Project number: 317871
Project acronym: BIOBANKCLOUD

**Project title:** Scalable, Secure Storage of Biobank Data  
**Project website:** [http://www.biobankcloud.eu](http://www.biobankcloud.eu)  
**Project coordinator:** Jim Dowling (KTH)  
**Coordinator e-mail:** jdowling@kth.se

**WORK PACKAGE 2:**  
Scalable Storage

WP leader: Jim Dowling  
WP leader organization: KTH  
WP leader e-mail: jdowling@kth.se

**PROJECT DELIVERABLE**

**D2.4**  
Secure, scalable, highly-available Filesystem integrated with the Object Model and Overbank

Due date: 31st May, 2015 (M30)
Editor
Jim Dowling (KTH)

Contributors
Salman Niazi, Mahmoud Ismail, Steffen Grohsschmiedt, Gautier Berthou, Jim Dowling (KTH)
Ricardo Mendes, Vinicius Cogo, Tiago Oliveira, Alysson Bessani (FFCUL)

Disclaimer
This work was partially supported by the European Commission through the FP7-ICT program under project BiobankCloud, number 317871.

The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose.

The user thereof uses the information at its sole risk and liability. The opinions expressed in this deliverable are those of the authors. They do not necessarily represent the views of all BiobankCloud partners.
Executive Summary

This deliverable presents our distribution of Hadoop, developed in the BiobankCloud project, called Hops (Hadoop Open Platform-as-a-Service) and its support for security as well as its integration with the Object Model (D3.5) and CharonFS (D4.3). HopsFS has been designed with the need to store high-coverage whole-genome sequencing (60x) files that can contain up to 250 GB of raw data as well as roughly an equivalent amount of aligned genomic data after processing. In previous deliverables (D2.2 and D2.3), we have described how extends Apache’s Hadoop Filesystem (HDFS) to build a scalable, highly-available storage system to support the storage of hundreds petabytes of genomic data. We added support for storing metadata in a highly available distributed database, MySQL Cluster.

A new contribution in this deliverable is Hops-YARN. YARN (Yet Another Resource Negotiator) is the resource management framework for Hadoop, enabling Hadoop V2+ to support a wide variety of data processing frameworks (such as Apache Spark and Apache Flink) and no longer only MapReduce. Similar to HopsFS, in Hops-YARN we moved the ResourceManager state from the heap of a JVM to a highly available distributed database, MySQL Cluster. This has enabled us to partition the ResourceManager services, and separately scale-out the ResourceTracker service, which is concerned with processing heartbeats from and commands to NodeManagers.

In this deliverable, we also present a user manual for HDFS’ command-line tools and provide more detailed performance evaluation figures, as well as a new file path caching optimization built using memcached. For deep filesystem paths (with 20+ path components), memcached gives us a 2x performance boost.

We also present the integration with the Object Model through automated indexing of filesystem objects in ElasticSearch. That is, extended metadata for files and directories can be generated by HopsFS and indexed by ElasticSearch, all while ensuring that the searchable metadata and the actual filesystem metadata are kept consistent. Extended metadata is kept strongly consistent with filesystem metadata using foreign keys in the database, while ElasticSearch indexing data is guaranteed to be eventually consistent with metadata in the database. Finally, we show how CharonFS integrates with HopsFS, with HopsFS as a datastore for CharonFS. This enables us to securely share data stored in HopsFS with other HopsFS clusters using public clouds.
Contents

1 Introduction

2 Hadoop Distributed File System
   2.0.1 NameNode limitations
   2.0.2 Storage limitations
   2.0.3 HDFS improvements

3 Hops-FS
   3.0.1 Hops-FS Architecture
   3.1 Distributed metadata
      3.1.1 Metadata partitioning
   3.2 Fine grained namespace Locks
   3.3 Multiple NameNodes
   3.4 Hops-FS leader election and group membership
   3.5 DataNodes
   3.6 Clients
   3.7 Erasure Coding
   3.8 Metadata security
   3.9 Combining Apache v2 and GPL v2 Open Source Licenses
   3.10 Removed and Deprecated Features
   3.11 Hops-FS User Manual
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12 Unsupported parameters and commands</td>
<td>15</td>
</tr>
<tr>
<td>3.12.1 Configuration Parameters</td>
<td>15</td>
</tr>
<tr>
<td>3.12.2 Administrator Commands</td>
<td>16</td>
</tr>
<tr>
<td>3.13 Hops-FS configuration parameters</td>
<td>16</td>
</tr>
<tr>
<td>3.13.1 NameNode configuration parameters</td>
<td>17</td>
</tr>
<tr>
<td>3.13.2 Client configuration parameters</td>
<td>20</td>
</tr>
<tr>
<td>3.13.3 Data access layer configuration parameters</td>
<td>21</td>
</tr>
<tr>
<td>4 Object Model and Data Security</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Metadata</td>
<td>23</td>
</tr>
<tr>
<td>4.1.1 HDFS Changes</td>
<td>23</td>
</tr>
<tr>
<td>4.1.2 Auto-updated Metadata using HDFS and the Database</td>
<td>24</td>
</tr>
<tr>
<td>4.1.3 Metadata Designer Tool</td>
<td>26</td>
</tr>
<tr>
<td>5 CHARON—Hops-FS integration</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>5.2 The CHARON File System</td>
<td>29</td>
</tr>
<tr>
<td>5.3 Integration Scenarios</td>
<td>30</td>
</tr>
<tr>
<td>5.4 Launching the chosen integration scenario</td>
<td>33</td>
</tr>
<tr>
<td>5.4.1 Hops-FS as a private repository in CHARON</td>
<td>33</td>
</tr>
<tr>
<td>5.4.2 CHARON managing external files</td>
<td>33</td>
</tr>
<tr>
<td>5.4.3 Sharing data</td>
<td>34</td>
</tr>
<tr>
<td>5.5 Deploying CHARON with Chef/Karamel</td>
<td>36</td>
</tr>
<tr>
<td>6 YARN</td>
<td>38</td>
</tr>
<tr>
<td>6.1 Hadoop YARN</td>
<td>38</td>
</tr>
<tr>
<td>6.1.1 Single point of failure</td>
<td>39</td>
</tr>
<tr>
<td>6.1.2 ResourceManager the Scalability Bottleneck</td>
<td>40</td>
</tr>
</tbody>
</table>
6.2 Hops YARN ........................................ 41
   6.2.1 MySQL Cluster Event API ....................... 42
   6.2.2 Hops YARN functioning ......................... 43

7 Evaluation ........................................ 44
   7.1 Hops-FS NameNode Performance .................. 44
   7.2 Effect of Number of RPC Handler Threads ....... 44
   7.3 Multiple NameNodes ............................... 45
   7.4 SubTree Operations Performance ................ 47
   7.5 File Path Length and Performance ................ 48
   7.6 Namespace overhead and Metadata scalability .. 49
      7.6.1 HDFS namespace usage ....................... 49
      7.6.2 Hops-FS namespace scalability ............... 51
   7.7 Erasure coding .................................. 54
      7.7.1 Resilience to failures ....................... 54
      7.7.2 Effect of block placement ................... 56
      7.7.3 Availability and degraded reads .............. 58
      7.7.4 Comparison to HDFS-RAID ...................... 59
   7.8 Leader Election Evaluation ...................... 60

8 Conclusions ........................................ 62
   8.1 Acknowledgements ................................. 62
Chapter 1

Introduction

Next-Generation sequencing (NGS) machines are already capable of producing over a petabyte of data per year, but the rate of growth in amount of data they can sequence is growing faster than Moore’s law. In 2014, Illumina introduced the HiSeq X Ten that can produce up to 1.4 petabytes (PB) of raw genomic data per year. The typical size of the raw data for a whole human genome (fastq files) with 30x coverage is 100 GB. However, even higher coverage (60x) is required if indel mutations are to be reliably identified using current technology. In addition to this, the data stored must be stored in redundant form to ensure that it can be reliably accessed when it is needed. For example, the 1.4 PB per year from an HiSeq X 10 would be stored as 5.2 PB in a cluster running the Apache Hadoop Distributed File system (HDFS). Many large-scale sequencing projects are starting or going to start in the next few years in Europe. In the UK, the Hundred Thousand Genomes project intends to sequence the whole genomes of 100,000 cancer patients, while in Sweden the LifeGene project intends to sequence the whole genomes of several hundred thousand individuals. The scale of the storage requirements for these projects is on the order of hundreds of petabytes. Currently, Apache’s HDFS does not scale to store that amount of data, due to a bottleneck in the metadata storage component (the NameNode). However, the option of partitioning the NGS data across multiple clusters is not an attractive option, as the Hadoop platform does not support the execution of analysis jobs over multiple clusters. With larger clusters, we can support analysis over larger sample sizes, and, thus, enable greater statistical power in making inferences on the NGS data available.

This document describes how the different deliverables D2.3 (Scalable Storage), D3.5 (Object Model) and D4.3 (CharonFS) come together to provide a Secure, scalable, highly-available Filesystem integrated with the Object Model and Overbank. Chapter 2 identifies the limitations of the Hadoop FileSystem. In chapter 3 we discuss the design, implementation, and evaluation of our own version of the Hadoop Distributed File, Hops-FS, where we externalized the metadata of HDFS to an open-source, in-memory, highly-available, distributed database called MySQL Cluster. In chapter 4 we describe how the file system integrates the Object Model (D3.5) and its data security. In chapter 5 we describes how CharonFS uses Hops-FS as underlying storage system. In chapter 6 we describe our new contribution Hops-YARN increases the scalability of Apache YARN by externalizing the ResourceManager state from the heap of a JVM to MySQL Cluster.
Chapter 2

Hadoop Distributed File System

In 2013, HDFS v2 introduced a highly available metadata architecture [12, 28]. The entire file system’s metadata is still stored in memory on a single node [26], but changes to the metadata edit log entries are now replicated and persisted to a set of (at least three) Journal Nodes using a quorum-based replication algorithm. The log entries in the Journal nodes are applied to a standby NameNode that will take over as primary NameNode when the active NameNode fails and all outstanding log entries in the Journal Nodes have been applied. Figure 2.1 shows the highly available HDFS architecture, with the eventually consistent replication of the NameNode state from the Active NameNode to the Standby NameNode. As the replication protocol is eventually consistent, when the Active NameNode fails, it may take tens of seconds for the Standby NameNode to take over as Active NameNode. Also, as the standby NameNode is not used to satisfy file system operations unless the Active NameNode fails, the NameNode is still a bottleneck. Moreover, the NameNode is slower in this HA (high availability) configuration for write operations, as it must now persist metadata updates to a quorum of Journal Nodes before returning to the client. As a result, this new version improves fault tolerance, but decreases the throughput of HDFS for write operations.

Moreover, the new HA NameNode architecture requires additional services and nodes in the cluster. It needs at least three Zookeeper instances on different machines to allow all nodes in the cluster to reliably reach an agreement on which NameNode is currently active and which is passive. In large clusters, there is limited free memory on the heap of the NameNode(s), so it is not possible to efficiently create a snapshot of the NameNode’s state. As such, an additional Checkpoint server is needed to periodically store a checkpoint of the NameNode’s state to disk. Note that this server should have at least as much RAM as the NameNode.

2.0.1 NameNode limitations

The Hadoop platform has received tremendous attention from the research community in recent years. This is due to the fact that it is the most advanced and complete open-source ecosystem for big data analytics. However, the Hadoop Distributed File System (HDFS) [27] has received relatively less attention from the research community and suffers from two main problems. First, the core design of HDFS NameNode has not changed since the project was first started: a
D2.4 – Secure, scalable, highly-available Filesystem integrated with the Object Model and Overbank

Figure 2.1: HDFS Architecture

A single NameNode is responsible for the maintenance of the file system metadata. Secondly, the reliability of the storage is only ensured by replication which is very storage inefficient and, in fact, has inferior high availability properties compared to erasure-coding replication in the case of multiple concurrent DataNode failures.

In order to reduce the software complexity of the NameNode and ensure its correctness and performance, the entire file system metadata is stored in the main memory of the NameNode. This makes it very simple to implement the file system operations, but this is not scalable and become a limitation for the platform. In fact, Hadoop clusters can grow very large and a single NameNode will have to serve thousands of DataNodes and hundreds of thousands of clients simultaneously while storing tens of gigabytes of metadata in RAM.

The problem with storing the namespace metadata in memory is two-fold. First, the namespace metadata cannot grow more than the main memory of the NameNode. Secondly, when the in-memory metadata grows to tens of gigabytes the application performance degrades due to JVM stop-the-world garbage collection events. Java garbage collectors are infamous for their poor performance for large heap sizes: the garbage collection thread consumes a significant amount of CPU resources and can pause applications for seconds at a time [26].

Moreover the namespace metadata is kept strongly consistent using readers-writer concurrency semantics, implemented using a global namespace lock (FSNamesystem lock), see figure 2.2. All the write operations are serialized, even if these operations intend to mutate different files in the different sub-trees of the namespace hierarchy. A write operation blocks all other operations;
and a large write operation can potentially block the namespace for a very long time. Some operators have reported that due to sudden influx of write-operations and JVM stop-the-world garbage collection, the NameNode can become unresponsive for tens of minutes [16]. Due to the coarse grained NameSystem lock the NameNode doesn’t scale well on multi-core hardware and the CPU is often underutilized for write intensive workloads [2, 3].

