

LenticularFS: Scalable filesystem for the cloud

Roberto Bampi

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Kista 02.10.2017

Thesis supervisors:

Assoc. Prof. Keijo Heljanko

Thesis advisor:

Prof. Jim Dowling

Author: Roberto Bampi

Title: LenticularFS: Scalable filesystem for the cloud

Date: 02.10.2017

Language: English

Number of pages: 5+57

Department of Computer Science

Professorship: Computer Science

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: Prof. Jim Dowling

The Hadoop platform is the most common solution to handle the explosion of big-data that both companies and research institutions are facing. In order to store such data, the Hadoop platform provides HDFS, a scalable distributed filesystem which runs on commodity hardware and enables linear scalability by adding new storage nodes. While storage capacity of the system can be increased by adding new storage nodes, the component that handles metadata for the filesystem, the namenode, is a single point of failure and cannot easily be replaced or linearly scaled. The Hops project provides an alternative implementation of the namenode, which increases performance and scalability by storing metadata on an external distributed NewSQL database called MySQL Cluster. With the new architecture, the system is much more scalable and can transparently manage the failover of namenodes which are now stateless components. HopsFS is, however, still limited to running within a single datacenter which can cause severe outages in case the entire datacenter becomes unavailable. Cloud native storage systems, such as Amazon's Simple Storage Service (S3), solve this problem by replicating data across different, geographically distant datacenters, so that the failure of any given zone does not cause data unavailability. The objective of this thesis is to enable HopsFS to work across geographical regions while, as far as possible, maintaining the semantics of a POSIX-style hierarchical filesystem. We leverage asynchronous replication functionality provided by MySQL Cluster to obtain replication of metadata across geographical regions and we present a detailed analysis on how to maintain the consistency properties of HDFS in such an environment. Furthermore, we analyze the issue of split brain scenarios and propose a way for namenodes to detect this condition and continue operating correctly. Finally, we discuss the changes to the codebase which are required to implement the proposed plan.

Keywords: HDFS, distributed file-systems, geographic replication

Preface

I would like to thank Jim Dowling for giving me the opportunity to work on such an interesting project, Salman Niazi and Mahmoud Ismail for the constant and priceless guidance during the planning and implementation and the rest of the team at RAISE SICS. I would also like to thank Keijo Heljanko for the feedback while working on the thesis document. I would like to thank my friends for their support and Fabio Buso in particular for the helpful brainstorming and feedback during the year. Finally, I would like to thank my family for their constant love and support, without which this thesis would not have been possible.

Kista, 02.10.2017

Roberto Bampi

Contents

Abstract	ii
Preface	iii
Contents	iv
1 Introduction	1
1.1 Outline	3
2 Background	4
2.1 The Hadoop Filesystem	4
2.1.1 Architecture	4
2.1.2 Read pipeline	6
2.1.3 Write pipeline	6
2.1.4 Block placement	7
2.1.5 Fault tolerance	8
2.1.6 Heartbeat	9
2.2 Scalability limitations of HDFS	9
2.3 HopsFS	11
2.3.1 Multi-namenode architecture	11
2.3.2 Group membership service and leader election	12
2.3.3 The (meta)Data Access Layer	13
2.4 MySQL Cluster	14
2.4.1 Network DataBase	15
2.4.2 SQL Nodes	18
2.4.3 Isolation levels and locking	18
2.4.4 Geographic clusters	19
2.4.5 Dealing with conflicts in optimistic configurations	22
3 Literature review	25
3.1 The Google File System	25
3.1.1 Architecture	25
3.1.2 Chunk management	27
3.2 Windows Azure Storage	27
3.2.1 Architecture	28
3.3 CalvinFS	31
3.3.1 Architecture	31
3.4 Summary	34
4 Contribution	36
4.1 Metadata management	37
4.1.1 Overview	38

4.1.2	Split brain detection	41
4.1.3	Conflict handling for network partitions	43
4.1.4	Summary	45
4.2	Block management	45
4.2.1	Placement policy	46
4.3	Adaptations	46
5	Summary	51
5.1	Future work	51
A	Code listings	55

1 Introduction

The Apache Hadoop project is by far the most well-known open source toolkit for the storage and processing of big data. Since its inception, the Hadoop project moved from a map-reduce framework to a generic set of loosely coupled services that can be used for many different kinds of computation. One of the most important components of this ecosystem and the focus of this thesis is HDFS.

Apache HDFS [22] is a distributed filesystem designed to store very large files and allow for programs and frameworks written in different languages to operate on the data. It is successfully deployed by many companies and it is capable of running on very large clusters. Its design uses a single node, the namenode, to centrally manage metadata for the whole cluster and this creates a limitation for both the scalability and robustness as the system as a whole. To improve robustness it is possible to run a second namenode which will act as a hot-standby, ready to replace the primary in case of problems, and then either trigger a manual failover or configure the cluster for automatic failover. While both methods improve the reliability of the system, neither does so without significant complications. First, both methods require the cluster operator to run additional services, the JournalNodes, just to keep the standby namenode in sync with the primary. In case of manual failover, the cluster operator must then manually verify and trigger the operation in case of problems, which is a slow and error prone procedure. In case of automatic failover, however, the cluster operator is required to configure and manage a Zookeeper cluster and a ZKFailoverController process on every namenode which significantly increases the complexity of the deployment as a whole. Furthermore neither solution improves the scalability of the system because all RPCs are still directed to the active namenode. The way Apache HDFS increases scalability is to allow the same set of datanodes to store data for multiple namenodes, a configuration known as Federation. In federation, however, all namenodes sharing the storage cluster are completely separate and cannot share files which limits its utility to situations where the namenode is overloaded by different applications that require access to different datasets.

The limitations described also present challenges for operators that want to run their HDFS clusters in public cloud environments such as Amazon Web Services (AWS), Google Cloud Platform (GCP) or Microsoft Azure. Public clouds offer virtual machines that are executed on hypervisors that are shared with other customers, and performance and reliability tend to be unpredictable as a result. Cloud providers also tend to provide reliability at a more abstract level than on-premise deployments. Whereas in a typical data-center the failure domains are machine, rack, and whole data-center, cloud providers have machines, availability zones and regions. Single instances in most cloud providers are considered unreliable and expendable, therefore proper cloud software should be resilient to the failure of any one instance by distributing or replicating processes onto multiple instances. The HDFS expectation that the machine hosting the namenode is stable and with a consistent performance

is therefore difficult to achieve in cloud environments, even when considering an high availability setup. To solve this problem, most providers offer managed Hadoop that can automatically create and manage clusters and lets the customer focus on writing the data processing pipeline. This does not, however, solve the problem of efficiently managing HDFS clusters in the cloud.

To store a large amount of data on the cloud, the most popular approach is to use provider-managed cloud storage solutions such as Amazon’s Simple Storage Service (S3), Google Cloud Storage or Azure Blob storage. These systems allow customers to use a simple API to upload, list, and retrieve millions of blobs which can be several terabytes in size each. Furthermore these services seamlessly scale without any user intervention and are priced according to the amount of data consumed and the bandwidth used to operate on them. While it may sound tempting to adapt applications to use cloud storage systems and forego HDFS, and hierarchical filesystems entirely, these system do not offer the primitives associated with traditional (distributed) filesystems. First, these systems are actually key-value stores that associate a key, the path name, to a value, the blob. While this helps with scalability, different keys can be mapped to different storage machines, which makes common operations such as listing the content of directories much slower and with a linear time increase with the number of entries in the store. Furthermore, to maintain their favourable scalability characteristics and fault-tolerance, they sacrifice data consistency for system availability in the face of network partitions [5], resulting in an eventually consistent system [26]. Eventually consistent systems propagate changes in the system asynchronously which may result in client retrieving stale data, such as a listing of a directory missing some newly created files or a payload fetch which still retrieves a recently deleted object. To allow these systems to offer consistency semantics equal to those of HDFS, some cloud providers, such as Amazon for their managed Hadoop offer (EMR), build additional software that expose a HDFS-compliant API while managing metadata in such a way that the overall system appears to have consistent metadata. The trade-offs are that this approach introduces further components that need to be managed and scaled, it worsens performance of the overall system because of the wait times required for the changes to propagate through the system, and it introduces the possibility of the metadata store becoming inconsistent with the underlying data store.

To solve the mismatch of HDFS with cloud environments, the Hops project provides a scalable, cloud-ready, protocol-compatible distribution of HDFS called HopsFS [18]. HopsFS solves the biggest architectural problem that limits both HDFS’s scalability and its fault-tolerance, the storage of filesystem metadata in the namenode process main memory. Unlike HDFS, HopsFS stores the metadata in a distributed, consistent NewSQL database called MySQL Cluster, which can scale to hundreds of machines and store hundreds of terabytes of metadata. By moving the metadata in an external component, the namenode effectively becomes a (mostly) stateless process which can be easily replicated on multiple machines, all connecting to the same metadata storage cluster. Aside from a clear improvement in availability,

all of the HopsFS datanodes can answer RPC requests traditionally directed towards the HDFS namenode, enabling horizontal scalability at the namenode layer. As demonstrated in [18] on a workload trace provided by Spotify, the improvements brought by the increased scalability allow HopsFS to perform 16 times the number of metadata operations in the same amount of time. Furthermore, the filesystem metadata is now accessible to other applications in a transactional SQL database, allowing other programs to consume and extend the model for their own purposes.

While HopsFS successfully improves on many of HDFS’s architectural pitfalls, The goal of this work is to enable a single HopsFS filesystem to be geographically replicated in up to two regions for fault-tolerance, while allowing clients in each data-center to perform all operations. MySQL cluster fully supports geographical replication, but the resulting system propagates changes between regions asynchronously. The main objectives for this projects are therefore threefold:

- investigate the properties of asynchronous replication in the metadata storage layer (MySQL Cluster),
- define the changes in behavior to the filesystem as a result of this work, if any, and
- implementation of the required changes in HopsFS.

The expected results is for the two regions to appear to clients as a single filesystem, while allowing clients in one data-center to keep working if the other data-center is unavailable for any reason.

1.1 Outline

In order to gain an understanding of the topics described in this thesis, Section 2 introduces 1) the Hadoop Filesystem (HDFS), as the system upon which HopsFS is built, 2) the main alterations to HDFS to increase scalability and reliability of the system (HopsFS), and 3) MySQL Cluster as the metadata storage layer for HopsFS.

Section 3 describes other distributed file systems and their approach to metadata handling.

Section 4 discusses the various challenges involved in geo-replicating the metadata storage layer and the proposed solutions with particular regard to the trade-offs in terms of filesystem behavior. It also describes the work done on the HopsFS codebase to allow the practical implementation of such a solution.

Finally, Section 5 draws conclusions and describes areas worthy of further exploration.

2 Background

2.1 The Hadoop Filesystem

The Apache Hadoop Filesystem [22], or HDFS for short, is a scalable, distributed filesystem written in Java and originally developed for the Hadoop MapReduce computing framework. Its design is heavily inspired by that of the Google File System (GFS) [11].

The system is designed to handle very large files, typically several gigabytes to terabytes in size, by partitioning them in *blocks* and storing the blocks on different machines. To increase reliability, blocks are replicated multiple times, three by default, on different failure domains. In a typical deployment, a block saved on a given machine will have another copy in the same rack and a final copy off-rack. Due to the high storage cost of this replication scheme, HDFS 3.0 (set to be released at the end of 2017) optionally supports the use of erasure coding to lower the overhead while maintaining desirable retention characteristics. Using either of the replication schemes effectively eliminates the need for RAID schemes on individual machines, as data retention is assured by the distributed filesystem itself.

Files in HDFS are expected to be accessed in a sequential fashion both during creation and during read operations and are considered mostly static. The only modification allowed on a file is appending to the end and this operation can only be performed by one client at a time. During read operations, the system supports the `seek` operation to read arbitrary portions of the file but it is a very inefficient operation that severely impacts throughput.

Clients interact with HDFS using a set of language-independent remote procedure call (RPC) endpoints. The RPC system achieves language independence by using Protocol Buffers, a mechanism that allows the description of protocol messages and interactions (functions) in a high level language. A protocol buffer specification, in the form of one or more `.proto` files, is compiled to target language code and then compiled (or interpreted) along with application files. In HDFS, RPC is used both for communication between clients and the system and for communication within the system itself.

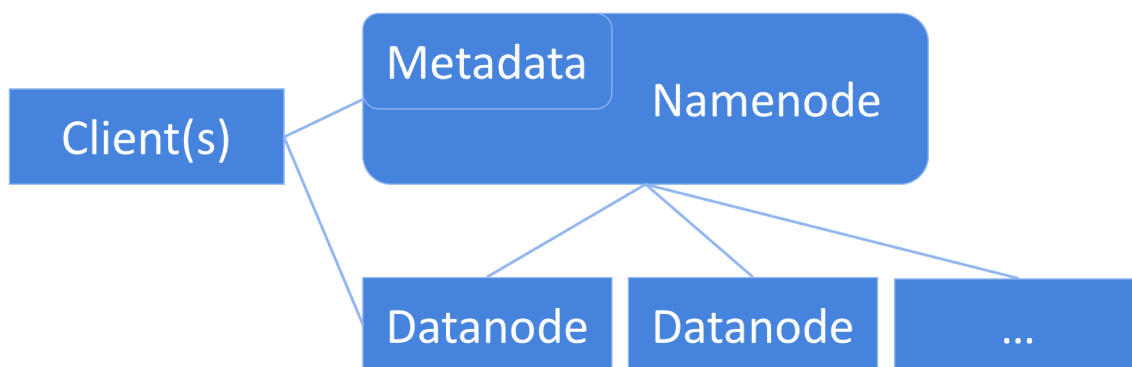
2.1.1 Architecture

The HDFS system contains three main components, as shown in Figure 1:

1. one *namenode*, with an optional hot standby copy,
2. a set of *datanodes*, and
3. *clients* interacting with the system.

