. This brought down the time taken to perform a write to around 2 minutes, which was still very poor.

At this point, I decided to store chunks outside of BerkeleyDB. At a high level, each write operation performed on a chunk now creates a new physical file on disk and BerkeleyDB only stores a mapping from data chunk identifiers to the correct physical file. This guarantees that we retain the transactionality properties provided by BerkeleyDB while keeping the actual data contents separate. This approach brought down the runtime to around 11 seconds, which is significantly better, yet still falls short of what the solid state drive is capable of.

In the final, fourth iteration, I used memory-mapped files to remove the need for a couple of memory copying operations that were being performed when transmitting chunk data over the network sockets. This brought down the runtime to around 6 seconds, which was exactly how much a local file system copy of a 1.5GB file takes.

One can thus conclude that BDFS manages disk resources efficiently.

Read performance was fairly consistent across all of the 4 iterations, with a slight increase in performance being observed from dropping BerkeleyDB.

Figure 4.1 summarises the write and read performance observed with the four different iterations.

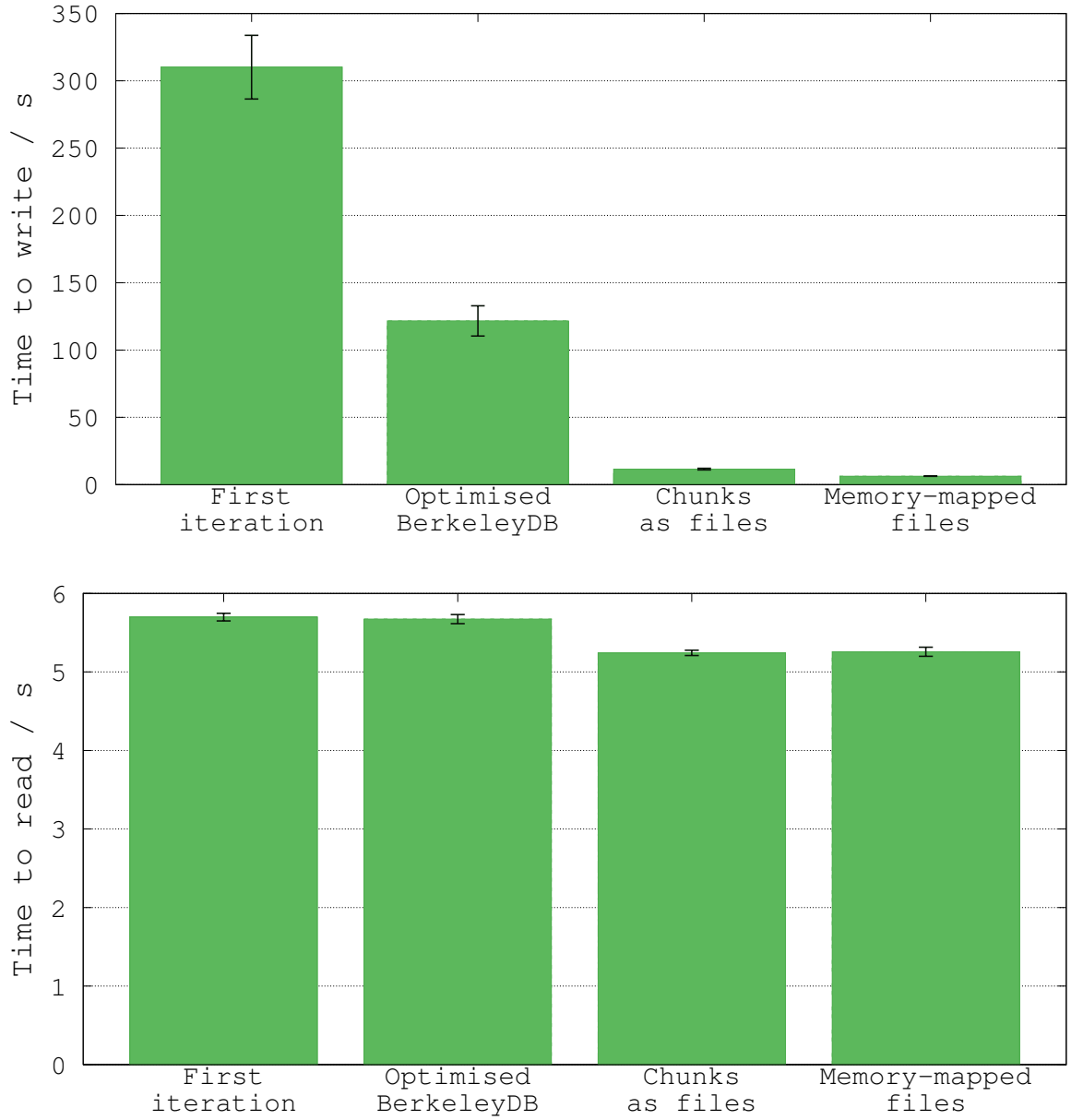

Figure 4.1: Evolution of local host data layer write and read performance over several implementation variants. Graphs show the time taken to write and read a 512*MB file to and from BDFS. Lower valued bars are better. Error bars denote differences observed over a sequence of 10 different test runs.*

4.2.2 Network performance

The second testing environment used consisted of two machines, the first one being the same laptop used earlier and the second one being a desktop computer running an up-to-date Arch Linux distribution on a 3.16GHz Intel Core 2 Duo processor with 4GB of RAM and a 750GB hard disk drive for storage. The two computers were interconnected using a reliable, 1 gigabit Ethernet connection.

It's worth noting that BDFS could run without any issues in a heterogeneous environment consisting of machines with hardware spanning a 5 year time gap and running completely different operating systems.

In this setup, we are interested in ensuring that BDFS does not introduce significant network overhead. The desktop machine runs 3 BDFS processes, one for each file system layer and it performs no replication. The laptop acts as a client transferring information to and from the distributed file system cluster.

We notice the same pattern as in the local host performance test where write times decrease with each iteration from around 6 minutes, to around 3 minutes, to 10.71 seconds, and, finally, to 10.19 seconds. For comparison, a simple file transfer followed by a flush to disk took 8.98 seconds on average.

Read performance was, as expected, similar across all iterations, at an average time of around 10 seconds. For comparison, a simple file transfer over the network, from the server to the client, took 5.65 seconds. By changing the BDFS client to request multiple different chunks in parallel, I was able to improve BDFS read times to 6.49 seconds. This may be explained by the fact that, in the parallel scenario, the requests may be processed by the server in separate threads eliminating latency due to request and response serialising, chunk file opening and reading etc.

We can see that both writing and reading can be performed within 15% of native file system performance, so we can conclude that BDFS does not introduce major overhead over a native file system. Furthermore, since each machine host will be serving independent shards, overall cluster throughput scales up linearly with the number of machines present. We demonstrate this for metadata layer operations in the following section.

Figure 4.2 summarises the write and read performance observed.

4.3 Metadata operation throughput

In order to test performance of the directory and file metadata layer, I've implemented a synthetic benchmark as part of the BDFS client, which opens any number of concurrent parallel connections to machines within the cluster and has each concurrent connection cycle through a sequence of file system operations. We distinguish between the write benchmark, which repeatedly creates a new directory, creates a new file and links another copy of the last created file, and the read benchmark, which repeatedly lists a directory and stats two files.

