Mounting External Storage in HopsFS

Gabriel Vilén

Delft University of Technology

Mounting External Storage in HopsFS

Master's Thesis in Computer Science

Distributed Systems Group
EIT Digital MSc Program Cloud Computing and Services
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Gabriel Vilén

28th September 2018

Author
Gabriel Vilén

Title
Mounting External Storage in HopsFS

MSc presentation
5th October 2018

Graduation Committee

prof. dr. D. H. J. Epema	Delft University of Technology
dr. J. S. Rellermeyer	Delft University of Technology
dr. J. Dowling	KTH Royal Institute of Technology
dr. A. Katsifodimos	Delft University of Technology

Abstract

A desirable feature of distributed file systems is to enable clients to mount external namespaces, allowing them to access physically remote data as if it was transparently stored in the local cluster. This feature is desirable in a variety of scenarios, such as in enterprise hybrid-cloud deployments, where some data may be stored remote on the cloud while other in the local file system. This feature introduces questions as how to design, implement, and handle various file system semantics, such as different consistency levels and operational performance between the remote and local storage. For instance, a POXIS-complient block based file system is strongly consistent and does not pose the same semantics as a more relaxed eventual consistent object-based cloud storage. Further, it is interesting to benchmark how such an integrated remote storage solution performs and what potential speedups that might be possible.

To examine such questions and provide the remote storage feature in HopsFS, a scale-out metadata HDFS distribution, this thesis presents an Proof-of-Concept (PoC) implementation that enables the local file system to be transparently integrated with an external remote storage. The PoC is storage provider agnostic and enables a HopsFS client to on-demand pull in and access remote data from an external storage into the local file system.

Currently, the solution works by mounting a snapshot of a remote storage into HopsFS, a read-only approach. For write and synchronization work, frameworks that layers stronger consistency models on top of S3 may be used, such as s3mper. Performance of the implemented feature has been benchmarked against remote Amazon S3 buckets, showing results of fast throughput for scanning and persisting metadata, but slower (4 Mb/s) for pulling data into the cluster through the Amazon cloud connector S3A. To improve performance, caching by replication of the remote files onto local disk storage has shown to drastically increase the read speed (up to 100x). The work done in this thesis shows a promising PoC that successfully serves remote blocks into a local file system, with future work including supporting writes, improved caching mechanisms, and reflecting off-band changes between the remote and local storage.

Preface

This thesis has been conducted as part of the EIT Master Program, Cloud Computing and Services, at the start-up Logical Clocks in Stockholm Kista, formed by members of the Distributed Computing Group at KTH Royal Institute of Technology and RISE SICS AB. Logical Clocks has the goal of bringing hyperscale Data Science to the Enterprise through Hops - a data platform (HopsWorks) backed by *the world's most* scalable filesystem (HopsFS) with scale-out compute, GPUs, and streaming analytics [7].

I would like to thank my supervisors for their feedback and remarks on the work; Dick Epema, chair of the Distributed Systems and Faculty of Electrical Engineering at TU Delft, Jan Rellermeyer, professor at the Distributed Systems depart at TU Delft, and Jim Dowling, professor at the department of Software and Computer Systems at KTH and CEO and co-founder of Logical Clocks. Further, thank you to Gautier Berthou, PhD student and admin of Hops, for the much appreciated help and advices, spanning from code implementation to bug hunting. To my colleague and friends from my bachelor program at KTH, and the master program at TU Berlin and TU Delft, thank you for all the collaborations and support that has gotten me this far. And finally, thank you for reading.

Gabriel Vilén
Stockholm, Sweden
28th September 2018

Contents

Chapter 1

Introduction

The massive global scale of Big Data is increasing every year. Popular statistics and infographics often showcase the amount of data, such as social media content, emails, or videos, being produced, uploaded, and stored daily on servers around the world. Such numbers are impressive, with trillions of gigabytes of data being created each day and many companies storing hundreds of terabytes of data [30]. The increasing volume of Big Data raises the need for scalable, reliable, and cost-efficient file systems and storages.

Distributed file systems, such as Google's File System (GFS) [24] and Hadoop's Distributed File System (HDFS) [5] have decreased the burden of storage and processing of Big Data by enabling distribution of storage and processing systems, such as MapReduce, to be built on top of the file system. Such systems work by distributing data and computation over many nodes, thus leveraging parallel computation that can span thousands of commodity machines in a cluster. To provide fault-tolerance, the file system's data is replicated over multiple nodes which take up considerable amount of excessive storage. Cloud storages, such as Amazon Simple Storage (S3) [3], where immense amount of data can be stored for a relatively cheap price, has aided in the reduction of storage cost for Big Data.

However, some data might not be suitable for storing on the cloud, such as sensitive or confidential information, or frequent accessed data for computing jobs (due to latency and cost of reading and writing to a remote cloud server). Providing an efficient integration between local and remote storage, such as a cloud storage, is a desirable feature in distributed file systems. Such an integration could be realized by being able to *mount* a remote cloud storage into the local file system, providing a seamless integration of mapping between local files stored in the cluster, and remote ones stored in the cloud.

This thesis introduces such an external storage solution in HopsFS [42], a scale-out metadata HDFS distribution developed at RISE SICS and Logical Clock AB.

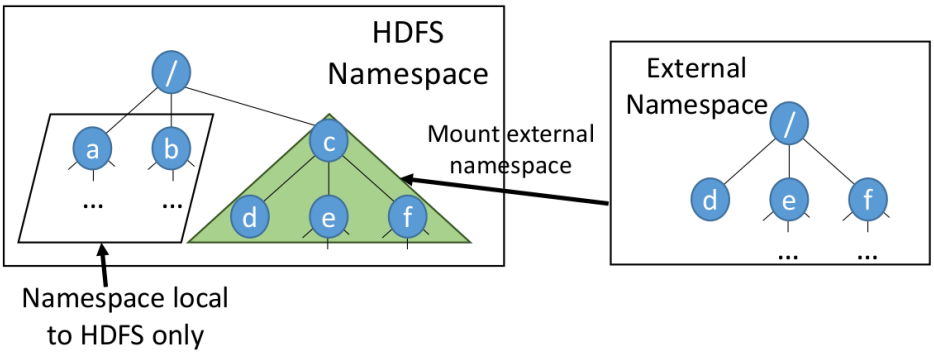

Figure 1.1: An example of a mounted external namespace in HDFS. The external namespace is mounted into HDFS, which is added under the same HDFS namespace as the local one. From HDFS-9806 [58].

1.1 Motivation and Scenarios

Currently, HopsFS, only supports storing data on their local environment, running on the start-up's cluster in the north of Sweden. For economical and practical purpose, it is desirable, both from a user and organizational perspective, to off-load and use the cloud for storage of data. For instance, an organization might already keep data in a remote storage which they wish to integrate into the file system, or they might want to backup archive data to a remote cloud storage. As different cloud storage and providers are becoming popular, it would be desirable to implement a provider agnostic solution that could be used with a range of different cloud providers, or even other types of remote storages, such as external HDFS clusters.

Specifically, from a client's perspective there are several scenarios where one benefit from using a mounted remote storage;

1. **Full cloud deployment.** The client transparently interacts with the file system which in turns communicated with the remote cloud store. From the client's perspective the data is stored in HopsFS. As opposed to current transient cluster implementation in HDFS, where a client can communicate directly to a remote storage (such as Azure HDInsight and Amazon EMR), a mounted remote storage could enable the file system to cache frequently accessed data locally, hold temporary data without persisting it to the backing store, and lazily write-back application data to improve performance [11].

2. **Hybrid Cloud Deployment.** A client might not be permitted to store all organizational data on the cloud, due to security or policy reasons. In such a scenario one can deploy a hybrid cloud scenario, where some data are stored locally in the file system and other in the cloud. Such clusters can still work in union, with workflows being coordinated and prioritized between different mounts.

3. **On prem-deployment.** Even if a client does not use the cloud, mounted storage deployment are still beneficiary. Organizations backing up their data between geographical regions for availability and fault-tolerance (such as Business Continuity Planning, BCP [11]) could use multiple mounts in one cluster to seamlessly integrate between regions.

Being able to locally cache and store the result from pulled data from a remote store will both increase the throughput, in terms of less round-trip-time of fetching data from the cloud, as well as reduce the cost of transferring data to and from a cloud provider. For instance, Amazon S3 (Simple Storage Service) charges a small volume-based fee each time a client moves data out or into their servers. Therefore, utilizing a local cluster as the computing medium for frequently accessed data is both desirable in terms of cost, scalability, and performance. However, as discussed in the next section, a seamless integration between a local and remote file system poses new questions; such as how to synchronize data between them and ensure availability.

1.2 Problem statement

The goal of this thesis is to develop a Proof-of-Concept (PoC) solution that enables Hops File System (HopsFS) to serve blocks from a remote external provided storage, which is to be mounted into a local HopsFS cluster. We may use the term ”provided”, ”remote”, or ”external”-storage interchangeable to describe a storage that is physically located outside the local cluster. This remote storage may be provided from a cloud vendor, another HopsFS cluster. The goal is to implement a provider-agnostic solution, which can be configured to connect to different cloud providers (e.g. Amazon S3, Microsoft's Azure, Google Cloud) through provider-specific connectors, provided by the vendor. We plan to implement the feature in HopsFS and test it against Amazon's object storage S3.

Specifically, HopsFS will treat blocks stored on the local disks of hosts managed by HopsFS the same as blocks stored in an external block store. The client could choose storage policy of the blocks, such as choosing offloading or backup of data to the cloud and storing more frequently used data in the local cluster. Currently, HopsFS has a periodic synchronization protocol to keep block data and metadata up to date. This introduces problems of ensuring the availability, consistency and integrity of block data stored in the external storage (such as S3). Such problems will have to be taken into consideration when developing the solution, we will investigate approaches taken into handling these problems. However, for the sake of scope, the PoC will focus on a *read-only* solution for serving blocks from a remote storage. Solutions for handling data consistency and synchronization will be investigated and discussed, but implementing such tools are scoped outside this project.

The research challenges of this thesis are to investigate and implement how a POSIX compliant filesystem (e.g. strong consistency, semantics) can be use with

an object store that does not enforce such consistency nor semantics, to enable applications and clients to operate the same regardless of underlying storage system. Is it viable, and how does such a solution perform?

In this thesis project we will research, implement, and evaluate such an integrated external storage solution. We state with following research questions;

1. **How can a distributed block based file system effectively be integrated with an external object based cloud storage?**

   The first research question (RQ1) asks *how* an integrated storage solution can be developed. It is critical to understanding the design of such an integration, as well as how it can be done in an efficient way. For instance, caching mechanisms might be a design choice to improve round-trip-time. We plan answer RQ1 through a practical approach of the design and implementation.

2. **What are the implications of such an implementation in terms of; data consistency, data availability, and performance?**

   The second research question (RQ2) examines *what* the consequences are of the provided solution. This will be done by benchmarking the solution for its performance and investigating aspects in terms of data consistency and availability. For instance, an S3 object storage have weaker consistency models (eventual consistency) than HopsFS, a hierarchical file system with POSIX semantics. This raises trade-off questions as of how to keep data consistent between the two, e.g. if data is assumed to be available at all time one will loose consistency. We plan to answer RQ2 through research and tests.

1.3 Contributions

The scientific contribution of this thesis is to provide research and practical-based knowledge of how an POSIX compliant filesystem effectively can be integrated with a remote storage that does not hold such semantics. This will answer questions of *how* this can be done and *what* consequences it imposes. For future development of remote storage solutions, this thesis work could serve as a foundation for understanding the requirements, implementation, and challenges of such an approach.

From a practical contribution, this thesis work will provide Hops File System with support of serving remote provided files. This will enable the local environment to be integrated with a cloud or other remote storage system, allowing clients to access remote storages through HopsFS. The thesis work will focus on investigating current solutions, adopting, implementing, and testing them in the HopsFS architecture. To test and benchmark the implementation against a live cloud vendor, cloud connectors, which are not currently supported in HopsFS, will have to be integrated into the distribution. Finally, a tool for scanning the external namespace (e.g. S3

bucket) to identifying remote files and directories, along with storing their metadata in HopsFS's Network Database, will have to be implemented.

Chapter 2

Background and Related Work

This chapter introduces a few relevant topics that the reader should be familiar with in order to follow along with the rest of the thesis. We examine the background and related work of distributed file system and cloud storages that enables the development of the integrated remote storage feature into HopsFS.

The chapter starts with a general introduction to what file systems are in Section 2.1 *Definitions of File Systems*, followed by a preceding look into earlier systems that has enabled the rise of HDFS in Section 2.2 *Preceding Distributed File Systems*. Section 2.3 *Hadoop Distributed File System (HDFS)*, details HDFS with important topics that has been forked off to HopsFS, which is discussed in section 2.4 *Hops*. Finally, Section 2.5 *Amazon Simple Storage Service (S3) and other Cloud Storages*, investigates remote cloud storages and how to deal with the different consistency semantics.

2.1 Definitions of File Systems

A file system is the method and structure of an operating system to organize files and data on a disk [25]. There are a multitude of different filesystems, such as NTFS, ext, fat, jfs, and many more. These have different semantics, features, and capabilities (which are far outside this discussion) but share the fact that they are all operating on the same partition (machine) and disk.

In a UNIX file system, an *inode* is a unique identifier for a file or directory. It is a data structure that contains information about the specific file or directory, such as its attributes and location. It includes *metadata* about the file, data describing the file such as access time, modification, owner or permission. The file, pointed out by the inode, is in term made up of *blocks*, a sequence of bytes which are determined by the *block size*; the smallest read and write unit in the file system. Larger block size increases performance for large files, since more time is spent on writing/reading and less spent on seeking to the next file, while it wastes the remaining unused block for small files (e.g. if a small file is 2 kilobyte (kb) with the block size of 32 kb, then one waste 30 kb). These trade-off depends on the intended application and use

of the file system.

In contrast to normal file systems, a *distributed file system* (DFS), sometimes called a network file system, is a file system that allows multiple nodes to share and access data over any location in a machine cluster. The nodes use network protocols to communicate with each other for file operations, which may have different access rights depending on the machine. The DFS nodes use the same semantics and interface to transparently operate on the shared files as local ones.

2.2 Preceding Distributed File Systems

This thesis works with HopsFS, a fork of an earlier HDFS distribution. However, there are a multiple of preceding systems that has inspired and enabled the development of the current file systems, with properties of interest to our remote storage feature. This section highlights some of these.

2.2.1 Andrew File System (AFS)

An early distributed computing environment developed at Carnegie Mellon University in 1983 was called Andrew [28]. Its distributed file system, Andrew File System (AFS), was a pioneer to modern distributed file system, presenting a location-transparent and homogeneous name space to the cluster machines. The goal was to build a system for high-availability, scalability, high-performance, and fault tolerant. One distribution at the universirty had over 25.000 nodes [39].

Built on top of a set of trusted servers, *Vice*, where each node ran the same UNIX operating system (Berkeley Software Distribution), the operating system intercepted and forwarded file system operations to user-level processes using the *Venus* process. Venus acted as the cache manager for Vice and stored a cache of remote files, reducing the amount of network operation to the Vice servers. AFS also had benefits over earlier file systems in terms of scalability and security, using Kerberos [41] (a network protocol for authentication) and permission based Access Control Lists (ACLs) [38] (lists of user or system permission for an object).

Many features from AFS can be found in today's distributed systems, where some might be interesting for the PoC in this thesis. For instance, the local caching idea from Venus will equally benefit a cloud integration. By preferring to read from local replicas instead of remotely hosted ones we will both minimize the network load as well as the economical fees of inter-cloud data transfer.