Storing all the namespace data on a single NameNode also makes the file system highly vulnerable. The file system becomes inaccessible when the NameNode fails. Hadoop V2 solved this problem with a hot swappable Standby NameNode that takes over as soon as it detects that the Active NameNode has failed. Yet, the time needed to propagate the logs from the main NameNode to the standby one is not negligible. From the time of detection of failure of the Active NameNode, the standby node can take tens of seconds to apply all the outstanding logs entries before serving read and write operations. Moreover, having a standby NameNode does not completely solve the single point of failure problem, the system can only tolerate a single NameNode failure. In case of failure of the Active NameNode, the standby NameNode becomes a single point of failure until the cluster administrators revive the failed Active NameNode. Although this problem can be resolved with even more redundant hardware (a standby for the standby), the costs involved and the fact the hardware does not contribute to the file system during normal operation, mean that operators typically go with just Active and Standby NameNodes.
Another problem with the Apache implementation of the NameNode is that it sometimes returns inconsistent results. Whenever the NameNode receives a write request it performs following steps:

1. acquires exclusive locks on the NameSystem,
2. performs permission checks and updates the namespace,
3. releases the Namespace lock,
4. saves the changes in the EditLog, and
5. finally, it notifies the client of the result of the operation.

As soon as the NameNode releases the namespace lock, the changes become visible to other clients and operations in the system. This can lead to inconsistent read operations. For example, consider a scenario where a client A creates a file \( F \). The NameNode makes the changes in the namespace and releases the namespace lock. The new file is now visible to the other clients in the system and a client B can read the file. If the NameNode fails before it persisted the creation of \( F \) in the Journal Nodes, the standby NameNode will take over without any information about the creation of the new file \( F \). If the client B tries to read the file \( F \) again will receive an error indicating that the file does not exist. Such inconsistencies arise because of the premature release of the namespace lock before mutations to the namespace have been persisted. To strengthen the NameNode’s consistency semantics, the namespace locks should only be released once the changes have been persisted in the EditLog. But persisting to the EditLog is slow (tens of milliseconds) and holding the namespace lock while the file system changes are persisted would massively decrease the throughput of the NameNode. The designers surmised that weaker-than-Posix semantics are tolerable for users, given the increased performance it brings.

### 2.0.2 Storage limitations

HDFS is designed to be deployed on commodity hardware and achieves high data availability by replicating the data block, typically three times producing a storage overhead of 200%. However, even a replication factor of three is not high enough for huge clusters and such clusters will lose data with a high probability in case of correlated failures [9]. It has recently been shown that higher data availability can be achieved using Reed Solomon (RS) Erasure coding [18, 23]. Moreover, RS erasure coding imply a smaller overhead on the system (40%) which saves valuable disk space.

### 2.0.3 HDFS improvements

The two limitations presented above need to be solved. There is great need for a file system metadata management that is highly available, scales horizontally, and supports multiple
concurrent write mutations. And HDFS storage reliability should not rely only on block replication. In this document we present our distribution of HDFS called Hops-FS. Hops-FS achieves horizontal scalability of metadata management by making the NameNode stateless. All the metadata is stored in a highly available, highly performant distributed database. This allows Hops-FS to have multiple stateless NameNodes which can process operations in parallel. The cluster administrator can increase or decrease the number of NameNodes depending on the workload and SLA requirements. All the clients in the system uniformly distribute their file system operations among all the NameNodes.

Hops-FS supports fine-grained *readers-writers* concurrency locking while strengthening the consistency semantics of HDFS. Unlike Apache HDFS, that locks the entire namespace to keep the namespace consistent, Hops-FS takes locks on the individual files or directories affected by each file system operation. This allows us to perform multiple file system read and write operations in parallel. Fine-grained locking and ACID transactions ensure that multiple namespace mutations do not violate the consistency of the file system. Additionally, Hops-FS supports RS Erasure coding to reduce the storage footprint and increase availability properties.
Chapter 3

Hops-FS

Hops-FS replaces the Active-Standby NameNode architecture with a set of stateless NameNodes that access a distributed, in-memory, replicated database: MySQL Cluster [6]. MySQL Cluster is a real-time, ACID-compliant transactional, relational database, with no single point of failure. In scalability tests with 30 nodes, MySQL Cluster has performed more than a billion transactional updates of 100 bytes in size every minute [5]. Externalizing the metadata to a database makes the NameNodes stateless and disconnects the metadata scalability from the size of the memory heap of a single JVM. Moreover, it allows Hops-FS to run multiple NameNodes in parallel, which increases the number of concurrent metadata operations and delivers very high throughput. This chapter gives a bird’s eye view of the Hops-FS architecture. A very detailed description about the internal workings of Hops-FS is included in the deliverable 2.3 titled “Scalable and Highly Available HDFS”.

3.0.1 Hops-FS Architecture

All the changes in Hops-FS preserve the semantics of Apache-HDFS so that existing applications can run on Hops-FS without any change. Although the semantics of Hops-FS at least as strong as HDFS, its metadata management is fundamentally different. Fig. 3.1 shows the architecture of Hops-FS. Unlike HDFS (see figure 2.1), Hops-FS has multiple NameNodes that manage the namespace metadata. All Hops-FS’ clients and DataNodes are aware of all NameNodes in the system. Whenever a NameNode fails the failed operations are automatically retried by clients and the DataNodes by forwarding the failed requests to a different live NameNode. In the remainder of this section we will present different architectural components of Hops-FS, starting from the metadata storage layer, followed by the NameNodes, the DataNodes and the clients.

3.1 Distributed metadata

All the NameNodes in Hops-FS are stateless and the namespace metadata is stored in MySQL Cluster® database. We have chosen MySQL Cluster for its high performance, high availability,
real-time performance, online scale-out and high throughput [6]. MySQL Cluster provides both SQL and native NoSQL APIs for data access and manipulation. Hops-FS mostly use the NoSQL APIs because it allows us to reach a high throughput by bypassing the MySQL Server and directly communicate with the storage layer. It also allows Hops-FS to control how the data is distributed among the MySQL Cluster DataNodes. However, Hops-FS uses the SQL APIs for operations on aggregate data (for example, calculating the total number of files or corrupt blocks in the system) as the MySQL Server collects statistics from DataNodes that enable it to perform these operations efficiently.

Making the NameNode stateless turned out to be a big challenge as the NameNode contains diverse and highly optimized data structures. In order to have multiple completely independent stateless NameNodes, we needed to persist many data-structures, including inodes, blocks, blocks-locations and files-leases. Along with these data structures a lot of the secondary data structures also have to be persisted. Some examples are different replication queues; invalidated, corrupted, and excess replicas list; the lease manager state, and global variables. Figure 3.2 shows the key entities in the Hops-FS metadata model. Files and directories are represented by the Inode entity. Each Inode contains a reference to its parent directory; which is used to construct a file system hierarchy. Each file contains multiple blocks whose information is stored in the Block entity. Each block is usually replicated multiple times. The location of each replica of the block is stored in a Replica entity. During its life-cycle a block goes through many phases. For example, when a DataNode fails some blocks will become under-replicated, such blocks are stored in the under-replicated blocks list (URB). The replication manager, periodically, selects some under-replicated blocks and tries to create more replicas for them. Blocks with ongoing replication are removed from the URB list and stored in the pending replication blocks list (PRB). Similarly, a replica of a block might have many states during its life-cycle. For example, if it gets corrupted will be moved to the corrupted replicas (CR) list. Whenever a client writes to a new block’s replica, this replica is moved to the replica under construction (RUC) list. Replicas that are in excess are stored in the excess replicas (ER) list and replicas that are scheduled for deletion are stored in the invalidation (Inv) list.
3.1.1 Metadata partitioning

MySQL Cluster supports distributed transaction processing, where a Transaction Coordinator (TC) is located at every DataNode in the cluster. Database transaction can be processed by any of the TCs in the cluster. When processing a transaction, a TC may communicate with TCs located on other DataNodes in order to retrieve data required for the transaction. As contacting other nodes implies costly network communications, MySQL Cluster is more efficient if all the data required by a transaction is located on the same host as the TC. We call transactions that can retrieve all their data locally network-aware transactions. Network-aware transactions require database tables to be partitioned by a user-defined partition key. If many different tables are all partitioned by the same partition key, then rows of all the tables with the same partition key value will all reside on the same DataNode. This means that if we have a transaction that will only access tables partitioned using the same partition key, all of the data will reside on the same DataNodes. If we are able to make sure that such network-aware transactions can on one of the DataNodes containing the transaction’s data, then the transaction will be able to read all of its data locally, and any updates will only affect other DataNodes in the same replica group. In MySQL Cluster, it is possible to control where a transaction is started by specifying a partition key value when starting a transaction. The primary key column is the default partition key, but it is also possible to specify secondary indexes as partition keys. In the case of Hops-FS, an efficient partitioning of the file system metadata is to have all the data related to a same file (inode) placed on the same DataNode. For this purpose, we partition all the tables containing file information by the Inode_id column and start all transactions on the DataNode storing the file data. If the DataNode crashes, another DataNode in the same replica group will take over as responsible for the partition key, and future transactions will be started on the new DataNode.

3.2 Fine grained namespace Locks

All operations in HDFS are path-based, and in an given file system operation only a subset of the path components are mutated. Most commonly, only the last component of the path is mutated. For example, the file operations “rm /dir1/dir2/dir3/dir4/file.txt”, “touchz /dir1/di2/file.txt” and ”mv /dir1/dir2 /.recycle/“ only mutate the last component in the path. HDFS NameNode uses a single, global namespace lock, that supports only two locking modes: shared and exclusive locks, to serialize all file system operations. This greatly reduces the number of concurrent filesystem mutations, see figure 2.2

Hops-FS uses fine grained namespace locks that increases the parallelism of file operations by locking only the files and directories that are being mutated by the file system operations. Figure
Figure 3.3: Hops-FS Architecture

3.3 show the internal working on Hops-FS namenode remote procedure call (RPC) mechanism. Using the new locking mechanism multiple operation (RPC handler threads) can run in parallel. A very detailed description about the new locking mechanism is included in the deliverable 2.3 titled "Scalable and Highly Available HDFS“.

### 3.3 Multiple NameNodes

Hops-FS supports multiple NameNodes. All NameNodes in Hops-FS are stateless and primarily provide an interface for reading and writing the namespace stored in the database (they also provide management, failover, and access control services). In Hops-FS, NameNodes can be easily added and removed depending on the load on the system. All client-facing file system operations can be performed by any of the NameNodes in the system. However, the execution of some management (housekeeping) operations must be coordinated among NameNodes. If these management operations are executed simultaneously on multiple NameNodes, they can potentially compromise the integrity of the namespace. For example, the replication manager service makes sure that all the blocks in the system have the required number of replicas. If any block becomes over-replicated, then the replication manager tries to remove replicas of the block until it reaches the required level of replication. If, however, we had several replication managers running in parallel without coordination, they might take conflicting/duplicate deci-
sions and a block might end up being completely removed from the system. Another operation that can affect the consistency of metadata is the lease recovery operation. Only one NameNode should try to recover a file that was not properly closed by the client. We manage the coordination of management operations between NameNodes by electing a Leader NameNode that is solely responsible for management operations.

3.4 Hops-FS leader election and group membership

Apache HDFS uses ZooKeeper [15] as a coordination service. In contrast, Hops-FS uses the database for membership management and leader election, thus reducing the number of services that need to be managed and configured by our system. We implemented both leader election and group membership services using the database as strongly consistent distributed shared memory. All NameNodes periodically write to a shared table containing information about the NameNode (the NameNode's descriptor). NameNode need to successfully update their descriptor to show that they are alive in the system. While updating its descriptor, a NameNode also reads all the descriptors for other NameNodes from the database to maintain a local history of the NameNode descriptors. Using the local history, the NameNodes can easily discover dead NameNodes: the nodes that fail to update their descriptor within a configurable timeout period. A NameNode is declared leader when it has the smallest Id among all the remaining alive NameNodes. The NameNode Id is a monotonically increasing number that is persisted in the database. Whenever a new NameNode joins the system, it gets its a new id by incrementing a global id counter stored in the database. Nodes that are too slow to update their descriptor (and are hence ejected from the system) rejoin when they find out that their history is too old. The leader NameNode is also responsible for removing dead descriptors from the database. Our recent publication [22] contains in depth working of the leader election protocol and its evaluation.

3.5 DataNodes

The DataNodes are connected to all the NameNodes in the system. Whenever a block is modified, the DataNode notifies a random NameNode. Every hour each DataNode sends a block report to the NameNodes. These block reports can be very large: they contain information about all the blocks stored on that DataNode. A DataNode with 12TB of storage capacity may contain up to hundred thousand 128MB blocks. This means that in a cluster of 10,000 DataNodes, an average of 3 block reports will be generated every second. Processing such large number of block reports is not possible with the single NameNode of Apache-HDFS. In Hops-FS the DataNodes uniformly distributes the block reports among all the NameNodes in the system.

Each DataNode regularly sends a heartbeat message to all the NameNodes to notify them that it is still alive. The heartbeat also contain information such as the DataNode capacity, its available space, and its number of active connections and updates the list of DataNode descriptors at the NameNode. The DataNode descriptor list is used by NameNodes for future block allocations, and due to the frequency of updates to it, we decided not to persist it in the database. It is
re-built on system restart using heartbeats from DataNodes. In the future, we intend to reduce the amount of heartbeat traffic by having DataNodes only send a single heartbeat message to a single NameNode.