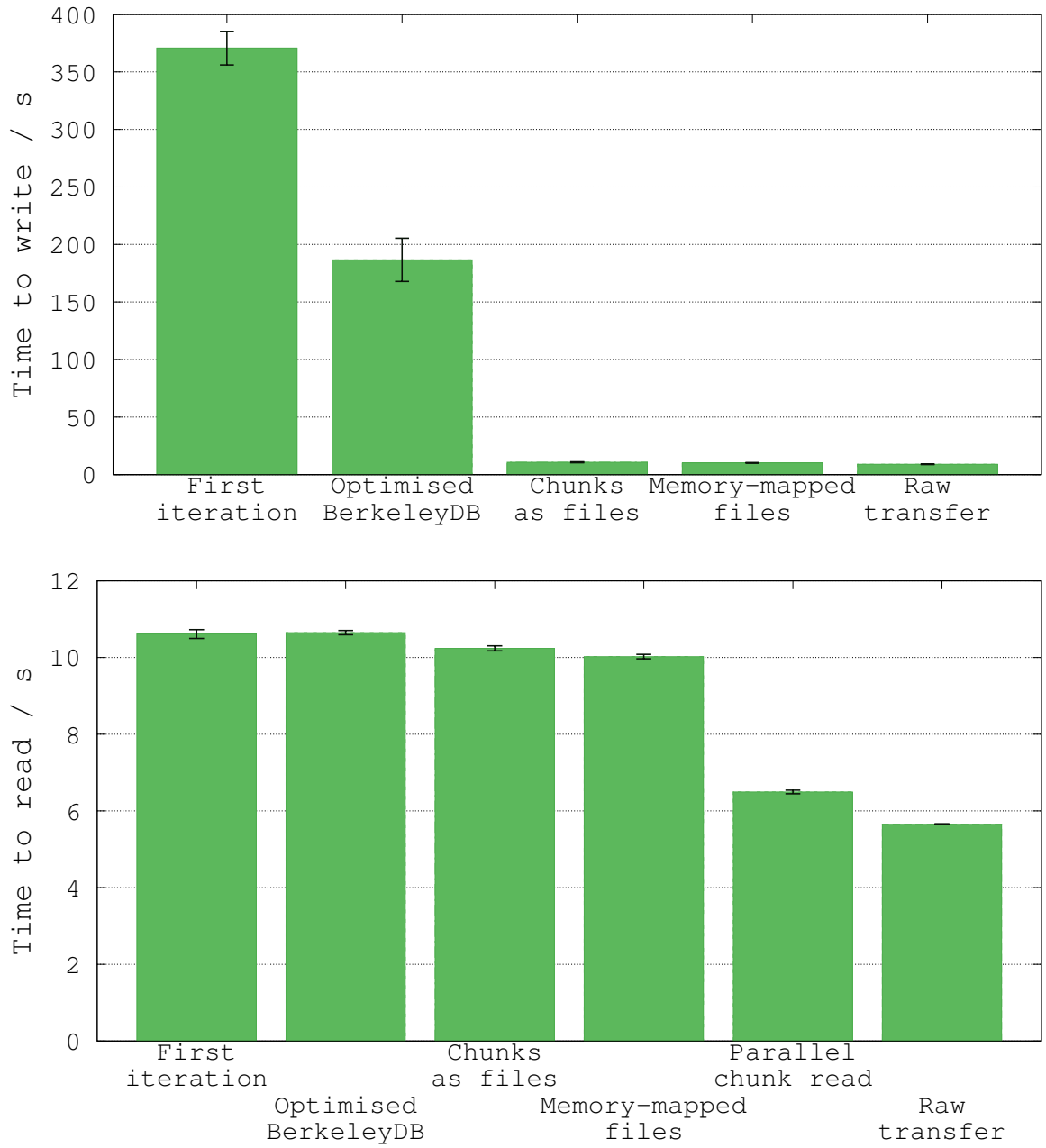

Figure 4.2: Evolution of local host data layer write and read performance over several implementation variants. Graphs show the time taken to write and read a 512*MB file to and from BDFS. Lower valued bars are better. Error bars denote differences observed over a sequence of 10 different test runs.*

These benchmarks are used to measure how many metadata queries per second can the cluster execute and see how this number changes under a variety of factors. All of these benchmarks are performed on a set of virtual machines running on Amazon's EC2 cloud service using hardware assisted virtualization. Each instance had 2 virtual CPUs composed of 6.5 EC2 Compute Units, 7.5GB of RAM and a 32GB solid state drive attached.

Each benchmark test run operated on an initially empty file system and measured the number of responses received every second for a period of 60 seconds.

4.3.1 Single host performance

I started out by measuring the throughput of a single host serving the entire key space and comparing how this changed as I varied the number of shards.

Read performance was pretty much constant throughout. I expected write performance to decrease slowly as the number of shards increased due to distributed transactions, but it actually went up. This is likely due to the fact that BerkeleyDB shards appear to be very fast when there is a small amount of data in them, but eventually stabilise as the amount of data increases. We might conclude then that given enough data, the differences observed will fade away. Additionally, each shard has a caching layer on top which is of fixed size, so the more shards a host serves, the more RAM it is using for caching purposes.

Each host achieved a throughput of roughly 1.400 write queries per second and 12.000 read queries per second for a single shard.

Figure 4.3 shows a comparison of write throughput under the different circumstances and figure 4.4 shows the same comparison made for read throughput.

4.3.2 Impact of replication

Next, I look into the performance impact introduced by replication. We expect replication to reduce write throughput since every single operation that is executed on a shard must be sent over the network and executed on another host before a response can be given to the client.

For the purposes of benchmarking this, I started off with a single shard and measured how throughput was being affected as that shard was being replicated to more and more machines. Read throughput was unaffected, at a steady 12.000 queries per second, since only the shard leader is able to process read queries.

Intuitively, we expect the biggest write performance decrease to occur when going from a shard being on a single machine to it being on two machines. This is the case because we need to incur the extra network latency penalty. As we increase the number of machines, we expect the impact on performance to be less and less because we can issue replication requests in parallel. However, since we do have to wait for the slowest machine, adding more machines does result in a performance degradation nonetheless.

This is exactly what was observed during benchmarking and can be seen in figure 4.5.