2.2.2 Coda

Building ontop of AFS, Coda [48] was developed at the same university but was focusing on development towards mobile distributed computing (mobile as in the terms of commodity machines, which was uncommon at the time). This moved the focus to support for disconnected operations, where a system can function properly without being connected to a network. This function is called ”hoarding”, and

enables Coda to cache all files that are accessed on the remote storage, specified in a text file. This raises a question how caching could be done in our feature, should be cache all the remote data as in Coda, or only cache the accessed data. The most probable answer would be to only cache the most accessed files. As ASF, Coda provides client-side caching, and mounts its data under the ”global name space” (/coda and /afs). All clients will be able to see the same files under the structure, similar to how a mount can be done in our integrated storage solution.

2.2.3 ZFS

A file storage is usually made up of logical volumes and physical drives. ZFS [47] is a unique system that acts as both of these, a file system combined with a logical volume manager. Therefore, it knows both the physical drives along with the complete status and health of the volumes. This enables ZFS to manage hundreds or thousands of physical drives as a single entity, providing almost endless (256 quadrillion zettabytes) scaling capabilities. As it was designed to run on a single server with an indefinitely number of pluggable storage drives, its goal was to make sure that data could not be lost due to malfunctioning. To achieve this, ZFS makes sure to uniquely verify and control each operation, taking necessary actions to correct them if needed. Further, it provides many features such as self-healing, snapshotting, cloning, and RAID capabilities [47].

For this thesis, ZFS is interesting since its capabilities of serving many heterogeneous storages are not unlike an integrated remote storage environment, where the remote storages may be of different types and characteristics. This is similar to the provider-agnostic approach that we wishes to achieve in our PoC, where the remote vendor might be another distributed file system, key-value store, or cloud vendor with different semantics.

2.2.4 Google File System (GFS)

Google File System (GFS) was developed to meet the increasing demand for growing data processing needs at Google [24]. It was driven by observing workloads in production at Google, which drove new design choices in comparison with traditional distributed filesystems.

First, failures are seen as the norm instead of the exception. GFS was designed to run on commodity hardware which drove the motivation of constant-monitoring, error detection, fault tolerance and automatic recovery as key aspects [24].

Further, as files of multiple gigabytes in size are standard at Google, block size and I/O operations were designed for large files. These files are mostly appended, rather than overwritten, with practically non-existent random writes. Once written, the reads are mostly sequential. Such observations lead to appending being the main optimization focus for performance and atomicity guarantees [24].

2.3 Hadoop Distributed File System (HDFS)

Hadoop's Distributed File System (HDFS) is the underlying file system for Hadoop's distributed computing softwares. The filesystem is very similar to GFS but open source. It was designed for storing very large files (terabytes) with streaming access to data stored on clusters of commodity hardware that may fail often [58]. Due to likelihood of failure, HDFS was designed for robustness with fault-tolerance in mind, with features such as configurable Replication Factors (the number of copies of each block), Rack Awareness (automatic rack placement on different physical racks), and High Availability (support for multiple backup Namenodes in case of failure). This section highlights the fundamental architecture of HDFS, displayed in Figure 2.1.

2.3.1 Blocks

The files in HDFS are divided into blocks that are replicated amongst the nodes in the cluster (default three per file). The blocks are typically 128 MB and stored as independent units. The reason for such large blocks are to minimize seek time and increase performance.

The advantage for abstracting files into blocks are several. Primarily, a file can be larger than any single disk in the network since the blocks are distributed among the machines. Block abstraction is also advantageous for storage management, since their size is pre-determined they simplify computing of storage size and handling of meta-data [58]. Further, blocks simplify replications as they may independently be shuffled around the cluster while their replication factor are being monitored. A block consist can be in any of the three replication states; under-replicated, replicated, or over-replicated. HDFS will automatically take action to replicate the under-replicated blocks or delete excess replicas, the block information are sent from the datanode to the namenode in a process called block reporting.

2.3.2 Namenodes and Datanodes

In HDFS there are two types of nodes; namenodes and datanodes. The datanodes store the actual data while the namenode manages the namespace and holds the metadata about the files and directories. When a clients interacts with HDFS it first asks the namenode for the file location, whom directs the client to the proper DataNode to read and write from. See Figure 2.1.

However, The namenode in HDFS suffer from scalability issues. For instance, since the metadata is stored in the heap of the namenode's JVM, the heap size is a limiting factor of the scalability of the namespace [52]. Another limitation of HDFS is single-writer, multiple-readers, due to global locking of the entire filesystem [42].

Further, there are logging mechanisms in HDFS where the namenode logs the changes of the metadata into journal servers to ensure high availability, using quorum-based replication. These are later read and reapplied asynchronously by the

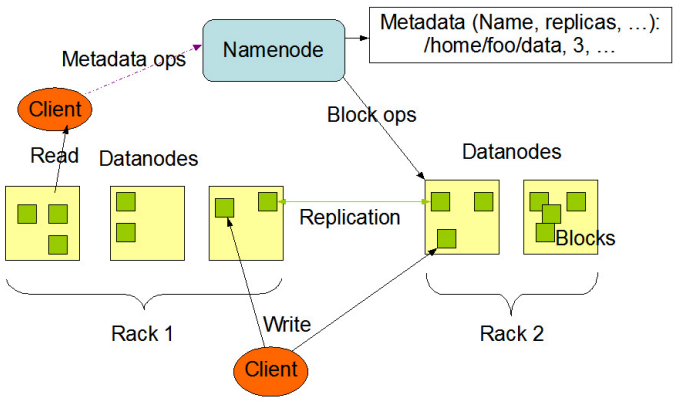

Figure 2.1: The HDFS architecture displaying clients writing and reading from a namenode and datanodes. From "Hadoop documentation. HDFS Architecture Guide" [45].

standby namenode. The configuration service Zookeeper is used to coordinate the failovers of the active to standby namenode, as well as providing agreements on the choice of the active namenode.

2.3.3 Heterogeneous storage types

The datanodes persist their blocks on underlying disk volumes, which are configured as mount points [1] in HDSF. Heterogeneous Storage in the idea of being able to use different kinds of storage media depending on their purpose in HDFS. The feature was introduced in JIRA-2832, *Enable support for heterogeneous storages in HDFS* [54]. For instance, machines with larger storage capacity but lower computing power might be used as archive storage, while faster machines with less storage can be primarily used for computing.

Heterogeneous storage added the new storage types ARCHIVE and RAM_DISK to the exciting types DISK and SSD. As the name suggest, the ARCHIVE type is used for offloading data and has large storage capacity but little computing power. The RAM_DISK was added for supporting writing single replica files in memory [54].

Previous Master thesis work has also been done at Logical Clock to implement the Heterogeneous Storage in HopsFS [35]. This work will act as a pillar for the addition of the PROVIDED storage type, which will point towards an external storage and be implemented in this thesis project.

2.3.4 Storage policies

To determine between which storage types to use to store the block replicas, heterogeneous storage policies [1] was introduced with HDFS 2.1, and improved upon in the heterogeneous storage release. They allow for specifying which preferable

underlying storage type the replica should be stored upon, as well as a fallback type for replica creation that does not succeed, which will be used if the preferred storage type is out of space. These policies can be enforced at the start-up or at any time during the lifespan of a file, and applies both to directories and all the files in them [1]. See table 2.1 of a list of the supported storage policies (before implementation of PROVIDED storage).

Policy name	Block placement	Fallback creation	Fallback replication
All SSD	SSD: n	DISK	DISK
One SSD	SSD: 1, DISK: *n* - 1	SSD, DISK	SSD,DISK
Hot	DISK: *n*	-	ARCHIVE
Warm	DISK: 1, ARCHIVE: *n* - 1	ARCHIVE, DISK	ARCHIVE, DISK
RAID5	RAID5: *n*	DISK	DISK
Cold	ARCHIVE: *n*	-	-

Table 2.1: Storage policies in HDFS.

2.3.5 View File System (ViewFS)

The View File System (ViewFS) is a relatively new ”file system” that was added to HDFS to provides it with the capabilities of supporting multiple namenodes and namespaces. ViewFS is a trivial filesystem in the sense that it only support linking of other filesystems and does not have any internal functions [22]. In original HDFS only one namenode is supported, which is a bottleneck for scalability and a single point of failure. With ViewFS, HDFS federation was introduced which enables multiple namenodes to operate independently without coordination of each other (see Figure 2.2). ViewFS may be used to create personalized namespace views and is analogous to mount tables in Unix systems [22].

In a HDFS cluster with enabled ViewFS, each namenode has an individual block pool, referencing the blocks it addresses. Mount points are specified in the standard Hadoop configuration files, with the prefix of `viewfs://clusterX`. For instance, this will make ViewFS look in the mount table for the cluster named ”clusterX”, see Figure 2.2.

ViewFS benefits of increased scalability and isolation and function as a generic storage service, which can be used for new file system to be added ontop of it. For instance, in our case of a mounted remote storage solution, we could have one remote storage mounted in one namespace which is controlled by a dedicated namenode.

2.3.6 Cloud connectors in HDFS

Currently, there are a few cloud connectors implemented in the Hadoop codebase that makes it possible for a client to read and write to an external cloud storage. These work by using their respective Java SDK to communicate to the remote

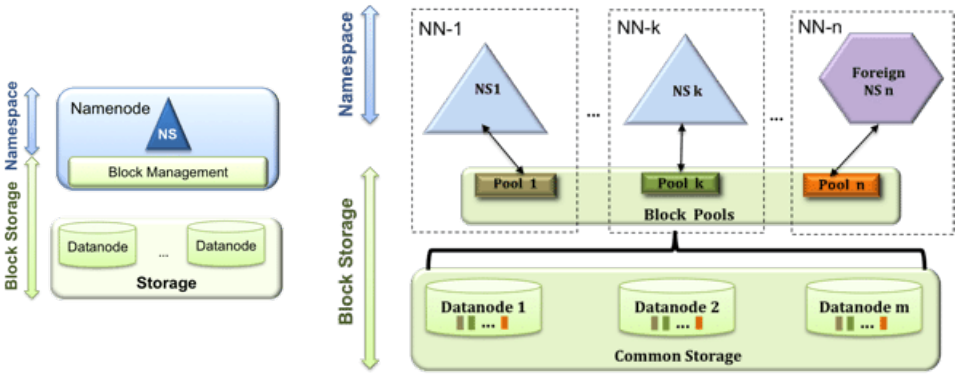

Figure 2.2: An HDFS cluster before (left) and after (right) ViewFS is enabled. The ViewFS enabled cluster displays multiple namenodes in one HDFS cluster, instead of the traditional single-namenode architecture. From "Hortonworks. An Introduction to HDFS Federation" [53].

cloud service. At the time of writing (august 2018), the supported connectors are; Amazon S3 (`s3a://`), Azure's WASB (`wasb://`), Azure ADLS (`adl://`), and Google Cloud Storage (`gcs://`) [27]. All of them have their separate module in the Hadoop codebase, for instance, the S3 connector is found in the `hadoop-aws` module.

The connectors work in a similar way where the client specifies a URL of the resource to access. In the case of S3, the resource is a bucket (a storage name space abstraction) with a following file name, hosted on Amazon Web Services (AWS). For instance, a client specifying the read path to `s3a://test-bucket/testFile.txt` will read the file `testFile.txt` from the bucket `test-bucket`. From the client's perspective, this is a transparent operation where she only specifies the URL and retrieves the files in the HDFS file system.

Behind the scenes, when a client performs a REST call to an Amazon service using S3A, the `s3a` prefix detonates that an instance of the S3(A) file system should be created, `S3AFileSystem`, an implementation of the abstract `FileSystem` class, see Figure 2.3, which is used to operate on the specified bucket address. Following, when the client specifies an operation on the resource, the S3 file system send an HTTP requests to the Amazon S3 REST API, returning the response to the S3 client. To do so, the user must have configured her AWS credentials so that the SDK can create a signature to sign the REST call [43]. For instance, a PUT request will add an object to the bucket which can be accessed through a GET operation or in the Web Management Console [51].

Since our Proof-of-Concept implementation is targeting testing with Amazon S3, we will discuss their three available plugins; S3, S3N, and S3A [55].

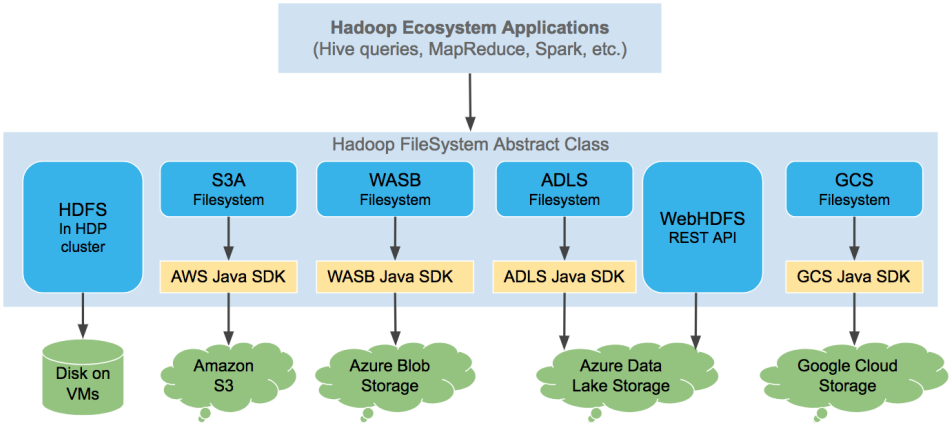

Figure 2.3: The cloud connectors in HDFS with their connection to their respective external cloud storage through Java SDKs. Each implement the abstract Hadoop FileSystem class. These include S3A, WASB, ADLS, and GCS. From "Hortonwork. Accessing Cloud Data" [27].

Amazon S3 connectors

S3 (`s3://`) was the first S3 plugin introduced in the Hadoop environment. It implemented a block-based filesystem that was backed by S3, and was introduced in 2006 when S3 had a limit on the size of their blobs [55]. Files were stored as blocks and it could support larger files than S3 could at the time. The s3 file system enabled Hadoop to be ran in Amazon's EMR infrastructure, using s3 as the persistent store [37]. However, its drawback was that it was incompatible with any other application's use of data in S3 (such as MapReduce) and also required that a new empty bucket was dedicated to the file system [55]. It has since then been deprecated and was removed in Hadoop 3.

S3N (`s3n://`) was the successor to S3 and was the first schema able to read and write native S3 objects ("n" for "native"). S3N introduced new features such as compatibility with standard S3 clients and support for partitioned uploads of multi-gigabyte files. It solved the compatibility problem of other application and made operating on Hadoop in EC2 easier [37]. However, it suffered to known issues, such as `seek()` being slow on large files due to reads always going to the end of a file [17]. These relied on third party dependencies that introduced new bugs in the code. It is no longer shipped with Hadoop 3.

S3A (`s3a://`) is the latest plugin and was introduced in Hadoop 2.7 as a replacement for the previous two. It has since then been the focus of all work and has seen improvements such as; support for IAM and environment variable authentication, encryption of AWS secrets in Hadoop credential files, metric collection for performance diagnostics and better error reporting and scalability [37].

The S3A client are under active deployment with security, scalability and performance being improved upon (for instance, see HADOOP-14831 [16] and HADOOP-

13204 [15]).

Mapping directories in object store

An issue that arises when mapping a file hierarchy to an object store is that the object store has a different semantic than file stores. For instance, Amazon S3 is not a file storage, but an object storage. This means that there is no knowledge of directories, which makes a direct mapping of directories structures difficult. To solve this, S3 connectors mimic the directories when a client creates (`mkdirs`), list, delete, or renames an directory [17]. The consequences are several; directories might lack modification time, meaning some applications that rely on this might not function as indented. Further, operations such as listings and renaming on directories might be slow and are non atomic, which can lead to problems in case of half-way failures.