3.6 Clients

The clients can send file system operations to any NameNode in the system. We support the following policies: fixed, round robin, and random. We can use these policies to load-balance client requests among all the NameNodes. If a file system operation fails because of a NameNode crash, the client removes the NameNode descriptor from its local list of NameNodes and retries the operation on a different NameNode. Clients periodically refresh their local list of alive NameNodes by querying a random NameNode. The refresh rate is configurable.

3.7 Erasure Coding

Apache HDFS stores 3 copies of your data to provide high-availability. So 1 petabyte of data actually requires 3 petabytes of storage. For many organizations, this results in onerous storage costs.

Hops-FS also supports erasure-coding to reduce the storage required by by 44% compared to HDFS, while still providing high-availability for your data.

Currently, the only alternative for HDFS is HDFS-RAID, which was developed by Facebook for an old version of HDFS, V0.19, and is no longer supported in the Apache HDFS distribution. In comparison to HDFS-RAID, our erasure-coding implementation, Hops-FS, is integrated with the NameNode, removing the need to periodically scan directories for broken files and reducing time to discover and repair from failures.

3.8 Metadata security

For performance reasons, MySQL Cluster does not support encrypted network channels between nodes in the cluster (clients, DataNodes, and Management nodes). Moreover, the data stored on the MySQL Cluster DataNodes is not encrypted - again for performance reasons. In a typical secure deployment, MySQL cluster is run on an isolated network, where all access is via applications that are also MySQL Cluster clients. The applications should support strong authentication and access control. Therefore, we recommend restricting physical access to MySQL Cluster using firewalls. Internally, MySQL Cluster nodes can use software firewalls to filter incoming network traffic that does not originate at known IP addresses/ports (the addresses of the other nodes in the cluster).

Hops-FS uses MySQL Server to perform aggregate queries on the metadata. Each NameNode
has a local instance of MySQL server running on the machine. Securing MySQL server is straightforward. The MySQL server instances are configured to only accept connections from the localhost and from other NameNodes in the system.

### 3.9 Combining Apache v2 and GPL v2 Open Source Licenses

We combine Apache and GPL licensed code, from Hops and MySQL Cluster, respectively, by providing a DAL API (similar to JDBC). We dynamically link our DAL implementation for MySQL Cluster with the Hops code. Both binaries are distributed separately.

Hops derives from Hadoop and, as such, it is available under the Apache version 2.0 open-source licensing model. MySQL Cluster and its client connectors, on the other hand, are licensed under the GPL version 2.0 licensing model. Similar to the JDBC model, we have introduced a Data Access Layer (DAL) API to bridge our code licensed under the Apache model with the MySQL Cluster connector libraries, licensed under the GPL v2.0 model. The DAL API is licensed under the Apache v2.0 model. The DAL API is statically linked to both Hops and our client library for MySQL Cluster that implements the DAL API. Our client library that implements the DAL API for MySQL Cluster, however, is licensed under the GPL v2.0 model, but static linking of Apache v2 code to GPL V2 code is allowed, as stated in the MySQL FOSS license exception. The FOSS License Exception permits use of the GPL-licensed MySQL Client Libraries with software applications licensed under certain other FOSS licenses without causing the entire derivative work to be subject to the GPL. However, to comply with the terms of both licensing models, the DAL API needs to be generic and different implementations of it for different databases are possible. Although, we only currently support MySQL Cluster, you are free to develop your own DAL API client and run Hops on a different database, see figure 3.4. The main requirements for the database are support for transactions, read/write locks and at least read-committed isolation.

Figure 3.4: Combining Apache v2 and GPL v2 Open Source Licenses
3.10 Removed and Deprecated Features

Hops-FS fundamentally changes the design of metadata storage for HDFS. Due to these changes some of the functionality of HDFS is no longer needed. In this chapter we specify the architectural components of HDFS that have been removed from Hops-FS. The only functionality that we have removed from Apache HDFS is the support for federated namespaces. The need for federated namespaces arose from the scalability limitations of HDFS (more specifically, the NameNode). However, the federated support as it stands, is extremely limited - there is no consistent global view over all federated namespaces, and there is no support for executing computations (such as YARN applications) over all clusters in the federation. There are also no guarantees provided for cross namespace operations, such as moving files. As such, and because Hops-FS has mitigated the scalability, we decided to remove support for federation. At this moment in time, we see little benefit and only extra complexity in HDFS federation.

- **Secondary NameNode**
  The secondary NameNode is no longer supported. Hops-FS supports multiple NameNodes and all the NameNodes are active.

- **EditLog**
  The write ahead log (EditLog) is not needed as all the metadata mutations are stored in the highly available data store.

- **FSImage**
  We don’t need to store checkpoints of the metadata (FSImage) as NameNodes in Hops-FS are stateless and metadata is stored in the external metadata store.

- **Quorum Based Journaling**
  Replaced by the external metadata store.

- **NameNode Federation**
  NameNode federations are no longer supported, due to reasons outlined above.

- **Viewfs**
  Viewfs is used by federated HDFS to view a namespace that contains multiple federated NameNodes.

- **ZooKeeper**
  HDFS uses ZooKeeper for coordination services. Hops-FS has replaced ZooKeeper with a coordination service built on distributed shared memory, see section 3.4 for more details.

There are a number of features of HDFS that have been introduced in HDFS v2 since 2014 that we do not support, due to us starting our work from the 2.0.4-alpha branch, including:

- **NFS**
  Given that HDFS is not POSIX compliant, this feature is currently low priority.

- **Snapshot**
  We are working on our own Snapshot algorithm based on the database.
• **Truncate**
  Truncate [HDFS-3107] was introduced in 2015 in HDFS 2.7. It enables files to be truncated to a specified length. This is useful for rollback operations.

• **Heterogeneous Storage**
  This feature is useful for clusters containing mixed storage technologies, such as spinning disk and faster SSDs. When a user creates a file, she can specify the desired storage type. See HDFS-2832 and HDFS-5682.

• **Encryption at Rest**
  From HDFS 2.6, HDFS files can be encrypted on disk using cryptographic keys stored in a KeyServer.

### 3.11 Hops-FS User Manual

Hops-FS is based on Apache Hadoop distribution. We have tried to make sure that Hops-FS does not break any semantics of HDFS, so that all the existing application and systems using HDFS can easily migrate to Hops-FS. Hops-FS supports most of the configuration parameters defined for HDFS [http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml]. In this chapter we will only point out the changes in the configuration parameters related to Hops-FS.

### 3.12 Unsupported parameters and commands

We have replaced HDFS 2.x’s Primary-Secondary Replication model with shared atomic transactional memory. This means that we no longer use the parameters in HDFS that are based on the (eventually consistent) replication of edit log entries from the Primary NameNode to the Secondary NameNode using a set of quorum-based replication servers.

#### 3.12.1 Configuration Parameters

Here are the parameters that are not used in the Hops-FS.

- **dfs.namenode.secondary.**: None of the secondary NameNode attributes are used.
- **dfs.namenode.checkpoint.**: None of the checkpoint attributes are used.
- **dfs.image.**: None of the FSImage attributes are used.
- **dfs.journalnode.**: None of the hadoop’s journaling attributes are used.
- **dfs.ha.**: None of the hadoop high availability attributes are used.
D2.4 – Secure, scalable, highly-available Filesystem integrated with the Object Model and Overbank

- **dfs.nameservices.***: None of the hadoop federation attributes are used.
- **dfs.namenode.num.extra.edits.***: None of the edit logs attributes are used.
- **dfs.namenode.name.dir.***: FSImage is not supported anymore.
- **dfs.namenode.edits.***: None of the edit log attributes are used.

### 3.12.2 Administrator Commands

Here are the commands that are no longer supported by the Hops-FS.

- **hdfs haadmin ***: None of high availability admin commands are supported since we don’t support Apache Hadoop high availability solution.
- **hdfs dfadmin -metaSave**
- **hdfs dfsadmin -saveNamespace**: The Namespace is already stored in a database, a dump could be created to save it to file instead.
- **hdfs dfadmin -restoreFailedStorage**
- **hdfs dfadmin -rollEdits**
- **hdfs dfadmin -fetchImage**

And here’s a list of command that could be supported by Hops-FS in the future.

- **hdfs dfsadmin rollingUpgrade ***: We don’t support rolling upgrade as of now, but we will added it later, since it needs to be adapted to our model.
- **hdfs dfadmin -allowSnapshot**
- **hdfs dfadmin -disallowSnapshot**

### 3.13 Hops-FS configuration parameters

Hops-FS supports multiple stateless NameNodes. All of the NameNodes can mutate the namespace. Hops-FS’ multi NameNode environment introduces a few new configuration parameters which are defined in the following subsection.
3.13.1 NameNode configuration parameters

- **dfs.quota.enabled**
  Quota can be en/disabled. By default quota is enabled.

- **dfs.leader.check.interval**
  The length of the period in seconds on which NameNodes run the leader election protocol. One of the active NameNodes is chosen as a leader to perform housekeeping operations. All NameNodes periodically update a counter in the database to mark that they are active. All NameNodes also periodically check for changes in the membership of the NameNodes. By default the period is to one second. Increasing the time interval would lead to slow failure detection.

- **dfs.leader.missed hb**
  This property specifies when a NameNode is declared dead. By default a NameNode is declared dead if it misses a HeartBeat. Higher values of this property would lead to slower failure detection.

- **dfs.block.pool.id**
  Due to shared state among the NameNodes, Hops-FS only support one block pool. Set this property to set a custom value for block pool. Default block pool id is HOP_BLOCK_POOL_123.

- **dfs.name.space.id**
  Due to shared state among NameNodes, Hops-FS only support one name space. Set this property to set a custom value for name space. Default name space id is 911 :)

- **dfs.ndb.setpartitionkey.enabled**
  Partition hints can be used to start transactions on a specific MySQL datanodes. If this parameters is set to false then the transactions will start on random MySQL Cluster datanodes. For performance reasons it is better to start the transactions on the datanodes that hold the data for the transaction.

Transactions Statistics

- **dfs.transaction.stats.enabled**
  Each NameNode collect statistics about currently running transactions. The statistics will be written in a comma separated file format, that could be parsed afterwards to get an aggregated view over all or specific transactions. By default transaction stats is disabled.

Listing 3.1: An example of the statistics file produced

```
Tx,PrimaryKey_hits,PrimaryKey_hitsRows,PrimaryKey_misses,PrimaryKey_missesRows,PrunedIndexScan_hits,PrunedIndexScan_hitsRows,PrunedIndexScan_misses,PrunedIndexScan_missesRows,IndexScan_hits,IndexScan_hitsRows,IndexScan_misses,IndexScan_missesRows,Batched_hits,Batched_hitsRows,Batched_misses,Batched_missesRows,BatchedPrunedIndexScan_hits,BatchedPrunedIndexScan_hitsRows,BatchedPrunedIndexScan_misses,BatchedPrunedIndexScan_missesRows,FullTableScan_hits,FullTableScan_hitsRows,FullTableScan_misses,FullTableScan_missesRows,FullTable_hits,FullTable_hitsRows,FullTable_misses,FullTable_missesRows,Hits,HitsRows,Misses,MissesRows,New,Modified,Deleted,Acquire,Processing,Commit,TotalTime
GET_STORAGE_INFO,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,28,0,20,48
LEADER_ELECTION,5,4,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,2,1,1,1,0,12,0,3,15
LEADER_ELECTION,4,4,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,5,2,2,0,1,0,2,3,4,9
```
If ResolvingCache is enabled, The NameNode will write another statistics file for resolving cache operations.

### Listing 3.2: An example of the resolving cache statistics file produced

<table>
<thead>
<tr>
<th>Operation, Elapsed, RoundTrips</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET, 10, 4</td>
</tr>
<tr>
<td>GET, 5, 5</td>
</tr>
<tr>
<td>GET, 5, 5</td>
</tr>
<tr>
<td>GET, 4, 5</td>
</tr>
<tr>
<td>GET, 5, 5</td>
</tr>
<tr>
<td>GET, 5, 5</td>
</tr>
</tbody>
</table>

- **dfs.transaction.stats.detailed.enabled**
  If enabled, The NameNode will write a more detailed and human readable version of the statistics. By default detailed transaction stats is disabled.

### Listing 3.3: An example of the detailed statistics file produced

Transaction: GET_STORAGE_INFO

---

VariableContext

- StorageInfo[PK] H=1 M=1
- N=0 M=0 R=0
- Hits=1(1) Misses=1(1)
- Detailed Misses: PK 1(1)

---

- Tx. N=0 M=0 R=0
- Tx. Hits=1(1) Misses=1(1)
- Tx. Detailed Misses: PK 1(1)

Transaction: LEADER_ELECTION

---

VariableContext

- HdfsLeParams[PK] H=4 M=1
- N=0 M=1 R=0
- Hits=4(4) Misses=1(1)
- Detailed Misses: PK 1(1)

---

HdfsLESnapshot

- All[FT] H=0 M=1
- ById[PK] H=1 M=0
- N=1 M=0 R=0
- Hits=1(0) Misses=1(0)
- Detailed Misses: FT 1(0)

---

- Tx. N=1 M=1 R=0
- Tx. Hits=5(4) Misses=2(1)
- Tx. Detailed Misses: PK 1(1) FT 1(0)

- **dfs.transaction.stats.dir**
  The directory where the stats are going to be written. Default directory is `/tmp/hopsstats`.

- **dfs.transaction.stats.writeround**
  How frequent the NameNode will write collected statistics to disk. Time is in seconds. Default is **120 seconds**.