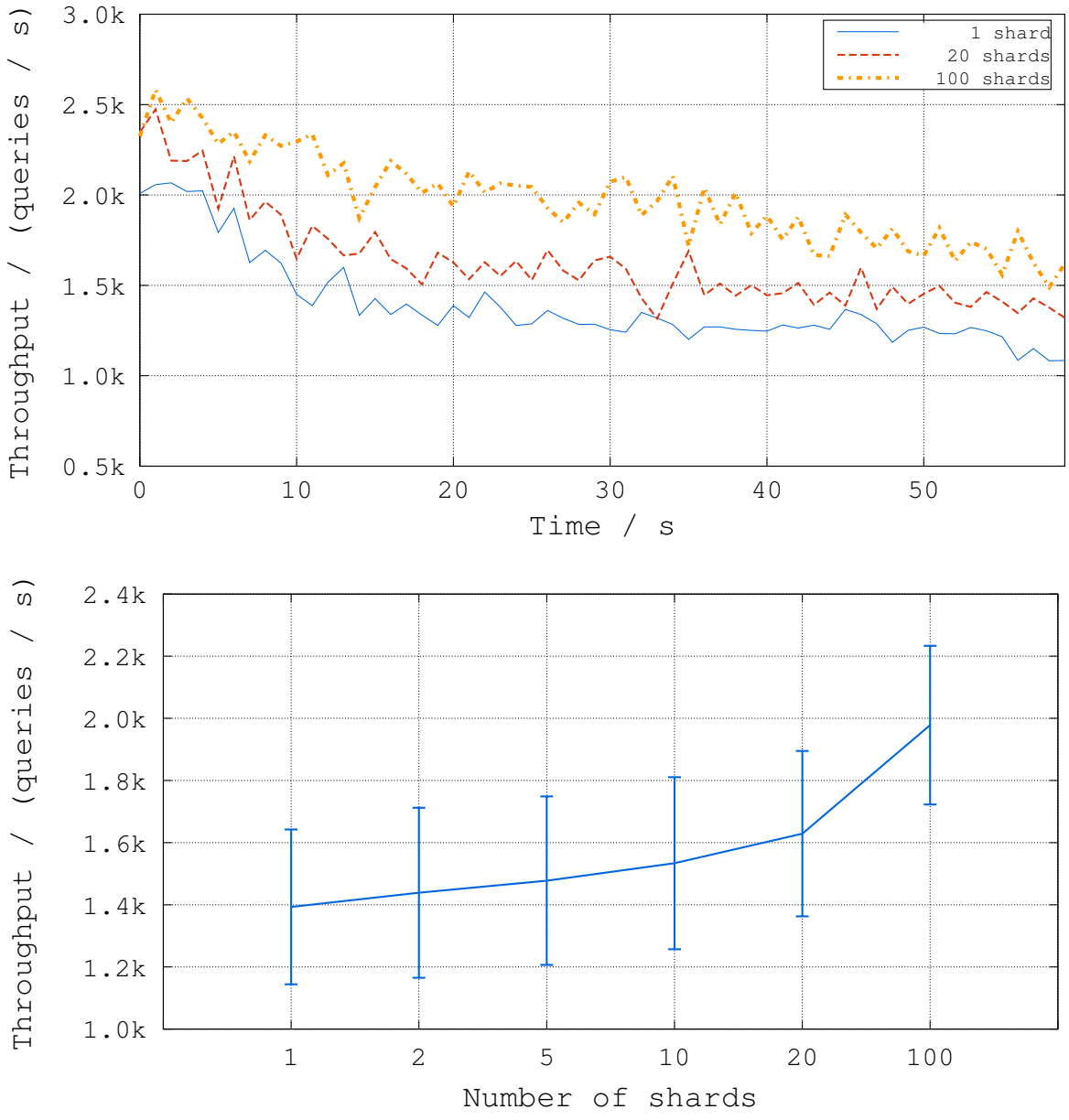

Figure 4.3: Metadata operation write throughput for a single host serving different numbers of shards.

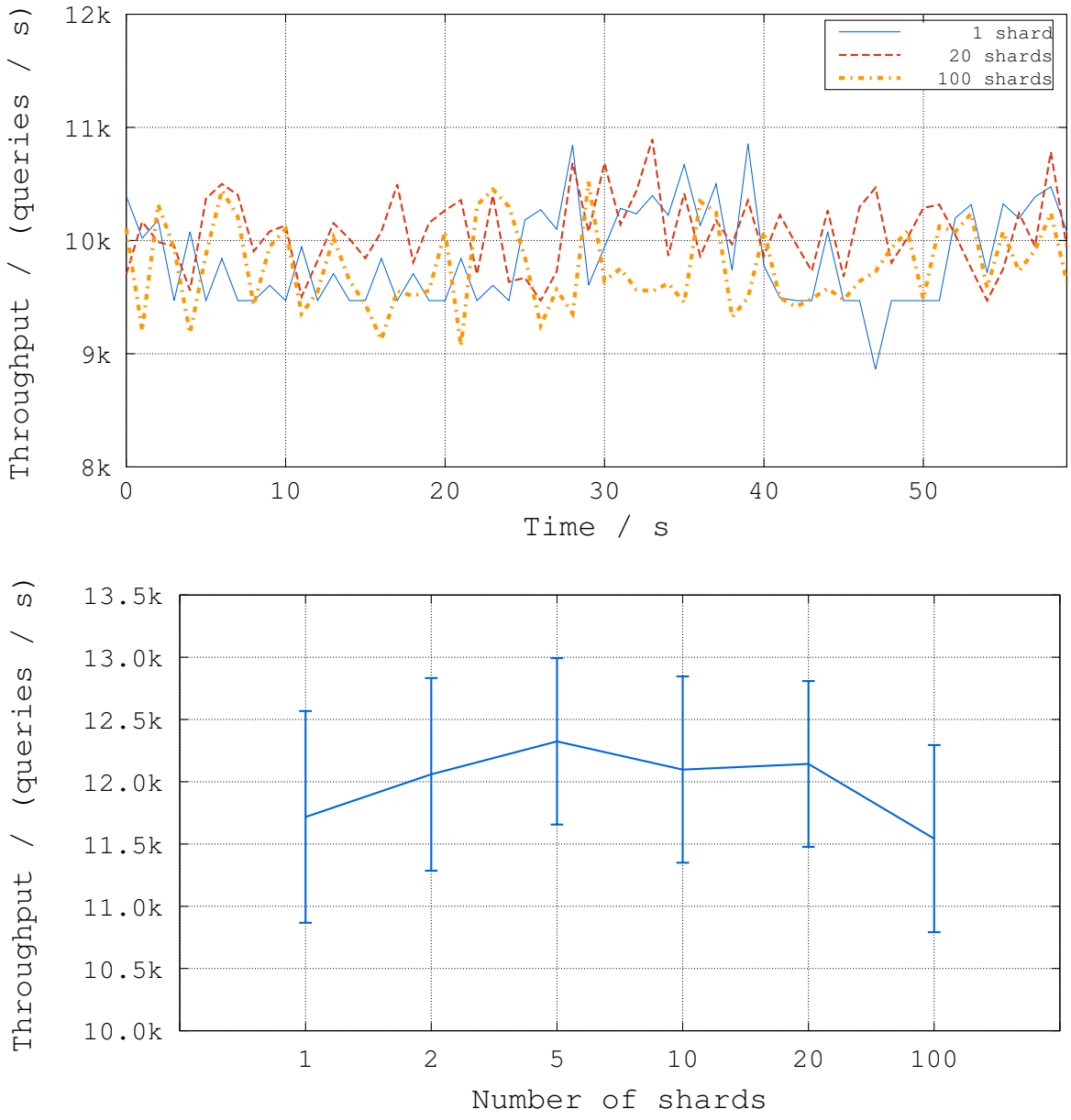

Figure 4.4: Metadata operation read throughput for a single host serving different numbers of shards.