Consistency issue

The arguable biggest drawback and problem of S3 is the consistency model. HDFS is strongly-consistent, meaning data is updated atomically, while S3 is eventual consistent (for performance and availability). This creates issues when updating and deleting files in the object storage, a change might not be directly reflected back to the client and can cause problems. New files might not be directly visible, delete and update might not propagate directly, and delete and rename operations takes proportional time to the number of files (due to recursive implementation) [17].

Authentication model

The authentication model of S3, and cloud storages in general, vastly differs from the one of HDFS. User authenticate to S3 buckets using AWS credentials, and the S3A plugin does not enforce any authentication.

Attempts have been done to try to solve these issues. S3Guard [40], developed by Apache, and S3mper [57], developed by Netflix, are features that try to deal with the eventual consistency issue by storing the object's metadata of the S3 bucket in a strongly-consistent, highly available database (DynamoDB). This also improve performance, since the metadata is cached in the Dynamo database [36].

However, such solutions increase complexity by relying on another system in order to handle the consistent metadata, requiring operations to go through the new storage before accessing the data. It also poses a consistency problem between the database and object store, regular synchronization are needed which brings extra network operations and cost.

Stocator, *An Object Store Aware Connector for Apache Spark* [56] is an attempt at solving these issues by taking advantage of object store semantics to achieve high performance and fault tolerance [56] for Spark applications. It was announced in 2017 and according to Vernik et al., it achieves better results than the Hadoop connector;"18 times faster for write intensive workloads and performs as much

as 30 times fewer operations on the object store" [56]. In essence, this is due to Stocator taking advantage of object store instead of file system semantics, thus it does not have to create and manipulate temporary directories and files. It might be an interesting option for connecting via Spark application, but is still in early stages.

In this thesis we will use their work as an reference and build it into the HopsFS.

2.4 Hops

To overcome some of the limitations of HDFS, HopsFS has been developed. It is an open-source fork of HDFS that stores the metadata in a separate Network Database (see section 2.4.2), instead of in the heap in the namenode. This solves the scalability issues of HDFS as well as providing faster queries performance. See Figure 2.4 for an architectural overview of HopsFS and HDFS.

HopsFS also differ from HDFS in its fundamental locking mechanisms. Instead of locking the entire directory tree (global lock), HopsFS uses more granular locking on each subtree of the directory structure. This enables multiple-writer, instead of only one writer at a time. Other changes include a Data Access Layer (DAL) that every namenode uses to communicate with the database, hopsFS clients providing load balancing between the nodes using *normal*, *round-robin* and *sticky* policies [42], and the database being used for leader election of the namenode.

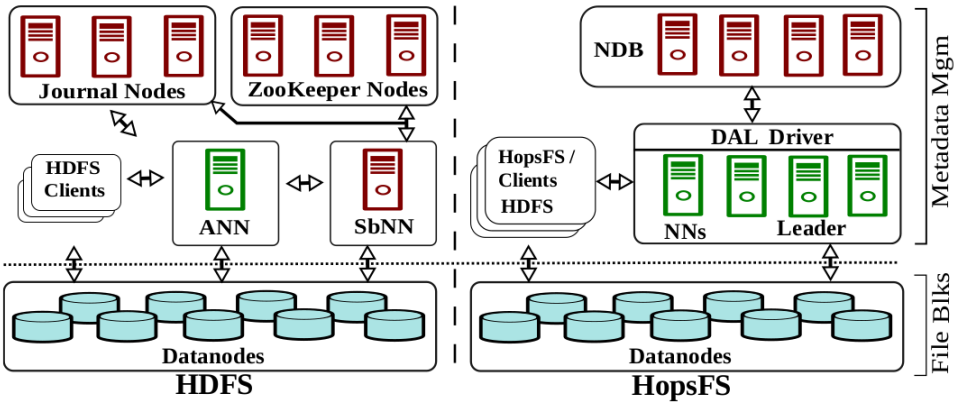

Figure 2.4: The HDFS and HopsFS architecture side-by-side. HopsFS support multiple namenodes (NNs) with the metadata stored in an external Network Database (NDB), instead of the active namenode (ANN) with standby namenodes (SbNN), and Journal/ZooKeeper Nodes for handling coordination. From "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases" [42].

2.4.1 HopsWorks

The data platform build on top of HopsFS is called HopsWorks. It is a web-based user interface used for collaborative data science where users can upload and execute

various big data jobs. HopsWorks supports running many services; such as Spark [21], Flink [18], Kafka [20], and support for notebooks such as Jupyter [34] and Zeppelin [19].

When a user uploads and store jobs in HopsWorks it sends the data down to HopsFS, it is executed on the cluster (local or remote), and the result displayed in the user interface (see Figure 2.5). HopsWorks further abstract Hadoop by providing two abstractions; *project* and *datasets*. Users manage membership of projects and scope access to the datasets. A user can have the role of ”Data Owner” (the administrator) or ”Data Scientist”. These membership and security should be taken into account in a provided store, where a remote storage medium (e.g. S3) might have other type of permissions and security models. Ideally, a user should be able to choose on what storage medium she wants to store the projects in the UI, such as on local DISK, SSD, RAM, or at a PROVIDED storage such as in S3. The datasets stored on the remote cloud storage that are mounted into HopsFS should therefore be displayed in the UI.

2.4.2 Network Database (NDB)

Network database (NDB) is the storage engine of MySQL Cluster, a shared-nothing, in-memory, highly-available, distributed, relational NewSQL database. MySQL Cluster is used as the storage medium of metadata for HopsFS and consists of a management nodes that manages the cluster, along with several NDB datanodes that stores the table's data and handles transactions. These nodes are grouped where the size of the group is determined by the cluster's replication factor. Each group is assigned a partition of the tables' data, and each node in the group holds a complete copy of the shards given to the data group. All NDB datanodes also have a transaction coordinator which enables queries to run in parallel with cross-partitioned transaction [42]. It is extensible in the way that new nodes can be added and removed at any time to a running cluster with rebalancing being done transparently by the cluster.

Moreover, HopsFS abstract the tracking of the *inodes* (files/direcories in HDFS) from the namenode into the NewSQL cluster. The inodes are partitioned into blocks and each one of them is replicated on a predefined number of machines. Their partitioned status (under-replicated, normal, or over-replicated) is translated into tables in the NewSQL cluster. For example, an inode is stored as a row in a *inodes* table and their associated blocks as rows in a *blocks* table. These are then used for HopsFS fine grained row locking mechanism. Each transaction is translated into a database operation, such as reading or writing from a block.

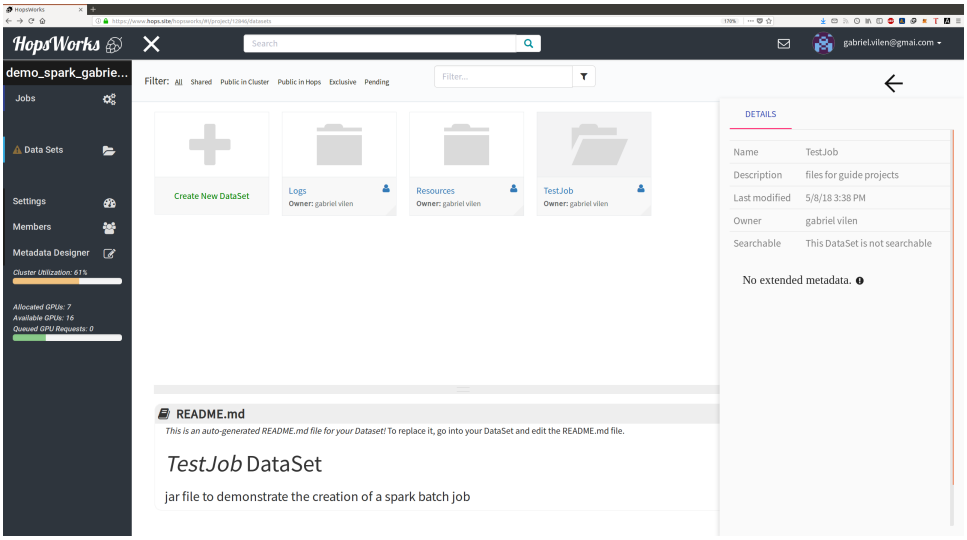

Figure 2.5: HopsWorks GUI for datasets stored in HopsFS. The web page displays the data sets which are shared between project members, such as data scientists and data owners.

2.5 Amazon Simple Storage Service (S3) and other Cloud Storages

S3, short for Amazon Simple Storage Service, is a highly-scalable, reliable, and durable cloud object store on Amazon Web Services (AWS). It guarantees ninety-nine with eleven nines (99.999999999%) of durability [3] and is designed to store any amount and form of data in *buckets*, the storage abstraction of S3. Data in the buckets are replicated across a minimum of three geographically separate physical facilities within an AWS region, and can further be replicated across multiple regions. Security, data-encryption, and access control are bundled with S3 by Amazon and supports multiple international security standards and compliance certifications. Further, AWS provides APIs to simplify transfer data between, out, and into the S3 clusters.

2.5.1 AWS Storage Gateway

Amazon Web Services has developed an option to integrate existing on-premises IT environment into S3. This feature is similar to the one proposed in this thesis, where S3 buckets can be mounted into an existing on-premises system. The gateway provides features for storing either files (File Gateway), blocks (Volume Gateway), or tapes (Tape Gateway).

The volume gateway is the most related ones to this thesis, which provides cloud-storages volumes that can be backed as Internet Small Computer System Interface (iSCSI) devices from the on-premises servers. The volume gateway can

be configured for either *cached* volumes, where all data is stored in S3 but a copy of frequent accessed data is retained locally, or *stored* volumes, where all data is stored locally and the Volume Gateway configured to backup snapshots to S3. The two configuration depends on use-cases, for instance, *cached* volumes offer more cost savings of storages, eliminating the need for scaling the on-premises servers, but provides higher latency access to the cloud stored data. On the other hand, *stored* volumes offers low-latency access to the entire dataset, and uses S3 as an offshore backup center.

Nevertheless, it is noteworthy that the Storage Gateway requires a Virtual machine to be installed on the on-premises clusters, where the file needs to be transferred to, for upload. It also includes Amazon's pricing model and the user has to pay for each transferred GB up and from S3 through the Gateway. In this thesis, we are looking for a more general open source alternative that is mounted directly *into* HopsFS, which does not require e.g. virtual machines.

2.5.2 Layering stronger consistency model on top of S3

From the S3 documentation,"Amazon S3 provides read-after-write consistency for PUTS of new objects in your S3 bucket in all regions with one caveat. The caveat is that if you make a HEAD or GET request to the key name (to find if the object exists) before creating the object, Amazon S3 provides eventual consistency for read-after-write. Amazon S3 offers eventual consistency for overwrite PUTS and DELETES in all regions." [12]. This means that data can be accessed at any time, from any node, and is used to achieve high availability in S3. However, it implies that an update to an object will *eventually* be available, it first has to propagate across the clusters. This could lead to problems of data-loss or stale data, especially in read or write heavy tasks such as data and analytic workloads [59]. A few libraries has been developed to deal with S3's eventual consistency "issue".

Netflix's S3mper

S3mper [57] is an open source library developed by Netflix that tries to deal with S3's eventual consistency issues in Netflix's HDFS distribution. It does so by storing the metadata of the object in an intermediate, read-after-write consistent, high performance, DynamoDB database. The idea of S3mper is to identify when a *list* operation to S3 returns inconsistent data and provide options to the job owner [57]. A secondary index is stored in the noSQL database, which keeps the up-to-date metadata to detects abnormalities. The "source-of-truth" is therefor still S3, with an additional layer of checking built on top (See Figure 2.6).

However, as S3mper requires an extra database for keeping data consistent, one needs to maintain another system. As the authors point out, the tool is not intended to solve every consistency problem, deletion of data outside of the Hadoop stack will still occur in divergent of the secondary index. Further, directory support is limited, since recursive listing is prone to consistency issues [57]. Obviously, the

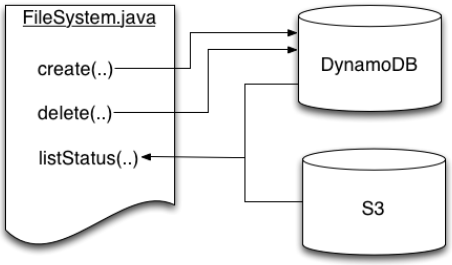

Figure 2.6: S3mper uses DynamoDB as an consistent metadata store between the client and S3. Inconsistent operation are proxied through DynamoDB instead of directly accessing the storage. From ”Netflix S3mper” [57].

client also has to pay for using the Dynamo database along with S3.

2.5.3 Amazon Elastic MapReduce File System (EMRFS)

Amazon Elastic MapReduce (EMR) [50] is a managed Hadoop framework running on dynamically scalable virtual machines, Elastic Compute Cloud (EC2) [49] instances, on AWS. It allows interaction with Amazon's data storages, such as on S3 and DynamoDB, with support for frameworks such as Spark [21], HBase [23], Presto [14], and Flink [18].

The EMR jobs run ontop of EMR File System (EMRFS) [2], Amazon's implementation of HDFS which guarantees read-after-write consistency with their storages. It is similar to S3mper in the sense that when a client creates a EMR cluster with consistent view enabled, a DynamoDB database is used to keep strong consistency. The read and write capacity of the database can be configured (default 500 read, 100 writes) depending on the number of object that should be ”tracked” and stored in the database. EMRFS only ensures that the objects in the tracked folders being are kept consistent [2]. Capacity and other configurations can be specified using the provided command line interface.

EMR is a popular choice for running MapReduce jobs on top of S3. However, it is a closed source implementation and does not support jobs running outside of AWS, such as in a local HDFS distribution. For the Proof-of-Concept implementation in this thesis, as we desires to pull data from S3 into the local HopsFS cluster, we primarily focus on storage of data and not MapReduce jobs. Running heavy I/O jobs is not desirable if the data is remote on S3, since this would require considerably network bandwidth and round-trip-time. Instead, running heavy I/O jobs should be done on locally cached data. If one desires to run such jobs, the remote data should be pre-fetched into the local HopsFS.

2.5.4 Google Spanner

Google has also tried several approaches to achieve consistent listing across their cloud service; global caching, porting S3mper, and storing persistent listings alongside the data. However, these were all suboptimal and required extensive porting work or relied on a single point of failure [59].

With Spanner [59], Googles globally distributed and strongly consistent relational database, the eventual consistency issues were solved. It is specialized for horizontal scaling whilst providing high availability and strong consistency. To provide the consistency guarantees it uses TrueTime [4], a highly available, distributed clock which runs on all Google's servers. TrueTime enables applications to generate monotonically increasing timestamps, a timestamp *T* that is guaranteed to be higher than previous *T'* iff *T'* was generated. This guarantee holds throughout all servers and timestamps. It is used by Spanner to assign timestamps to all transactions, enabling its strong consistency guarantees. The ordering is given to all clients across all regions due to Spanner keeping multiple immutable versions of data (multi-version concurrency control), enabling consistent reads and non-blocking writes [4].

The consistency guarantee provided by Spanner is called *external* consistency, where the system behaves as all transactions are executed sequentially even though they run in parallel. Specifically, if one transaction is finished before another one starts to commit, the system guarantees that no clients can see the effects of the second transaction but not the first. According to Google, even though the strong consistency guarantees and behaviour of a single-machine database, Spanner still provides performance in the likes of databases with weaker consistency models such as eventual consistency [4].