The namenode is the central entity responsible for storing and applying modification to the system’s metadata. Metadata stored in the namenode includes

Figure 1: HDFS components



information on the state and health of the cluster, information on how files are stored and replicated and finally information on the state of operations being executed. Background threads in the namenode are also responsible for initiating periodic maintenance tasks such as block re-balancing and lease recovery. It is a server application written in the Java programming language and it accepts commands from clients via the RPC interface mentioned above. The namenode can also publish commands that will be executed by datanodes using the heartbeat mechanism. The heartbeat mechanism is described in detail in Section 2.1.6. In HDFS the namenode is a single point of failure and, in the event of namenode failure, manual fail-over to a hot-standby is required to restore service availability.

Datanodes are processes that handle physical storage of file blocks on disk. A datanode is oblivious to the concept of file and only stores blocks. To increase scalability, the datanode communicates directly with clients during read operations and with clients and other datanodes during write operations. It also periodically reports its health and the integrity of the blocks it manages to the namenode via heartbeats.

Clients are all programs that interact with the system to create, append or read files stored in the distributed file system. As mentioned above, clients may be written in any language for which a protocol buffer implementation is available. Depending on the operation type, client may interact with both namenode and datanodes to complete an operation.

While for some operations, such as listing the content of a directory, the client only performs a RPC call to the namenode, read and write operations require the client to contact both the namenode and datanodes. HDFS follows a single-writer multiple-reader semantic for files, which means that there can be an arbitrarily large number of clients reading a file but only one writing data to it.

2.1.2 Read pipeline

When performing a read operation on a file, the client begins by contacting the namenode to get the addresses of the datanodes containing the first block of the file. The list of datanodes holding a copy of the requested blocks is returned by the namenode sorted by proximity to the client requesting it according to the block placement policy. The concept of proximity and how blocks are distributed onto datanodes is explained in Section 2.1.4. The client then contacts the first datanodes to start reading the block. If the connection to the datanode fails at any point during the operation, the client connects to the next datanode in the list and remembers the failed datanode so that it does not try to attempt a connection to it during following block reads. If the checksum of the block read by the client is different from the expected one, the client communicates the checksum mismatch to the namenode before connecting to the next datanode in the list. Once the client fully reads a block, it contacts the namenode to get the location of the next block and starts the process again. In the actual implementation the client fetches several block locations with every call, further reducing the load on the namenode for client read operations. It is worth mentioning that, on recent versions of Hadoop, the client can sometimes bypass the datanode completely and read the data directly from the local filesystem. This operation is called a short-circuit local read. The operation is only possible when the client is co-located on the same machine as the data-node housing the particular block requested, but this is often the case with data-aware frameworks such as MapReduce.

2.1.3 Write pipeline

Writes on HDFS are performed by one client at a time. To maintain single-writer semantics, the client acquires a lease (essentially a lock) on every file it intends to write to. The lease is periodically renewed by the client for as long as it is writing to the file. If the lease is not renewed for a set amount of time, for instance because the client holding the lease crashed, it will expire. There are two types of expiration times: soft, set at one minute and hard, set at 60 minutes. When a lease expires after a soft timeout, it becomes available for other clients to claim through a procedure called *lease recovery*. On the other hand, when the hard limit for a lease expires, the namenode forcibly performs *lease recovery* by closing the file, thereby making it available for new clients. To decrease the network traffic generated by periodic lease renewal procedures on the namenode, a single lease renewal RPC call renews all the leases associated with the client performing the request.

Once the client acquires a lease, it contacts the namenode to get a new block id and a list of datanode to write data to. The client will only write data and control messages such as `close`, to the first datanode which will then replicate the message to the second datanode in the list and so forth until there are no datanodes left. Acknowledgments follow the same path in reverse, and are delivered in a single call to the client by the first datanode. Finally, when the client closes the file, the lease

is removed from the datanode and the block is closed by sending a close message through the pipeline. The system is able to recover from failures during writing by performing *pipeline recovery*. Depending on the phase where the failure happens, the client can require a new set of datanodes from the namenode or exclude some of the datanodes from the pipeline.

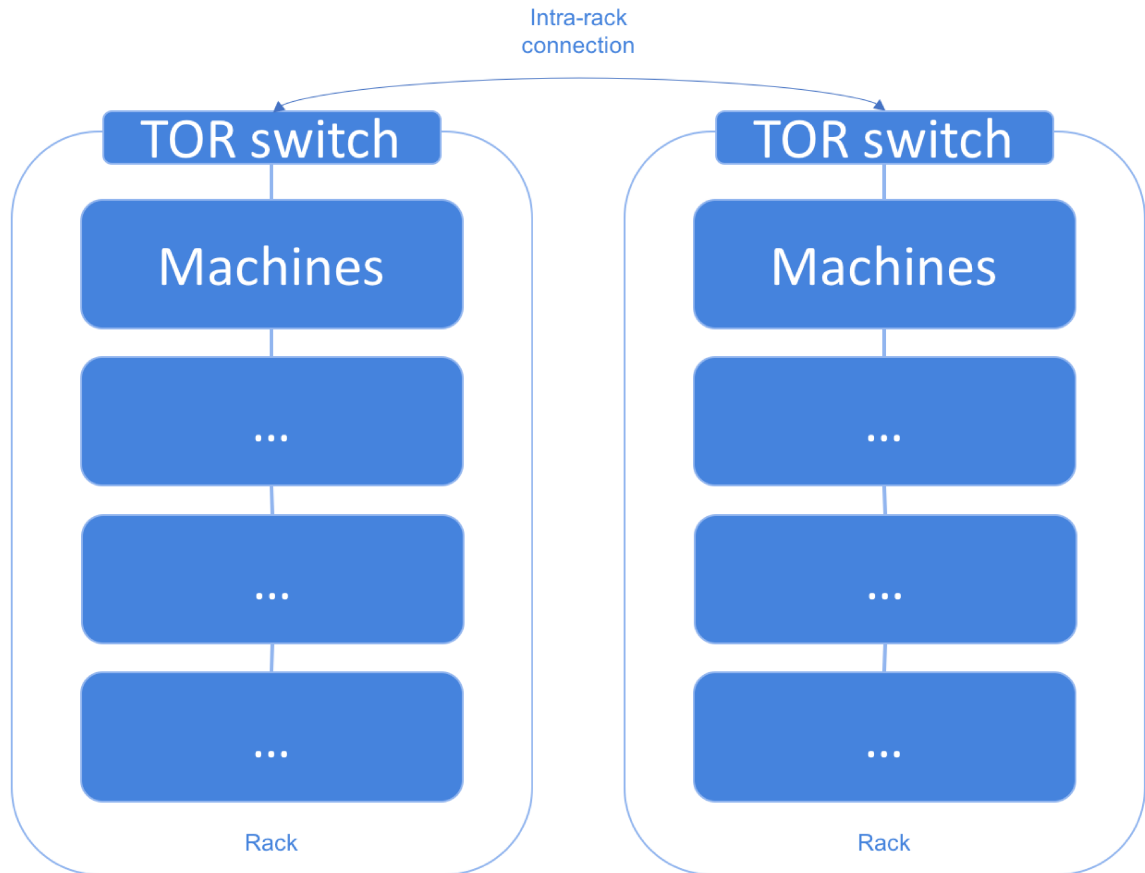
2.1.4 Block placement

Apache HDFS stores a configurable number, three by default, of copies of each data block. There are two primary reasons for this: i) to be able to withstand failure of a single data node holding the block and ii) to increase throughput by allowing different readers to read different copies of the same block. To fulfill both purposes it is important to consider the placement of blocks in the context of the overall network topology where HDFS is deployed. In a typical deployment, HDFS data nodes will be installed in server blades which will be installed in a rack. Machines in a rack will be connected to the network via a TOR (top of the rack) switch, which will provide both connectivity between machines in the rack and connectivity to the other racks via a higher level switch as shown in Figure 2. This type of deployment assumes that inter-rack connections are lower latency and have more bandwidth, while intra-rack connections are more expensive both in terms of bandwidth and latency. In this scenario, each rack represents a separate failure domain, as failure of the TOR switch, loss of connectivity to the higher level switch, or power failure effectively isolates all the machines in the rack from the network. To avoid the scenario where the loss of a single rack compromises the availability of all the replicas of a block, HDFS distributes the replica of a block across racks, provided that the cluster operator provides the namenode with information on placement of datanodes.

As part of the setup for a write pipeline, the namenode provides the client with a ordered list of datanodes to write data to. If datanode rack placements are configured in the namenode, datanodes are selected as follows:

- If the client is in the cluster, like a MapReduce job, and there is a datanode on the machine, the first block is placed on the same machine as the client.
- If, on the other hand, the client is not part of the cluster, the first block is placed on a random node as there is no way to compute a distance metric between the client and the datanodes.
- The second block is placed on a machine in a different rack than the first block.
- The third block is placed on another machine on the same rack as the second machine.
- The fourth block, if present, is placed on a different machine on the same rack as the first machine.
- If any more blocks are present, they are randomly distributed.

Figure 2: Machines in a rack, connected by top-of-the-rack switches



This distribution scheme minimizes the amount of inter-rack transfers necessary to spread the blocks on more than one availability zone.

2.1.5 Fault tolerance

Apache HDFS is designed as a modern distributed system and as such, datanode failure is treated as a routine event and handled transparently without the need for manual intervention. Datanodes periodically communicate their health status to the namenode by sending heartbeat messages. When a datanode stops sending heartbeat messages, the namenode considers it failed, and therefore it regards all the blocks stored on it as not accessible to the cluster.

To maintain the correct number of replicas for every block the namenode runs a background thread called the *replication monitor*. The replication monitor periodically checks the number of replicas for every block in the system and performs remedial action if the number is lower or higher than required.

- In case the block is **over-replicated**, the replication monitor schedules the deletion of the excess replicas in such a way that the remaining copies still

fulfill the block placement policy.

- In case the block is **under-replicated**, for example as a result of datanode failure, the replication monitor schedules the creation of new replicas according to the block placement policy.

The operations scheduled by the replication monitors are executed by datanodes and are transmitted to the relevant datanodes via the heartbeat mechanism.

2.1.6 Heartbeat

The mechanism used by the datanodes to communicate their status to the namenode is to send periodic heartbeat RPC messages to the namenode. The interval of time between heartbeats can be specified in the configuration file of HDFS but by default it is three seconds. Responses to heartbeat messages from namenode to datanodes also optionally contain commands for datanodes to execute, such as the deletion of blocks, the re-replication of a block to another datanode, and so forth. The main advantage of delivering commands as responses to heartbeats instead of sending commands from the namenode to the datanodes is that it allows a single namenode to manage a far greater number of datanodes, removing a bottleneck to scalability.

2.2 Scalability limitations of HDFS

A study conducted regarding the scalability limitations of HDFS [23] concluded that HDFS can manage an estimate 1 petabyte of data per gigabyte of metadata. While Apache HDFS can be scaled to manage multi-petabyte clusters, its single-active namenode design effectively limits both the amount of metadata and the number of queries per second (QPS) a node can process, to the largest machine it can be installed on. The amount of metadata is limited because they are stored as Java objects in the Java Virtual Machine (JVM) heap space, which is itself limited by the amount of main memory available in the machine. Furthermore, Java objects have a 8 to 12 byte header which is used by the virtual machine, increasing the memory requirements even further. The amount of QPS that the system can process is limited by both the number and speed of processors in the machine, the connection between clients (including datanodes) and the namenode itself, and the number of alterations that the system can apply to the metadata. Metadata objects are, in principal, only altered in two ways: from periodic processing by the namenode and as a consequence of RPCs invoked by clients and datanodes. Given that any number of these alterations can happen in parallel, the namenode protects the metadata with a global lock, the *FSNamesystemLock*, which can be acquired by an arbitrary number of threads in *read* mode, but requires exclusivity when acquired in *write* mode. All operations that require modification of the metadata are therefore executed serially, further lowering the amount of queries per second that the namenode can process.

Storing metadata in the JVM heap is also problematic due to increasingly long garbage collection pauses that freeze the entire process as the heap grows in size.

2.3 HopsFS

HopsFS [19] is a fork of Apache HDFS created with the explicit goal of solving the biggest scalability and availability limits that are inherent to the single-namenode nature of the system: i) the amount of metadata limited by the main memory of the machine running the namenode process, ii) the number and speed of processors in the machine, iii) the amount and latency of bandwidth between the namenode and its clients, iv) the coarse grained locking that requires a global lock to alter any piece of metadata, and v) the long garbage collection pauses can block the entire process for long periods of time as heap grows. To do so, HopsFS decouples the responsibility of managing metadata from the namenode and places it in a separate distributed system called MySQL Cluster. MySQL Cluster is a distributed, consistent (CP), in-memory relational database management system (RDMS) that can be operated and scaled independently from the hadoop cluster(s) it stores metadata for. Data stored in MySQL Cluster’s distributed storage engine (NDB), is divided between nodes participating in the cluster, allowing capacity to be increased by adding more machines to the cluster. Unlike more traditional RDBMS, where data is stored on disk and only loaded in memory at query time, data in MySQL Cluster is stored in-memory and persisted to disk as a recovery mechanism, allowing very fast query execution. By moving metadata to such a system, all of the issues regarding the memory limitations of a single system are automatically solved. The gains are even more significant with regards to the amount of queries per seconds that the system can manage. Decoupling metadata management from the namenode makes it a stateless component, which can be horizontally scaled and enables downtime-free failover, which is described in the following section. Furthermore, compared with the approach of having a global lock for all metadata, a relational system such as MySQL Cluster can have much more fine grained locks allowing, for instance, parallel modification of the information of any number of different files. This is possible because relational databases structure data as tuples in a table and each tuple (or set of tuples as defined by a query) can be separately locked. Unlike memory-managed applications, MySQL Cluster also does not suffer from garbage collection pauses, avoiding the pitfall in performance as the amount of managed metadata grows larger.