**Note:** All the times written in the statistics files are in milliseconds.
Resolving Cache configuration

Each NameNode caches the path metadata (inode ids) in a resolving cache for later use. Note this is not same as transaction cache used in the HopsFS NameNode Dal Implementation layer. Resolving Cache entries have longer life than the individual transaction caches.

We support different implementations for the resolving cache; INodeMemcache, PathMemcache, OptimalMemcache and InMemory.

1. **INodeMemcache**: for each path component “INode” we associate a key \((parentId, Name)\) with a value \(INodeId\).

2. **PathMemcache**: for each path, we associate a key \(md5(path)\) with list of \(INodeIds\).

3. **OptimalMemcache**: sits in a middle ground between INodeMemcache and PathMemcache. We divide the path into parentPath and file then we associate \(md5(parentPath)\) with list of \(INodeIds\) till parent, and associate \((fileparentId, fileName)\) with file\(INodeId\).

4. **InMemory**: The same as INodeMemcache, but instead of using Memcache, we use a ConcurrentLinkedHashMap with LRU.

Resolving cache common configuration:

- **dfs.resolvingcache.enabled**
  Enables/Disables the resolving cache for the NameNode.

- **dfs.resolvingcache.type**
  Resolving cache type, could be INode, Path, Optimal, InMemory. Default is InMemory.

Memcache specific configuration:

- **dfs.resolvingcache.memcached.server.address**
  Memcached server address.

- **dfs.resolvingcache.memcached.connectionpool.size**
  Number of connections to the memcached server.

- **dfs.resolvingcache.memcached.key.expiry**
  It determines when the memcached entries expire. The default value is 0, that is, the entries never expire. Whenever the NameNode encounters an entry that is no longer valid, it updates it.

InMemory cache specific configuration:

- **dfs.resolvingcache.inmemory.maxsize**
  Max number of entries that could be in the cache before the LRU algorithm kick in.
Quota manager configuration parameters

- `dfs.namenode.quota.update.interval`
  In order to boost the performance and increase the parallelism of metadata operations the quota updates are applied asynchronously. The quota update manager applies the outstanding quota updates after every `dfs.namenode.quota.update.interval` milliseconds.

- `dfs.namenode.quota.update.limit`
  The maximum number of outstanding quota updates that are applied in each round.

Distributed unique ID generator configuration

ClusterJ APIs do not support any means to auto generate primary keys. Unique key generation is left to the application. Each NameNode has an ID generation daemon. ID generator keeps pools of pre-allocated IDs. The ID generation daemon keeps track of IDs for inodes, blocks and quota entities.

- **Batch Sizes**
  When the ID generator is about to run out of the IDs it pre-fetches a batch of new IDs. The batch size is specified by these parameters.

  - `dfs.namenode.quota.update.id.batchsize` Prefetch batch size for Quota Updates. As there are lot of quota updates in the system the default value is set to 100,000.
  - `dfs.namenode.inodeid.batchsize` Prefetch batch size for inode IDs.
  - `dfs.namenode.blockid.batchsize` Prefetch batch size for block IDs.

- **Update Threshold**
  These parameters define when the ID generator should pre-fetch new batch of IDs. Values for these parameter are defined as percentages i.e. 0.5 means prefetch new batch of IDs if 50% of the IDs have been consumed by the NameNode.

  - `dfs.namenode.quota.update.updateThreshold` Threshold value for quota IDs.
  - `dfs.namenode.inodeid.updateThreshold` Threshold value for inode IDs.
  - `dfs.namenode.blockid.updateThreshold` Threshold value for block IDs.

- `dfs.namenode.id.updateThreshold`
  It defines how often the IDs Monitor should check if the ID pools are running low on pre-allocated IDs.

3.13.2 Client configuration parameters

- `dfs.namenodes.rpc.addresses`
  HOP support multiple active NameNodes. A client can send a RPC request to any of the active NameNodes. This parameter specifies a list of active NameNodes in the system. The list has following format [hdfs://ip:port, hdfs://ip:port, ]. It is not necessary that this
list contain all the active NameNodes in the system. Single valid reference to an active NameNode is sufficient. At the time of startup the client will obtain the updated list of all the NameNodes in the system from the given NameNode. If this list is empty then the client will connect to ‘fs.default.name’.

- **dfs.namenode.selector-policy**
The clients uniformly distribute the RPC calls among the all the NameNodes in the system based on the following policies. See section 3.6 for more details.
  
  - ROUND_ROBIN
  - RANDOM

By default NameNode selection policy is set of ROUND_ROBIN

- **dfs.clinet.max.retries.on.failure**
The client will retry the RPC call if the RPC fails due to the failure of the NameNode. This property specifies how many times the client would retry the RPC before throwing an exception. This property is directly related to number of expected simultaneous failures of NameNodes. Set this value to 1 in case of low failure rates such as one dead NameNode at any given time. It is recommended that this property must be set to value \(\delta=1\).

- **dfs.client.max.random.wait.on.retry**
A RPC can fail because of many factors such as NameNode failure, network congestion etc. Changes in the membership of NameNodes can lead to contention on the remaining NameNodes. In order to avoid contention on the remaining NameNodes in the system the client would randomly wait between \([0,MAX\_VALUE]\) ms before retrying the RPC. This property specifies MAX\_VALUE; by default it is set to 1000 ms.

- **dfs.client.refresh.namenode.list**
All clients periodically refresh their view of active NameNodes in the system. By default after every minute the client checks for changes in the membership of the NameNodes. Higher values can be chosen for scenarios where the membership does not change frequently.

### 3.13.3 Data access layer configuration parameters

- **com.mysql.clusterj.connectstring**
  Address of management server of MySQL NDB Cluster.

- **com.mysql.clusterj.database**
  Name of the database that contains the metadata tables.

- **com.mysql.clusterj.connection.pool.size**
  This is the number of connections that are created in the ClusterJ connection pool. If it is set to 1 then all the sessions share the same connection; all requests for a SessionFactory with the same connect string and database will share a single SessionFactory. A setting of 0 disables pooling; each request for a SessionFactory will receive its own unique SessionFactory. We set the default value of this parameter to 3.
- **com.mysql.clusterj.max.transactions**
  Maximum number transactions that can be simultaneously executed using the clusterj client. The maximum support transactions are 1024.

- **io.hops.metadata.ndb.mysqlserver.host**
  Address of MySQL server. For higher performance we use MySQL Server to perform a aggregate queries on the file system metadata.

- **io.hops.metadata.ndb.mysqlserver.port**
  If not specified then default value of 3306 will be used.

- **io.hops.metadata.ndb.mysqlserver.username**
  A valid user name to access MySQL Server.

- **io.hops.metadata.ndb.mysqlserver.password**
  MySQL Server user password

- **io.hops.metadata.ndb.mysqlserver.connection_pool_size**
  Number of NDB connections used by the MySQL Server. The default is set to 10.

- **Session Pool**
  For performance reasons the data access layer maintains a pools of pre-allocated ClusterJ session objects. Following parameters are used to control the behavior the session pool.

  - **io.hops.session.pool.size**: Defines the size of the session pool. The pool should be at least as big as the number of active transactions in the system. Number of active transactions in the system can be calculated as \((\text{num_rpc_handler_threads} + \text{sub_tree_ops_threds_pool_size})\). The default value is set to 1000.

  - **io.hops.session.reuse.count**: Session is used \(N\) times and then it is garbage collected. The default value is set to 5000.
Chapter 4

Object Model and Data Security

Unlike traditional POSIX compliant filesystems Hops-FS lacks native support for file searching. For example, users may be interested in finding all data created by a certain user. Deliverable D3.5 discusses in detail how we have implemented support for searching metadata by automatically exporting metadata created from both the frontend UI and the NameNode in HDFS to Elasticsearch. The metadata can then be searched using free-text search on an Elasticsearch cluster. In this section we discuss Object Model and mechanisms through which the Hops-FS filesystem objects are exported to an external database for searching.

4.1 Metadata

As Hops directories are represented as rows in the Inodes table in the database, we can maintain the integrity of metadata by designing metadata tables that have a foreign key to the target directory (or file). Typically, removal of a directory or a file will also remove its associated metadata (foreign key on delete cascade). The rows in the metadata tables will be added to ElasticSearch using the JDBC river plugin.

4.1.1 HDFS Changes

An extra metadata attribute is enabled for DataSets. DataSets for which this flag is set are automatically indexed for ElasticSearch. An extra RPC has been added to set or unset the metadata attribute for an inode (and its subdirectories).

Security

Hops-FS supports permissions on an inode-basis. Hops-FS permissions follow the POSIX schema, where an inode has an owner and group associated with it. Each inode has three scopes of access, user, group and world access, each access scope is defined by three bits for
read, write and execution access.

In our Object Model, we introduce new notions; Study and DataSet and new roles; data provider and researcher. With these new additions, it became unrealistic to depend just on the current Hops-FS permissions without modifications to support a secure implementation. Moreover, implementing a new scheme might break the semantics of Hops-FS for other applications. That’s Why, we implemented a simple schema for creating/sharing DataSets and Studies based on the normal Hops-FS permissions without adding any more sophistications to Hops-FS.

Our proposed scheme is, for each LIMS user, we create a per-Study HDFS user identity for each Study the user is a member of. A LIMS user will always interact with HDFS in the capacity of his/her per-Study identity. For each interaction with HDFS, the LIMS intercepts the operation and determines the HDFS identity to use based on the logged in user and his/her active Study. The LIMS then passes the operation to HDFS as the newly determined user.

With each user having a per-Study HDFS user identity for each Study (s)he is a member of, each user has as many HDFS user identities as the number of Studies (s)he is a member of. Also, we define per-dataset groups as well. Finally, to enforce the data provider and researcher roles, we need to create a new role per Study. Our proposed access control scheme is discussed in more details in D3.5.

4.1.2 Auto-updated Metadata using HDFS and the Database

HDFS can auto-update metadata for files and directories by checking if the metadata flag is set. If the flag is set it inserts a row for every inode mutation or access in the tables, below. ElasticSearch will pull these operations using the river to update indexes. The initial state of a directory is identified and reported when enabling the auto-update.

**inodes_ops table**

Ordinary HDFS does not enable users to do advanced searches on its file metadata. Hops resolves this limitation by keeping track of file operations such as create, delete or rename and providing them to a search provider such as elasticsearch. To achieve this, all operations on auto-update enabled paths are logged in the table structure shown below. Note that renames are represented as a sequence of adds and deletes.

**Note:**

- A rename operation is expressed by a DELETE and a following ADD for the same inode id
- When reading values of this table, one needs to ensure to read all entries for a given dataset and inode id and to apply them in logical time order
D2.4 – Secure, scalable, highly-available Filesystem integrated with the Object Model and Overbank

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset_id</td>
<td>The inode id of the dataset folder.</td>
</tr>
<tr>
<td>inode_id</td>
<td>The inode id of the file or folder that was modified.</td>
</tr>
<tr>
<td>logic_time</td>
<td>Logic time providing happened-before ordering for events on the same dataset and inode ids.</td>
</tr>
<tr>
<td>operation</td>
<td>0 for ADD 1 for DELETE.</td>
</tr>
</tbody>
</table>

Table 4.1: INode Operations Table.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inode_id</td>
<td>The id of the inode that was accessed.</td>
</tr>
<tr>
<td>user_id</td>
<td>The id of the accessing user.</td>
</tr>
<tr>
<td>unix timestamp</td>
<td>The access time.</td>
</tr>
</tbody>
</table>

Table 4.2: Access Times Table

access_time table

Ordinary HDFS does keep track of access and modification times. However, it does not keep track of access times for specific users. In metadata auto-updates enabled, Hops keeps track of access times for individual users, thereby empowering users to discover and search advanced access patterns. This is made possible by logging every user access to a log table as shown below, which can then be consumed by a client application such as Elastisearch.

Note:

- Might contain multiple entries for the same inode id. Update only if greater.

file_size table

With standard HDFS, file sizes are not easily searchable. In order to make size quickly searchable by the user, Hops keeps track of size modifications in metadata auto-update enabled directory trees. The size information is stored in the table shown below and thereby available for a consuming application such as ElasticSearch.

Note:

- Might contain multiple entries for the same inode. We do currently not have operations

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inode_id</td>
<td>The id of the inode that was modified.</td>
</tr>
<tr>
<td>new_size</td>
<td>The new size.</td>
</tr>
</tbody>
</table>

Table 4.3: File Size Table
that make files smaller, so apply only if greater than the known value.

4.1.3 Metadata Designer Tool

The Metadata Designer is used to design MetaData templates. It supports the creation of new templates (from scratch or from existing templates). If an existing template has some DataSets that use it, it will not be editable - it has to be saved with a new name. The Tool is a Kanban Board (Trello) with Lists as groupings of metadata, and cards as metadata attributes.