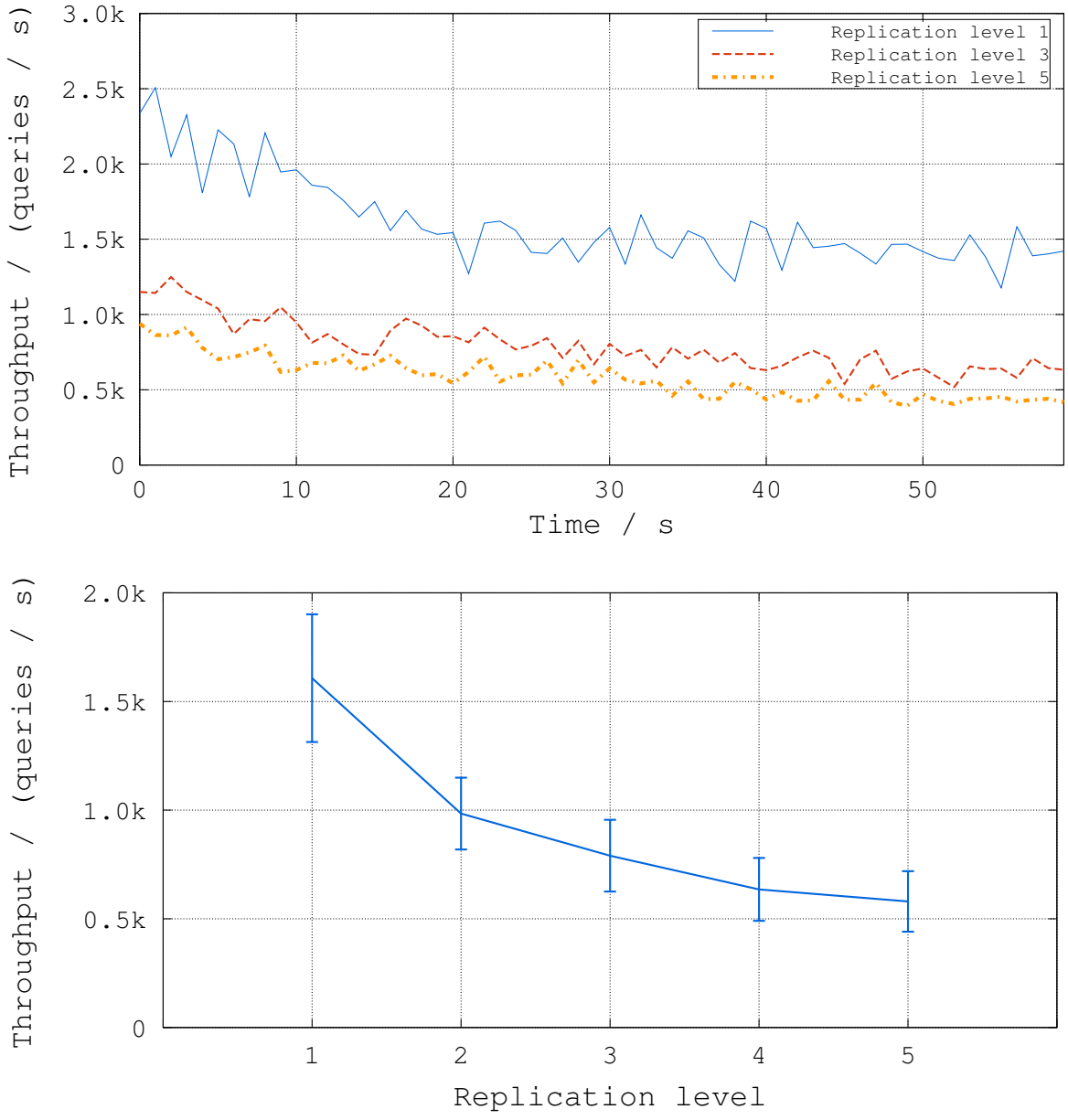

Figure 4.5: Metadata operation write throughput as a function of replication factor.

4.3.3 Multiple host performance

Finally, I look at how throughput scales when the number of machines dedicated to the cluster is increased. For this test, I vary the number of BDFS machines that are part of the cluster and partition the key space evenly among them, without enabling replication.

We would expect performance to scale linearly with the number of machines and that is indeed what happened. Interestingly, after two machines, the BDFS client was no longer able to issue read requests to the cluster at a fast enough rate in order to saturate it and as such multiple parallel clients needed to be employed. Figure 4.6 shows both read and write throughput scaling almost perfectly linear as more machines are added to the cluster.

4.4 Fault tolerance

In order to test BDFS' tolerance to machine failures and recovery I booted up a cluster and purposefully caused failures while running the write benchmark in the background. I performed tests under two different configurations: a 3 machine cluster with a single shard and a 5 machine cluster with 5 shards. Both of these ran with a replication factor of 3 and a strict factor of 2, meaning an entry was considered committed if it was replicated to at least two machines.

20 seconds into the benchmark, the host replicating a shard was killed. This caused a drop in throughput as the leader machine would try to reconnect to the kiled machine for a second until the coordination service removed it from the shard. After 5 seconds the host would rejoin the cluster, which does not affect throughput as the leader does not need to wait for the host to catch up. After 10 seconds, the shard leader would fail causing throughput for that shard to drop to 0 briefly until a new shard leader is designated. Finally, after 5 seconds, the host would rejoin the cluster.

Due to the quality of the network connecting the EC2 virtual machines, the heartbeat timeout used by the coordination service was set at 1 second. This meant that failure detection took roughly between 1 or 2 seconds. In a high speed home network, I've managed to successfully run a cluster in full load using a heartbeat timeout of only 0.15 seconds.

Figure 4.7 demonstrates the behaviour described.