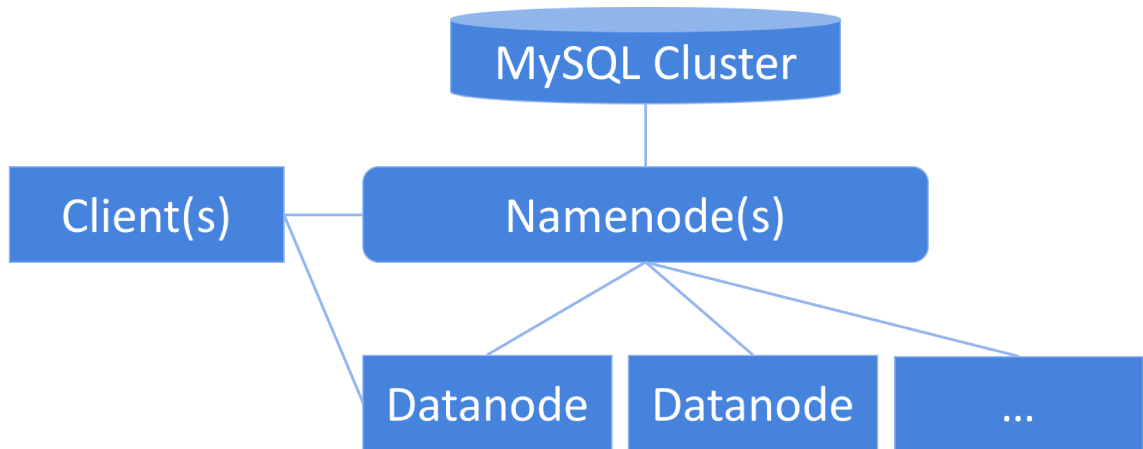
2.3.1 Multi-namenode architecture

The namenode, which is now a client of the metadata storage system, performs metadata queries, both in terms of reading and modifying, using an interface called the (meta)Data Access Layer or DAL, which internally connects to the distributed storage system in an efficient fashion. This allows multiple HopsFS namenodes to run in parallel, each serving a subset of the client requests to the overall system. The architecture of the resulting system is shown in Figure 3.

While most client operations can be directed to any one namenode, the block reports from datanodes and the daemon threads must be handled carefully. In a Apache HDFS namenode, background daemon threads are responsible for a variety

of functions including block re-balancing and lease recovery. If allowed to run on every node in the cluster, these daemon threads would interfere with each other's work, causing unpredictable results. To maintain the behaviour of Apache HDFS, HopsFS must guarantee that only one node in the cluster executes daemon threads at any given time. To this end, all namenodes active in a cluster participate in a leader election algorithm, explained in detail in Section 2.3.2, that will elect a single namenode to be leader at any given time. Additionally, the leader election procedure serves as *failure detector* within the set of namenodes, allowing clients to retrieve a list of live nodes. Datanodes fetch an updated list of active namenodes before every heartbeat message, allowing them to keep an updated view of participants in the cluster and the current leader node's identity. As metadata about the state of datanodes, the action they should execute, and the blocks they contain is stored in MySQL Cluster, any namenode can process heartbeats or block reports from any datanode. Heartbeats are sent by the datanodes to different namenodes in a round-robin fashion, while the namenode to send block reports to is obtained by performing a RPC call to the leader. The leader will assign the block report to a namenode that has the capacity to process it and has the lowest number of block reports to process at the moment where the RPC is sent.

Figure 3: Architecture diagram of HopsFS. The metadata storage engine is provided by MySQL Cluster



2.3.2 Group membership service and leader election

As previously mentioned, HopsFS maintains a list of active namenodes in the system and elects one leader which executes functions critical to the proper functioning of the overall system. Apache HDFS, when configured in High Availability mode (HA) also require such a system and leverages Zookeeper to perform these functions. HopsFS eliminates the reliance on Zookeeper by leveraging the metadata storage system as shared memory and implementing reliable failure detection and leader election [18]

using MySQL Cluster distributed engine (NDB). The failure detector implemented therefore complies with the requirements for the weakest failure detector for solving consensus [7].

2.3.3 The (meta)Data Access Layer

The Data Access Layer, DAL for short, is the Hops component that allows the delegation of metadata handling to the MySQL Cluster database. To achieve this, the component provides two distinct pieces of functionality: i) management of the life-cycle of database connection, including various optimizations to reduce network round trips and, ii) abstractions that allow engineers working on Hops to convert all memory metadata accesses in database operations in a convenient way. Formally, the data access layer provides only the interfaces that Hops itself uses to describe accesses to metadata, delegating the implementation of database access to a further library that provides concrete implementations. Given that in Hops only one such implementation exists (*hops-metadata-dal-impl-ndb*), this chapter will consider both DAL and DAL-implenentation as a whole without making the distinction explicit.

Connection management The DAL provides management of the life-cycle of database connections to a MySQL Cluster cluster. Specifically, upon configuration, the DAL creates two persistent connectors to the same MySQL Cluster cluster: one that connects to NDB using the native protocol and the ClusterJ Java library and one that connects to SQL nodes using the standard JDBC MySQL driver. The reason to use both a SQL driver and the native NDB protocol is that, while the NDB protocol is very fast at performing primary-key based operations, more complex operations such as joins and deletes are not supported and can only be executed through the SQL nodes. Given that the performance of Hops is determined mostly by how fast it accesses metadata, the DAL must be as performant as possible. To achieve better performance, this part of the DAL library implements optimizations aimed at reducing connection overhead, thus allowing a greater number of operations per second. The main technique for this is connection pooling, which associates each open connection to a thread that will use it for all operations. By allowing a thread to re-use the same connection for all operations, the overhead of opening the connection is effectively eliminated. Connections are only closed in case of shutdown of Hops or errors on the connection itself, in which case the connection is re-opened at the next use. The connector itself is provided to clients as a global object, accessible to any component that requires it and it is initialized and configured in `Namenode.java`.

Database access Aside from managing database connections, the DAL provides abstractions that are used to convert all memory metadata accesses into accesses to the metadata storage layer. The main abstraction provided is the *request handler*, a structure that provides information on the type of operation being performed (the `OPCODE`) and the procedure to execute on the metadata, whether read-only or

a modification. When the handler is executed it performs the procedure in the context of a database transactions where errors will be handled by rolling-back the transaction itself, guaranteeing atomicity of metadata modifications.

The DAL provides two types of request handlers, the *lightweight request handlers* which execute the operations as described above and the *transactional request handlers* that apply most metadata modifications in memory before committing them to the database with the goal of reducing database round-trips.

In a *lightweight request handler*, shown in Listing 6, every modification to the metadata is concretely executed as a database query, causing a large number of network operations. In case of transaction handlers with very large number of modifications, the network round-trips rapidly become the performance bottleneck.

To increase performance in modification heavy handlers, *transactional request handlers*, shown in Listing 5, operate in a different way with the goal of reducing network operations to a minimum. Transactional request handlers introduce a lock acquisition phase which is executed before the code for the transaction itself. In this phase, the DAL acquires locks on all the specified rows and materializes them as objects in the DAL memory. Upon execution, the handler operates on the in-memory representation of the objects either by modifying or deleting existing ones or by creating new ones through the **EntityManager** class. At the end of the perform phase, the objects are divided into four categories: 1) unmodified, 2) created, 3) deleted, 4) modified, and the required operations are executed in batch on the database. The perform phase is, therefore, still executed in the context of a database transaction, with the possibility of rollback in case of errors, but all operations on the database are executed at the end. Given that all of the materialized rows are locked for the duration of the database, there can be no conflicts upon commit at the end of the handler. Note that the handler can request read locks as well as write locks and, in that case, the rows locked in read mode cannot be modified.

Replacing all memory accesses with transaction handlers which acquire the minimum amount of locks required to perform the operation, HopsFS achieves a much more granular level of concurrency compared to the in-memory global lock, which allows it to execute a much greater number of concurrent operations.

2.4 MySQL Cluster

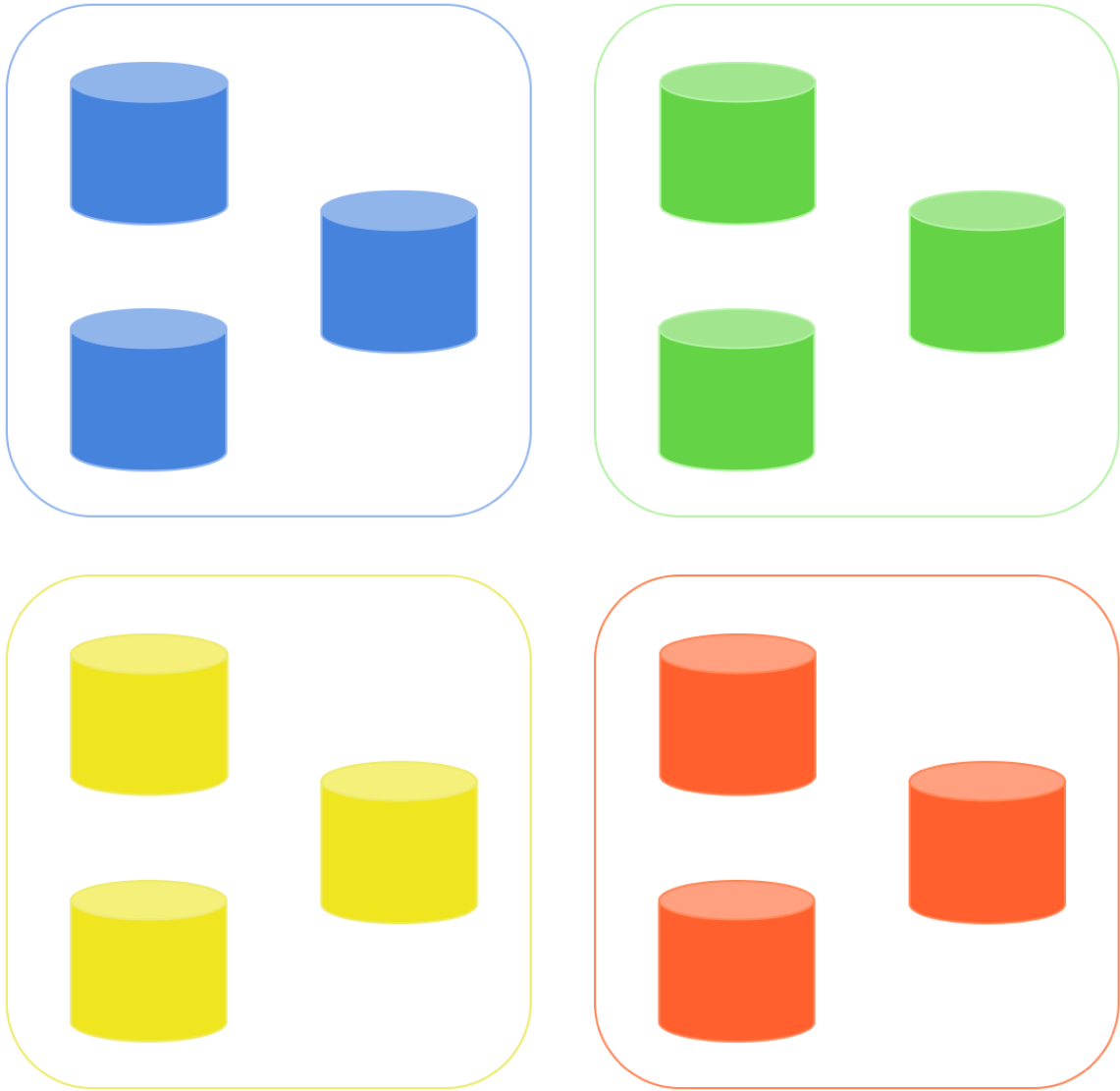
HopsFS delegates the storage and querying of metadata to an external database called MySQL Cluster. MySQL Cluster is a in-memory, distributed, consistent, relational database management system (RDBMS) currently developed by Oracle. The sources for the system are released under the terms of the GNU General Public License (GPL), but development is driven by Oracle without external contributors. MySQL Cluster is the combination of the MySQL relational database management system and a distributed table storage system called Network DataBase (NDB) [21, 1]. As such, any program that is able to use MySQL as the database can be migrated on a MySQL Cluster system with minimal modifications. In this system,

data storage and query processing are handled by NDB while MySQL nodes act as front-ends by parsing and interpreting SQL queries.