2.5.5 Azure Data Lake Store (ADLS)

To meet the broad spectrum of Big Data analysis at Microsoft, Azure Data Lake Store (ADLS) [46] has been developed. It is a successor of Cosmos and a fully-managed, elastic, scalable and secure filesystem which supports both Cosmos and HDFS semantics. It brings together components from both Cosmos and HDFS to create a unified platform for internal and external workloads, acting as a new warehouse for data and analytic at Microsoft.

ADLS was designed with a microservice arcitecture, where services for transaction, data, block management etc. are decoupled from the physical clusters of the stored data. This architecture differs from the transitional HDFS clusters where each service is running on the cluster machines (such as on the namenode).

ADLS is similar to a heterogeneous and provided storage concept in the way that it seeks to provide tier and locality transparency. It acts as a HDFS compatible filesystem layer between Hadoop workloads operating in the cloud (on Azure public compute), and on-premise HDFS clusters, through Azure's cloud connectors. The work has been largely adopted and inspired by the Hadoop open-source community,

such as by the tiered storage, discussed in 2.3.3. The motivation of implementing tiered storage were three-fold [46];

- The overhead of moving data from the cloud storage into the local compute tier negatively effects the analytic job performance.
- Fine-grained file managers are required to be ran to determine where the jobs are stored to collocate computing.
- Users must explicitly manage data placement, consider security and access control as data moves between services and software boundaries.

This was simplified by ADLS by letting the system manage data locality between clusters based on higher-level policies specified by the user. Policies to balance storage cost and set the trade off between local and remote stored data are handled by the user, whilst security and compliance is handled by ADLS. If relevant data is only available on the remote storage it is automatically fetched on-demand into the required cluster machine.

Further, ADLS supports the concept of *partial* files, a sub-sequence of a file that can be stored on different storage medium. A file in ADLS is internally represented by an unordered collection of partial files, each uniquely identified by an ID. This concept is not exposed to users and enables splitting of files onto different storage tiers, each with their own storage provider. ADLS is interesting since it interoperates the provided storage concept which we wish to implement in HopsFS.

Chapter 3

Method of Provided Storage Implementation

In this chapter we explain the method of the remote provided storage implementation to answer RQ1. We are basing the work on HDFS-9806 [11], adopting and extending it to work with HopsFS, while highlighting the design choices and challenges of the implementation.

The chapter starts by explaining the general design of the solution in Section 3.1 *Design*, followed by the architecture in Section 3.2 *Architecture*, and finally the details of the implementation in Section 3.3 *Implementation details*.

3.1 Design

In the Proof-of-Concept implementation, the client will only enable retrieving and reading from the external store, thus reducing handling of race-conditions and inconsistent data. Writing to the external store needs efficient handling of different consistency levels, which research proved requires external tooling (such as S3mper or S3Guard, chapter 2.5.2). Handling this will be out of scope of the PoC. However, integrating and layering such a consistency framework should not be difficult since they are designed to plug into Hadoop distributions that are using S3 connectors. It may also be possible to use HopsFS's Network Database (chapter 2.4.2) as the single source of truth for keeping consistent metadata, although, this would require frequent synchronization between the external cluster and HopsFS - which may not be desirable in a cloud scenario due to multitude of network requests. For the moment, we will focus on how to design a system for access, read, and store data from an external storage, to answer RQ1.

3.1.1 Provided storage type

In HopsFS, storage types specify the type of storage where a block resides, currently these are `DISK, ARCHIVE, SSD, RAM_DISK`. In the PoC, we add a new stor-

age type called PROVIDED amongst the existing ones which specifies that a block is stored on a remote *provided* storage.

A datanode in the cluster can be configured to host a provided storage (we call such a datanode a *provided* datanode), which enables it to access the PROVIDED blocks through a *provided store client*. The client is provided specific and depends on the remote storage to be used, such as S3A in the case of an S3 storage (see Section 2.3.6). The PROVIDED storage type makes no assumption of the underlying implementation besides that the block is coming from an external namespace and should be treated as such.

The namenode assumes all PROVIDED blocks from a datanode as valid, based on the assumption that the remote store has high durability and availability (e.g. 99.99% and 99.999999999% for S3 [26]). This eliminates the need for the provided datanode to include the PROVIDED blocks in the *block report*, the periodic synchronization of blocks and metadata between datanode and namenode, allowing scaling to large external storage by paging in data as needed with minimal overhead [11]. If we instead choose to include all the remote blocks in the reporting, it would case significant overhead when using a large remote storage.

3.1.2 Alias map design

Currently, HopsFS makes no assumptions of the physical locations of where current block are stored. The identifier pointing out that a block resides on a remote namespace, is the PROVIDED storage type of the block. However, to be able to identify the location of such a remote block, e.g. that resides at a different URL than the local file system, a mapping between the local block and the remote object has to be implemented. This map is called an *alias map* [11] since it maps block aliases to block locations, and modelled as a key-value store. The alias map maintains the mapping of local PROVIDED blocks to remote object in the external store and is used to lookup block location before consulting the remote storage client (see Figure 3.1).

Specifically, an object in a remote store is modelled and corresponds to a *file region* in the local storage. The *file region* is identified by an *URL* that points towards the data, an *offset*, a *length* and a *nonce*. This information is stored as a tuple in the alias map (see Table 3.1). The *nonce* can be thought of as an unique identifier of the metadata that identifies the version of the file, similar to a generation stamp in HDFS. The nonce is based on the assumption that the external store exposes modification information by timestamps, unique IDs per file/directory, or other means, i.e. if the file version is changed the *nonce* will differ. The *nonce* can be used to detect block data inconsistencies, for instance when serving read requests.

Initially, the alias map can be stored in-memory as an service running in the file system and shared between all datanodes in the cluster. A csv text file can be used to hold the mapping between the blocks and file regions (see Table 3.1). However, storing the map in a text file will not scale to multiple provided datanodes. Further, adopting the design to HopsFS, the alias map may be implemented and

ID	URL	offset	length	nonce
10725	`s3a://bucket-1/file_0.txt`	0	1024	1001
10726	`s3a://bucket-1/file_1.txt`	0	512	1001
10727	`s3a://bucket-2/script.js`	0	2048	1001

Table 3.1: An alias map example containing the ID, URL, offset, length, and nonce to uniquely identify a block in a remote storage.

stored in the external Network Database (NDB) instead of in-memory, reducing the memory consumption of the Java Virtual Machine (JVM). Such an implementation makes sense for production environments, where a client might store hundreds or thousands of files in the remote storage, while in a small PoC environment the extra memory consumption of storing the alias map in the NDB will be indistinguishable.

3.1.3 Design benefits

The above design choices have several benefits;

Primarily, the new `PROVIDED` storage type is **storage agnostic**. Instead of a provider specific solution, this general type enables pluggable storages, such as different kind of cloud providers or key-value stores to be used under the same storage type. The namenode is only aware that the block resides in a provided storage, and the only changes of serving a block from another storage is changing the URL prefix (s3a://, wasb:// etc.) to use that client and cloud connector. The provider specific implementations are then handled by the provider client, such as S3A. Overall, this enables code re-usability and decoupling as it builds on the heterogeneous storage type.

The implementation **extends features from HDFS**. Since the provided storage is participating as a storage tier added to the heterogeneous storage, it can uses familiar features and policies, such as storage-level quotas which limit on the number of files and space in a directory and storage placements policies (Section 2.3.4).

The design is **transparent** to users and applications. These do not need to know if the data is local or remote since this is abstracted by the unified namespace where data locality is handled by HopsFS. This enables applications to operate as usual without having to care for the underlying storage type.

Finally, enabling **caching and prefetching** of local data from the remote store could be added to the solution to improve performance due to minimizing remote storage round-trip-time. Frequently accessing a remote versus local block might not be ideal for performance reasons, and the performance impact will be benchmarked in the following evaluation chapter. For now, prefetching is not part of the PoC.

3.1.4 Design challenges

The design of a final solution poses multiple challenges and questions, such as;

How to synchronize metadata without copying data? We want to dynamically page in the blocks on demand, without having to copy data from the remote store. For instance, if the remote storage contains 100.000 files, we do not want to move in all the redundant data due to storage capacity overload. This requires policies for prefetching and evicting local replicas (i.e. caching eviction).

How to mirror changes in remote namespace? Since data in the remote namespace can be changed and modified by other users, this should be identified by the client and reflected accordingly. The cluster should be able to handle out-of-band churns in the remote storage in a defined way.

How to handle writes consistently? New writes has to maintain file-to-object mapping and make sense to a provided storage. A direct writing of blocks to the object store will not work because of the different underlying storage models.

How to enable dynamic mounting? Mounting of the remote storage should be done in an efficient and clean behaviour. Common accepted standards and concepts for mounting should be kept, such as directories and hierarchies of installed files.

As discovered, the design challenges are many of an provided storage solution, motivating our design to focus on the core functionality of being able to mount and read from a generic external namespace into the local cluster. The mounting will be done with a recursive creation tool (fs2img), discussed further in the implementation, and we will focus on supporting translating, store, and read the files from the provided storage into HopsFS.

3.2 Architecture

This section handles the architecture of the provided storage Proof-of-Concept implementation. We try to keep the architecture close to HDFS-9806 [10] to enable easier code maintenance (e.g. simplifying integration of future upgrades and reflections in the Hadoop code base). However, since HopsFS is an older fork of a HDFS release (around version 2.3, whilst currently being upgraded), the code between the two has heavily deviated, and newer features that exists in HDFS does not exist in HopsFS, and vise verse. The fundamental architecture between the two is also different, since HopsFS relies on storage of metadata in an external database instead of in the JVM of the namenode (see Chapter 2.4 for more details). Therefore, a lot of work has gone into reviewing and adopting a multitude of filed issues (JIRAs) to fit their needs in the HopsFS architecture to be able to support provided storages. We highlight two crucial JIRAs and changes that have been implemented and adopted into HopsFS;

1. *HDFS-9809 - Abstract implementation-specific details from the datanode* [31].

   This JIRA aims at abstracting away file specific details from the datanode to enable storage medium that do not follow HDFS semantics. For instance, a file stored in S3 might not have its data represented as Java Files (as default

in HDFS) but as an URL. We model this as a URL pointing to a bucket, with an offset pointing to a specific block (see design choices above). Further, volumes might not be divided into directories as assumed by a local HDFS cluster. This JIRA abstracts such details to be provider handled based on filesystem semantics.

2. *HDFS-7878 - API - expose a unique file identifier* [13].

   Files in remote provided storage that do not enforce HDFS semantics will have to be uniquely identified. This JIRA exposes a unique inode ID as an API, which is used as the *nonce* for correctness in the provided storage. The nonce can be thought of as a generation stamp of the block. This makes sure the correct version is returned and have not been modified, preventing reading of stale data. The nonce will play an important role when synchronizing between remote and local storages, such as when reflecting changes from the local data in the remote storage. For the moment, we ignore the nonce since we do not reflect changes in the remote store.

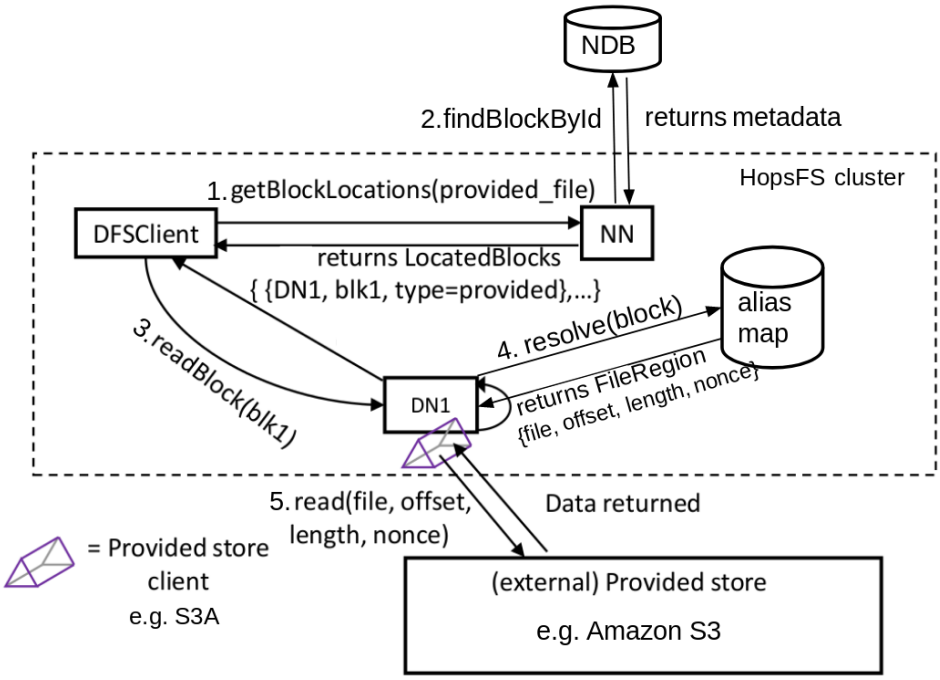

Figure 3.1: An architectural overview of a HopsFS cluster with enabled external provided storage. A read scenario from a client to a remote block is displayed in the image. Modified from ”HDFS-9806” [11]

3.2.1 Read scenario

The flow of reading data from the external provided storage in a HopsFS cluster, displayed in figure 3.1, is discussed below;

1. The file system client, `DFSClient`, calls `getBlockLocations(provided_file)` to the namenode. To identify blocks available from the provided storage the namenode (NN).

2. The namenode in turn calls the Network Database (NDB) with `findBlockById` to retrieve the metadata about the datanode reporting the block(s) for the provided file. The namenode returns the metadata of the located blocks, containing the datanode(s) (DN1) where the block(s) (blk1) are addressable along with its storage type (`PROVIDED`).

3. The client consults the datanode (DN1) to read the block (blk1) in `readBlock(blk1)`. The datanode will now resolve the physical location and file region of the provided block by consulting the shared alias map.

4. The datanode addressing the provided block consults the alias map, which could be running as a service in the memory of the namenode or be stored in a csv file, to retrieve the FileRegion which corresponds to the provided block in the remote storage. The file region, containing the tuple to identify the remote object in provided storage, is returned.

5. The datanode reads the external provided storage with his *provided store client*, using the metadata returned from the *alias map* as parameters in `read(uri, offset, length, nonce)`. This step is provider specific as it depends on the provided store client, e.g. S3A, to call its respective REST API. The data is returned back through the provider client. At this final step, data could potentially be cached in the local disk storage of the datanode.

3.3 Implementation details

To enable the provided storage implementation new classes has been integrated and a few architectural changes has been made to fit the HopsFS architecture. Figure 3.2 displays a diagram over most of the new classes that has been added, or modified, to support storage and handling of provided blocks. We will in this section go into the various details of the most important changes. Many classes build upon previous abstractions and thus re-use the same functionality.

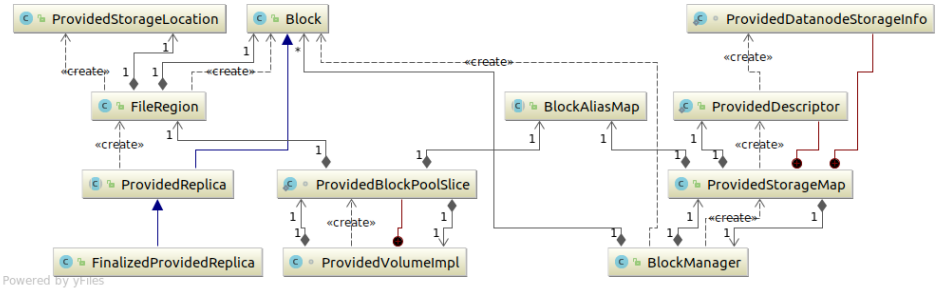

Figure 3.2: An overview UML diagram of the provided storage architecture. It displays the relationship of new and changed classes that has been adapted to support the provided storage feature.