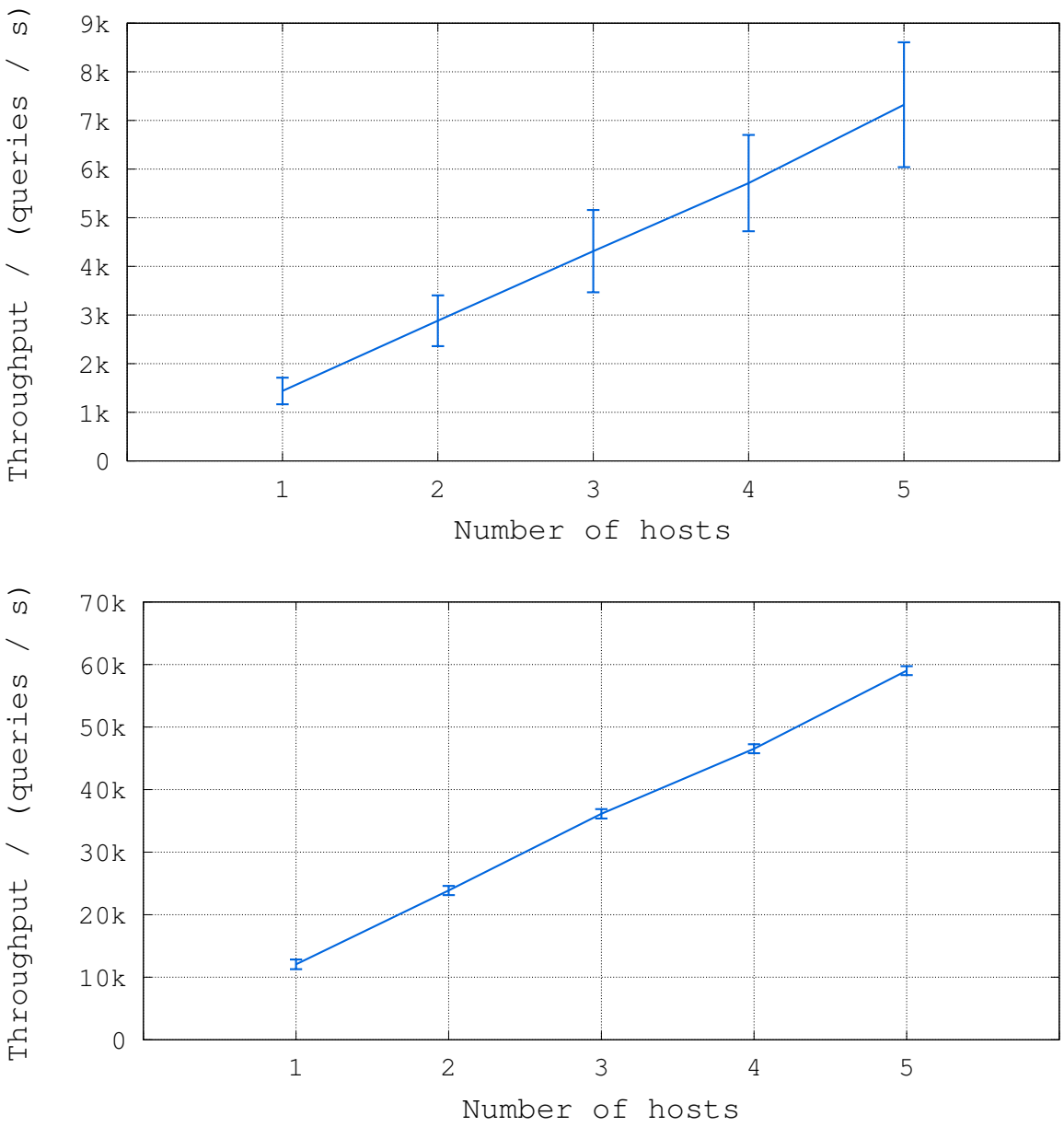

Figure 4.6: Metadata operation throughput as a function of cluster machines. The top graph shows changes in overall write throughput as the number of machines in the cluster is increased, while the bottom graph shows changes in overall read throughput.

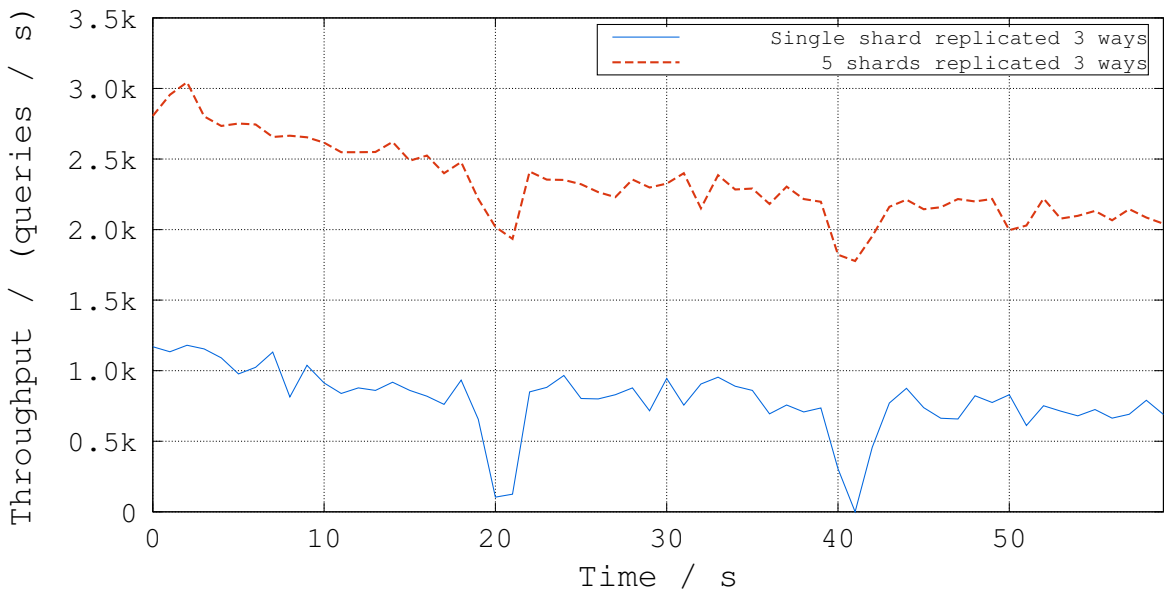

Figure 4.7: Throughput of BDFS clusters undergoing forced failures.

4.5 Summary

In this chapter I showed that BDFS fulfilled all of the success criteria put forth in the original proposal as well as the extension success criteria mentioned in the Preparation chapter. I have demonstrated succesful operation of BDFS in multiple environments and configuration setups, showed that it performs well when it comes to data transfer throughput when compared to a native file system and evaluated its performance in metadata operations. As expected, BDFS scales linearly with the number of machines in the cluster and increasing replication factor degrades performance only in a sublinear fashion. Moreover, I've shown BDFS to be fault tolerant and be able to detect and recover from machine failures within seconds.

Chapter 5

Conclusion

This dissertation has described the design, implementation and evaluation of the BDFS distributed file system. Looking back to the start of the project, I must admit I was intimidated by the amount of unknowns surrounding the project, its inherent complexity and the difficulties of implementing a functioning distributed system. I have to say, after going through the process, that had I fully realised the amount of work required to carry this project to completion, I would have tempered my ambitions. Nevertheless, I thoroughly enjoyed this journey and it led to a tremendous sense of satisfaction upon every single milestone achievement.

5.1 Achievements

All of the core success criteria set out in the original project proposal were achieved and surpassed. BDFS demonstrated a working fault tolerant file system which distributed both file data as well as directory structure and file metadata across multiple machines. It eliminates the single point of failure present in the metadata layer in the original HDFS [1] file system and in the original GFS [2] file system. It provides fault tolerance via replication and raises the ceiling on the number of files that can be stored in the file system. It was shown to scale linearly both in terms of throughput and in terms of capacity with the number of machines allocated to it and it was shown to use network and disk resources efficiently.

From a personal point of view, I have gained a significant amount of insight into distributed systems, concurrency and networking both from reading several academic research papers on the topics and from running into and dealing with multiple and very diverse practical issues. Additionally, I feel I have gained a significantly better ability to estimate, plan out and execute large scale projects as a result.

I consider the project a success both from the standpoint of what it was set out to achieve and demonstrate and from the standpoint of my personal development.

To my knowledge, BDFS breaks new ground in how it distributes and shards file system metadata. Whereas the approaches taken by HDFS Federation [8] and MapR [7] build on top of existing infrastructure and are as a result more limitted or

require more manual configuration and tuning, BDFS was built from the ground up with this in mind.

5.2 Lessons learnt

My main takeaway from this project is that, even though I felt I was reserved and I conciously took on what I considered to be a big project, it is extremely easy to underestimate the complexity of implementing such a large-scale system. It is often the case that what is very simple to describe in a few words hides a number of edge cases which need to be carefully thought through and handled. This is especially true in the context of a system that is subject to concurrency, with many days being spent hunting down subtle bugs that would only crop up every so often in unpredictable ways. Furthermore, I have learned that I should not always put complete faith in the open source external libraries I am using as they might themselves contain bugs as was the case with libev's handling of asynchronous events.