MetaData Objects in JSON

Each file/directory can be associated with a JSON metadata object (metadata designer object):

```json
{

description: ....,

  groups : [

   { name :Personal Details,

       attrs : [

        {       label : DoB ,

               type : selection ,

               description: Year of Birth,


        },

        {                         label :First Name

               type : String ,

               description: Enter your Firstname,

               length: 100

        }

    ],

    { name :Payment Details Fileset,

        attrs : [

          {                           label :Credit Card,

                type : selection ,

                description: Select your type of Credit Card,

                options: { Visa , Mastercard }

          }

        ]

  }

}
```

The current metadata schema associated with a template
```json
"name": "MainBoard",
"numberOfColumns": 2,
"columns": [
  {
    "id": 1,
    "name": "SampleInfo",
    "cards": [
      {
        "id": 5,
        "title": "cardtitle",
        "find": false,
        "required": false,
        "description": "fielddescription",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      },
      {
        "id": 6,
        "title": "title",
        "find": false,
        "required": false,
        "description": "description",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  },
  {
    "id": 2,
    "name": "Info",
    "cards": [
      {
        "id": 9,
        "title": "d",
        "find": false,
        "required": false,
        "description": "d",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  }
],
"name": "MainBoard",
"numberOfColumns": 2,
"columns": [
  {
    "id": 1,
    "name": "SampleInfo",
    "cards": [
      {
        "id": 5,
        "title": "cardtitle",
        "find": false,
        "required": false,
        "description": "fielddescription",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      },
      {
        "id": 6,
        "title": "title",
        "find": false,
        "required": false,
        "description": "description",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  },
  {
    "id": 2,
    "name": "Info",
    "cards": [
      {
        "id": 9,
        "title": "d",
        "find": false,
        "required": false,
        "description": "d",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  }
],
"name": "MainBoard",
"numberOfColumns": 2,
"columns": [
  {
    "id": 1,
    "name": "SampleInfo",
    "cards": [
      {
        "id": 5,
        "title": "cardtitle",
        "find": false,
        "required": false,
        "description": "fielddescription",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      },
      {
        "id": 6,
        "title": "title",
        "find": false,
        "required": false,
        "description": "description",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  },
  {
    "id": 2,
    "name": "Info",
    "cards": [
      {
        "id": 9,
        "title": "d",
        "find": false,
        "required": false,
        "description": "d",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  }
],
"name": "MainBoard",
"numberOfColumns": 2,
"columns": [
  {
    "id": 1,
    "name": "SampleInfo",
    "cards": [
      {
        "id": 5,
        "title": "cardtitle",
        "find": false,
        "required": false,
        "description": "fielddescription",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      },
      {
        "id": 6,
        "title": "title",
        "find": false,
        "required": false,
        "description": "description",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  },
  {
    "id": 2,
    "name": "Info",
    "cards": [
      {
        "id": 9,
        "title": "d",
        "find": false,
        "required": false,
        "description": "d",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  }
],
"name": "MainBoard",
"numberOfColumns": 2,
"columns": [
  {
    "id": 1,
    "name": "SampleInfo",
    "cards": [
      {
        "id": 5,
        "title": "cardtitle",
        "find": false,
        "required": false,
        "description": "fielddescription",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      },
      {
        "id": 6,
        "title": "title",
        "find": false,
        "required": false,
        "description": "description",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  },
  {
    "id": 2,
    "name": "Info",
    "cards": [
      {
        "id": 9,
        "title": "d",
        "find": false,
        "required": false,
        "description": "d",
        "fieldtypeid": 2,
        "sizefield": {
          "showing": false,
          "value": "0"
        }
      }
    ]
  }
]
```

Download MetaData Template as JSON Object

Users should be able to download a MetaData template (Board) as a JSON object.

Upload MetaData Template as JSON Object

Users should be able to upload a MetaData template as a JSON object and it will automatically populate the Board and Lists in the MetaData designer tool.
Chapter 5

CHARON—Hops-FS integration

Chapter Authors:
Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Alysson Bessani (FFCUL).

5.1 Introduction

Previous chapters introduce and position Hops-FS as the BiobankCloud solution for effectively storing and processing data in a single datacenter. CHARON is a file system devised in the same project to store and share data from several biobanks using multiple cloud providers and storage repositories in a secure, reliable, and efficient way. Additionally, CHARON allows biobanks to extrapolate the capacity of their private datacenters to the virtually infinite capacity of public clouds without compromising the individuals’ privacy. Integrating Hops-FS and CHARON is vital for the BiobankCloud platform in providing a complete solution for storing, sharing, and processing data from one or more biobanks. The present chapter overviews the CHARON file system, discusses scenarios where the integration of Hops-FS and CHARON is more appropriate, and describes how to use them together.

5.2 The CHARON File System

CHARON is a cloud-backed file system capable of storing and sharing big data in a secure, reliable, and efficient way using multiple cloud providers and storage repositories. It is secure and reliable because it does not require trust on any single entity, and it supports the storage of different types of data in distinct locations to comply with required privacy premises. Two distinguishing features of CHARON are its serverless design (no client-managed server is required in the cloud) and its efficient management of large files (by employing prefetching, cache, and background writes). The complete description of CHARON architecture and its internal protocols is available in the Deliverable D4.2 of the BiobankCloud project [7].

Figure 5.1 illustrates a deployment scenario where two biobanks store their data in local repos-


Resilient Cloud-of-Clouds Storage

Public Cloud A

Public Cloud B

Biobank 1

Biobank 2

Figure 5.1: CHARON overview.

Itories, in single public cloud providers, and in a resilient cloud-of-clouds. In this scenario, the namespace tree has six nodes: directories d1 and d2, and files A, B, C, and D. The namespace is maintained in the cloud-of-clouds, together with file B. File D, less critical, is kept in a single cloud. File A is stored locally because it cannot leave Biobank 2. File C is shared between the two sites (e.g., in the same country), thus being stored in both of them.

The BiobankCloud platform will be able to process data available in the Hops-FS, which has a view of all data being stored in a single datacenter. CHARON will perform the inter-datacenter tasks and the processes between biobanks and public clouds. In the next section, we discuss the integration between these two storage systems and evaluate the most appropriate integration scenario for the BiobankCloud platform.

5.3 Integration Scenarios

The CHARON File System is able to store data in three different locations: a single public cloud, a cloud-of-clouds (composed of a set of different cloud storage providers), and a private repository. Furthermore, the private repository can be a disk in the machine running the CHARON system, a disk in a remote machine (e.g., a private data center), or any other distributed storage system. With these characteristics in mind, configuring the Hops-FS as a private repository of CHARON is a promising approach to integrate the two BiobankCloud systems. This approach allows LIMS’ tenants to share the data located in the Hops-FS, as well as give them the opportunity to store more sensitive/critical data in a secure and reliable cloud storage backend (e.g., a cloud-of-clouds instead of a single public cloud). However, integrating these two storage systems is not straightforward since some requirements must be satisfied first.

The following descriptions present different integration scenarios and explain their up and downsides.
Scenario 1—Hops-FS as a private repository in CHARON. In this first scenario, CHARON acknowledges and manages all data stored in Hops-FS. Users must interact with CHARON interfaces whenever they want to access or modify any file, as depicted in Figure 5.2. It means that the computation with Hops platform has to read and write data using the CHARON system.

CHARON stores all data in chunks of 16MB to improve the system management and performance. However, such approach is incompatible with the applications running on Hops, which would require modifications to be aware of such file split. CHARON could store data as whole files in the Hops-FS private repository to overcome this limitation, which would also avoid the overhead of having an additional CHARON layer during computations. However, all write operations would still have to be performed through CHARON to correctly update the metadata associated with files.

Despite the feasibility of this approach, it might be inefficient in terms of management time and storage space. Files from sequencing data community may size hundreds of gigabytes each. Copying them to Hops-FS through CHARON would take significantly longer and would spend more storage space since data is stored both in the cache of CHARON and in the Hops-FS.

Scenario 2—CHARON monitoring a specific folder in Hops-FS. In this second scenario, Hops-FS runs independently from CHARON, and CHARON monitors a specific folder in Hops-FS constantly. Users must interact only with the interfaces provided by Hops-FS whenever they want to access or modify any file, as depicted in Figure 5.3. Applications computing on top of Hops run unmodified.
CHARON monitors a specific folder in Hops-FS (e.g., the “charon” folder in Figure 5.3) to identify updates in files from this directory. Users only need to move or copy a file to this folder, which is when CHARON becomes aware of this file and creates the respective metadata entry. This type of file is considered an external file since they are not managed by CHARON but are monitored by it. Users can read, write, and compute data directly in the Hops-FS with this approach. CHARON only reads data—and provides it to Hops-FS—from files that are shared with other biobanks, or are stored in external facilities (e.g., public clouds).

Similarly to the previous integration scenario, the present case might be inefficient in terms of management time and storage space. Copying or moving large files is always time-consuming. In this case, files also would be duplicated if clients copy (instead of moving) them to the “charon” folder. The solution for these two limitations encompasses creating symbolic links on the “charon” folder to the files of interest instead of copying or moving them.

**Scenario 3—Users controlling what CHARON should monitor in Hops-FS.** In this third scenario, Hops-FS also runs independently from CHARON, but CHARON does not monitor any specific folder in Hops-FS. Instead, CHARON will only start monitoring files and folders when clients indicate to do so. Users again interact only with the interfaces provided by Hops-FS whenever they want to access or modify any file, as depicted in Figure 5.4. However, they are responsible also for indicating to CHARON the resources it has to monitor, which is done through a command (“charon.manage”) with the path of the file or folder to be monitored. Furthermore, CHARON only reads the data of external files when sharing the file with other biobanks or storing data in external facilities (e.g., public clouds).

![Figure 5.4: Scenario 3—Users controlling what CHARON should monitor in Hops-FS.](image)

**Chosen integration scenario.** We opt to integrate Hops-FS and CHARON using a composite integration scenario devised from the first and third scenarios previously presented. In our approach, clients can choose between interacting with CHARON interfaces (Scenario 1) or inform CHARON to monitor a file or folder of interest (Scenario 3). Thus, we provide the best of the two approaches, where:

- Users write and read small or medium files from Hops-FS using CHARON (Scenario 1), which saves computation costs of continuously monitoring these resources.
- Users indicate CHARON to monitor large files (Scenario 3). It avoids copying it to CHARON, which saves management time and storage space.
The Deliverable D4.3 [8] from BiobankCloud project presents a tutorial on how to configure, mount, and use CHARON to store and share data. In the next section, we describe how to implement the chosen scenario for integrating CHARON and Hops-FS.

## 5.4 Launching the chosen integration scenario

In this section, we describe the steps that need to be performed to configure and deploy CHARON integrated with Hops-FS. Additionally, we also discuss how users interact with the tools and commands to share data using CHARON.

First, we expect users have fulfilled the prerequisites, downloaded CHARON, and configured the system properties as explained in the Deliverable D4.3 [8] of the BiobankCloud project. Second, users must have a functional Hops-FS instance running on their system, which can be obtained following the steps presented in the previous chapters. The following descriptions focus on launching the chosen integration scenario using CHARON and Hops-FS.

### 5.4.1 Hops-FS as a private repository in CHARON

To use Hops-FS as a private repository in CHARON you need to set the `config/hopsfsRep.properties` file with the path of the directory to be used for storing the data inside Hops-FS. In order to use a directory called “charon-files” to do that, the content of this file must look like this:

```bash
hopsfs:/charon-files
```

As you can see, this materializes the Scenario 1 defined in Section 5.3.

### 5.4.2 CHARON managing external files

CHARON is able to manage files/directories that are external to the system. Users only need to inform CHARON that some resource should be managed by it. The system will immediately create the metadata for this resource and will periodically verify if there were modifications on it. One design choice that appears here is that these kind of resources are set as read-only inside CHARON making the system just a sharing agent for them. Given this, to add an external managed folder/file to CHARON, one only needs to execute the following command in the project’s root folder:

```bash
./charon.manage <resource-to-be-managed-path> <charon-containing-folder>
```

In this command, the `resource-to-be-managed-path` parameter is the path of the folder to be managed, and the `charon-containing-folder` is the name of this folder inside the CHARON mount point. For example, if we want the folder `/home/user/studies`
to be managed by CHARON in such a way that its content will appear in it as a folder called external/studies, the command would looks like the following:

```
./charon.manage /home/user/studies external/studies
```

Note that we are also able to define a specific file to be managed. An example of this would be the addition of the /home/user/studies/study1.doc file to the external managed resources of CHARON in the same directory of the previous example. In this case, the command would look like:

```
./charon.manage /home/user/studies/study1.doc external/studies
```

Finally, in the case of Hops-FS resources, they can be added to the CHARON external managed resources by executing the same command explained before adding the prefix hopsfs: to the path of the Hops-FS resource in the resource-to-be-managed-path parameter of the command. In this way, to make CHARON manage the Hops-FS’s studies folder in a folder called external/studies, the command to be executed would look like:

```
./charon.manage hopsfs:/studies external/studies
```

As can be seen, this command materializes the Scenario 3 defined in Section 5.3.

### 5.4.3 Sharing data

In order to share data, there are some information that must be exchanged between the CHARON instances. Figure 5.5 shows this process for two instances of the system deployed in two sites (Site 1 and Site 2 in the figure). The procedure is the following: in step 1, Site 1 generates a site_id which contains the information other sites need to know to share data with him and send it to Site 2 (step 2 in the figure). This token is a text file that is filled by the client with info about itself, the clouds it uses, and the site in which the system is deployed. After receiving the site_id, Site 2 is able to share data with Site 1. Given this, when the former wants to share data with the latter it first generates a share_token and then send it to the Site 1 (steps 3 and 4 in the figure). Contrary to the last token, this one is generated automatically by CHARON when a folder is shared. Finally, when an instance gets the share_token (step 5) it is able to access the shared data.