Provided storage map

The class that stores the map of local and provided storages is `ProvidedStorageMap`, which allows management and multiplexing between datanode's local to provided storages. The namenode keeps the only reference to the map (in the `BlockManager`), and uses it when a node is to be added, retrieved, or removed from any storage. The class contains logic for checking if a storage is of the `PROVIDED` type, and if so, will receive the provided storage information from the provided datanode descriptor, `ProvidedDescriptor`, a class that keeps track of datanodes with provided storages.

For example, if a client wants to receive information about a file stored on a provided storage, the namenode consults the map for the storage, and returns the storage information from the provided datanode descriptor which addresses the provided storage. Basically, the logic for retrieving the provided storage map from the provided datanode descriptor is bundled in this class.

Specifically, the class contains the inner classes `ProvidedDescriptor`, used to keep track of datanodes with provided storage (kept in a list), `ProvidedDatanodeStorageInfo`, an implementation of a provided datanode storage, and a builder for creating provided located blocks (blocks that are associated with datanodes). To summarize, the class is used for keeping track of the provided datanodes and associated provided storages.

Provided volume

The client to a provided storage has been implemented in `ProvidedVolumeImpl`. Its purpose is to create and store provided replicas and it is used when building provided volumes. It is a subclass of a general file system volume, `FsVolume`, which represents a general underlying volume to store replicas in, and therefore maintains the same functionality, except for the following changes;

The inner class `ProvidedVolumeDF` is added to keep track of usage statistic for provided volumes. For now, this simply includes the space used of the provided

volume (represented as a long).

The other new inner inner class, `ProvidedBlockPoolSlice`, represents a portion of the block pool stored on a provided volume. This is the important class that stores and interacts with the alias map and replica map to resolve blocks to file regions (step 4 in Figure 3.1). The provided volume also contains the iterators to iterate over the provided blocks in the file region.

3.3.1 Alias map implementations

The alias map contains the mapping between blocks and file regions in the provided storage and is shared between the namenode and datanodes. Currently, the implementation supports storing the mapping between blocks and file regions in a text file, `TextFileRegionAliasMap`, or in-memory of the namenode in the `InMemoryAliasMap`. Figure 3.3 displays the overview architecture of the alias map in terms of classes and protocols (see appendix A. for larger illustrations).

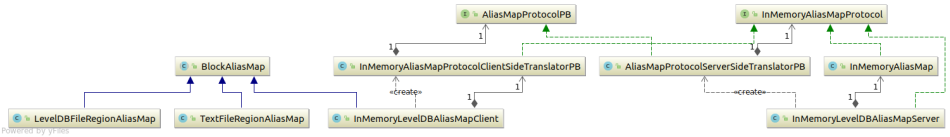

Figure 3.3: A UML diagram of the complete alias map architecture, containing the block and in-memory alias map implementations.

Block alias map

The abstract class `BlockAliasMap` is the super class of the block alias map implementations; `TextFileRegionAliasMap` and `LevelDBFileRegionAliasMap` with its client `InMemoryLevelDBAliasMapClient` (see Figure 3.3). This abstract class provides a template for how to interact with the alias map implementation through a generalized reader and writer. The reader has functionality for resolving a block to a corresponding alias and the writer stores the alias with the associated block.

In practice, the alias can be a file region or any other identifier, depending on the alias map implementation. The client will send the reader the provided block to be resolved in `Optional<U> resolve(Block b)`, the reader iterates over the alias, and if found, returns the matching alias of the block. Similarly, the generalized writer takes an alias in `void store(U alias)`, appends the associated block with the values of the alias, and stores it in the implementation specific way. Figure 3.4 displays the simple flow for reading and writing to an alias map, and the simple API for interacting with the alias map can be found below;

```
/*
* @param b is the block to resolve in the alias map.
* @return BlockAlias corresponding to the provided block.
```

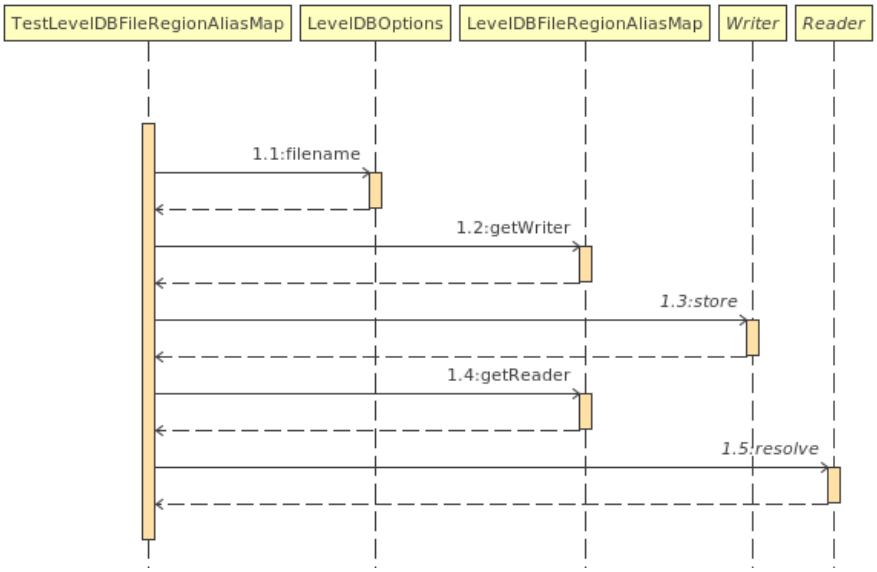

Figure 3.4: Sequence diagram of the read and write flow to an level DB alias map.

```
*/
public abstract Optional<U> resolve(Block b) throws
    IOException;

/*
* @param alias is the general file region to store in the
    alias map.
*/
public abstract void store(U alias) throws IOException;
```

Text file region alias map

A text implementation of the alias map is `TextFileRegionAliasMap`. It stores and loads the mapping of blocks to file regions as *blockID, path, offset, length, nonce* in a `.csv` file, and uses its reader and writer to interact with the file. The class is suitable for testing, but since it only stores the mapping in a physical file it will not scale beyond a single node, and is not suitable for multi-node production environments. The client can specify the path to the file as well as the delimiters to use (defaults to comma) to point the location of the map.

In-memory alias map

The `InMemoryAliasMap` implementation is better suited for production environment than the previous text version. Instead of keeping the alias mapping in a file, it runs an alias map service in the memory of the namenode, storing the mapping in a

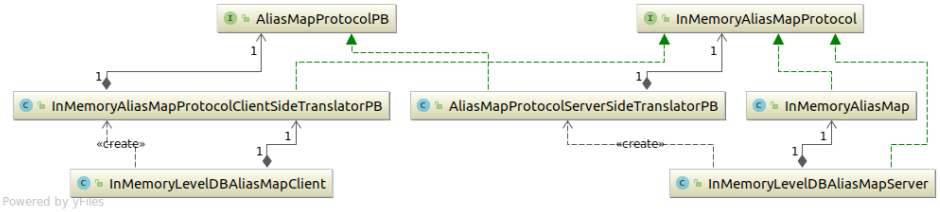

Figure 3.5: The complete in-memory alias map architecture containing the server, client, and translators

levelDB [9], a key-value store that provides an ordered mapping from string keys to string values. The key is stored as a byte array representing the given block, and similarly, the value a byte array of the file region. These pairs are converted to and from protocol buffers (protobuf), a language and platform-neutral mechanism for serializing structured data [8]. Utility methods are used to convert the block and file region to and from the protocol buffers byte arrays.

In-memory alias map protocol

This is the top hierarchy interface of the in-memory alias map implementation. The client and the server implements the protocol and use it to communicate with the map. The protocol is simple, containing method for listing, reading, and writing a block to location;

```
IterationResult list(Block marker)
void write(Block block, ProvidedStorageLocation location)
ProvidedStorageLocation read(Block block)
```

Noteworthy is that the iteration result of the list is an inner class (`IterationResult`), containing a batch of file regions and the next block to read from. The returned block is used as a marker by the client to know where to start the next listing from. The `ProvidedStorageLocation` is the implementation of the file region, but serves the purpose of addressing a location through a *path, offset, length, nonce*.

Client and server side translators

The `InMemoryAliasMapProtocolClientSideTranslatorPB` translates and sends request from the client to the sever (see Figure 3.5). For readability we will refer to this class as the ”client translator”. Similarly, we refer to the server side translator, `AliasMapProtocolServerSideTranslatorPB`, as “server translator”. The client translator configures the client side RPC proxy, through the common interface `AliasMapProtocolPB`, connecting with an statically configured socket IP address. The client translator then serves the purpose of translating client requests to and from protocol buffers and sending them to the in-

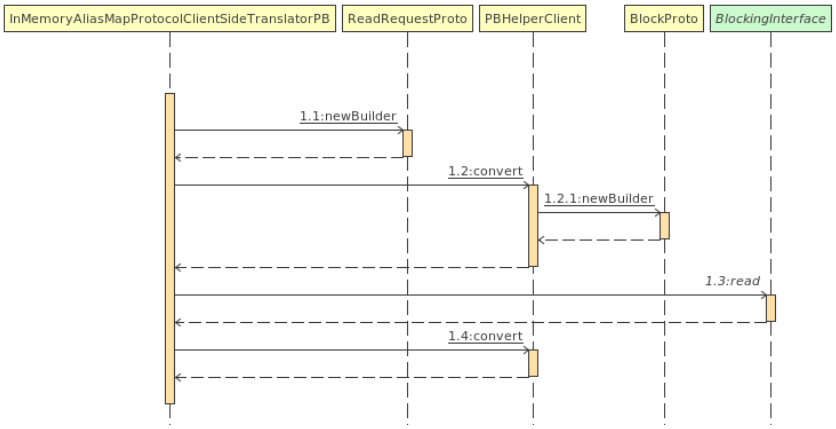

Figure 3.6: A sequence diagram displaying the read request flow from the client translator.

terface. Similarly, the server translator implements the `AliasMapProtocolPB` and decodes the messages on the server side and forwards them to the in memory alias map implementation, `InMemoryAliasMap`.

Figure 3.6 displays the flow diagram of a simple read request from the client translator. First, the translator uses a builder pattern to build the protobuffers (1.1). Then, the translator proceeds to converts the client request (list, read, or write) into protobuffers, using the helper class `PBHelperClient` (1.2). With the converted protobuf, the translator calls the RPC proxy `AliasMapProtocolPB`, which implements the `BlockingInterface`, with the read request (1.3).

In-memory alias map server

For the client to interact with the `InMemoryAliasMap` an RPC server is needed. This is done through the `InMemoryAliasMapServer`, which implements the `InMemoryAliasMapProtocol`. The class holds the RPC server for the alias map, as well as the reference to the actual `InMemoryAliasMap`. Further, it contains functionality for starting and stopping the RPC server, listing the aliases, and calling the `resolve` and `store` methods in the `InMemoryAliasMap`. The reference to the class is kept by the namenode whom also controls when to start and stop the RPC server.

In-memory alias map client

The client of the `InMemoryAliasMapServer` is used by the datanodes and the image creation tool, `fs2img` (see section 3.4) to store and retrieve file regions based on given block. Specifically, as this client class extends the abstract `BlockAliasMap`, it contains the same interaction functionality as previously described.

3.4 Image creation process

A tool for creating snapshots of namespaces introduced in HDFS with the provided storage is `fs2img` [33]. The tool iterates over a given remote filesystem by URL and creates a mirror *image* of the filesystem while populating the alias map, mapping blockIDs in the generated image to file regions in the scanned remote filesystem. As pointed out earlier, file region contains the `URL, offset, length` that uniquely identifies the remote sequence of bytes in the remote storage, along with the `nonce` that verifies that the region is unchanged since creation. Once created, the namenode and datanode has to be configuration to reference this address space and restarted to update their view of the remote namespace.

The scanning tool works by sending an *list* request to the remote storage client's API, retrieving a tree-hierarchy of the directories and files in the namespace. For each scanned node it saves the metadata as an inode and block file entry, writing the content to an image file. The file is then scanned upon cluster start up by the namenode, which uses journal logs to append the new snapshot information.

However, this approach of the fs2img does not work in HopsFS, since it does not support journal logs nor image snapshots, these are unnecessary as the metadata is stored in the Network Database (chapter 2.4.2). We redesigned the tool to, instead of creating a snapshot and storing the metadata in a file, write it directly to the database. This different approach can enable fs2img to be executed on-demand over the remote namespace, adding metadata as needed to the database and cluster. We would not need to restart the cluster to gain new information or add out-of-band chunks, as we can simply rerun the tool in a live cluster (or invoke a thread that periodically runs it).

Example run in HopsFS

Suppose we have a bucket called ”provided-test”, containg a folder called ”hej”, with a file inside it called ”build_sh.txt” (the extension is irrelevant). We now want to pull this namespace into the local HopsFS cluster. To do this we can run the following command:

```
$ fs2img -b TextFileRegionFormat s3a://provided-test/
```

Executing the above command, the tool first realizes that the prefix `s3a://` means that the S3A file system client should be used. It will create an instance of the class, `S3AFileSystem`, which is set as the file system client to interact with during the run. Then it sends an HTTP *list* request through the client which consults to the Amazon S3 API, returning the metadata hierarchy of the `s3a://provided-test/` bucket. It proceeds to iterate over the hierarchy, saving each inode (directories or files) and its blocks (data object) to a data structure. When done, the structure with inodes, blocks, and users, are persisted to NDB through the corresponding ClusterJ clients. The metadata of the file has now been fetched from the S3 bucket and stored in the alias map and persisted in the database.

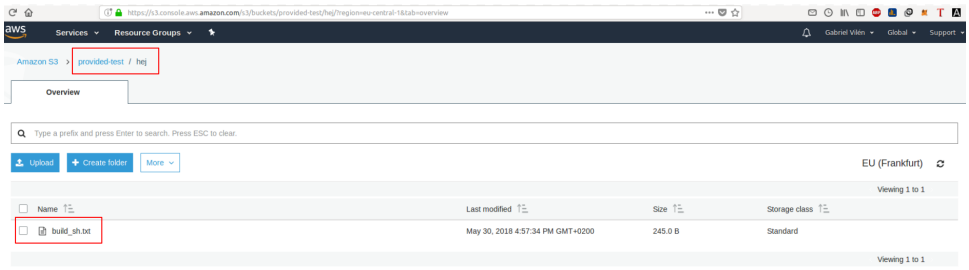

Figure 3.7: A screen shot from Amazon S3 displaying a file stored in a directory, about to be pulled into the local HopsFS through the scanning (fs2img) tool.

The above run is visualized in Figure 3.7, and the result can be seen by inspecting the created alias map csv file that contains the blockID, path, offset, length of ”build_sh.txt”:

```
1073741825, s3a://provided-test/hej/build_sh.txt, 0, 245
```

Chapter 4

Evaluation and Analysis

This chapter evaluates and analyses the implemented remote storage solution to answer RQ2. We start by benchmarking the performance of the implementation against a remote S3 bucket in Section 4.1 *Benchmarks*, followed by the unit tests in Section 4.2 *Tests* that has been implemented to ensure the correctness of the solution. Section 4.3 *Discussion* finishes the chapter with an analysis of the data availability and consistency aspects from RQ2.

4.1 Benchmarks

To test the performance of the provided storage solution, a couple of benchmarks have been concluded. This section details and discuss the various ones.