2.4.1 Network DataBase

NDB is a in-memory share-nothing database which runs as a distributed application on a set of nodes. NDB can either be used on its own or as part of the MySQL Cluster system, in which case a set of MySQL nodes act as clients, accepting client connections in the MySQL wire protocol, parsing the SQL and executing them using the native NDB protocol. Its share-nothing architecture relies on message passing between nodes participating in the cluster instead of disk or memory sharing like other distributed databases. Furthermore, unlike traditional databases, NDB holds all data for tables in main memory. Each NDB cluster contains two sets of nodes: i) data nodes, `ndbd` and `ndbmtd`, which contain the data for tables and participate in queries and commit protocols ii) management nodes, `ndbmgt`, which provide parameters to data nodes in order to form and maintain clusters and, typically, act as *arbitrators* during split brain protocol. In NDB tables are divided into *partitions* and partitions are assigned to node groups. In order to compute the partition any row belongs to, the default strategy is to take the hash of the primary key modulo the number of node groups, though this behaviour can be modified at table creation time. The system can be configured, by tuning parameters in the management node(s), to replicate each data partition multiple times. Aside from creating redundancy in case of data node failure, multiple data nodes will be able to serve reads for the partitions stored in the node group, linearly increasing the number of read queries per second that the system can serve. If replication is set to one, only one copy of the data is available in the system and, if the data node storing the partition fails, the data is permanently lost. If replication is set to a value higher than one the cluster is divided in logical units called node groups. The number of node groups N_g formed is controlled by $N_g = \frac{N_t}{R}$ where N_t is the total number of data nodes in the system and R is the replica factor. This also implies that, by setting the replica factor R , the number of datanodes in the cluster must necessarily be a multiple of R itself. Every write for a data partition will be replicated on every node in the assigned replica group so that, in case of failure of any node, the system will still be able to serve all the requests, albeit at a slower rate. Figure 4 shows an example scenario for a cluster with $N_t = 12$ and $R = 3$.

Figure 4: Node group configuration for cluster with $N_t = 12$ and $R = 3$



In order to provide clients with synchronous transaction, which maintain data consistency, NDB employs a two phase commit protocol (2PC) [4] when committing transactions. A 2PC protocol run is divided into two phases: 1. a *prepare* phase, where nodes execute the operations specified in the transaction and prepare for a commit 2. a *commit* phase, where nodes commit the operation. After a successful *commit* phase, the data is durable and cannot be rolled back by any node. To initiate a transaction, a client contacts any data node. Data nodes contain a *transaction coordinator* process, which identifies the partitions involved in the transaction and initiates the 2PC protocol. If any node involved in the transaction fails, the transaction coordinator will abort the transaction after a short timeout of 5 seconds by default. Clients of the system are expected to handle transaction failure, and the typical strategy is for the client to retry the transaction at a later time.

To detect failures, data nodes arrange themselves in a virtual ring and send heartbeat messages to the next node in the circle. If one node fails to acknowledge a heartbeat three consecutive times, it is considered failed and the cluster enters a split brain protocol, during which it is not able to accomplish any work. The purpose of a split brain protocol is to identify and designate a subset of nodes in the cluster that still have a complete copy of all partitions and can therefore continue to function, albeit in a degraded fashion. To identify the *sub-cluster* that continues to function, each sub-cluster executes a series of checks:

- if the sub-cluster includes all nodes from any node group, this is the only possible functional sub-cluster and can **continue to operate**,
- if the sub-cluster does not contain at least one node in each node group, this sub-cluster is not functional and can **shut-down**,
- if the above conditions are both false, there is more than one functional sub-cluster, **defer the decision to an arbitrator**.

In order to avoid a split brain scenario, where two or more subsets of the cluster continue to apply diverging modifications to the data in parallel, the arbitrator allows only one of the functioning clusters to continue. The arbitrator select only one cluster by only replying positively to the first subset contacting it, instructing all following sub-clusters to shutdown. If a sub-cluster cannot contact the arbitrator within a predefined amount of time, it shuts itself down, guaranteeing that *at most* one sub-cluster will be live during split brain protocol. The role of arbitrator can be fulfilled by both management nodes and SQL nodes, which are explained below, but management nodes have higher priority compared to SQL nodes. Given that, without an arbitrator the whole cluster fails upon failure of a single node, more than one node can fulfill the role of arbitrator, albeit not at the same time. If an arbitrator fails during normal cluster operations, the datanodes agree on another, selected from a list of arbitrators and associated priorities. This list is specified at cluster configuration time and can only be updated by a management node by applying a configuration change. All the nodes that shut down as part of the split brain protocol must re-join the cluster through a management node upon restarting. It is worthy to note that, while the cluster is effectively able to access all data in the aftermath of a split brain protocol, the reduced capacity of one or more node groups can cause load spikes for the nodes that are left.

NDB data nodes store all partition data in main memory. In case of data node shutdown, either planned or unplanned, all the partitions on the node are lost. While a restore procedure can, in principal, fetch copies of the partitions from other data nodes in the same node group this will either 1) take a very long time if the goal is to minimize the impact on the other working nodes in the node group 2) consume most of the bandwidth on the working nodes left in the node group, further worsening the strain caused by a reduced number of nodes in the group . To limit the amount of bandwidth required by a node restore procedure, data nodes periodically checkpoint

state to durable storage. Checkpoints to durable storage are achieved by periodically flushing to disk a log, called the *REDO* log, which contains all the transactions committed between the last flush and now. To obtain a consistent snapshot of the system, one where no committed transactions have a dependency on uncommitted transactions, all the data nodes coordinate using a global checkpoint protocol (GCP). GCP enables data nodes to flush REDO logs in such a way that the resulting snapshot is globally consistent. Due to the way the REDO log stores changes, without any other mechanisms to limit its growth, the on-disk snapshot would effectively grow without bounds. To prevent this, data nodes also run a local checkpoint (LCP), which persists a snapshot of the state of the partitions in the system to durable storage. With a complete snapshot available, the node can discard the portion of REDO log coming before the local snapshot, as in case of restore the local state is used to reconstruct the in-memory state of partitions. In case of restore a data node i) loads the most recent local checkpoint ii) applies all the transactions from the REDO logs iii) requests the newest transactions from other nodes in the group . By using a combination of LCP and REDO log, the node therefore reduces the amount of data to transfer from a complete snapshot to only some transactions. The above techniques only help if the node starts the restore relatively promptly as the local snapshots will be quickly invalidated when other nodes erase their REDO logs to create a newer LCP.

2.4.2 SQL Nodes

SQL nodes are MySQL server instances that can create and interact with tables using the `NDBCLUSTER` engine. Tables created with such an engine are stored on a NDB cluster. SQL nodes participate in the cluster as clients and can also be elected arbitrators, though usually with lower priority compared to management nodes. Any number of SQL nodes can be connected to the same NDB cluster to better distribute the load and increase the availability of the service for MySQL clients. While MySQL server itself is modified to connect and participate in an NDB cluster, clients can connect using standard MySQL client libraries, which allows unmodified applications to take advantage of the scalability and performance benefits of MySQL Cluster.

2.4.3 Isolation levels and locking

NDB only supports transaction isolation level `READ_COMMITTED`, which guarantees that uncommitted values will never be read. While reading an uncommitted value is impossible, NDB implements `READ COMMITTED` on a row-by-row basis, which makes it entirely possible for a transaction to commit some updated values while another transaction is reading them, resulting in the second transaction observing a subset of values before the transaction and the rest after. In concrete terms, whenever a data node receives a read request, it will always return the most recently committed value.

In order to obtain stricter forms of serialization, NDB allows transactions to set row-level locks, both shared and exclusive, which are released upon transaction commit or roll-back. Row level locking is the fundamental mechanic that allows HopsFS to provide consistent filesystem operations to clients as well as enable the use of NDB as shared memory for the leader election processes.

2.4.4 Geographic clusters

While a single NDB cluster offers strong consistency and good performance in the context of a data-center network, MySQL cluster also offers a variety of options to extend a cluster to more than one data-center.

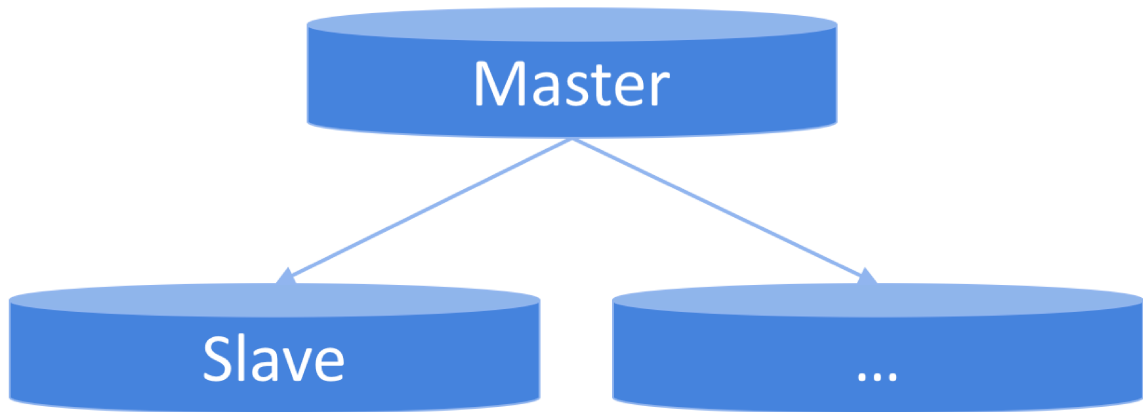
The obvious solution to the problem of geographic clusters would be to set up data nodes in all locations and join them in a single cluster the same way it would be done in a single data-center. This is not, however, a viable solution in most scenarios due to both assumptions in NDB and in the way data-center networks are designed. NDB assumes all data nodes are running in a interconnected network where the latency and bandwidth to contact any other node in the cluster is generally constant, and it leverages this assumption to provide on-line transaction processing typical of a online transaction processing system (OLTP). Timeouts for transactions are very short (5 seconds by default) and the failure detection mechanic is also sensitive to increased latency as it may confuse a latency spike with node failure. A multi-data center network, on the other hand, would have very low latency and high bandwidth between nodes in the same data-center but comparatively higher latency and lower bandwidth between nodes in different locations. Furthermore, connections between nodes in different locations would all share very few channels, while internal data-center networks tend to be very well connected.

The better alternative for geographical replication in MySQL Cluster is to use asynchronous replication features built into MySQL. Asynchronous replication techniques are used in standard SQL databases such as MySQL and PostgreSQL to achieve a variety of functions such as performing analytics without compromising the database running online processing or creating standby replicas, ready to be promoted should the master fail. In asynchronous replication a node referred to as master publishes a log-like stream of operations it executed, in the order they were executed. A set of other nodes, referred to as slaves or followers, consume the log of operations and apply the same operation to the local representation of the data. The state of followers is therefore consistent with the state of the master at some point in the past, even in case of master failure. This technique is asynchronous because the master does not wait for followers before reporting success to the client, thus maintaining the low latency operational characteristic of a online database. High latency only affects this process in that the state of followers on high-latency links will lag further behind the master's state. In the MySQL Cluster system, asynchronous replication and the conflict detection and resolution functions associated, which are illustrated later, are delegated to SQL nodes which propagate events to other SQL

nodes and apply the necessary changes to the tables in NDB. This technique is used to implement many different systems, depending on the application's requirements.

Master-follower A very common technique used to implement replication in systems where write traffic is mostly constant but read traffic is variable, is to have a simple single master/multiple follower system (shown in Figure 5), where write traffic is only directed to the master and read traffic can go to any of the available followers. Such a system offers eventual consistency semantics for reads as followers may serve an outdated view of the master's state. If a stronger consistency level is required, for instance by a client which needs a read-your-writes level of consistency, the only solution is to perform such reads on the master node. While this system is relatively simple to implement and operate it is unsuitable for systems where write traffic (or consistent read traffic) increases to the point where it can no longer be handled by a single master.

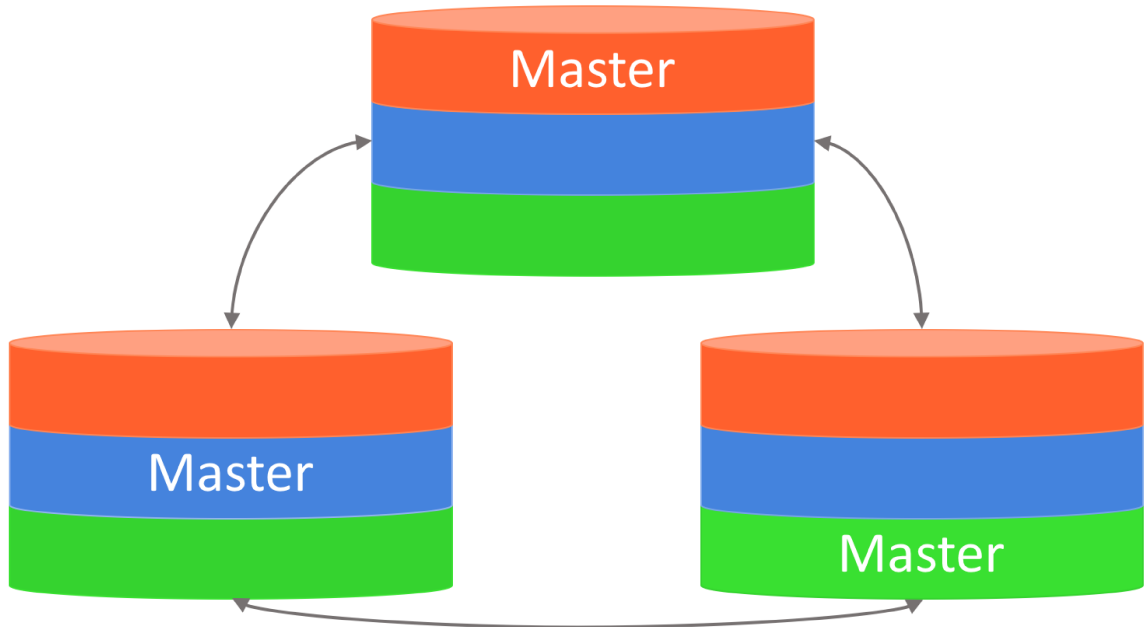
Figure 5: Master-follower topology



Balanced master-follower A solution which allows better scaling of write traffic is the so called Balanced master-follower or Partitioned Active/Active scheme. In this scenario all of the nodes are configured as master for some partition of the data, while they act as followers for all other partitions, as shown in Figure 6. Given that master or follower in this case is ambiguous we refer to nodes as either active or passive for a given partition. Data partitioning can be done at the database, table or even row level, depending on the application. The application is also responsible for selecting a partitioning scheme that distributes load evenly on all partitions. Aside from increased complexity in the application, the system also requires a sophisticated routing system which guarantees that write operations always access the correct node which is active for the partition. Fail over also requires care as one of the nodes may become overloaded if it becomes active for too many partitions. While this system improves on the write scalability limits posed by the master-follower design, it still

requires complex routing to perform writes. The routing is necessary so that, when two applications try to write conflicting data, the writes are effectively serialized at the active partition, which will reject one of the two writes thus implementing consistent writes.