Notice that steps 1 and 2 are executed only once when a CHARON instance introduces himself to other, and steps 3, 4 and 5 are executed every time some instance share a folder with other. The exchange of both site_id and share_token are not suppose to be done by the system. However, CHARON is able to perform this task using the users’ e-mail. We are studying if there is some way to perform this task within the BiobankCloud LIMS.

As explained before, the site_id must be filled by the client. To do that, one just needs update the fields of the /config/site.id file. This file must look like:
D2.4 – Secure, scalable, highly-available Filesystem integrated with the Object Model and Overbank

Figure 5.5: Exchange of tokens needed to share data.

id=4
email=user@gmail.pt
addr=127.0.0.1:11000

driver.type=AMAZON-S3
canonicalId=*****************************************

driver.type=GOOGLE-STORAGE
canonicalId=user@gmail.com

driver.type=RACKSPACE
canonicalId=user, user@gmail.com

In this file, the fields must contain:

- **id**: The unique identifier of the CHARON instance. This id is defined in the **client.id** field in config/charon.config file.

- **email**: The email of the system client defined in the config/charon.config file.

- **addr**: IP and port where the system listens requests from the private repository, which are defined in config/charon.config file.

- **driver.type** and **canonicalId**: The **driver.type** identifies the cloud storage provider, and a **canonicalId** is used by other instances to give this client access to their data. Both must be defined for each cloud account used to store both file system’s data and metadata.

The **canonicalId** format differ from cloud to cloud, which will be explained in the following descriptions together with an explanation on how to obtain them.
In the case of **Amazon S3**, in AWS control panel go to top left corner and click on your username and Security Credentials. Next, click on “Account Identifiers” and you will find the value of “Canonical User ID” there.

For **RackSpace**, the canonicalId is composed by the service account username and the client e-mail, separated by a comma. This e-mail could be the one defined in the email field on the top of the file.

In **Google Storage** this identifier is the Google account email.

Notice that there is no information about **MS Windows Azure**. This happens because in this service the technique used to give grantees access does not need any client specific information [7].

Given this, when a client wants to share a directory it can use a tool we provide in the project directory by executing the following command:

```
./charon.share <charon-dir> <peer-id> <permissions>
```

In this command the charon-dir parameter is the CHARON managed directory the client wants to share, the peer-id is the client with which the directory will be shared and the permissions define the capabilities the peer will have over that directory. In this way, to share a directory called studies/shared with the client with the id 5 giving him read-only permissions, the command to be executed should be this:

```
./charon.share studies/shared 5 r
```

The execution of this command generates the share_token for this shared folder and after that the resources inside the studies/shared folder will be accessible for the client 5. This share_token will be created in the directory defined by the share.tokens.directory field in the `/config/charon.config` file. As explained before, this token must be sent to the client with which the directory was shared (in our example client 5), and it must put it into a folder called NewSNSs in the project’s root directory. The config/NewSNSs folder is monitored by CHARON in order to add new shared folders to the system.

### 5.5 Deploying CHARON with Chef/Karamel

The previous section has described how to manually configure and install CHARON. However, this process can be hard and confusing, specially for users that are non-computer experts. To facilitate the CHARON deployment, we provide a Chef cookbook [1] which automatically downloads and executes CHARON and its dependencies. Moreover, this cookbook can be used through Karamel [4] (which was developed under the BiobankCloud Project) to deploy CHARON together with other cluster software. Note that, although the CHARON Chef cookbook and the Karamel framework automate the installation of CHARON, the users still need to create the clouds accounts and find the required API keys, as explained in Section 5.4.
CHARON Chef cookbook. For using the CHARON’s cookbook the user must download it at GitHub in the following link: https://github.com/biobankcloud/charon-chef; extract it; and put it in a desired folder (e.g., charon_cookbook). It provides two recipes: one for installing the software and all the dependencies (charon_cookbook/recipes/install.rb), and another for running the system (charon_cookbook/recipes/default.rb). For running this cookbook without using karamel (using Chef commands), the users must fulfill first the attributes in the Chef attributes file (which can be found in charon_cookbook/attributes/default.rb).

Using Karamel. The first thing to do is to download the last stable version of Karamel available on its web page [4]. After that, for deploying CHARON through Karamel the user only needs to download the “charon.yaml” file on the root of the CHARON’s cookbook at GitHub. This file defines the cluster to be deployed, including the size of the cluster (1 machine by default), the cookbook GitHub link, the Amazon EC2 machine to be used (m3.medium by default) and the location of that machine (Ireland by default).

After running the Karamel framework, users will be able to load the “charon.yaml” file to Karamel interface. Before launching the cluster, the users must configure the CHARON attributes in that interface, for example the cloud account’s API keys, the user email and the mount point of the file system. When launching the cluster, Karamel will first execute the recipe to install the software, and then the recipe to execute it.
Chapter 6

YARN

In modern clusters the network is a bottleneck. As a result, it would be highly inefficient to download 1000 genomes from HDFS to a single computation machine in order to run a 1000 genomes analysis. To solve this problem big data systems distribute the computation on the datanodes. This way, the data does not have to travel in the network and computation is run in parallel on several machines. This provides better performances than centralized solutions that download the data on a single machine [11], but this also introduce some resource allocation complexity: it often happens that several clients want to simultaneously run computation on the same data or on data hosted by the same datanode. It is the role of the resource manager to ensure that the clients can simultaneously run computation in the cluster without interfering with each other.

In the Hadoop stack, the resource manager is Hadoop YARN [29]. Hadoop YARN was introduced in 2013, in Hadoop v2. It is a young project still going through developments. An important part of these developments intend to resolve YARN scalability, reliability and responsiveness issues. As we will see in section 6.1, these issues are inherent to YARN centralized architecture. In section 6.2 we present Hops YARN, the version of YARN we are developing to solve these problems and enable clients to run interactive workloads.

6.1 Hadoop YARN

On the model of HDFS architecture, the Hadoop YARN architecture, presented in figure 6.1, is centralized. YARN architecture is composed of two main components: the ResourceManager and the NodeManagers.

NodeManagers They run on the nodes that run users’ computation, usually the datanodes. Each computation node runs one NodeManager whose role is to monitor the usage of the node and to ensure that clients access to the node resources according to the ResourceManager decisions.

ResourceManager It runs on a dedicated node. This node centralizes all the usage information
provided by the NodeManagers and uses this knowledge to share resource between the clients that want to run computation on the cluster.

When a client wants to run an application on the cluster, they send this application to the ResourceManager. The ResourceManager finds some available resources on one computation node and starts an ApplicationMaster on this node. This ApplicationMaster is then charged of negotiating with the ResourceManager in order to obtain resources to run the application. As we can see, all the clients have to contact the ResourceManager in order to start applications, all the applicationMasters have to contact the ResourceManager to obtain resources and all the NodeManagers send information to the ResourceManager in order to inform it of their current state. This centralized architecture has two main advantages: first, the ResourceManager has a complete view of the cluster state, this allows it to take optimal decisions when attributing resources [24]; second, it is easy de design new scheduling policies to optimize the cluster usage according to the type of usage required [10, 13]. On the other hand, this architecture has two main problems: the ResourceManager is a bottleneck and a single point of failure.

### 6.1.1 Single point of failure

In order to solve the single point of failure problem, a replication mechanism has recently been introduced in YARN. The principle of this mechanism is to run two ResourceManager: a main ResourceManager and a standby ResourceManager. The main ResourceManager run the tasks
described above and stores a copy of its state in ZooKeeper. As long as the main ResourceManager is running, the standby ResourceManager does nothing. When the main ResourceManager crashes, the standby ResourceManager pulls the state of the crashed ResourceManager from ZooKeeper and becomes the new main ResourceManager. This solution is limited by the small throughput of ZooKeeper, which results in the impossibility for the main ResourceManager to store its full state. As a result, the main ResourceManager only stores a partial state in Zookeeper, and the standby ResourceManager must restart all the running applications when a failover happens. This can result in the loss of hours of computation. In order to solve this problem, the Hadoop community is working on a mechanism that would allow the standby ResourceManager to rebuild the missing parts of the state by getting information from the NodeManagers. Getting the information and rebuilding the state is a slow process and can result in the cluster being frozen during a long time after a failover.

6.1.2 ResourceManager the Scalability Bottleneck

The second problem of Hadoop YARN architecture, the bottleneck, is not addressed by the solution mentioned above: as long as the main ResourceManager does not crash, the standby ResourceManager stay inactive and is a waste of resources. This bottleneck problem has two results: the size of the cluster is limited and the responsiveness of the cluster is limited. Even if most of the existing Hadoop clusters are too small to reach the size limitation, there exists some large cluster, such as Yahoo’s cluster, that reach the limits of this architecture. The responsiveness problem comes from the mechanism used to exchange messages between the ResourceManager and the NodeManagers. In order to ensure that the ResourceManager will never be overloaded, all the exchanges between the ResourceManager and the NodeManagers are following a heartbeat mechanism. When setting up a cluster, the administrator decides on a frequency at which the NodeManagers will send messages to the ResourceManager. This way, there is no risk of pike utilization of the ResourceManager and it is easy to modulate throughput at which the ResourceManager will have to handle messages by modulating the frequency of the heartbeats. In order to avoid overloading the ResourceManager, the administrator must choose a low frequency. This result in the ResourceManager needing time to be informed of any change in the state of the cluster and to propagate resource attribution to applications. As a result, starting an application and getting resources for this application is slow and the cluster is not interactive.

A bottleneck problem also appears in the internal architecture of the ResourceManager. This architecture is represented in figure 6.2. In this figure, each square represent a single thread. Each thread is equipped with a queue and the threads communicate with each other by putting events in each other queue. This system provides some parallelism. But, as we can see, the RMContext Queue play a central role: all the messages received by the ResourceManager result in at least one event in the RMContext Queue. As a result, the RMContext Queue is a bottleneck inside of the ResourceManager which is itself a bottleneck in the YARN Architecture. In order to reduce the bottle neck aspect of YARN we propose to give an active role to the standby ResourceManager, to offload the load on the main ResourceManager.
6.2 Hops YARN

Our goal when designing Hops YARN, our implementation of YARN, is to solve the Hadoop YARN bottleneck and fail over problems while keeping YARN’s advantages: full view of the cluster state when taking resources allocation decisions and ease of implementing new advance scheduling policies. Additionally to the centralized architecture, the state of the art contains two other architectures: two-level architecture [14] and shared state architecture [24]. But, none of these architectures guaranties both of YARN architecture advantages: in the tow-level architecture the resource allocation decisions are taken with only a partial view of the cluster state and in the shared state architecture it is complicated to implement new advances scheduling policies.

We propose to reduce the bottleneck aspect of the centralized architecture by creating a new architecture in between the centralized architecture and the shared state architecture. This architecture is presented figure 6.3. In this new architecture the different tasks of the centralized architecture’s ResourceManager are distributed among two types of nodes: The Scheduler and the ResourceTrackers that use MySQL Cluster for communication. The Scheduler handles the messages from the Clients and from the ApplicationMasters and implements the resource allocation logic while the ResourceTrackers handle the messages from the NodeManagers and update the cluster resources view. This allows Hops YARN to keep the ease of implementing new advance scheduling policies, by still having the scheduling logic in a centralized node (the Scheduler), while distributing the load generated by the NodeManagers’ heartbeats on the Re-
sourceTrackers. Distributing this load allows Hops YARN to handle bigger clusters and higher
heartbeats frequencies, making it more interactive.

### 6.2.1 MySQL Cluster Event API

In order to provide fail over and to share information between the ResourceTrackers and the
Scheduler, we use MySQL Cluster NDB. On top of taking advantage MySQL Cluster char-
acteristics presented in section 3.1 (high throughput, reliability and scalability) Hops YARN
also take advantage of the event API provided by NDB. The NDB event API allows a node to
subscribe to event messages containing the updates done on the rows of a given table. This
event messages are periodically sent to the subscribing node by the datanodes of the database.
The periodicity at which the event messages are sent is a parameter of the database (by default
100ms), and each period is called an epoch. The events messages contain the table updates
that happened during the last epoch and the epoch number. If no transaction modified the table
during the last epoch, the message only contains the epoch number. The epoch number is atom-
ically incremented on all the datanodes, it is used for the client to be able to correlate together
the updates happening on the different datanodes within a same epoch. As the event messages
contain all the updates done by transactions successfully committed during the last epoch, the
subscribing node can rebuild a consistent state of the database, once it has received all the event
messages for one epoch. Moreover, as the epoch number is atomically incremented at the end
of each epoch, it can be used by the client to verify that it did not miss any message.
6.2.2 Hops YARN functioning

Hops YARN works as follow. Each NodeManager is associated with a ResourceTracker in a load balanced way. The NodeManagers send their heartbeats to the ResourceTrackers they are associated to. When a ResourceTracker receives a heartbeat, it handles it, creates an answer based on its in-memory view of the cluster, sends this answer to the NodeManager and commits to the database the new view of the resources modified by the heartbeat. Then, the database streams the updates to the Scheduler, thanks to the NDB event API. The Scheduler uses the received events to update its in-memory view of the cluster. When the Scheduler receives a request for resources, either from a client or from an ApplicationManager, it handles it, takes resource allocation decisions based of its in-memory view of the cluster, answers to the request and commits to the database the new state of the affected resources. When the commit is done, the database uses the NDB event API to stream the updates to the ResourceTrackers. The ResourceTrackers use this information to update their in-memory view of the NodeManagers they are associated to. These updated views are then used to answer to the next NodeManagers heartbeats.