The benchmarks has been performed against an S3 bucket hosted on the AWS availability zone `EU-WEST-1` (ireland), accessed from a local HopsFS distribution (in Stockholm, release version 2.8) with a latency to the bucket of 30ms, through the cloud connector S3A (release version 2.7). The machine running the benchmarks is a commodity laptop with [Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz x 4, 8GB RAM, 350 MB/s SSD Storage] on Ubuntu 16.04 with a network download rate of about 210 mbit/s. It should be noted that these tests are by no means extensive as they are executing on a single-machine environment, running them in a live production cluster will likely produce different results. However, they should act as a solid base line for understanding the performance of the implementation.

The setup of the benchmarks is following; n datanodes has been created, with at least one serving a `PROVIDED` location ($DN_{provided}$), and n - $DN_{provided}$serving a `DISK` location (DN_{disk}). The $DN_{provided}$has been configured to point its storage location to the remote S3 bucket, whilst the DN_{disk}points it towards the disk on the local machine (note that we do not specify an SSD storage type, but since our local machine does not have HDD drives, as default for DISK storage, the DN_{disk}will use SSD). They both use the Network Database for storage of metadata and operate as normal datanodes in HopsFS.

4.1.1 Read over size performance

This benchmark has been concluded to test the average read performance of files of increasing size from the remote S3 storage versus files that resides in the local HopsFS file system. As seen in Figure 4.1, 99 files of increasingly size of 1 kilobyte - 90 megabytes has been read from the storage, each file has been read ten times, with the average plotted in the graph.

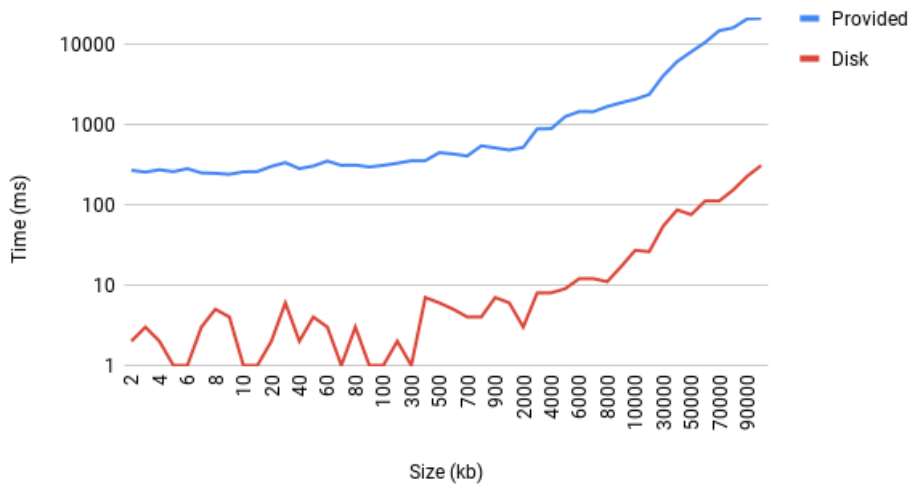

Figure 4.1: Graph displaying the time to read bytes from a remote (provided) storage versus local (disk) file. The x-axis displays the increasing storage size in kilobytes, with the read time in milliseconds on the y-axis.

As seen in the graph, the read performance of the $DN_{disk}$is almost a magnitude faster than of the $DN_{provided}$(note the logarithmic axis for comparison). We see that the read from the provided location has a fixed starting bound of around 150 ms, likely due to the latency of setting up of the connection to the remote S3 storage through S3A. The read time increases at around 1 megabyte file size (1000 kilobytes in the graph) and continues to grow linearly. This shows us that files of $<$1 MB might not make sense for frequent reading from a remote storage, due to the overhead.

4.1.2 Replication as caching

Due to the substantially slower performance for reading from the remote storage, caching the blocks on a local storage is desirable for speed up. However, due to time constrains, automatic caching mechanisms of remote replicas has not been implemented in our PoC (current work in Hadoop is being done on this, see HDFS-13069 [29]). In spite of this, replication of a `PROVIDED` block can be enforced from the client by setting the replication of the block to $>$1, prompting HopsFS to replicate the block to disk on a DN_{disk}. Subsequent reads from the block will then

go to the DN_{disk} and the disk speed up will be achieved (the next datanode to read from are sorted by network distance, and `PROVIDED` storages are always marked as being furthest away from the client).

The graph in Figure 4.2 shows this behaviour when a remote 100 MB file is replicated on a DN_{disk} (after read number 10 on the x-axis). An interesting behaviour is that the performance of subsequent reads to the local replica is even further increased, achieving a throughput of over 1000 MB/s (notice the drop from 4553, 1848, to 141 ms). This is due to HopsFS utilizing the Operative System (OS) Buffer to cache the local replicas in RAM. On average, the replication of ten remote versus local reads achieves a speed up of 39x in our benchmark, see Table 4.1.

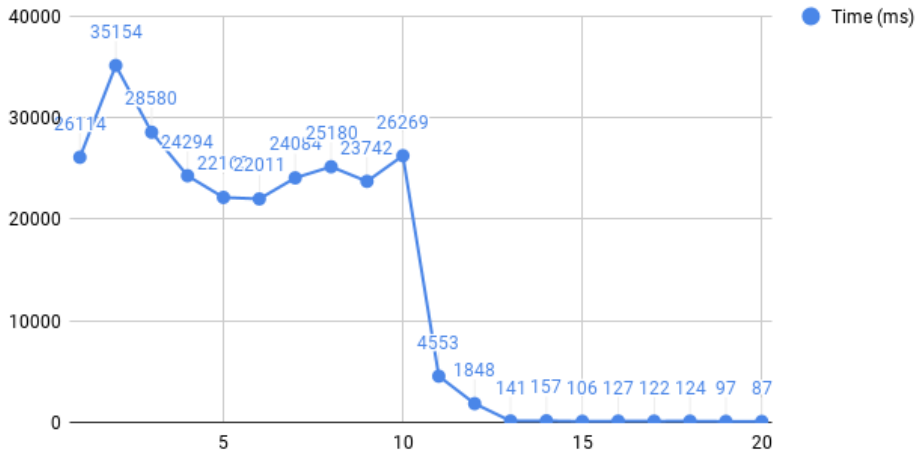

Figure 4.2: Graph displaying the time to read a file from an external storage which gets replicated onto a local disk. The x-axis display each read, and the y-axis the time in millisecond of each read.

Replica location	Average read time	Throughput	Speed up
Remote	25759 ms	4 MB/s	1x
Local	736 ms	156 MB/s	39x

Table 4.1: Average read time remote and disk replica.

Observing the speed up by replicating a block from $DN_{provided}$ to DN_{disk}, it raises the question of how long time such a replication takes. Figure 4.3 displays the replication time of replicating a block of size 0 - 80 MB from a $DN_{provided}$ to DN_{disk}. The minimum replication time of the benchmark is 6.5 seconds for an empty file, with an linearly increasing replication time (the copying the blocks) of about 3.5 MB/s.

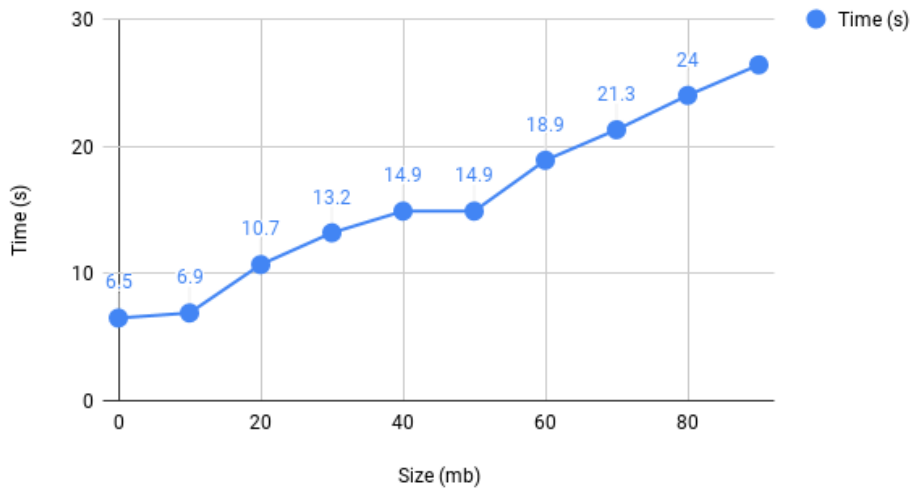

Figure 4.3: Graph of time to replicate a file from a provided datanode ($DN_{provided}$) to a disk datanode (DN_{disk}). The x-axis shows the size in megabytes of each file, and the y-axis the time to replicate the file.

4.1.3 Scanning tool performance

The tool responsible for iterating over the remote namespace, *fs2img*, has been benchmarked by scanning the bucket. The benchmark measures a run of the tool over the bucket; the time of scanning the remote bucket and storing the object metadata (user, group, inode, and blocks) into the Network Database (NDB). The NDB is running in a cluster in the same building as where the test have been concluded, with a latency of 5 ms.

Figure 4.4 displays a bar chart of scanning an increasingly large bucket. The test has been done to measure how the performance of an increasingly large bucket affects the scan and persist time of the *fs2img* tool. In the graph, the the linearly increasing *Storage size (mb)* displays the total size of the bucket, where the bars consist of the time of scanning the bucket, *Scan time*, with the additional time of persisting the found metadata to the NDB, *Persist time*. Since the scan time is dependant on network requests (HTTP HEAD) through the S3A to retrieve the object's metadata, the spiky behaviour is likely caused by a fluctuating network connection, where latency spikes resulted in a ten folded increase of scan time. However, the persist time constantly totals of a small amount of the total run time, as seen by the red bar top in the graph.

The take away is that the size of the storage and number of objects minimally affect the *fs2img* runtime, running the tool over a small storage performs the same as running it over a large one. Further, the performance is unaffected by the storage size as the HTTP HEAD requests only retrieves the metadata and not the actual file content itself. The majority of the impact seemingly relies on the network connection. Figure 4.5 displays this behaviour better, as bucket with many buckets

hardly impact the scan and persist time.

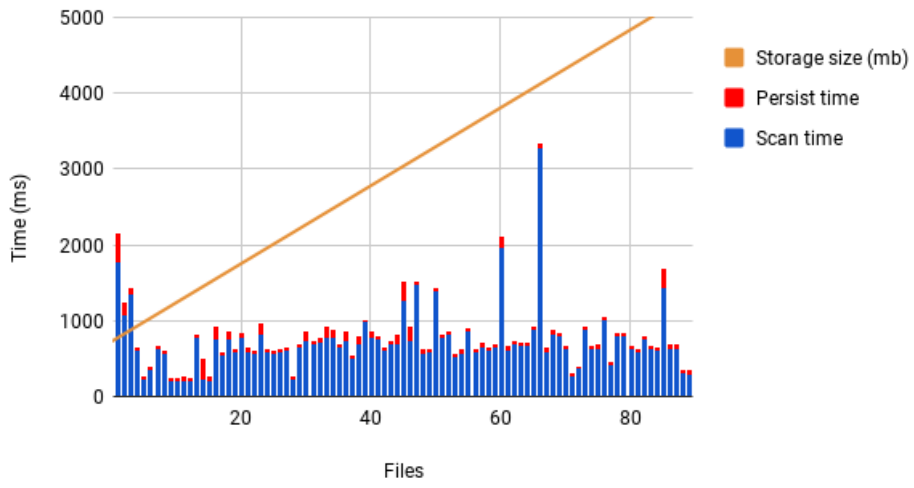

Figure 4.4: Graph displaying time to scan a remote bucket (scan time) with time to persist the object into the database (persist time) over an increasing remote bucket size (storage size). The x-axis display the number of files, and the y-axis the time in milliseconds of scanning and persisting the file's metadata.

4.2 Tests

To test the correctness of the provided storage solution, a multitude of unit tests has been implemented and some existing tests has been modified. This section details the most important ones.

Scan and storage tests

For testing the scanning and creation of the remote namespace, `testImageLoad` scans a provided directory, creates the image, and compares the file status of the created image with a local mirror. The test mocks a directory on disk as provided, creates an image of the disk, and compares it with the actual directory. This test that the tool loads the image correctly.

Block reporting tests

To verify that the $DN_{provided}$correctly reports the blocks and information to the namenode, `testProvidedReporting` will trigger the heartbeats of the datanodes that updates the namenode provided capacity and knowledge of the provided volumes. It will check that the multiple $DN_{provided}$are discoverable through the namenode, that their volumes and information (used capacity and block pool) is correct, and that the blocks are of the `PROVIDED` type and resides on a $DN_{provided}$.

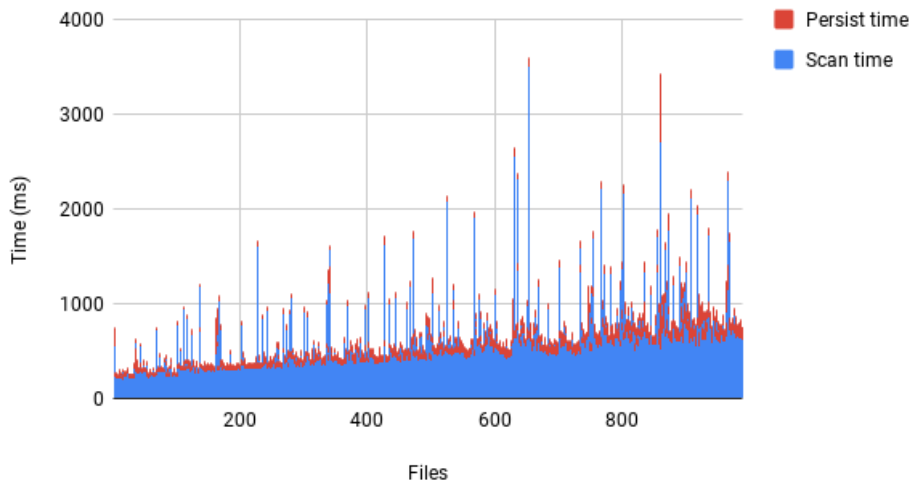

Figure 4.5: Graph of the time to scan a remote bucket (scan time) over an increasing number of files. The x-axis display the number of files, and the y-axis the time in milliseconds of scanning and persisting the file's metadata.

Replication and restart tests

Replication has been tested through `testDefaultReplication` and `testProvidedReplication`. The former checks that the blocks has the correct number of storage locations after increasing the replication factor of the block. This is done through comparing the number of hosts of the block and the expected locations. The latter is similar, but sets a replication of the block after it has been created. We then constantly poll the block until we see that is has been replicated on a new storage. This is the same idea which is used for benchmarking the provided replication. Noteworthy here is that a datanode can have multiple storages (since the heterogeneous storage patch), so one $DN_{provided}$may contain a `DISK` storage aswell. However, it will get replicated to another datanode which chooses to put the replica on its disk (due to choosing nearest storage policy as discussed earlier).

Failure tests

To test the datanode failure and restart, `testProvidedDatanodeFailures` will create n $DN_{provided}$with associated blocks, stop the datanodes, and make sure the namenode detects that they are dead. We then check that the locations are removed from the provided blocks and when all n $DN_{provided}$are dead, the block report count of the provided storage should be reset to zero. As long as there is one $DN_{provided}$alive that can reference the provided storage, we should not modify the block report count. When one $DN_{provided}$is restarted, the block report should go up to one, and the block address should correspond to the address of the $DN_{provided}$.

Location tests

The number of locations for a provided block has been tested in `testNumberOfProvidedLocations`, where multiple files and $DN_{provided}$ are created and stopped in sequential order to verify that the number of locations are correct per file. The `testNumberOfProvidedLocationsManyBlocks` functions in a similar way, however, it increases the number of blocks per file to ten from one, to check that the discovered locations work for multiple blocks per file. The location tests also include test for checking that the $DN_{provided}$ works with multiple racks, `testProvidedWithHierarchicalTopology`.