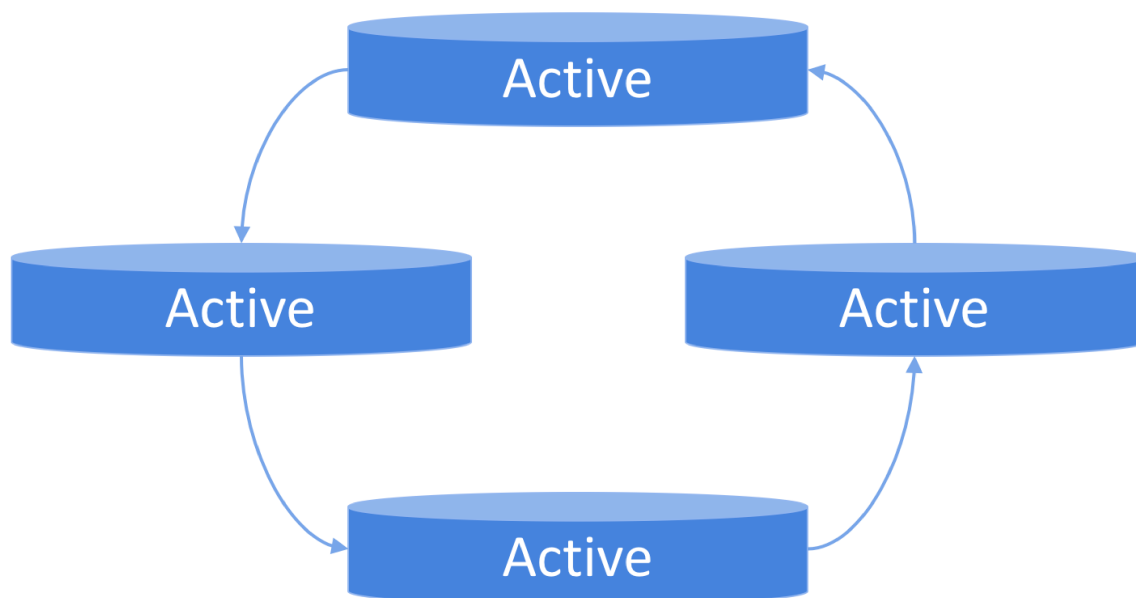
Figure 6: Balanced master-follower topology



Optimistic Active Active In many parallel applications, conflicting operations are relatively rare, when compared to the total volume of write traffic. While pessimistic systems such as master-follower or balanced master-follower simply make it impossible for write conflicts to happen, at the expense of limited write scalability or more complex routing logic, optimistic systems handle conflicts as they arise. The core principal behind this assumption is that, while handling conflicts may be expensive, it is a very rare operation and the total cost is therefore inferior to the constant cost of serializing writes in a master/active node. An optimistic active active replication system, shown in Figure 7, allows write traffic on any node part of the cluster. When a transaction is committed, it is propagated asynchronously to the other nodes which will check for conflicts. If a conflict is detected it is rolled back. Clients of such a system must accept that they may read data which is stale or not durable because it may be rolled back, but the overall system is much more scalable and does not require complex routing as both reads and writes can be executed at any node.

Hybrid Optimistic Active Active A hybrid optimistic active active system, shown in Figure 8 works like its purely optimistic counterpart with the additional

Figure 7: A topology with two MySQL Cluster clusters replicating with active to active configuration



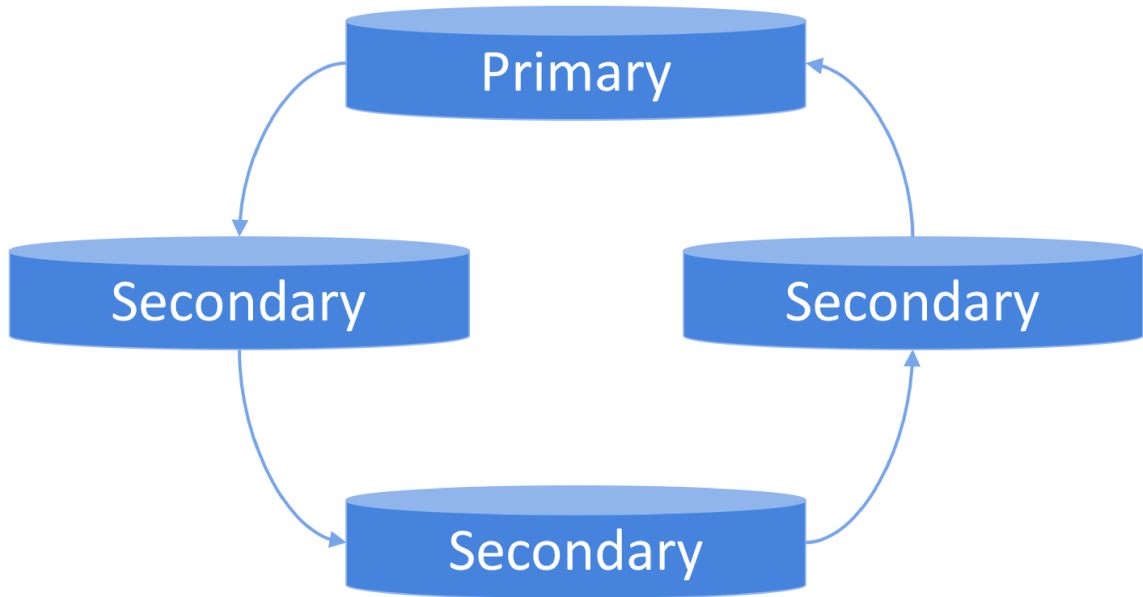
advantage of defining an active node for each partition. Writes performed on the active node for the partition will be durable and will not be rolled back, while writes on other nodes will be rolled back if they are in conflict with those originating on the active node. This system still enables the benefits of the optimistic active active scheme, by routing traffic to the passive nodes while allowing applications that require write consistency or read-your-write semantics to be routed to the active node for the partition.

2.4.5 Dealing with conflicts in optimistic configurations

Optimistic systems require the ability to detect conflicting updates and reject them in order to avoid diverging replica states. Rejection is not, however, the only strategy to avoid diverging replica states, as many conflict can instead be merged by the application depending on specific application logic. In the simple case of a numeric value which is set to 1 and is incremented by 1 on each replica, rejecting one of the updates would result in a value of 2 but an application could merge the conflicting updates, producing a correct value of 3.

MySQL Cluster provides a sophisticated mechanism to both detect and resolve (merge) conflicts in a optimistic active active replication scheme. All of the available methods are documented in the MySQL Cluster documentation website [2], but this section gives an overview of the `NDB$EPOCH` and `NDB$EPOCH_TRANS` merging functions, which can detect and resolve conflicts at a transaction granularity. The `NDB$EPOCH` functions can only be applied in the context of a hybrid optimistic active active

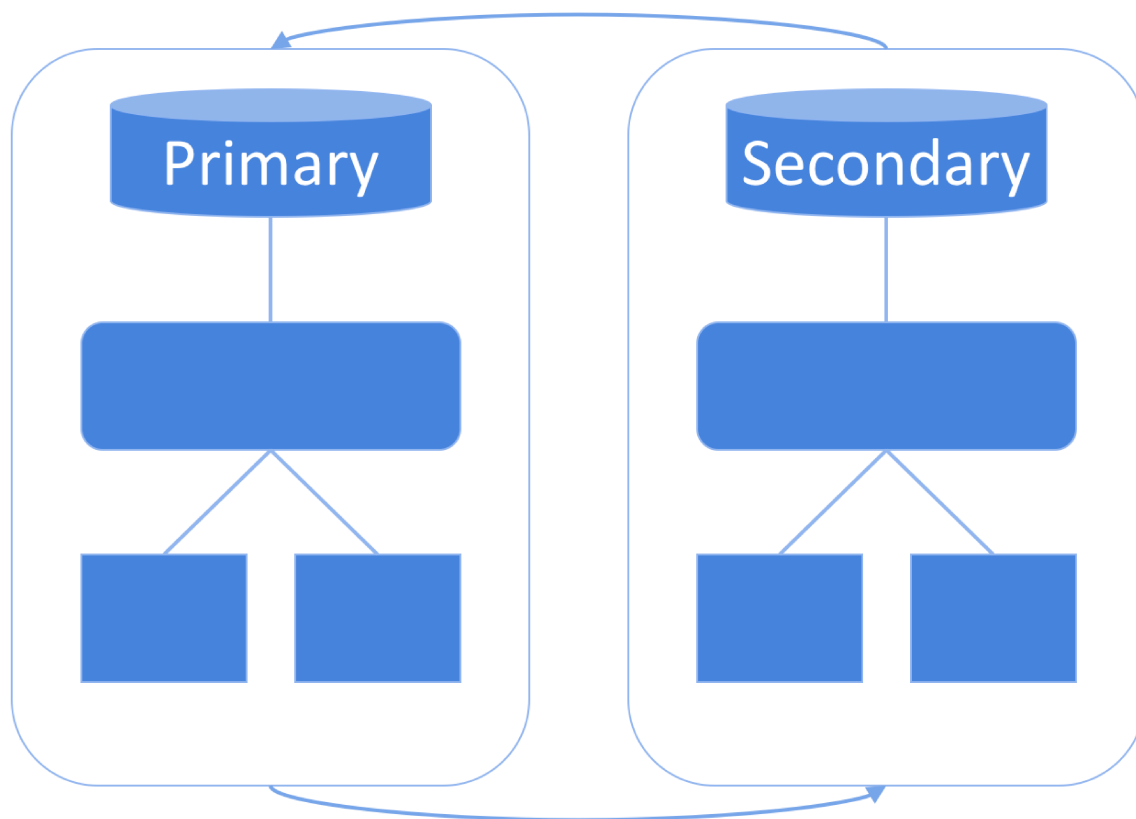
Figure 8: Hybrid optimistic active to active topology



system with two replicating clusters as shown in Figure 9. Partitioning of data in this case can be either at table or database level, but this chapter will assume only one database configured as active on one cluster and passive on the other. We will refer to the active cluster as *primary* and to the passive cluster as *secondary*. As previously mentioned, the hybrid optimistic setup implies that all writes performed on the primary cluster are durable, while all writes on the secondary cluster may be rolled back. As shown in Figure 9, one SQL node per cluster acts as both master publishing changes for the other cluster to consume and follower fetching changes from the other cluster, though this is not the only possible set-up. The resulting setup is a circular replication scheme, which is necessary for the conflict detection and merging functions to work.

Conflict detection is performed on the primary cluster, as any conflicts caused by the primary cluster, which would be handled by the secondary cluster, are automatically “won” by the primary cluster. When a conflict is detected the master emits events that re-align the secondary cluster with the state of the primary cluster, in effect rolling back any conflicting changes. Here `NDB$EPOCH` and `NDB$EPOCH_TRANS` diverge in behaviour: while the former only re-aligns the rows causing a conflict, the latter rolls-back the entire transaction and any transactions that depend on it before applying the master state. Aside from aligning the secondary state to that of the primary, MySQL Cluster also provide a powerful mechanism for applications to merge conflicting state: the *exception tables*. Exception tables are created on the primary cluster with the same name as the tables conflict resolution is enabled for but with the additional `$EX` suffix. These tables include a variety of columns

Figure 9: A topology with two MySQL Cluster clusters replicating with active to active configuration



including all of the columns of the primary key for the original table, some columns containing information on the conflict itself, and optionally any other column from the conflicting table as well as meta-columns containing values before and after the exception itself. If such tables are present when conflict resolution is enabled, every time a conflict is detected and the secondary state is re-aligned, the value for the aligned rows in the secondary is saved in the exception tables. The application can then poll the exception tables to implement any desired merging behaviour. Going back to the example of two concurrent transactions adding one to a value in a row, an entry for the secondary would be added in the exception table and the application would then be able to increment the counter again and delete the row from the conflict table to merge the results and obtain the correct value.

Finally, it is important to note that conflict detection is sensitive to replication latency between the two clusters and the longer replication latency is, the more likely it is for the clusters to produce conflicting transactions.