As we said previously, the NDB event API messages can be used to rebuild a consistent state of the database. In our case, this means that once the Scheduler or the ResourceTrackers have received all the event messages corresponding to an epoch they can build a consistent view of the state of the cluster. The only limitation is that the database guaranties that two updates being reported in two different epochs have happened in the same order as their respective epoch, but it does not give any guaranties about the order of two updates happening within the same epoch. This is not a problem for the updates committed by the Scheduler and streamed to the ResourceTrackers. In fact, the ResourceTrackers only subscribes to events concerning the resources of the NodeManagers they are associated to and these resources are updated by the scheduler at most once every heartbeat. As a result these resources should not be updated two times within the same epoch. Moreover, the resource updates are commutative. Ordering the updates is more important for the updates committed by the ResourceTracker and streamed to the Scheduler. In fact, some of these updates correspond to NodeManagers registration and the Scheduler need to be aware of these registrations before to receive any updates due to heartbeats from the corresponding NodeManagers. To order this updates, each ResourceTracker is equipped with a counter that is atomically increased each time the ResourceTracker commit an update. The value of this counter is committed to the database at the same time as the resources update. This value is then used by the Scheduler to order updates committed by a ResourceTracker. As each NodeManager is associated to one ResourceTracker, ordering by this value guaranty that the NodeManagers’ registration updates will be handled by the scheduler before any heartbeat updates.

Using the event API is efficient and implies a low charge on the Scheduler and the ResourceTrackers. Moreover, as the full state of the cluster is stored in the database Hops YARN provide transparent fail over both for the Scheduler and for the ResourceTrackers. As a result, Hops YARN provides the same properties as Hadoop YARN while providing transparent failover and while being more scalable and more responsive.
Chapter 7

Evaluation

7.1 Hops-FS NameNode Performance

In order to stress test the NameNodes and evaluate their read and write throughput, we designed a custom benchmark. Standard benchmarks, such as NNThroughputBenchmark, are designed to test the performances of a single NameNode and are not suited to evaluate the performance of a multi-NameNode system. Another limitation with NNThroughputBenchmark is that it does not include the cost of network communication between the clients and the NameNode, and it is also limited in that the whole benchmark runs on a single JVM. NNThroughputBenchmark also cannot balance load among different NameNodes.

We designed our benchmark to determine the throughput of read and write operations on a cluster containing several NameNodes. The benchmarks runs multiple clients distributed across multiples hosts. The clients are remotely controlled by a central operator that starts and stops all of them at roughly the same time. To measure the write throughput of the NameNodes, the clients send create file requests to the NameNodes, where the requests are load balanced across the NameNodes and the clients send requests in a loop in order to create as many files as possible. We then aggregate the number of files created by each client to obtain the number of files created per minutes. We proceed in the same way for read-operations.

7.2 Effect of Number of RPC Handler Threads

The amount of parallelism is controlled by dfs.namenode.handler.count parameter. It determines the number of handler threads that process the incoming clients requests. In Apache HDFS as the entire namespace is in the main memory the RPC handler threads do not block during operations. Therefore, in Apache Hadoop the number of threads are directly proportional to the number of CPU cores in the system. In Hops-FS a handler thread access the database multiple times during an operation. This blocks the thread and the throughput drop. By increasing the number of handler threads Hops-FS can achieve high throughput. Figure 7.1 shows how the throughput of Hops-FS is effected when the number of handler threads are in-
increased. The experiment was performed on 24 core Intel(R) Xeon(R) CPU E5-2420 machine clocked at 1.90GHz with 256 GB RAM. The Throughput linearly increases when the number of handler threads are increase from 1 to 80 threads. However, further increasing the number of threads degrades the performance due to threads switching and thrashing and simultaneous direct memory allocation by concurrent threads.

![Figure 7.1: Effect of number of handler threads on Hops-FS throughput.](image)

### 7.3 Multiple NameNodes

Thanks to this benchmark we measured the performances of Hops-FS according to the number of NameNodes in the system. All the experiments were performed on 9 bare-metal machines running CentOS 6 (7x(12 core Xeon E5-2420, 256 GB of RAM) and 2x(12 core Intel Xeon X5660, 40 GB RAM)). All the hosts are connected by a 10-gigabit ethernet switch with no network bonded interfaces. As the experiments were designed to stress the NameNode and not to test the DataNodes, we only needed three DataNodes for all experiments.

**Setup for Apache HDFS:** the number of NameNodes was set to two (one active and one standby NameNode). Three Quorum Journal Nodes were used for storing the EditLog, and 7,000 clients distributed on 12 hosts were used to send file operations requests to the NameNode.

**Setup for Hops-FS:** We varied the number of NameNodes from one to seven. The metadata was stored in a deployment of MySQL Cluster containing 2 Database DataNodes, and 7,000 clients distributed on 12 hosts were used to send file operations requests to the NameNode.

Figure 7.2 shows the results of this benchmark when evaluating the write throughput of the system. We can see that with only two NameNodes Hops-FS can create 3200 files/second, which is more files per second than Apache HDFS. Our solution scales linearly when increasing the number of NameNodes. The system saturates with six NameNodes and adding more NameNodes does not increase the performance of the system.

Similarly, the performance file read operation increases linearly and the system saturates with six NameNodes. Figure 7.3 shows that Hops-FS can read 22,000 files per second with just four
Figure 7.2: Hops-FS File Creation Performance.

NameNodes. We are confident that with larger cluster setup we can easily achieve even higher read throughput.

Figure 7.3: Hops-FS Read Performance.

Figure 7.4 shows the performance of the rename operation. Although the performance linearly increases it remains lower than the performance of HDFS. This is primarily because in Hops-FS rename operation follows a more complicated SubTree operations code path. Detailed discussion on as to why Hops-FS has more complicated rename operations can be found in D2.3. We are optimizing the rename operation and hope that we will be able to match the performance of the HDFS. Similarly, figure 7.5 shows the performance of the delete operation. Although the performance linearly increases it remains lower than the performance of HDFS because delete operation also follows a complicated SubTree operation code path.
SubTree Operations Performance

Operations like move, rename, and delete operations affect all the descendants of an inode manipulated by these operations. For example, move and rename operations will change the absolute paths of all the descendant inodes, while the delete operation will remove all the descendant inodes. One potential implementation of these operations would be to take exclusive locks on all the descendants of the inode that is being manipulated by the operation. This will ensure that no other operation can access the subtree while the operation is ongoing. However, a directory may have millions of descendants and this isn’t feasible in current in-memory online transaction processing (OLTP) if we assume that each operation is encapsulated in a single transaction. This is because large number of locks cannot be acquired before practical TransactionTimeout limits being exceeded. Secondly, large transactions consume significant amounts of the memory, and transactions on tens of millions of inodes require all those inodes to be kept.
in memory, which again may not be feasible for in-memory OLTP systems. Lastly, acquiring large numbers of locks affects the performance of the distributed lock manager of the database management system which will result in poor performance for concurrent operations. In the following section, we introduce a new method for locking and isolating the entire subtree without relying on the distributed lock manager of the database management system.

As we cannot implement large subtree operations as single transactions, we have developed our own protocol that provides the same consistency, isolation, and durability guarantees as transactions, but where operations are not atomic. In our protocol, subtree operations do not execute all-at-once, but rather as a series of batched operations, where each batch makes progress towards completing the subtree operation and when all batches have been executed, the subtree operation will have completed. Other concurrent clients are not affected by ongoing large subtree operations, and we maintain the same isolation semantics as Apache HDFS for the subtree operations. For more details see D3.5 (Scalable Storage). Table 7.1 show the performance of subtree operations on a directory containing 100,000 files. Hops-FS’ performance of these operations is considerably low because large amounts of data is read over the network and the operations divided into thousands of small transactions. In case of delete operation the directory is deleted incrementally using 100,000 transactions. HDFS out performs Hops-FS as all the data is readily available in the memory.

<table>
<thead>
<tr>
<th>Operation</th>
<th>HDFS</th>
<th>Hops-FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete SubTree Operation</td>
<td>97 ms</td>
<td>25636 ms</td>
</tr>
<tr>
<td>Rename SubTree Operation</td>
<td>55 ms</td>
<td>4775 ms</td>
</tr>
</tbody>
</table>

Table 7.1: Performance of SubTree Operations on a directory containing 100,000 files.

## 7.5 File Path Length and Performance

One of the most expansive tasks in Hops-FS is resolving the path components. Taking into consideration that most of Hops-FS operations starts with resolving the path components for the operation path of interest, optimizing the path resolving is a big deal.

Normally, if we assume that we are creating a file “test.txt” in “/a/b”. Given that in our database scheme each INode, has a composite primary key that consists of parentId and name. So, to resolve the parent directory for “test.txt” we will have to go to the database 3 times. First to read the root “/”, then “a” using Id of the root, then “b” using Id of “a”. Moreover, increasing the path length will increase the number of roundtrips to the database which will slow down the operation.

One solution is to cache parentIds for each given path, and then send a one batch to the database to read the path components data. We guarantee consistency by validating the cache after doing the batch read.

One more reason to cache path components is to enable distribution awareness in NDB. In Hops-Fs, all the data related to an INode file is residing on the same machine, so giving a hint to the NDB transaction where this file could be will enhance the performance of Hops-FS. When
we start a transaction the path is only known, and to enable distribution awareness, the id of the last component should be known as well. We could accomplish this easily using our caching layer.

In our experiment, we used one NameNode and one experiment machine which runs 100 Hops-FS clients simultaneously, and a small NDB cluster consisting of two datanodes. NDB cluster and the NameNode are connected through 10 Gigabit connection.

Figure 7.6 and 7.7 shows the average time taken by a transaction using different Caching implementations.

Figure 7.8 show the time elapsed by each one of the caching implementations to resolve a path for different number of clients. Figure 7.9 shows the number of round trips. We can conclude from these figures that InMemory cache have the best performance and the second best if we are going for Memcache based implementation is OptimalMemcache.

7.6 Namespace overhead and Metadata scalability

Storing the metadata in a database causes expansion in the amount of memory required to store the data. We call the amount of extra memory required the memory expansion factor. The extra memory is used to store primary and secondary indexes as well as pad-out both rows and columns. In this section we evaluate the memory expansion factor and compare the scalability of the namespace of Hops-FS and HDFS in terms of number of files they can manage.

7.6.1 HDFS namespace usage

As discussed in Chapter 2, HDFS keeps all the metadata state in the RAM of the NameNode. In order to allows the NameNode to store as many metadata as possible the developers of HDFS have put a huge effort in designing data structures that reduce the memory footprint of the metadata [25, 26]. We will now evaluate the size of each of these data structures.

The vast majority of the NameNode’s memory is used by Inodes and BlockInfos (cf:Figure 7.10). The memory footprint of each of these entities has been estimated by the past [25], but these estimations are outdated.

In order to conduct this evaluation we assume a 64-bit JVM. We then use the known size of Java primary types and the size of the main Java basic data structures overheads, listed in Table 7.2, to compute the size of each of the NameNode data structure classes. The results are presented in table 7.3, with $L$ the length of the file name and $R$ the replication factor of the blocks.

We can now evaluate the amount of memory required to stored the metadata for a file. In production environments, Yahoo observed that each file contains on average 1.5 blocks [26] with a replication factor of 3. Armed with this information, we took a number of assumptions to enable us to build a simple model that estimates the memory consumption of each file: each file has a name of length 20; each file contains two blocks and the replication factor is set to 3. Given
Figure 7.6: Performance of Writing new empty files to Hops-FS while using different Caching implementations.

these assumption each file in HDFS will consume \((128 + 20) + 2 \times (88 + 24 \times 3) = 468\) bytes. Se we can store metadata for up to 2.1 million (2136752) files in 1GB of main memory.
7.6.2 Hops-FS namespace scalability

We will now estimate the memory consumption of the same metadata in a database. Storing metadata in a database incurs some additional costs. These additional costs come from normalizing the stored entities and storing more information for efficient data retrieval: primary keys, partition keys and indexes. Additionally in MySQL cluster all rows are 4 bytes aligned, that is, if a row is 21 bytes long then it will actually consume 24 bytes in the memory. In order to evaluate this over cost we need to know that in MySQL Cluster each index use 16 bytes per record and that the overhead of a variable sized columns (to store tables) varies from 4 to 12 bytes. In order to calculate the size of each entity we use a tool called sizer. Sizer accurately measures the memory usage of the entities stored in MySQL Cluster and is used to evaluate large scale cluster deployments [17].

Like HDFS, the entities that consume the most memory are INodes, Blocks, Replicas and Block.Lookup. Inode entity contains five variable length columns. The columns client.machine,
client_node and client_name stores information about the client that is writing data to the file. Once the file is closed these variables are emptied. Similarly, name and symlink store up to 3000 byte long file names. However, in typical production systems, filenames are short [20]. Table 7.4 shows that a row in an INode table, including all the overhead costs, can take up to 6586 bytes when the file is open. However, when the file is closed, the variable columns, client_machine, client_node and client_name, will be empty. As a result, assuming that the file-
name is 20 characters long, each INode row will require 306 bytes of memory, see equation 7.1.

\[
6586 - 3000(symlink) - 2980(filename) - 100(client\_name) \\
- 100(client\_node) - 100(client\_machine) = 306\text{bytes}
\] (7.1)

A row in the Block table takes 114 bytes (see table 7.5). Many internal operations of Hops-FS contain only information about the block_id. In order to efficiently determine to which file the block belongs we maintain an inverse block lookup table. This block lookup table has one row for each block in the system. The cost of storing one row in the block lookup table is 66 bytes (see table 7.6). Similarly each replica of a block uses 90 bytes (see table 7.7).