From a software engineering standpoint, I have seen first hand the benefits of performing rigurous, systematic testing of every significant feature as it is being implemented both automatically via the use of unit testing and end-to-end testing and manually via memory usage error detecting tools. I felt I have improved my knowledge of the latest C++ standard by quite a lot and improved my coding style as a result.

5.3 Future work

The BDFS implementation is complete in the sense that it is working and it fulfils the success criteria for the project as well as several extensions. Nonetheless, there is scope for some further work on it, as described below.

- **Performance tweaks:** There is great scope for improving BDFS performance. One can look at building a custom persistent storage solution that does not rely on BerkeleyDB which was shown to not perform very well in committing large amounts of data.

  The current format is inefficient as it uses BerkeleyDB to store a transaction log which could very well be stored in a simple file and it relies on BerkeleyDB's transactionality features to update both the log and the database entries atomically. Since we already have a transaction log for the purposes of replication and distributed transactions, we are already able to implement local transactions ourselves more efficiently by bypassing BerkeleyDB altogether.

  Other areas one can look at to improve performance are the use of datastructures with more fine grained locking, implementation of an in-memory caching layer, use of DirectIO to gain raw access to disk and use of `iovec`s to enable receiving and sending of messages over multiple sockets in parallel.

- **Shard balancing:** The mechanisms described in section 3.2 and appendix A.1 for redistributing workload among cluster machines are relatively simple and there is great scope for devising more complex, better performing solutions.
- **Implementation of lazy-move operation:** Described in appendix D, this method of implementing the move operation should increase its performance without having a major impact on any of the other operations.

Bibliography

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Storage Conference*, 2010. http://storageconference.org/2010/Papers/MSST/Shvachko.pdf. i, 1, 53, 68

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP*, 2003. http://research.google.com/archive/gfs-sosp2003.pdf. i, 1, 53, 68, 69

[3] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *OSDI*, 2004. http://research.google.com/archive/mapreduce-osdi04.pdf. 1, 68

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *OSDI*, 2006. http://research.google.com/archive/bigtable-osdi06.pdf. 1, 68

[5] A. Ryan, "Under the Hood: Hadoop Distributed Filesystem reliability with Namenode and Avatarnode," 2012. https://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920. 3

[6] A. Foundation, "HDFS High Availability with NFS," 2014. http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithNFS.html. 3

[7] "Design, Scale and Performance of MapR's Distribution for Apache Hadoop," in *Hadoop Summit 2011*, 2011. http://www.youtube.com/watch?v=fP4HnvZmpZI. 3, 53

[8] A. Foundation, "HDFS Federation," 2014. http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/Federation.html. 3, 53

[9] A. Fikes, "Google Storage Architecture and Challenges," in *Faculty Summit 2010*, 2010. http://static.googleusercontent.com/media/research.google.com/en//university/relations/facultysummit2010/storage_architecture_and_challenges.pdf. 3

[10] D. Fetterly, M. Haridasan, M. Isard, and S. Sundararaman, "TidyFS: A Simple and Small Distributed File System," in *USENIX ATC'11*, 2011. http://research.microsoft.com/pubs/148515/tidyfs.pdf. 4

[11] L. Lamport, "The part-time parliament," in *ACM Transactions on Computer Systems 16*, 1998. http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf. 4, 17

[12] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," 2014. https://ramcloud.stanford.edu/wiki/download/attachments/11370504/raft.pdf. 17

[13] J. Highsmith and M. Fowler, "The Agile Manifesto," 2001. 19

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *USENIX'10*, 2010. https://www.usenix.org/legacy/event/usenix10/tech/full_papers/Hunt.pdf. 24

[15] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance.," *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 335–348, 1989. 71

Appendix A

Cluster coordination service

A.1 Balancing shards across machines

First of all, we want to be able to reason about the computational resources involved in serving or replicating individual database shards. We refer to a numerical value associated with these resources as the **load** caused by a shard.

The load of a machine represents how much work it needs to perform to be part of the BDFS cluster and can be viewed as the sum or the individual shard load values. These values can be computed as a function of several parameters including the amount of data stored within the shard, the amount of requests per second that hit the shard and the read to write request ratio.

For the purposes of my implementation, I chose the simpler approach of assuming that load across all shards is evenly distributed and that the amount of read and write requests are fairly balanced. As such, I associate a load value of 2 for serving a shard and a load value of 1 for replicating a shard. Under some use-cases, this simplification might prove suboptimal, but the approaches described work equally well for a more complex load function.

Our goal is then to distribute load as evenly as possible across the cluster.

If a new machine is joining the cluster, we can take the opportunity to allocate under-replicated shards to under-loaded machines. This can be done by repeatedly choosing the least loaded machine and assigning the highest load under-replicated shard to it.

When a machine recovers from failure, for any shard that becomes over-replicated, we can repeatedly choose those machines which replicate the shard and the most loaded and remove them from the shard.

Once that is done and we have updated load values for each individual host, we can partition the machines into two groups: those that have below-average-load and those that have above-average-load. We can then choose to move a shard from the an over-loaded machine to an under-loaded machine as long as this does not result in either machine “swapping sides”. This can be done by repeatedly identifying the pair of heaviest-loaded and lightest-loaded machines that are able to swap a shard

and performing the swap until convergence. While this approach is not necessarily optimal, it yields reasonable results as it tries to repeatedly close the gap between the most over-loaded and most under-loaded machine.

A.2 Network partition tolerance

In the event of a network partition, we have already established that ZooKeeper will continue to function as long as at least a majority of its machines are still operational within the same partition. All BDFS machines that are not connected to this functioning ZooKeeper partition will pause and cease to accept user requests until connection is re-established. All hosts that are still connected to ZooKeeper will simply operate with their reduced quorum.

The shard replication protocol has two configurable parameters, one controlling the desired level of eventual replication, `ER`, and one controlling the desired level of strict replication, `SR`. For a strict level of replication of 3 and an eventual replication level of 5, a transaction is considered committed once it is confirmed by at least 3 machines of the 5 which store the shard. As such, it is possible that 2 machines are lagging behind slightly, which means that in the event of a partition, we need to have at least `ER - SR + 1` machines alive within the shard in order to be sure that we did not miss any committed updates. Thus, the coordinator will only ever assign a machine to serve a shard if the proper quorum is met and if the machine is the most up to date (the one with the highest last log entry) out of those present.

Upon network recovery, I simply treat the new hosts as individual recovering failing hosts and bring them back up to speed with the changes that have occurred in the meantime.

This makes BDFS network partition tolerant by guaranteeing that “split-brain” behaviour does not occur and by ensuring that partial function is possible only when we can be sure that no data has been lost.

Appendix B

Distributed transactions

B.1 Per-shard transaction log

As described in the section 2.7, I extend the log system used for consistency purposes in order to implement transactions occurring over multiple shards potentially across different file system layers.

All log entries store a description of the operation that needs to be performed on the file system as well as information about the identity of the user requesting the operation. In addition, in order to support distributed transactions, we distinguish between three different types of log entries: committed, checkpoint and aborted.

Committed log entries signify the fact that the operation has fully completed execution and thus the client requesting it can be informed of success once the updates have been replicated as desired.