3 Literature review

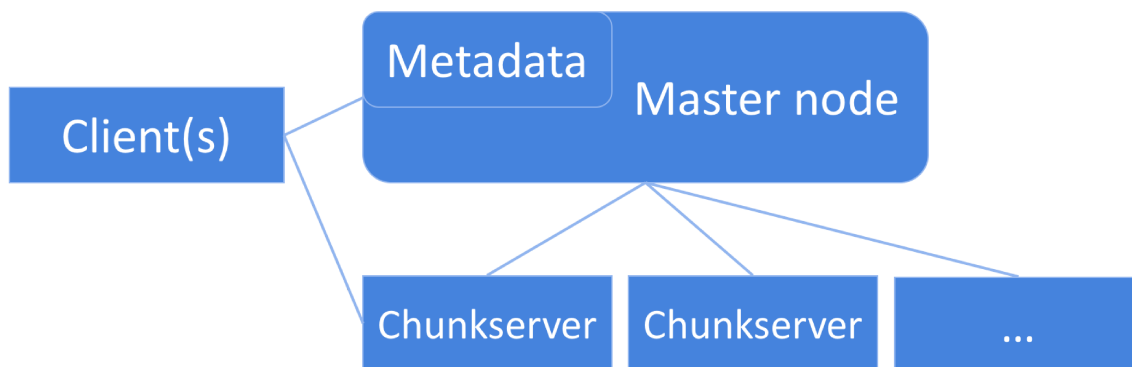
This chapter analyzes the prior art in the space of distributed file-systems with a particular focus on the handling of metadata. Given that Apache HDFS has already been described in detail, it is omitted from this section and three others are described instead.

3.1 The Google File System

The Google File System (GFS) [11] is the distributed file-system employed internally by Google until it was replaced by Colossus, another internal system (which remains unpublished at the time of this writing). The design of GFS and MapReduce [10] directly drove the implementation of the Hadoop platform and, as such, there are many similarities between GFS and HDFS.

3.1.1 Architecture

Figure 10: GFS architecture diagram



GFS is designed as a distributed non-POSIX-compliant file-system for distributed applications and it is built by clustering a set of general-purpose machines as shown in Figure 10. It is a distributed system with three distinct roles:

- a *master node*, which maintains metadata on files and the state of the system,
- a set of *chunkservers* which perform storage of data on disk and directly interact with clients, and
- clients which read and write data from/to the system.

In the system files are divided in chunks, which can grow up to 64 megabytes before another chunk is created. The master stores both the hierarchy of files and directories, as well as information on which chunks form any given file. Multiple

copies of each chunk (three by default), referred to as replicas, are managed by different chunkservers operating in different failure domains. Chunks are stored as regular files in the chunkserver's local filesystem. Replication is employed to maintain data availability in the face of chunkserver failure, as well as to provide a limited form of load balancing by allowing different clients to read different replicas of the same block. The location of replicas is stored, alongside all other file metadata, in the master main memory.

As in HDFS, the single master node is the main scalability and reliability bottleneck for the system and, as such, many techniques employed in GFS have to goal of reducing interactions with this node to a minimum and increase its reliability.

Reads The chunk size is purposefully very large compared to local file-systems, as any read request must first contact the master to learn the location of block replicas. When responding to such a query, the master sends location data about several following blocks in the file and this information is cached by clients for a short period of time, to avoid excessive master involvement in sequential read scenarios. After learning the location of blocks, the client can complete the read operation with no further involvement from the master node, by contacting the relevant chunkservers directly.

Writes In order to minimize the master's involvement in write operation, the systems grants block leases to chunkservers. When a client requests to mutate a block, either by writing or appending to it, the master selects three (assuming a default replication factor) chunkservers to receive the mutation: a *primary* and two replicas. The primary is granted a new lease to alter a block with data received by the client for as long as the lease is valid, unless it was already holding a lease for the specified block. The chunkserver can periodically renew the lease by contacting the master node if it is still receiving data from clients. At this point, the client pushes the data to all the chunkservers and waits for a confirmation that all of the replicas received the data. When a confirmation is received the client contacts the primary and requires a write operation. The primary serializes all writes (there may have been concurrent writes) and then applies them to the file stored on the local disk. After it applied the state to the local file it contacts the replicas and asks them to write the changes in the same order. Once it receives confirmation from all replicas, the write is finally acknowledged to the client.

Reliability In order to increase the master reliability, all metadata mutations are persisted on a disk-based log which is both kept on the local machine and replicated to a number of others. Client operations that involve metadata modifications are not acknowledged before this flush is completed. Given that such a mutation log would grow without bounds, it is periodically compacted into a snapshot. The snapshot is created by serializing the current master metadata on disk in a format that can be directly used to restore a master without any parsing. When a master needs to

recover from a crash, first it loads the most recent snapshot, then it applies all the modifications in the log before accepting any client queries. This mutation log is also used to keep several “shadow masters” up-to-date with the state of the master. Shadow masters cannot perform any metadata mutation but they can serve read requests, even in the event of master failure. This mechanism is used both to scale the system further by delegating reads to a shadow master and to grant a read-only service during recovery of the master.

3.1.2 Chunk management

While client-initiated operations are optimized to involve the master infrequently, some periodic operations are necessary to keep the cluster in a healthy state. As previously mentioned, GFS uses replication to maintain data availability in the face of chunkserver failure. However, if chunks with fewer than three replicas are not replaced, eventually all replicas will be unavailable. To prevent this, the master periodically queries all chunkservers for the list of all chunks they are holding and instructs the chunkservers to re-replicate the ones with fewer than the specified number of replicas (three by default). Finally, chunk deletion is also handled asynchronously, if the master detects any chunks that are not tracked in its memory metadata, the corresponding chunkserver is instructed to delete the chunk from disk.

3.2 Windows Azure Storage

Windows Azure Storage [6], WAS for short, is a system developed by Microsoft for the Azure cloud platform and it is in production since 2008. Unlike a classic distributed file-system where the only primitive offered is the file, WAS offers three different primitives to clients:

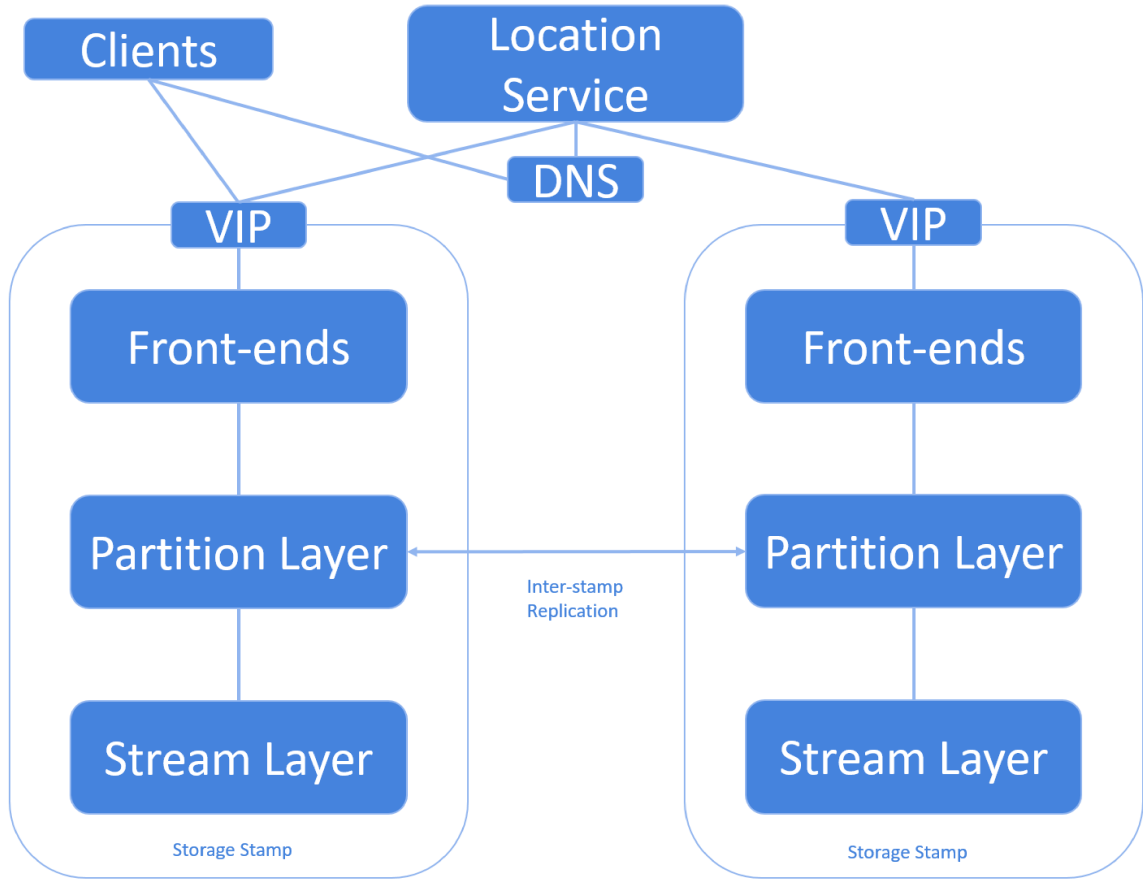
- a blob storage to process unstructured data,
- a table storage to process structured data in tuples, and
- a queue system to build message-passing based systems.

Typically data flowing into and out of the system is saved in blob storage, sent to workers as queue items and processed using the table store.

WAS was designed around a global namespace which allows clients to access data in any deployment in the world using the same addressing scheme. Data in the system can be accessed with a url built from three components: *account name*, *partition name* and *object name*, which can uniquely identify all objects available in the system worldwide. While account name is used to identify the client, identification of the data objects varies according to the type of object: blobs are uniquely identified by partition name, tuples in table storage are identified by a composite primary key (partition name, object name) and for queues, the partition name identifies the queue and the object name the specific message within that queue.

3.2.1 Architecture

Figure 11: Windows Azure Storage architecture diagram



The architecture of the system, shown in Figure 11, includes two separate systems: i) the location service (LS) and ii) the storage stamps (SS). The location service handles the global namespace and as such, the lifecycle of accounts and the management of storage stamps. This involves managing the association of accounts to storage stamps, and coordination of the replication of account data between stamps for disaster recovery purposes (async replication). The location service is itself distributed on two geographical zones for redundancy and fault tolerance.

Storage stamps, on the other hand, perform the physical storage of data and respond to all user queries. Each storage stamp is a cluster of racks, where each rack is a separate failure domain with redundant connectivity and power supply, and can store up to 30 petabytes (at the time the paper was published). The target utilization for a storage stamp is 70% and if it raises accounts are migrated to other stamps. Each storage stamp is divided into three layers which perform different functions:

- the *Stream Layer* stores data on disk, its design is very close to that of other

distributed file-systems,

- the *Partition Layer* implements the higher level data abstractions discussed, provides transactions and strong consistency for objects, caches data and uses the Stream Layer to store the data for the objects, and finally,
- the *Front End Layer*, a stateless component that performs routing of requests to the appropriate Partition Layer process and streams large objects directly from the Stream Layer as an optimization for large files.

The Stream Layer The *Stream Layer* implements the basic storage primitives for the system and it is accessed by the Partition Layer (the client). Its design is that of a append-only filesystem, and the interface provided to clients offers the usual operations: i) open, ii) close, iii) delete, iv) rename, v) read, vi) append, and vii) concatenate. Operations in the stream layer work on streams, large files built as a list of pointers to extents. Extents are physical file stored on the NTFS filesystem that contain the data, as a list of blocks.

Blocks are small data units (up to 4MB) with a check-sum and they are the minimum unit the system operates on. Reads and writes operate on whole blocks and when written, the blocks are atomically appended to an extent. Writes also support appending multiple blocks as an atomic operation, a “multi-block” write. Reading less than one block is also not supported, as read operations verify the checksums for block themselves (and the checksum cannot be verified by reading only a part of the block). If less than a block is requested by clients an entire block is loaded into memory and the extra data is simply discarded.

Extents are just a list of appended blocks that can grow up to 1GB in size. Extents, much like blocks in HDFS, are the unit of replication in the stream layer and, unless there are errors, there are three copies of each available in the system. Unless an extent is last in a particular stream, it is sealed. A sealed extent can no longer be appended to and is completely immutable. Sealed blocks can also be erasure coded, depending on policy. Erasure coding in WAS is described in detail in a separate paper [14]. To avoid excessive fragmentation of small objects, the stream layer appends multiple objects to the same block or the same extent, depending on the size.

Streams are the file-like primitive provided to clients by the stream layer. Every stream has a name in the name-space of the stamp (which is maintained at the stream layer), and it is a list of pointers to extents. Representing streams as list of pointers enables a very efficient concatenation operation, where two or more streams can be merged by just concatenating the list of pointers but without modifying the existing extents. All of the extents in a stream but the last are sealed.

The stream layer is organized as two different components:

- the Stream Manager (SM), a component similar to HDFS’s namenode and

- the Extent Nodes (EN), components that perform a function similar to that of HDFS datanodes.

The *Stream Manager* is a group of nodes, coordinating using Paxos, that performs functions equivalent to those of a HDFS namenode. Such functions include assigning extents, both primary and replicas, to extent nodes, performing periodic polling and re-replication of under-replicated blocks and the storage of metadata on streams. Streams are managed solely by the Stream Manager as a set of pointers to extents stored by Extent Nodes.

Extent Nodes, on the other hand, manage the physical storage of extents on disk. Each node completely manages a set of disks where extents are saved as NTFS files. For each extent the nodes also store an index that identifies block boundaries within the stream. Extent nodes also perform synchronous replication of extents to other nodes both during client writes and during re-replication as scheduled by the Stream Manager.

The Partition Layer The *Partition Layer* builds upon the storage primitives of the Stream Layer to provide higher level APIs to application developers. Clients that access Windows Azure Storage can only use operations provided by the Partition Layer and cannot access the Stream Layer directly. The APIs provided to external clients allow users to store data and manipulate it in three different types of objects: i) blobs, ii) tables and iii) queues. Additionally, the Partition Layer provides transactional behaviour for all supported data models, load-balancing and object namespacing within the stamp and finally, inter-stamp replication for disaster recovery and balancing purposes. The inter-stamp replication works by asynchronously replicating all data for an account from a primary stamp, where all the queries are routed by the LS, to a secondary stamp in a different geographic region. The secondary stamp can be promoted to primary both if the primary fails (disaster recovery) or if its load raises above a set threshold (load balancing).