To conclude, in Hops-FS a file with two blocks and a replication factor of 3 will take 1206 bytes in memory, (see eq. 7.2). MySQL Cluster replicates the data to provide high availability of the stored data. By default MySQL Cluster replicates the data twice; which means a file in Hops-FS will actually take 2412 bytes. Hops-FS can store 0.4 million (414593) files in 1GB of metadata.
### Memory Requirements for Main Data Structures (Objects) in HDFS

<table>
<thead>
<tr>
<th>Object</th>
<th>Used memory in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INode</td>
<td>$64 + L$</td>
</tr>
<tr>
<td>INodeFile</td>
<td>$128 + L$</td>
</tr>
<tr>
<td>INodeDirectory</td>
<td>$144 + L$</td>
</tr>
<tr>
<td>INodeDirectoryWithQuota</td>
<td>$176 + L$</td>
</tr>
<tr>
<td>Block</td>
<td>$32$</td>
</tr>
<tr>
<td>BlockInfo</td>
<td>$88 + 24 \times R$</td>
</tr>
</tbody>
</table>

Table 7.3: Memory requirements for the main data structures (objects) in HDFS.

\[
306(INode) + 2 \times 114(Blocks) \\
+ 2 \times 66(Lookup\_Table) + 6 \times 90(Replicas) \\
= 1206\text{bytes}
\] (7.2)

Hops-FS uses 514% of the memory used by the active HDFS NameNode to store the metadata for each file. However, for a highly available (HA) deployment with an active, standby, and checkpoint NameNode, the expansion factor becomes only 171%. Nevertheless, HDFS’ namespace scalability is limited by the size of JVM heap of the NameNode. Yahoo has one of the biggest Hadoop clusters in the world, containing 4000 DataNodes. In this large cluster, the NameNode’s JVM uses 100 GB of heap space. Yahoo’s Hadoop cluster cannot scale to larger sizes because increasing the size of the JVM heap beyond 100 GB is not practical, due to stop-the-world garbage collection events. At such scales the JVM pauses are very significant and the throughput of the NameNode can drop dramatically on JVM pauses. With current JVM technology, it is reasonable to conclude that HDFS’s namespace cannot scale that much beyond a couple of hundred gigabytes. Meanwhile, Hops-FS has no such limitations. MySQL Cluster supports up to 48 DataNodes, which allows it to scale up to 12 TB of data in a cluster with 256 GB RAM on each DataNode. As shown in figure 7.11, Apache-HDFS is limited to 213.6 million files on a 100 GB JVM, while Hops-FS can store up to 4.9 billion files using 12 TB of data (see figure 7.11), where the metadata has two copies (replicas) for high availability. Finally, the metadata overhead in Hops-FS also has utility for system operators, as the data tables contain a large number of indexes that support arbitrary queries on the metadata using SQL - a feature not supported by HDFS. If users don’t wish to perform online queries on MySQL cluster, the cluster data can be replicated to a MySQL server, where offline analytics queries can be run.

### 7.7 Erasure Coding

#### 7.7.1 Resilience to Failures

Two experiments were executed in order to evaluate the reliability of Hops-EC and to compare it to standard replication (or block triplication, as we call it). In each experiment, 100 random randomly sized files with 10 to 400 blocks with 4MB of random data were stored on 18 DataNodes, before failing $n$ DataNodes. Failures were simulated by shutting down DataNodes.
The first experiment evaluated the effect of node failures when using block triplication and the default block placement policy. Blocks were spread evenly though nodes, by removing the writing node after file creation and waiting for the cluster to achieve triplication again. Figure 7.12 shows the results of the experiment. Three node failures are enough to corrupt all replicas of some blocks for 33% of the files, making the file unavailable. Having four or five node failures a majority of files becomes unavailable. This drastic result is most certainly caused by the small size of the cluster, in combination with a large number of blocks per file. Thus, many blocks of a file are stored on each node and hence not many failures can be tolerated.

The second experiment encoded the same files with a \((10, 6, 5)\)-LRC code and applied a replication factor of one, before introducing the same node failures as in the first experiment. As the results in figure 7.13 show, up to four node failures can be tolerated without any file corruption. This is achieved by the block placement and the encoding of Hops-EC. With five nodes failing, the failure rate is similar to the triplication scenario. Thus, the reliability of Hops-EC is superior to triplication.
### 7.7.2 Effect of block placement

In order to evaluate the benefits of placing the blocks of each stripe and its parity blocks onto different DataNodes, the experiments of creating 100 files with a random size of 10 to 400 blocks, with 4MB of data each, were repeated. Instead of failing nodes, blocks were randomly deleted in order to simulate failures without enforced block placement. The same cluster consisting of 18 DataNodes was used and the files were created with the same seed as in the experiment with failing nodes, resulting in the same files. Having 18 DataNodes, one node stores roughly 5% of all blocks. Hence, instead of failing one node, 5% of all blocks are randomly deleted.

The results from the experiment are illustrated in figure 7.14. While a block loss of 5% can still be tolerated in this case, a 10% loss already leads to a significant amount of corrupted files. Block loss factors of 15% and 20%, respectively, is disastrous and causes corruption of a majority of files. Comparing this result to the one benefiting from the block placement strategy, shown in figure 7.13, it is obvious how much a robust block placement strategy improves reliability. With the placement strategy of Hops-EC, not a single block would have been lost, even with 20% of lost blocks. The block placement strategy is, therefore, a very efficient way of increasing resilience to failures.
Table 7.7: Replica table memory consumption.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage_id</td>
<td>int(11)</td>
<td>4</td>
</tr>
<tr>
<td>replica_index</td>
<td>int(11)</td>
<td>4</td>
</tr>
<tr>
<td>inode_id</td>
<td>int(11)</td>
<td>4</td>
</tr>
<tr>
<td>block_id</td>
<td>bigint(20)</td>
<td>8</td>
</tr>
<tr>
<td>primary key index</td>
<td>hash index</td>
<td>16</td>
</tr>
<tr>
<td>storage_idx</td>
<td>btree index</td>
<td>16</td>
</tr>
<tr>
<td>Row Size</td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>Row Size with overhead</td>
<td></td>
<td>90</td>
</tr>
</tbody>
</table>

Figure 7.11: HDFS and Hops-FS namespace scalability. HDFS’ scalability is limited by the fact that the JVM cannot handle much more than 100 GB, while Hops-FS can support up to 5 billion files.

Figure 7.12: Triplication: Corrupted files per failed node.
7.7.3 Availability and degraded reads

As described earlier, Hops-EC ensures erasure-coded files are repaired transparently during reads, repairing lost blocks while blocking the client until the repair has completed. To evaluate the performance of degraded reads an experiment was conducted comparing the read performance of a healthy file to a scenario with one missing block and a scenario with one missing node. A file size of 10GB and a block size of 64MB was used, while reading the file with a single client. A (10, 6, 5)-LRC code was used for file encoding. The experiment was executed on the cluster of 18 nodes. Figure 7.15 shows the results of the experiment.

For one lost block, the read duration increases by 30 seconds from 1.5 to 2 minutes. Considering that more than 98% of failures in large distributed file systems are single block failures [19], this should not have a big influence on the expected read performance. Looking at the scenario in which a full node failed, the read time increases significantly. This is due to the fact that the cluster consisted of only 18 nodes and the failed node stored a sizeable percentage of the

---

**Figure 7.13:** Hops-EC: Corrupted files per failed node.

**Figure 7.14:** Hops-EC: Corrupted files during random block loss.
total set of blocks in the system. Also, when the recovery file system detects a block failure, it will retry to read the block multiple times, after waiting for a few seconds, before starting the reconstruction process. Hence for an increasing number of lost blocks, the wait time also increases linearly. Considering a realistic cluster size of several hundred nodes, a single node failure should not cause many losses in a single file and the degraded read performance should be comparable to the single block failure experiment. Consequently, having a transparent repair mechanism is an effective way of ensuring file availability after failures.

![Diagram of read duration](image)

**Figure 7.15: Read duration of a 10GB file**

### 7.7.4 Comparison to HDFS-RAID

Table 7.8 shows a summarized feature comparison of Hops-EC and HDFS-RAID. It can be seen that Hops-EC continues to support most of the features of HDFS-RAID while adding support for important features such as a flexible API, enforced block placement and prioritized repairs. The most important feature that is currently not supported by Hops-EC is appending to encoded files. However, it will most certainly be supported in the near future.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Hops-EC</th>
<th>HDFS-RAID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible API / Support for custom strategies</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Detection of failures as early as possible</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Low overhead to maintain the state</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Prioritized repairs</td>
<td>✔️</td>
<td>partially</td>
</tr>
<tr>
<td>Enforced block placement</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Support for Hadoop 2</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Transparent repairs</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Configurable and extensible codecs</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Grouped encoding</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Support for append</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>HAR support for parity files</td>
<td>n/a</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Table 7.8: Comparison of Hops-EC and HDFS-RAID**
7.8 Leader Election Evaluation

We have implemented the leader election using in-memory, highly-available, distributed database called NDB (Version 7.4.3), the storage engine for MySQL Cluster [21]. NDB is a real-time, ACID-compliant, relational database with no single point of failure and support for row-level locking. We use the native Java API for NDB, ClusterJ, as it provides lower latency and higher throughput than the SQL API that uses the MySQL Server.

All the experiments were performed on nodes behind a single 1 Gbit switch, where the network round trip time between any two nodes is in single digit millisecond range. The NDB setup consisted of six data nodes (6-core AMD Opteron 2.6 GHz, 32GB RAM) with replication factor of 2. We compare our solution with a leader election solution implemented using ZooKeeper. The ZooKeeper setup consisted of three quorum nodes (6-core AMD Opteron, 32GB RAM). We used the leader election library for ZooKeeper (Version 3.4.6) from the Apache Curator project (Version 2.7.1). Each ZooKeeper client creates a sequential ephemeral node in predetermined directory. Each node registers a watch (callback request) for its predecessor. Upon a node failure its successor is notified. The successor checks if there are any nodes with smaller sequential number. If there are no smaller nodes available then it elects itself as the new leader; otherwise, it registers a new watch for its new predecessor.

In the experiments the initial heartbeat round time was set to 2 seconds and Maxmhb was set to 2. To accurately determine the failover time all the clients were run on a single machine (12-core Intel Xeon 2.8 GHz, 40 GB RAM). All experiments were performed fifteen times and the graphs show the average results, with the error bars showing the standard deviation of the results. In each experiment N processes are started. When all processes have joined the system, the round time is continuously monitored for changes. If it does not change for a certain time (three minutes) then the system is considered stable. After the system has stabilized the leader process is repeatedly killed 50 times to measure failover time.

Figure 7.16a shows the relation between network size and the time to elect a new leader. Up to 200 processes the service consistently elects a new leader in around five seconds. However, when the network sizes increases beyond 200 nodes the time to elect new leader also increases. This can also be observed in figure 7.16b which shows the relationship between round time and the network size. For network sizes up to 200 processes, all the processes manage to update the counter before they are suspected by the leader process. However, when the network size increases beyond 200, contention on the registers prevents some processes from writing to the shared register for consecutive heartbeats. The leader processes detects contention on the registers when an evicted process raises the evict flag. The leader process increases the heartbeat delay to release the contention on the registers, which has the side-effect of also increasing the leader failover time. In the experiments, the heartbeat delay increment (Δ) was set to 50 milliseconds.

In the implementation of leader election using ZooKeeper, the time to elect a new leader is determined by two configuration parameters: tick time and session timeout. We set these values as low as possible to quickly elect a new leader in case of a leader failure. The lowest allowable values for tick time is 2 seconds, and session timeout is 4 seconds. In order to accurately determine the fail over time all leader election processes were run on one (12-core Intel Xeon 2.8 GHz, 40 GB RAM) machine. Up to 400 processes ZooKeeper constantly elects
Figure 7.16: Performance of leader election service with the default configuration settings for NDB (MySQL Cluster). Figure 7.16a shows the average time to elect a new leader when the current leader process fails. Figure 7.16b shows the increase in the heartbeat round time when the leader detects contention on the registers.

a new leader in six seconds. However the time to elect new leader starts to drop if we increase the number of clients on the same machine. This is because of the contention on the CPU and main memory because of which the processes slowed down. When a leader is killed it may have already skipped a heartbeat. This results in quicker re-election of a new leader. Due to memory limitations we could not add more than 800 processes in the experiment.
Chapter 8

Conclusions

This document, we showed how the different deliverables D2.3 (Scalable Storage), D3.5 (Object Model) and D4.3 (CharonFS) come together to provide a Secure, scalable, highly-available Filesystem integrated with the Object Model and Overbank. We discussed the design, implementation, and evaluation of our own version of the Hadoop Distributed File, Hops-FS, where we externalized the metadata of HDFS to an open-source, in-memory, highly-available, distributed database called MySQL Cluster. We have showed how the file system integrates the the Object Model (D3.5). We also showed how CharonFS uses Hops-FS was underlying storage system. In addition to that we described our new contribution Hops-YARN that increases the scalability of Apache YARN by externalizing the ResourceManager state from the heap of a JVM to MySQL Cluster.

8.1 Acknowledgements

This work funded by the EU FP7 project “Scalable, Secure Storage and Analysis of Biobank Data” under Grant Agreement no. 317871.
Bibliography