Checkpoint log entries signify the fact that the operation has reached a point in its execution where it must now wait for some reason or another. This could, for example, be because it needs to acquire a lock or because it needs to issue a sub-operation request to a remote shard and await a response from it as well. Checkpoint log entries also need to store some information about the point at which execution of the transaction was blocked waiting.

Finally, aborted log entries signify the fact that for some reason or another the operation has had to be aborted and the effects of all previous checkpoints were reverted.

An operation can thus result in zero, one or multiple checkpoint log entries followed by a final entry which can either be committed if the operation was successfully performed or aborted if the operation failed. Note that if no checkpoints were saved to the transaction log, there is no need to commit a single aborted log entry as there are no changes that need to be reverted. In order to identify which entries belong to which transactions, we additionally store within entries the identifier of the last entry that was part of the same transaction, if it exists.

The transaction logs are replicated per-shard and thus if the machine serving a shard fails, the one that replaces it can traverse the transaction logs and resume

execution of any transactions that were ongoing as described in section 2.7.

B.2 Transaction isolation

It can be the case that multiple operations touching the same entries within the same shard can be issued by different clients and we need a way of ensuring that they do not interfere with each other. I employ a simple strategy whereby one can only fetch or update an entry in a shard if it first obtains a lock on it.

Implementation-wise, for each shard that a machine serves, I maintain an in-memory set of entries which are currently locked. This can be implemented efficiently using a simple hash table supporting concurrent access. Requests that are waiting for a locked entry are simply added to a queue and signaled once the resource becomes available.

There is no need to replicate the lock state to other machines storing this shard. If the machine serving a shard fails, then each lock is either part of a distributed transaction which will have checkpointed after acquiring its locks, or part of a local operation that is not committed.

It can be the case that file system operations touch multiple shards at the same time as well. This can occur, for example, when creating a new file, since we need to atomically create a new file node within a shard found in the file layer and link that file to a directory which is an operation performed in a shard in the directory layer. I avoid the problems that usually come up with having a distributed locking system by enforcing that locks may only be held **locally**. This means that operations must be split into multiple smaller ones, each locking different bits of the file system. These are executed as part of a distributed transaction. Since we do not acquire all the necessary locks at the beginning, sometimes the transaction will need to be aborted and rolled back.

B.3 Ordering of operations and snapshotting

In order to implement replication, we need a way of signaling to each replicating machine what operations to perform on its copy of data in order for it to be consistent with the data stored by the machine serving the shard. It is important to note that the order in which operations on the same pieces of data are performed does matter.

The approach taken is to fully serialise all operations that are committed on a shard and store them in a log. Each operation is assigned a unique identifier and, for any two operations identified by IDs txn_1 and txn_2, $txn_1 < txn_2$ if and only if the first operation occurred before the second operation.

In our implementation, transaction IDs are only 32-bit integers, so if a cluster operates for long enough it is possible that we end up exhausting the 32-bit space allocated. In addition, it is the case that many of the operations found within a

transaction log become redundant once data is overwritten and we end up wasting disk space. As such, it makes sense to regularly perform **snapshotting** to avoid both of these problems.

When a snapshot is performed, we ensure that all machines storing a particular shard are up to date, we increment a snapshot version counter and reset the transaction log. It is then possible to determine whether or not two machines are lagging behind one another by first comparing their snapshot versions and then, on equality, comparing their last transaction log entry numbers.

Appendix C

BerkeleyDB data stores

In this chapter we look at the alternative data stores that BerkeleyDB provides:

- **B-Tree** provides an unrestricted key-value data store where both keys and values are binary data blobs. Implementation-wise, the database is backed by a specialised on-disk B-Tree. This makes for a logarithmic bound on the time complexity required to fetch or update a value and it also allows one to traverse the entries in increasing or decreasing order of their key values.
- **Hash** provides an unrestricted key-value data store similar to **B-Tree**, the only difference being that the underlying implementation is backed by a hash table. This makes for an amortized constant bound on the time complexity required to fetch or update a value, allowing for more data to be efficiently stored, but it does mean that traversing entries is no longer performed in-order.
- **Queue** provides a key-value data store in which keys are restricted to 32-bit integers and in which values are padded or truncated to a fixed size that is established at database creation. The more rigid structure allows very efficient implementations of the get and update operations and allows per-entry rather than per-database mutual exclusion schemes. As its name suggests, **Queue** is optimised for use as a highly concurrent persistent queue that can be used, for example, in a producer-consumer model. Keys within the queue are not fixed to the value specified at insertion, but instead denote the position of the elements in the queue.
- **Recno** provides a key-value data store in which keys are restricted to 32-bit integers and in which values can be binary data blobs or arbitrary size. This datastore stands somewhere in between **Hash** and **Queue**. It can be configured to restrict values to be of fixed size, thereby increasing performance. Additionally, it can be configured such that keys are either fixed or they represent logical numbers as in the case of **Queue**.

In all three file system layers, we use the Recno format for the transaction log database. The use of Recno as opposed to Hash stems from the fact that log entries

can simply be 32-bit integers as opposed to the larger keys required for addressing directory, file or data chunk entries. For this reason, all other databases are stored in Hash format. The use of Hash as opposed to B-Tree is motivated by the slightly better performance of get and update operations and by the fact that in-order traversal is not a requirement for our purposes.

Appendix D

Recursive move operation

Section 2.4 discusses two approaches to sharding the directory layer, the second of which optimises for access operations at the expense of recursive move operations.

Within the context of this approach, which ended up being implemented in BDFS, I describe one way of implementing the move operation.

A more efficient implementation would involve using a lazy-move mechanism. Instead of actually performing the move, a special entry is added to the new location and the old directory is marked as moved. Lookups for the old directory behave as if the directory did not exist, while lookups for the new directory will retrieve the special entry and trigger an actual move to be performed. When performing the move, all subdirectories are marked as moved as well and their entries are converted to these special entries. This approach has the advantage that it reduces the amount of work performed per move operation by a significant factor, but it does increase implementation complexity as there are quite a few different edge cases to consider.

Appendix E

Project Proposal

Computer Science Project Proposal

The BDFS Distributed File System

Bogdan Cristian Tătăroiu, Churchill College, bct25

Originator: Ionel Gog and Malte Schwarzkopf

17^{th} October 2013

Project Supervisor: Ionel Gog & Malte Schwarzkopf

Director of Studies: Dr J. Fawcett

Project Overseers: Prof P. Robinson & Dr R. Watson

Introduction and Description of work

In recent years we have seen a significant increase in interest around frameworks which enable distributed processing of relatively large sets of data, both as the result of reduced costs of computer hardware allowing individuals and companies to store larger amounts of data over more machines and as the result of the inherent shift towards parallel applications caused by the advent of multi-core CPU's.

One such framework, Hadoop, started off as a result of scientific papers published by Google detailing its internal systems. The first of these papers, published in 2003, describes the Google File System [2], a distributed, fault-tolerant file system designed to run on large clusters of commodity hardware and provide a solid foundation upon which data-intensive applications can be built. The next paper, published in 2004, describes the first system built on top of GFS, the MapReduce programming model [3] which enables processing and generating data sets in a highly distributed manner by dividing the input data set into smaller subsets and modeling real world tasks in terms of a sequence of functional map and reduce steps that can be run in isolation on each subset of data. The last of these papers, published in 2006, describes BigTable [4], a distributed storage system built on top of GFS with support for structured data and high performance database queries.