All internal state for the Partition Layer is stored and processed in Object Tables (OT), an internal abstraction providing SQL-like tables that can grow to several petabytes. All user-facing abstractions, as well as internal functions are stored in such tables, which are in turn persisted by the Stream Layer. The Partition Layer manages OTs by dividing them in ranges and assigning ranges to nodes. The Partition Layer is itself organized as a set of three different components:

- a Partition Manager (PM) that splits the object table and assigns it to Partition Servers. It manages failures of Partition Servers and does load balancing by re-assigning partitions to other servers,
- Partition Servers (PS) which serve requests for the partition of OTs they are managing, and
- a Lock Service (LS) which provides a Paxos [16] lock service used to elect a

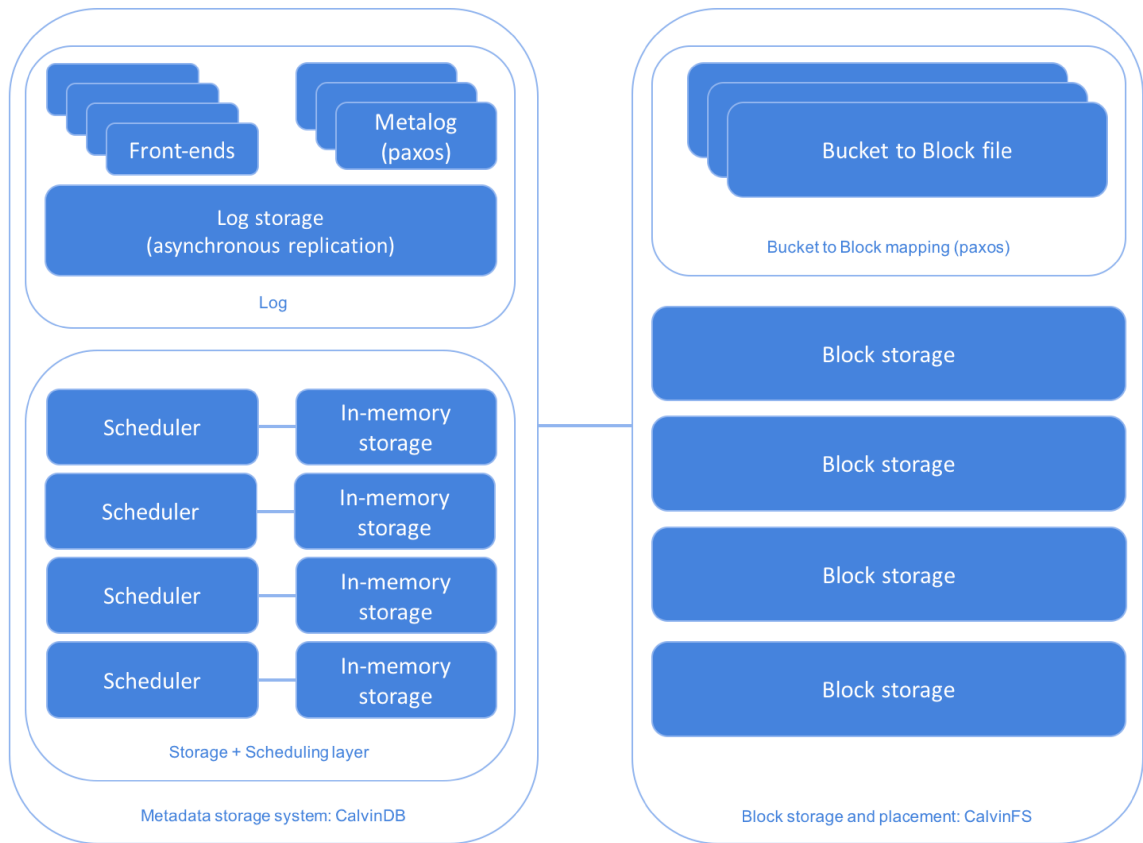
Partition Manager from the set of Partition Servers and to detect failures in Partition Servers.

3.3 CalvinFS

CalvinFS [24] is an experimental distributed file-system created with the goal of exploring the performance and scalability implications of metadata storage in a (distributed) database system. The distributed database is CalvinDB [25], created by same of the same authors, a share-nothing distributed database with support for high-performance distributed transactions.

3.3.1 Architecture

Figure 12: CalvinFS architecture diagram



Much like previously discussed distributed file-systems, the system is designed around two components, as shown in Figure 12:

- a *metadata store* which maintains information about file-system hierarchy, permissions, file-to-block mappings and replica locations, and

- a *block store* which stores physical data on disk in blocks and provides primitives to read and write such blocks.

Metadata store The *metadata store* is CalvinDB, extended with a number of filesystem-specific operations. CalvinDB is itself divided into three components: i) a *log* which maintains an ordered list of transactions with parameters, ii) a *storage* layer which stores database data and provides local transaction semantics, and iii) a *scheduling* layer which performs local execution of transactions. Each of these components is exposed to others through a standard interface and can therefore be replaced independently.

In CalvinDB, the **log** maintains a complete and ordered list of transactions and transaction parameters, such that, by replaying all transactions from this log the database can be reconstructed. The log is completely distributed and is divided in two logical components: front-end servers and the metalog. Front-end servers accept transaction requests from clients and batch them before writing such batches in the distributed storage. Once the batch is safely stored (and replicated) in the storage, the system generates a unique ID in the batch and writes it in the metalog. The metalog is a ordered sequence of unique batch IDs maintained by a set of servers running a Paxos consensus protocol for consistency. In order to “replay” transactions, the system traverses the metalog extracting the unique IDs and executes the transaction batches in that order.

The **storage layer** organizes the storage of database data. As all the other components, the storage layer is an interface and any implementation fulfills the following criterias: i) provides read and write primitives that execute on the node, ii) provides a placement manager that, for every request, provides a storage node where the operation can be executed and iii) allows the definition of custom transactions that include both read/write primitives and other deterministic application-specific logic. The ability to define custom transactions is particularly powerful in the context of a distributed file-system as it provides the opportunity to define more high-level operations such as `CreateFile(path)` that will be serialized in the log along with all their arguments (`path` in this case). The implementation used in CalvinFS provides a in-memory key-value store which supports versioning of keys and uses consistent hashing of keys to determine placement of values.

The **scheduler** drives local query execution and one process is therefore executed alongside every storage layer node. Unlike most other database systems which employ a pessimistic locking scheme and wait for locks when they are acquired by another transaction, the scheduler in CalvinDB uses a protocol called deterministic locking that analyzes the entire transaction, determines the read/write set and executes it only when it is safe to do so without additional checks. The actual execution is performed by the storage node when the scheduler forwards the transaction to it. The absence of a distributed commit protocol, usually required by other database systems in this scenario, greatly increases scalability and reduces latency. It, however, limits the type of transactions that can be executed to those for which the read/write

set can be determined statically (without executing the query itself). Given that some operations, such as recursive change of permissions, require transactions for which the read/write set cannot be statically determined, a system called OLLP (Optimistic Lock Location Prediction) is used to estimate the read/write set. The implementation of OLLP for CalvinFS executes the query without performing writes (a dry-run) and then annotates the transaction with the read/write set obtained. If the read/write set provided by the dry-run is different from the one obtained during actual execution due to changes in the data, the new set is appended to the transaction and the transaction is restarted.

File-system metadata organization CalvinFS stores file metadata as an association between a key, the full path of the object, and a value containing various fields:

- Type: either file or directory,
- Permissions: unix-like permissions for the object and all ancestors,
- Content (directory): a list of all files contained in the directory including subdirectories and
- Content (file): A mapping between byte-ranges in the file and block IDs.

Given the flat organization of files and the fact that all files store permissions for all the ancestors, changing permissions of a directory is potentially a very expensive operation as it involves changing all the descendants. Recursive queries are also very expensive for the same reason.

Block store Block storage in CalvinFS differs from the file-systems previously discussed in two significant ways: block allocation and block assignment. Blocks in CalvinFS are completely immutable and can range from 1 byte to 10 megabytes. Every write operation generates an entire new immutable block and appends it to the file metadata. A background process periodically rewrites and compacts blocks in order to reduce fragmentation but blocks are otherwise completely immutable. Each block is assigned a global ID which is stored in the metadata and, in order to be assigned to a set of machines, the block ID is hashed and the hash is used to identify a bucket. Each bucket is assigned to a set of machine and those machine are responsible for all the files whose ID hash is in the bucket. The mapping of machine to buckets is maintained in a Paxos replicated store and is additionally cached on all machines.

Geographical replication All of the components discussed above can be executed in geographically distant data-centers and the system assigns replicas in a way that minimizes disruption during failures and network partitions. Most operations only

need a quorum of machines to acknowledge before returning to the client, therefore the latency of the overall system in the case of three geographic areas (the typical case) depends on the two areas with the lowest latency to the client (the third will eventually ack).

3.4 Summary

In this section we analyze three different distributed file-systems with a focus on how they handle metadata management.

The Google File System (GFS) paper [11] directly influenced the design and implementation of HDFS and the similarities between the two are therefore extensive. Like HDFS, GFS only uses a single master node and maintains the entire file-system metadata in main memory. For fault-tolerance all metadata operations are recorded in a log, which is propagated to other machines that build a in-memory state from it. Such machines can either be used as backups in case of master failure and as read-only replicas that can serve any read operation from clients. Both master backups and read-only replicas aim to increase fault-tolerance of the system but do not handle the scaling use case. In order to scale GFS, Google eventually adopted a solution virtually identical to HDFS federation by allowing multiple masters to control a shared pool of chunkservers. The limitations of GFS eventually prompted the design of other systems with better scalability and performance such as BigTable and later Colossus [17]. BigTable [8] is an extremely scalable distributed storage systems for structured data and it is built on top of GFS. Colossus [17], on the other hand, is the successor to GFS and it employs a distributed master design with metadata stored on BigTable and allows for more granular file operations by adopting a 1 megabyte size for its chunks. While this significantly increases the amount of metadata for the master to handle it is better suited for real-time applications for which GFS was not originally designed for.

Windows Azure Storage [6] introduces a high-performance append-only filesystem that is capable of supporting the three core abstractions that are offered to users by the system. The file-system, called Stream Layer (SL) in Windows Azure Storage, is very similar in design to both GFS and HDFS and it provides reliable storage to the abstractions built by the upper layer. It operates in a single zone and a single cluster called a Stamp. Replication in the Stream Layer is performed in a similar fashion to both GFS and HDFS, extents (chunks in GFS, blocks in HDFS) are synchronously replicated to a set number of replicas before any operation is acknowledged to the client. Metadata is stored in a Paxos replicated group where each machine stores and mutates the state synchronously. Reliability across Stamps (and therefore availability zones) is only provided by the upper layers which replicate entire objects asynchronously to another Stamp in a different zone for disaster recovery or load balancing purposes. In order to increase scalability past the limits of a single Stamp, the application must use and coordinate multiple independent Stamps without any support by the system.

CalvinFS is the only file-system analyzed here that natively supports deployment in multiple availability zones both to increase reliability and to increase scalability. It does however optimize for a very different use case than typical distributed file-systems and that is an extremely large number of small files. Furthermore, due to the way it handles the hierarchical nature of a file-system tree, operations that need to modify large sub-trees are required to modify each child and are therefore slow and expensive. Finally, this is the only solution that is completely experimental and has not been validated with real-world usage.

While all the papers analyzed in this section introduce some interesting concepts, very few are directly applicable to our problem due to the peculiarity of how replication works in MySQL Cluster. However, concepts not directly relating to metadata replication, such as erasure coding in WAS, provide interesting insights in how to handle such tasks.

4 Contribution

The goal of this work is to plan for an extension to HopsFS that leverages the geographical replication capabilities built into MySQL Cluster and illustrated in Section 2.4 to build a geographically distributed file system that transparently appears to clients as a single name-space and maintains most of the consistency properties that clients expect. Furthermore, clients running in or near the closest geographical location, the *local* cluster, are expected to continue to function, possibly at reduced capacity, in case other, *remote*, geographical locations fail or become unavailable for any reason. This also implies that operations from clients in or near the local clusters should be processed in the local data center as much as possible to avoid saturation of the egress links that connect the different locations together.

Two data-centers are considered separate geographical locations if they are different, distant buildings that are serviced by different utilities such as power companies and internet service providers, and are therefore unlikely to be all affected by local catastrophic events such as loss of power or a localized earthquakes. This requirement also influences network topology in that two machines in different geographical locations may only be able to connect to each other through a virtual network which connects to other data-centers through the external connection. Due to the use of the external connection, packets travelling on the virtual network are subject to both additional overhead caused by the virtual networking protocols and routing on the open internet. Such a topology implies that connection between machines running in different data-centers are subject to higher latency, often orders of magnitude higher, and lower, more expensive bandwidth compared to a connection between two machines in the same geographical zone. A partial exception to this rule are cloud provider's Availability Zones (or just Zones depending on the provider specific terminology), which fulfill the requirements of different geographical locations but are connected by low-latency dedicated fibers and allow machines in two different zones to communicate with latency and bandwidth parameters similar to those of machines in the same zone. They achieve this result by placing different data-center buildings just hundreds of kilometers from each other, connecting them to different power providers and ISPs and providing dedicated connection between the data centers themselves. Cloud provider zones are, however, insignificant to our goal as a system designed to run in the former scenario will only perform better when deployed in the latter.