Alias map tests

The alias map has been tested through a variety of tests. The in-memory alias map which stores the alias map in a levelDB, has been tested in `testInMemoryAliasMap`. The test creates a temporary directory for storing the in-memory alias map and an alias map server to recieve the Remote Procedure Calls (RPCs). Reading and writing of the alias map is verified by creating blocks and associated provided storage locations, `ProvidedStorageLocation`, which are written and read from the levelDB instance by converting sent protocol buffers to and from java object.

The text alias map, `TextFileRegionAliasMap`, has been used in the majority of tests and benchmarks and similar tests has been done against it. These include setting the reader and writer options (e.g. codex) and testing the read, list, and write interaction against the csv file.

S3 tests

A few tests have been implemented to test the replication and reading against a remote live S3 storage, both directly through a provided replica and a $DN_{provided}$. The tests create and write some content into a local file, uploads it to S3 through the S3A connector, and asserts that a client can read the file content through the $DN_{provided}$. In turn, the replication tests ensures that the objects in an S3 bucket can be properly replicated onto local datanodes.

As mentioned earlier, the consistency model of a remote storage might not be consistent with that of the POSIX compliant file system. For instance, S3 provides read-after-write consistency for new objects but eventual consistency for overwrite and deletes [12]. Since all our tests created new objects, there were no recorded consistency issue. To test the number of encountered issues, another test was created that deletes an object in a remote bucket followed by trying to list it by running *fs2img*. If the deleted object was found by the tool, an consistency issue would be found for that object. The test was performed on 1000 files in a bucket, deleting one at a time followed by executing the tool. Surprisingly, not a single consistency issue was encountered in our test. Obviously, more extensive tests would have to be ran to measure how often such consistency issues appear in live production environments,

where I/O is heavier. We encourage future work to use the existing frameworks that implements a strongly consistent metadata layer on top of S3 to deal with this potential issue (see section 2.5.2).

4.3 Discussion

In this section we discuss some of the aspect that has not been tested, but should be taken into consideration for future development and live production deployment of the remote storage solution. We investigate the consistency and data availability aspects of RQ2.

4.3.1 Handling consistency issues

One consistency issue arises if data is changed out-of-band in the remote storage. This will be critical to handle to prevent race-conditions (such as TOCTOU, Time Of Check to Time Of Use [44]) when writes are to be enabled, but are less of an issue in current implementation. For this, the *nonce* (the file version identifier) may be used to ensure that the right version of the remote replica is updated. The nonce value may further be used to verify read requests, such as to implement an operation to open file by nonce or id (see HDFS-7878 [13]). Specifically, when a client wants to read a block from a remote storage, the namenode could return and embed the nonce in the located block response. The nonce could then be passed to the $DN_{provided}$which uses it for the *open-file-by-nonce* operation. Another approach is to store the nonce in the alias map and let the $DN_{provided}$look it up to ensure that it is equal to the generation stamp of client's block request.

To handle changes in the remote namespace, these would first have to be reflected in the local cluster. This will requires synchronization protocols for periodically updating HopsFS' view of the remote storage, pulling in and removing outdated metadata from the database and updating the alias map. Divergence between the current and remote cluster will have to be resolved and merged, such as addition of new metadata and deletion of cached (local) replicas. Currently, our PoC implementation does not take out-of-band changes into consideration, we instead generate a fresh snapshot of the remote namespace each time when manually run the *fs2img* tool. Potentially, the scan could be automatically executed to update the remote metadata in NDB. However, this would require locking the database to prevent race conditions during a live cluster run, which might not be desirable due to performance impact.

4.3.2 Data availability

The question of data availability and consistency can be looked at two-fold, either we simultaneously provide availability but loose consistency, or we simultaneously provide consistency but loose availability (CAP theorem [6]). In our solution, we do not take inconsistencies into consideration and data is available at all time, a

client requesting a read form a remote object will always have it returned through a $DN_{provided}$. A potential issue arises when the remote block is replicated onto a local DN_{disk}. The block data is still available as future reads will go to the DN_{disk}, and this replica is now consider to be the single source of truth. As long as the local replica is not modified, there are no issues. However, if the replica is modified to another state there will be inconsistency between the local and remote one. How should this be be reflected in the remote storage? For instance, if we assume that the blocks should be kept synchronized between the local and remote storage, these blocks will have to become unavailable for some time, which may be for a significant time depending on network bandwidth and amount of the changes. Synchronization also introduces another problem; moving data out and into the S3 cluster cost money (S3 transfer fee), from an economical perspective it is desirable to keep inter-cluster IO to a minimum. For now, we avoid these problems as we do not allow writes on remote replicas, but such questions would have to be taken into consideration for future work.

4.3.3 Handling security

As mentioned in the introduction, one scenario in deploying the remote storage is a hybrid cloud scenario where secure data is in the cluster and less secure on the cloud. Another is to enable clients to store secure data on the cloud, a full-cloud deployment. To do this we must enforce some kind of security and access rights over the remote data. The problem is that the remote storage often does not have the same security as the local file system. For instance, HopsFS has POXIS security semantics (i.e. user-group-everyone), while S3 uses bucket-wide access credentials and user policies (such as bucket read, list, or write-only access). The security model depends on the policies of the remote storage, and while mapping these to the POSIX model would be desirable, there are no easy provider-independent way to do this. Currently, HopsFS needs to be configured with the S3 security credentials which will grant HopsFS permission to access all objects in the remote bucket. This impersonation approach will share the security credentials (i.e. access keys) amongst all users in the cluster, which might not be desirable in a large environments with many users. Another approach is to authenticate the requests through a client token, having the client supply his credentials through the cluster and proxy them through the $DN_{provided}$, supplying it with a necessary token that acts on behalf of a client.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis has presented a new feature in HopFS that allows the distributed file system to be integrated with an external remote storage. This external storage is said to be *provided* and could in essential be any other type of storage, such as another distributed file system (e.g. HopsFS or HDFS) or a live cloud storage (e.g. Amazon S3, Google Cloud Storage, or Microsoft Azure). The presented feature enables a client of HopsFS to access, retrieve, and store her cloud data transparently and seamlessly from within the local file system.

The first research question of the thesis asked; *How can a distributed block based file system effectively be integrated with an external object based cloud storage?* (RQ1). We answer this through the implementation where we show how this feature can and has been designed and adopted in HopsFS. Since the HDFS and HopsFS architecture fundamentally differers, the work has included research of implementation differences, redesigning the way of scanning, handling and storing the provided storage's metadata, and implementing cloud connectors (S3A) for testing the solution with live remote (S3) storages.

The second research question; *What are the implications of such an implementation in terms of; data consistency, data availability, and performance?* (RQ2), has been partly answered by evaluating the feature from an performance aspect, which included benchmarking of the read, scan, replication, and failover -time. Since the PoC implementation is read-only and we mount a static snapshot through the *fs2img* tool, we have avoided synchronization with consistency and data availability issues. However, for future work we have seen that external frameworks can be used to layer stronger consistency ontop of the implementation.

We have seen that the PoC scales well, a large remote storage (bucket) does not significantly increase the scan or database persist time due to the scan tool scanning and persisting the remote object's metadata and not its actual content, resulting in much smaller I/O. The content is pulled in on-demand when read from the client and the observed bottleneck has been network, due to inter-cluster I/O relying on

network bandwidth which may heavily fluctuate the performance. The reads from the bucket has showed to scaled linearly with an average throughput of 4 MB/s (using S3A version 2.7). Replication ”caching” of the remote block onto local disk (SSD) storage has proved to be very beneficial, with an average speed-up of 39x over ten remote/local runs, followed by even further speed-up by the Operating System (OS) Buffer caching the local replicas in RAM. Comparing reading from local to remote we achieving a 100x read throughput.

5.2 Future Work

The developed Proof-of-Concept solution has been the first stage of how to integrate and transparently use a remote storage from a local file system cluster. As such, there is still future work to be considered until the feature is fully production ready.

Caching mechanisms

Currently, caching of a remote block is done by manually setting the replication factor of the block to be higher than one. This will prompt HopsFS to replicate the remote block from the external storage onto a local disk one. Future reads will be directed to the new local datanode hosting the block, which functions as a cache. However, there are no automatic caching mechanisms in place that performs such steps when reading from a remote storage. This will pose a problem for future write scenarios, where writes would have to be directed through this local datanode, potentially introducing high load and bottlenecks on the datanode. Each provided datanode having their own local cache with caching mechanisms, such as supporting lazy-writes or read/write-through caching, should be considerable for future development.

Reflecting remote changes

If an object changes in the remote storage (i.e. another application or user edits it), the local replica is not modified. Similarly, a local change is not reflected to the remote storage. Implementing uni or bi-directional synchronization could be done by using *endpoints*, reflecting a directory whose content is to be kept in sync with the remote storage (see HDFS-12090 [32]). These would have to be synchronized asynchronously and propagate metadata and blocks between the remote and local storage, and tools for attaching such endpoints would have to be implemented. This requires policies of how often to and what data to synchronize. For instance, an uni-directional remote to local reflection could be done by periodically execute *fs2img* and only paging in new data (which would have to be checked against the *nonce* value). Depending on the remote provider, bandwidth and cloud transfer fees might occur, which should be taken into consideration when developing this functionality.

Writes to remote replicas

Supporting writes is important for an production ready remote storage integration, where a client may want to write new values to the block which is to be uploaded to the remote cloud storage. However, as discussed earlier, this poses questions such as how to deal with inter-cluster synchronization, availability, and consistency issues. For instance, when the remote cluster diverges from the local one, protocols will have to deal with merges and inconsistent states. Handling writes was scoped outside this PoC but will be a considerable future project.

Bibliography

[1] Arpit Agarwal. Heterogeneous storage policies in hdp 2.2. *https://hortonworks.com/blog/heterogeneous-storage-policies-hdp-2-2/*, January 2015.

[2] Amazon. Amazon emr documentation. consistent view. *https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-consistent-view.html*, 2018.

[3] Amazon. Amazon simple storage service (s3) homepage. *https://aws.amazon.com/s3/*, Retrieved: february 2018.

[4] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 331–343. ACM, 2017.

[5] Dhruba Borthakur et al. Hdfs architecture guide. book. hadoop apache project. volume 53, 2008.

[6] Julian Browne. Brewer's cap theorem. *J. Browne blog*, 2009.

[7] Logical Clock. Company team. *https://www.logicalclocks.com/team/*, Retrieved august 2018.

[8] Google Developers. Google's protocol buffers. From: *https://developers.google.com/protocol-buffers/*, Retrieved june 2018.

[9] Google Developers. Level db. From: *https://github.com/google/leveldb*, Retrieved june 2018.

[10] Chris Douglas, Virajith Jalaparti, Sriram Rao, Thomas Demoor, and Ewan Higgs. Allow hdfs block replicas to be provided by an external storage system. jira hdfs-9806. https://issues.apache.org/jira/browse/HDFS-9806?attachmentSortBy=fileName, 2017.

[11] Chris Douglas, Virajith Jalaparti, Sriram Rao, Thomas Demoor, and Ewan Higgs. Tiering hdfs over external storage systems. *PDF*, 2017.

[12] Chris Douglas and Sergey Shelukhin. Api - expose a unique file identifier. jira hdfs-7878. *https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html*, 2017.

[13] Chris Douglas and Sergey Shelukhin. Api - expose a unique file identifier. jira hdfs-7878. *https://issues.apache.org/jira/browse/HDFS-7878*, 2017.

[14] Facebook. Presto website. distributed sql query engine for big data. *https://prestodb.io/*, Retrieved august 2018.

[15] Apache Foundation. Uber-jira: S3a phase iii: scale and tuning. jira hadoop-13204. *https://issues.apache.org/jira/browse/HADOOP-13204a*, 2017.

[16] Apache Foundation. Uber-jira: S3a phase iv: Hadoop 3.1 features. jira hadoop-14831. *https://issues.apache.org/jira/browse/HADOOP-14831*, 2018.

[17] Apache Foundation. Hadoop-aws module: Integration with amazon web services. hadoop docs.
http://hadoop.apache.org/docs/current/hadoop-aws/tools/hadoop-aws/index.html, Retrieved mars 2018.

[18] The Apache Software Foundation. Flink website. stateful computations over data streams. From: *https://flink.apache.org/*.

[19] The Apache Software Foundation. Apache zeppelin website. web-based notebook that enables data-driven, interactive data analytics and collaborative documents with sql, scala and more. From: *https://zeppelin.apache.org/*, Retrieved august 2018.

[20] The Apache Software Foundation. Kafka website. a distributed streaming platform. From: *https://kafka.apache.org/*, Retrieved august 2018.

[21] The Apache Software Foundation. Spark website. lightning-fast unified analytics engine. From: *https://spark.apache.org/*, Retrieved august 2018.

[22] Apache Foundations. Hadoop *documentation*. hdfs federation. *https://hadoop.apache.org/docs/r3.0.3/hadoop-project-dist/hadoop-hdfs/Federation.html*, Retrieved august 2018.

[23] Lars George. *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system, 2003.

[25] Linux System Administrators Guide. What are filesystems? From: *https://www.tldp.org/LDP/sag/html/filesystems.html*, Retrieved august 2018.

[26] Cloud Guru. What is the availability on s3. aws certified solution architecht. From: *https://acloud.guru/forums/aws-certified-solutions-architect-associate/discussion/-KMFsMGnjQHHoiESSnuP/what-is-the-availability-on-s3-question*, Retrieved june 2018.

[27] Hortonworks. Chapter 2. the cloud storage connectors. From: *https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-3.0.0/bk_cloud-data-access/content/intro.htm*, Retrieved august 2018.

[28] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, Feb. 1988.

[29] https://issues.apache.org/jira/browse/HDFS 13069. Enable hdfs to cache data read from external storage systems. *https://issues.apache.org/jira/browse/HDFS-13069*, January 2018.

[30] IBM. The four v's of big data. infographic. http://www.ibmbigdatahub.com/infographic/four-vs-big-data, 2004.

[31] Virajith Jalaparti. Abstract implementation-specific details from the datanode. jira hdfs-9809. *https://issues.apache.org/jira/browse/HDFS-9809*, 2017.

[32] Virajith Jalaparti. Hadoop jira. handling writes from hdfs to provided storages. *https://issues.apache.org/jira/browse/HDFS-12090*, Retrieved august 2018.

[33] Virajith Jalaparti and Chris Douglas. Introduction to amazon s3. amazon s3 data consistency model. *README file*, 2006.