Hadoop was developed internally at Yahoo, who at the time were seeing similar data processing needs to Google and sought to provide equivalent frameworks to its own developers in the form of HDFS [1], Hadoop MapReduce and HBase respectively. It has since been open sourced and adopted by many companies around the world as the canonical framework for data processing.

Through this wide adoption, a number of problems with each of the three components have been identified originating both from lack of feature parity, with the Hadoop implementation trailing behind the Google papers, and from inefficiencies present in the implementation, poor architectural choices or companies requiring different use cases which were not relevant for the author's original purposes.

Since GFS/HDFS sits at the foundation, it makes sense to try to address issues with it first. One complaint with the reference open source implementation of HDFS is relatively low throughput compared to machine limits, which seems to stem from inefficient implementation rather than poor architectural design. Other complaints which stem from decisions made when designing GFS include the presence of a single point of failure in the file meta-data layer and poor performance when dealing with a very large number of small files. Other limitations include the lack of support for random access writes or other POSIX file system features like symbolic or hard links.

I propose writing an implementation of a distributed file system from scratch having the design decisions behind GFS/HDFS as a starting point while trying to address as many of the issues identified above as possible. One particular focus point will be on distributing the file meta-data layer across multiple machines as a means towards improving performance around very large number of small files.

The core of the project will consist of implementing a file system which distributes both file data as well as file system structure and metadata across multiple machines and is capable of at least reading existing files, creating new files and appending to existing files while keeping file contents replicated over more than one location. Addressing each of the issues highlighted above as well as providing a more complete feature set would constitute optional, but highly desirable extensions.

Starting Point

To complete the project, I will draw from

1. Papers published on GFS and HDFS as well as articles and presentations made by companies (for e.g. MapR) trying to address some of the issues pointed out in the Introduction.
2. Part IB Courses from the Computer Science Tripos, specifically the Concurrent and Distributed Systems and Programming in C and C++ courses from IB Lent.
3. Previous Programming Experience from summer internships at Facebook, Twitter and Dropbox.

Resources Required

Development will primarily be done on my personal laptop running Mac OS X and desktop computer running Linux. An unintended side-effect of this will likely be enforced cross-platform support.

I intend to do most of the development in C or C++, version controlled using Git, hosted on GitHub and backed up to Dropbox. The project will likely be released as open source either from the start so as to permit external code review or after graduation.

For the purposes of deploying, testing and benchmarking the distributed file system I will make use of a cluster of SRG test machines within the computer laboratory as well as Amazon EC2 academic credits provided by my supervisors.

Work to be done

The objective of this project is to successfully demonstrate a working file system which distributes both file data as well as file system structure and metadata across multiple machines and is capable of at least reading existing files, creating new files and appending to existing files while keeping file contents replicated over more than one location. The core design of the file system will be based on the GFS [2] paper and other published work around HDFS.

GFS and HDFS share a very similar architecture, which is described very summarily in the following paragraph. A cluster consists of a single master node and multiple chunkservers. Files are partitioned into fixed-size chunks which are each identified by an immutable and globally unique chunk handle assigned by the master at creation. Chunkservers store chunks on local disk, and each chunk is replicated a number of times across multiple chunkservers. The master maintains all file system metadata, including the directory hierarchy, access control information and mappings from files to chunks and maintains all of it in memory for performance reasons.

The underlying assumption here is that the system should store a modest number of very large files, which means support for small files can be improved. One problem with small files is that usually there are a lot of them and this is where the master becomes a bottleneck since each file and block requires at least some amount of bytes of metadata to be stored in RAM. Therefore, by distributing the file system metadata across multiple machines we should be able to lift this limit by orders of magnitude. In addition to this, one optional extension could consist of figuring out a way to coalesce multiple small files into a single block, thereby improving locality when streaming data sets which had been previously partitioned into many small files.

The fact that the GFS master is a single point of failure means potential data-loss as well as increased periods of down time. A good optional extension would be to provide some form of fault tolerance for the metadata layer of the file system, perhaps reusing the same type of fault tolerance present in the data layer.

For the purposes of evaluation, we can look both at artificial tests measuring for e.g. read and write throughput within various conditions, performance around creating a very large number of files or behaviour under different level of concurrency and at integrating with various other data processing frameworks such as PowerGraph, Naiad or even Hadoop MapReduce.

Success Criterion for the Main Result

As described in the previous section, the project will be a success if it will be a working file system which

1. Distributes file contents, metadata and directory structure over multiple machines.
2. Provides replication for file contents resulting in fault tolerance against single node failures.
3. Is able to read existing files, create new files and append to existing files.

Possible Extensions

As pointed out in the previous sections:

1. Try to improve performance when compared against current HDFS implementation. Could be done by starting multiple parallel connections on read, having the master signal chunkservers to prefetch data into RAM or bypassing the Linux file caching layer and implementing own policy.
2. Coalesce multiple small files into smaller number of blocks to improve locality.
3. Use information theoretic dispersal algorithms [15] to provide fault tolerance in a more space efficient manner to the current hard replication method.
4. Implement redundancy within the file metadata layer.
5. Implement additional POSIX file system semantics (random write, symbolic and hard links) as well as unsupported GFS semantics (concurrent append, snapshots).

Timetable: Workplan and Milestones to be achieved.

0. October 8^{th} - October 25^{th}

Preparation (reading papers and other literature around GFS, HDFS and distributed file systems, identifying which issues to tackle) and submission of project proposal.

1. October 28^{th} - November 10^{th}

Implementation work commences, further research made into specific libraries or frameworks to use, and into establishing a reasonable protocol for communication between various layers of the file system and clients.

Milestone: Basic server-client architecture up and running.

2. November 11^{th} - November 24^{th}

Work on a distributed file metadata layer and client routing mechanisms.

Milestone: Ability to create and manipulate files within the file system without storing content.

3. November 25^{th} - December 8^{th}

Work on file contents layer.

Milestone: Ability to write data chunks and have them be replicated.

4. December 9^{th} - December 22^{nd}

Integrate the two layers and perform end-to-end testing.

Milestone: Have a fully functioning core file system.

5. January 6^{th} - January 19^{th}

Work on various extension elements with a focus on performance tweaks and easy to implement functionality. Start working on progress report.

6. January 20^{th} - February 2^{nd}

Write synthetic tests to compare BDFS and HDFS on various metrics.

Milestone: Progress report submitted. Obtain first set of artificial test data. Test BDFS fault tolerance.

7. February 3^{rd} - February 16^{th}

Integrate BDFS with at least one other data processing framework and perform benchmarks against HDFS.

Milestone: Obtain first set of more real world test data.

8. February 17^{th} - March 2^{nd}

Buffer period for rectifying potential issues uncovered during testing. If no problems identified, then continue work on more extension elements.

9. March 3^{rd} - March 16^{th}

Another buffer period for rectifying potential issues uncovered during testing. Begin writing dissertation and work on more extension elements, time permitting.

10. March 17^{th} - March 30^{th}

Finalise first draft of the dissertation while putting finishing touches on codebase and performing more benchmarks as needed.

Milestone: First draft of the dissertation submitted to supervisors.

11. March 31^{st} - April 30^{th}

Last minute touches on codebase and dissertation. Work done as needed and as time permits.