[34] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[35] Bram Leenders. Master thesis. heterogeneous storage in hopsfs., 2016.
[36] Steve Loughran. Using s3guard for amazon s3 consistency. *Hortonworks blog*. *https://hortonworks.com/blog/s3guard-amazon-s3-consistency/*, 2017.
[37] Steve Loughran and Sanjay Radia. The history of apache hadoops support for amazon s3. *https://hortonworks.com/blog/history-apache-hadoops-support-amazon-s3/*, October 2016.
[38] Microsoft. Access control lists. From: *https://docs.microsoft.com/en-gb/windows/desktop/SecAuthZ/access-control-lists*, retrieved august 2018.
[39] Phillip Moore. When your business depends on it. the evolution of a global file system for a global enterprise. Stanford *PDF*.
[40] Chris Nauroth. Apache hadoop s3guard jira. *https://issues.apache.org/jira/browse/HADOOP-13345*, 2016.
[41] B Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications magazine*, 32(9):33–38, 1994.
[42] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. Hopsfs: Scaling hierarchical file system metadata using newsql databases. https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi, 2017.
[43] Amazon official documentation. Amazon s3 rest api introduction. From: *https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html*, Retrieved august 2018.
[44] OWASP. Toctou definition. time of check to time of use. *https://www.owasp.org/index.php/Time_of_check,_time_of_use_race_condition*, Retrieved september 2018.
[45] Hadoop Apache Project. Hdfs architecture guide. hdfs official documentation. From: *https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html*, 2008.
[46] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 51–63. ACM, 2017.
[47] Ohad Rodeh and Avi Teperman. zfs-a scalable distributed file system using object disks. In *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 207–218. IEEE, 2003.
[48] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on computers*, 39(4):447–459, 1990.
[49] Amazon Web Services. Amazon ec2 website. secure and resizable compute capacity in the cloud. launch applications when needed without upfront commitments. *https://aws.amazon.com/ec2/*, Retrieved august 2018.
[50] Amazon Web Services. Amazon emr website. easily run and scale apache hadoop, spark, hbase, presto, hive, and other big data frameworks. *https://aws.amazon.com/emr/*, Retrieved august 2018.
[51] Amazon Web Services. Aws management console. From: *https://aws.amazon.com/console/*, Retrieved august 2018.
[52] K. V. Shvachko. Hdfs scalability: The limits to growth. In *The Magazine of USENIX*, volume 35, no. 2, page 616.
[53] Hortonworks. Suresh Srinivas. An introduction to hdfs federation. *https://hortonworks.com/blog/an-introduction-to-hdfs-federation/*, Retrieved august 2018.

[54] Suresh Srinivas. Enable support for heterogeneous storages in hdfs - dn as a collection of storages. jira. *https://issues.apache.org/jira/browse/HDFS-2832*, 2014.

[55] Gil Vernik. Advantages and complexities of integrating hadoop with object stores. ibm site. *https://www.ibm.com/blogs/cloud-computing/2017/05/integrating-hadoop-object-stores/*, Retrieved mars 2018.

[56] Gil Vernik, Michael Factor, Elliot K. Kolodner, Effi Ofer, Pietro Michiardi, and Francesco Pace. Stocator: An object store aware connector for apache spark. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 653–653, New York, NY, USA, 2017. ACM.

[57] Daniel C. Weeks. Netflix s3mper. blog. *http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html*, 2014.

[58] Tom White. Hadoop the definitive guide, 2015.

[59] Zhihong Yao. How google cloud storage offers strongly consistent object listing thanks to spanner. blog. *https://cloudplatform.googleblog.com/2018/02/how-Google-Cloud-Storage-offers-strongly-consistent-object-listing-thanks-to-Spanner.html*, 2018.

Chapter 6

Appendix

The appendix includes a number of sequence and UML diagram that has been scaled-up for readability in the thesis. Some of these has been extended with more detailed information.

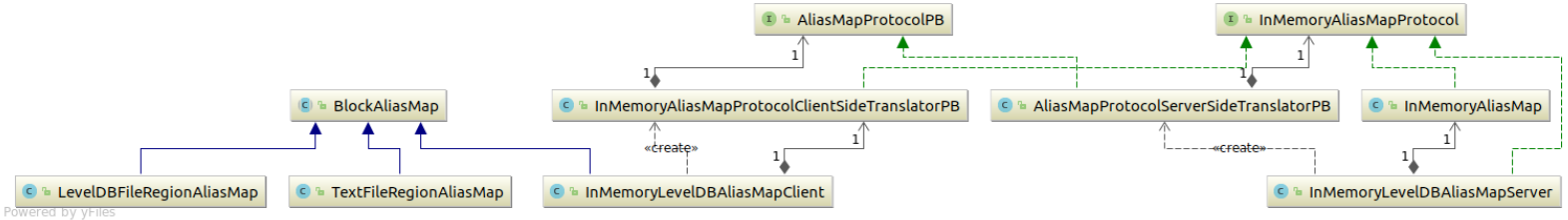

Figure 6.1: The alias map architecture

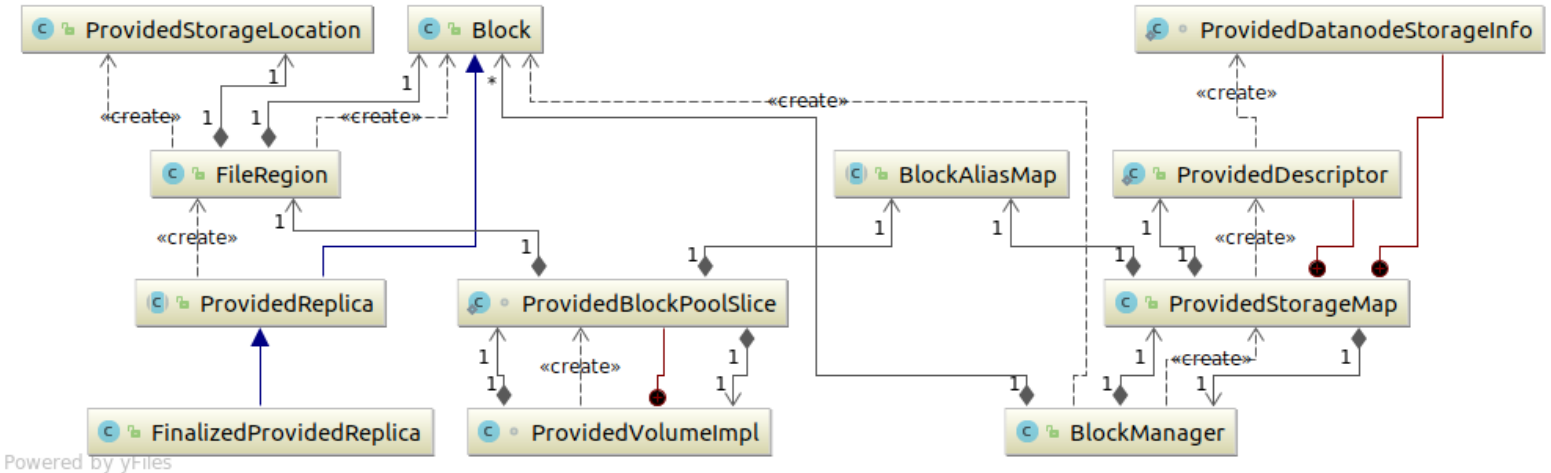

Figure 6.2: The provided storage architecture

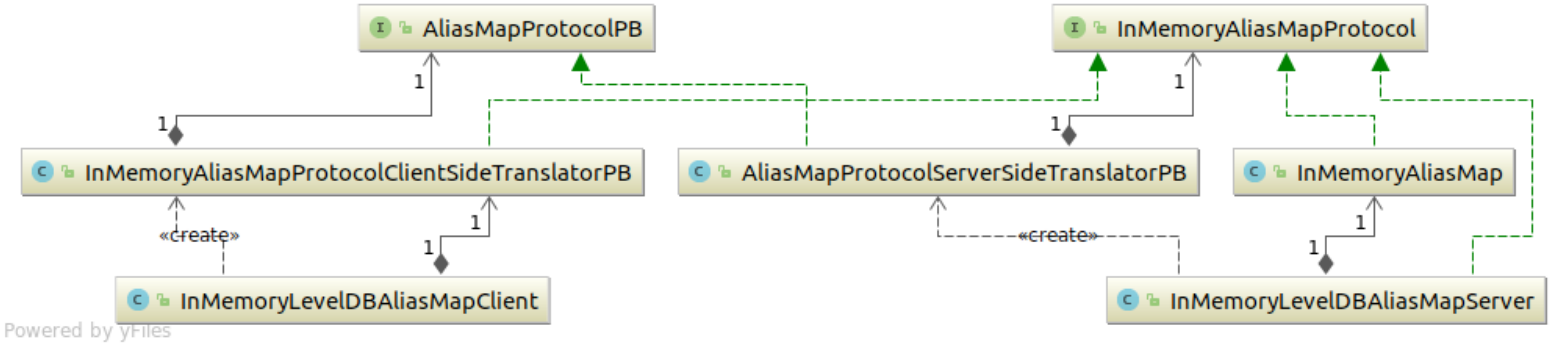

Figure 6.3: The in-memory alias map architecture

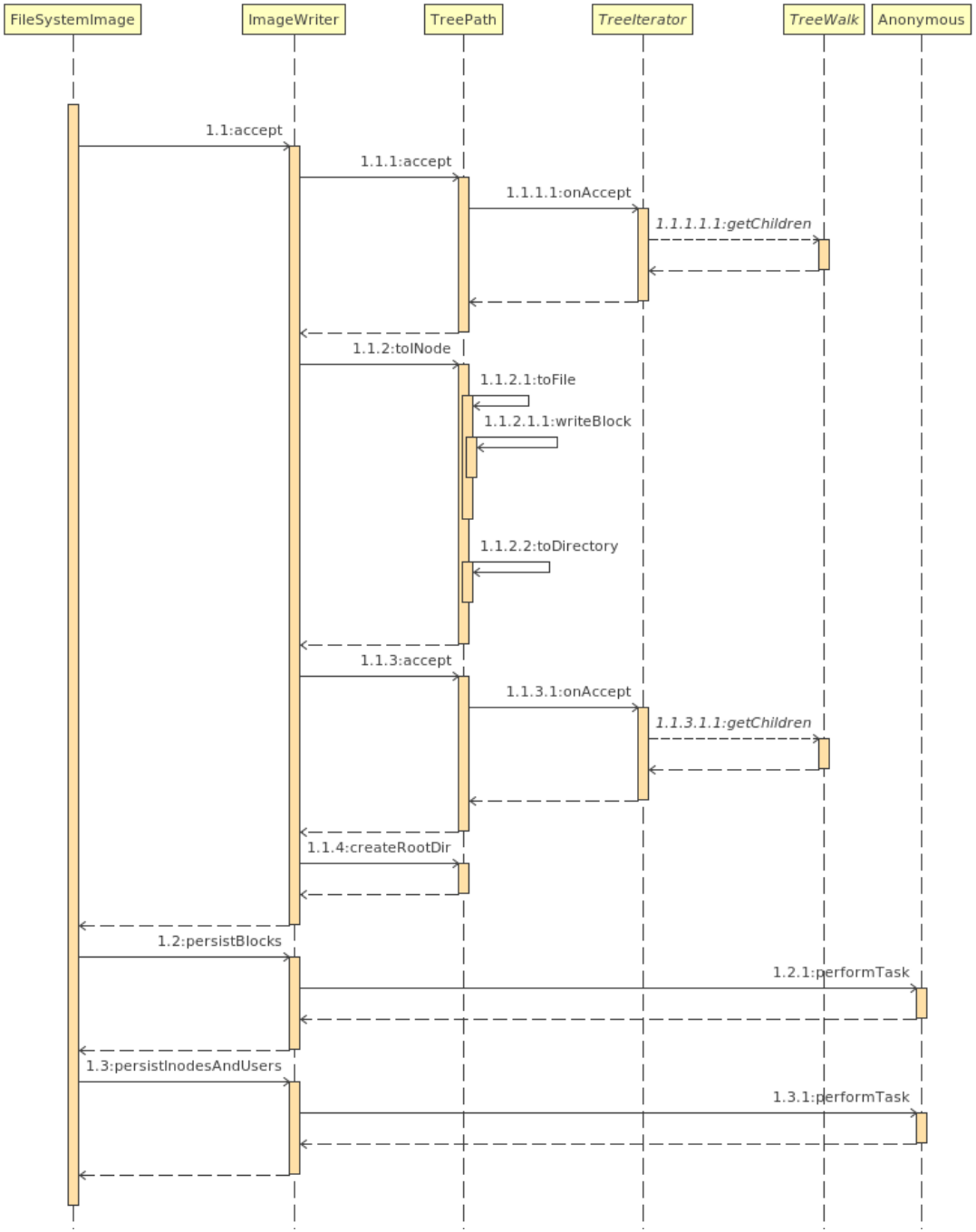

Figure 6.4: A sequence diagram displaying a complete fs2img run from external inode and block creation to persisting the information in the database.

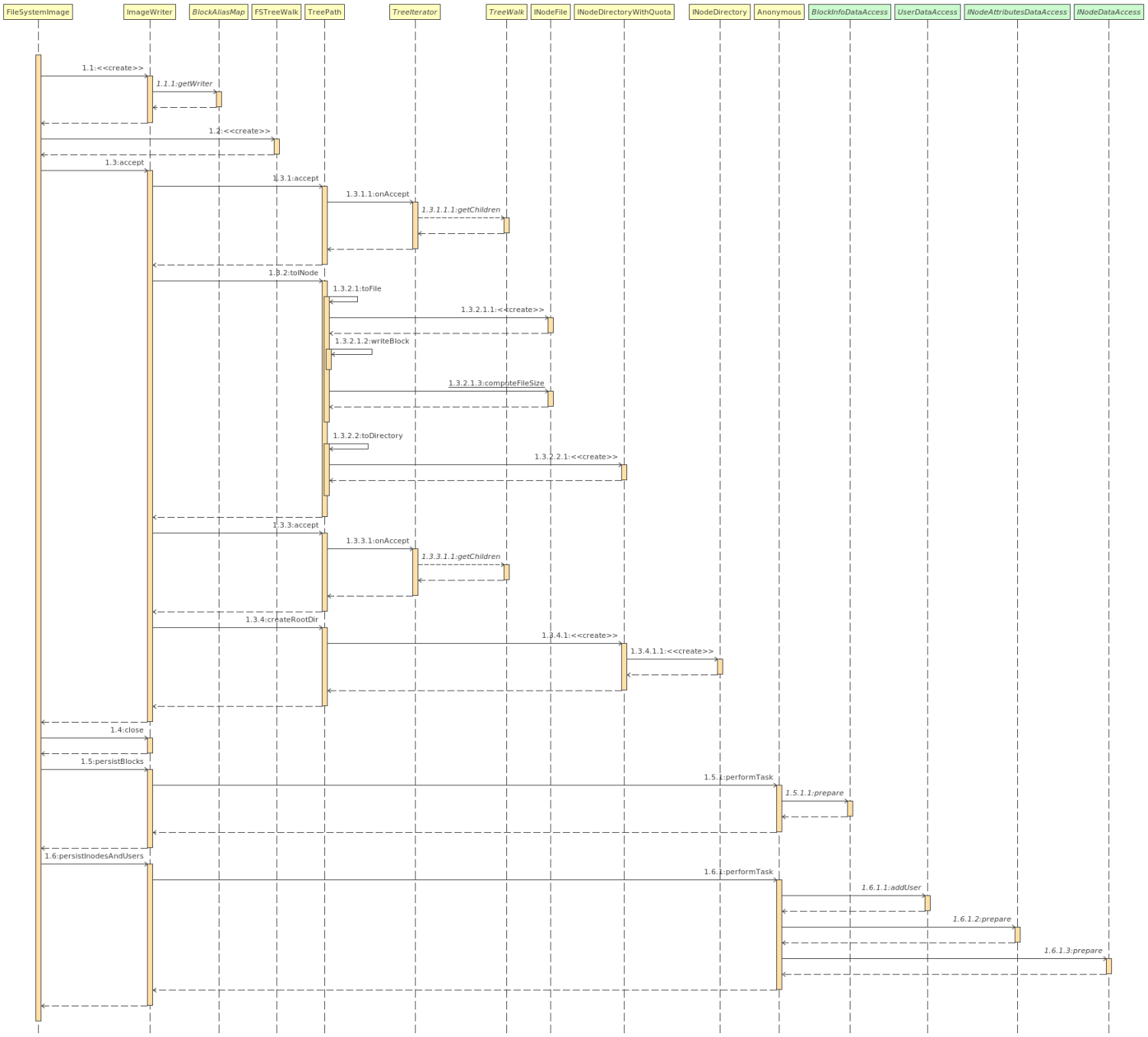

Figure 6.5: Fs2img extended sequence diagram.