As shown in Section 2.1 and 2.3, the HopsFS architecture involves three main components:

1. a set of namenodes which process client and datanode RPC requests as well as performing background periodic maintenance tasks such as re-replication of blocks which keep the cluster in the correct state,
2. a set of datanodes which store block data and checksums and report their status to the namenodes using heartbeats, and

3. a metadata storage cluster which stores and handles modification of the cluster metadata by the namenodes.

In order to allow clients to perform operations on the local cluster, which is one of the key objectives of the project, each of the clusters needs i) a complete copy of all metadata, ii) a complete copy of all filesystem data, and iii) running instances of all the components required for the system to function on its own. If this were not true, operations on the local cluster would require very expensive connections to a remote cluster, operations that would fail and render the local cluster inoperable in the event of remote cluster failure. Replicating the infrastructural components is by far the simplest task, as it only involves the deployment of a complete cluster in the other geographical location plus some configuration to connect the clusters. Management of filesystem metadata and blocks are, however, very complex problems and the focus of this thesis.

4.1 Metadata management

In HopsFS, file system metadata are stored and processed by a MySQL Cluster cluster. As discussed in Section 2.4, MySQL Cluster supports a variety of asynchronous schemes that can be used to replicate transactions between different geographically separate clusters, without impacting the liveness and latency of the running NDB cluster. The hybrid active active replication scheme allows different metadata clusters and namenodes to operate on separate copies of the metadata, which is asynchronously distributed to all clusters in the replication ring. Per limitation of the conflict function selected (`NDB$EPOCH_TRANS`), only two clusters can be set up in this configuration, limiting the replication to two geographical areas. In order to further simplify the basic design, one cluster is designated as the active partition for all data, while another is designated as the passive. Following this, we will refer to the clusters as *primary* for the cluster active for all partitions and *secondary* for the cluster passive for all partitions. All transactions committed on the primary cluster are durable, while transactions committed on the secondary cluster may be re-aligned if they are in conflict. Re-aligning involves undoing the conflicting transaction, as well as any transactions depending on it and the applying the changes originated on the primary. As previously mentioned, conflict tables, which are only present on the primary cluster, will contain the conflicting values for the rolled-back rows, allowing applications that access the database to react to conflicts in specific ways.

While asynchronous propagation of transactions fulfills the requirement of maintaining a complete working copy of all data in both clusters, it undermines a number of processes in the namenode that rely on the consistency properties of the NDB database. Due to the lack of row-level locking, for example, it would be entirely possible for HopsFS to grant a lease on a file in both the primary and secondary database concurrently, breaking the single-writer semantic of HDFS (and HopsFS). To avoid this issue and maintain the appropriate level of consistency for the filesystem, namenodes in the secondary zone are allowed to perform direct connections to the

primary metadata cluster to execute operations which require strong consistency properties and locking. Operations require such strong consistency properties to maintain the single writer semantics of HDFS, which means that all operations that modify file and block metadata will be routed to the primary cluster. While routing write operations to the secondary cluster may appear to be problematic in terms of traffic flowing between zones, the analysis of the workload provided by Spotify and described in the HopsFS paper [18] as well as similar traces provided by Yahoo [3] and LinkedIn [20], show that such operations only make up for less than 5% of the total volume, allowing this approach to be considered. The only situation where not all operations are going to be committed on the primary, and therefore in a consistent and durable fashion, is when the two clusters are unable to communicate with each other; a condition known as split brain that can be caused by one of two events:

- one of the two clusters fails or
- both clusters are online, but cannot communicate to each other, a situation known as a network partition.

A network partition can manifest in different ways but in the context of this paper we define it as a complete inability of nodes in the first cluster to connect to any node in the second cluster and vice-versa. While the definition is very specific, and network partitions can typically manifest in a variety of more subtle ways, in this case the specificity is also supported by our model of geo-replicating databases which implies that all traffic between zones is carried by a virtual network running on the external connection. In case of failure on this particular (virtual) link, all connectivity between data centers would effectively be cut, and, as shown in Microsoft’s study on network failures [12], links between data-centers take the longest to repair. Because we can assume that split brain scenarios are going to last a non-negligible amount of time to repair, regardless of cause, determining when such an event is happening is necessary to allow clusters to adapt their behaviour.

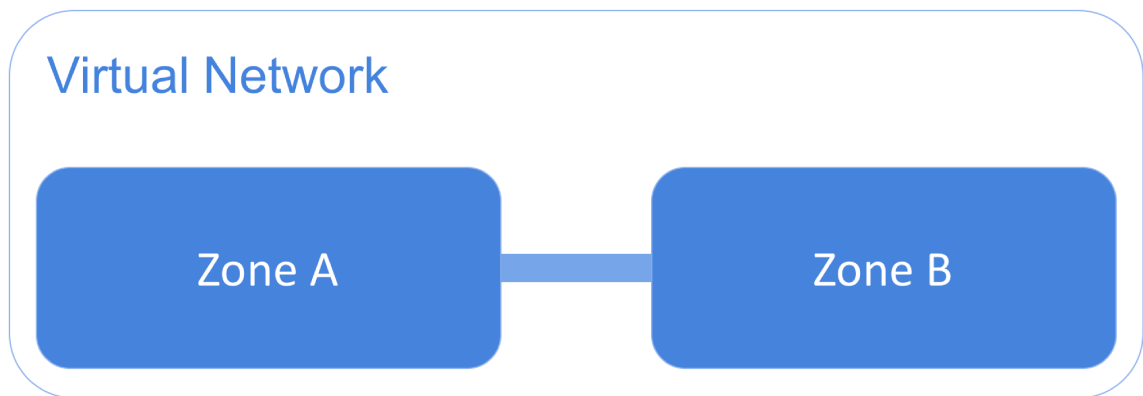
4.1.1 Overview

As previously mentioned, there are three possible states the system can be in at any given time. This section provides a high level overview of the three states and the expected behaviour and trade-offs in each while a detailed account of the mechanics that allow the system to detect its state and react accordingly is provided in the following sections.

In **nominal operating conditions**, where the connection between the different geographical areas is functioning properly, all namenodes apply metadata modifications directly to the primary cluster and execute read operations on the local cluster as shown in Figure 13. This type of system, which is conceptually similar to a master-slave topology, is extremely effective in read-intensive workloads because it delegates all read operations to the local cluster. Due to the way operations are

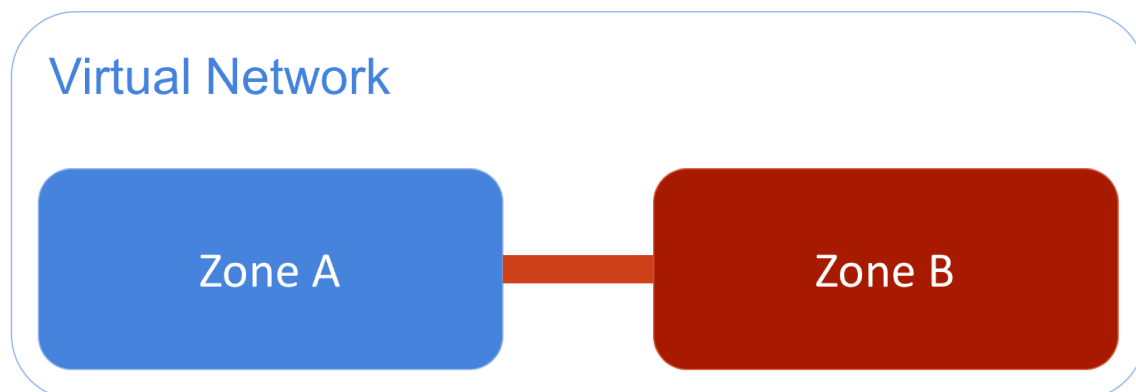
handled on the secondary cluster, it is, however, possible for clients connected to this zone to modify metadata in such a way that newly written data will not be immediately available for local reads. If a client creates or appends content to a file and then tries to read said file, due to the asynchronous nature of the replication, there is no guarantee that metadata will be available on the secondary cluster and, as a result, the client may read stale data or not find the file at all. To guarantee that a client process metadata modification is immediately available for read from the same process, a behaviour commonly referred to as *read-your-writes*, the client can optionally perform *fully consistent reads* by reading metadata from the primary cluster. Because this operation will put further strain on the primary cluster, it is to be used only when read-your-writes behaviour is required for correctness such as when using files as locks. In summary, the system in nominal condition behaves as it would in a single cluster scenario except for the handling of reads on the secondary cluster, where all metadata will be delayed due to asynchronous replication.

Figure 13: System in nominal conditions: reads go to the local database, while writes are directed to the primary.



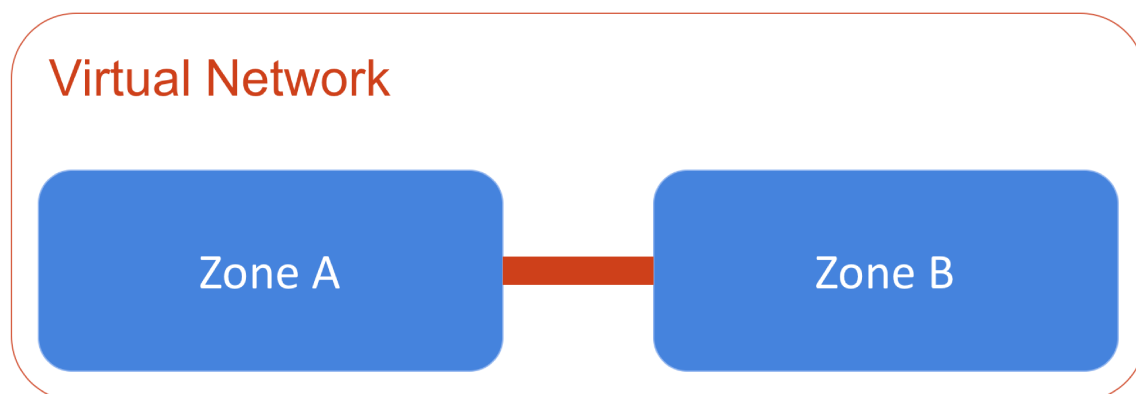
During a **cluster crash**, a split brain scenario caused by the failure of either cluster, the remaining cluster can operate normally and execute all possible operations (shown in Figure 14). In case the failed cluster is the secondary, the namenodes in the primary zone can continue without any modification to the behaviour but, if the failed cluster is the primary, the namenodes in the secondary zone require some adaptations. Given that, in nominal operating conditions, namenodes in the secondary cluster execute all operations on the primary cluster, in case of failure of the primary zone a failover to the local cluster is required. Once the primary cluster becomes available again, the secondary cluster invalidates all leases and both clusters wait for database replication to completely synchronize the primary zone with the secondary before allowing modifications again. This process is necessary to guarantee that no conflicts are caused during transitions between states. To summarize, in case of cluster crash the surviving cluster operates normally except for some contingencies during state transitions which are necessary to prevent conflicts.

Figure 14: System during a cluster crash: one of the two clusters behaves normally while the other is unavailable.



During a **network partition**, a situation where the two clusters function correctly but cannot communicate with each other as shown in Figure 15, concurrent modifications to metadata must be only allowed in such a way that prevents uncorrectable conflicts from happening. To guarantee the absence of conflicts that cannot be automatically solved, the system in this state does not allow clients to perform a subset of metadata modifications. More specifically, the only metadata modification allowed is the creation of new files and the subsequent append of data to such files, an operation for which automatic conflict resolution is possible in a deterministic fashion. Any other operation is disallowed and an error is sent to the client attempting it.

Figure 15: System during a network partition: both clusters are active but the connection between them has failed.



4.1.2 Split brain detection

In order to detect whether the system is operating normally or it is suffering from a split brain we propose two different procedures, one for the primary and one for the secondary cluster, that allow namenodes to detect split brain scenarios with a minimum of internal coordination. Coordination is provided by the leader election procedure described by Niazi et al. [19], and discussed previously which is already present within HopsFS.

Detection on primary cluster To detect a split brain scenario on the primary cluster, we need to ascertain whether or not we are able to communicate with any node in the secondary cluster. While we could implement a distributed failure detector to check for liveness of nodes in the secondary zone, HopsFS already exposes the failure detector built into the leader election procedure. In the context of multiple data-centers, the leader election procedure is extended to include both namenodes from the primary and secondary cluster and a field in every row of the election table, to indicate the cluster the node belongs to. Nodes from the secondary cluster connect directly to the primary cluster to perform leader election which means that, both in case of network partition and secondary cluster failure, the nodes would eventually be marked as not live by the failure detector. With all of the prerequisites in place, the algorithm to detect network partitions on the primary cluster is illustrated in Algorithm 1.

Algorithm 1 Split brain detection: primary cluster

```

1:  $fd \leftarrow leaderElection.getFD()$   $\triangleright$  get the failure detector from the leader election
2:  $liveNodes \leftarrow fd.getLiveNNSet()$ 
3: for  $node$  in  $liveNodes$  do
4:   if  $node.getCluster() == SECONDARY$  then
5:     return ok
6:   end if
7: end for
8: return detected

```

Detection on secondary cluster The secondary cluster cannot rely on the same procedure as the primary cluster because, by definition, if a network partition happened or the primary cluster crashed, the connections of the namenodes to the primary clusters would be lost (and the leader election procedure would not run). We can, however, treat the loss of connection as a signal that a network partition or cluster crash is occurring, but only if all nodes in the secondary cluster are not able to reach the primary. One possible solution would be to have a table on the local database where namenodes write the status of their connection to the primary metadata cluster. Given that namenodes can fail at any time, however, old

entries from crashed nodes could actually result in false positives, impeding the other namenodes from detecting the partition. In order to only query live namenodes we can use the same leader election component that we leverage in other parts of the system, running the algorithm on the local instance of NDB and only allowing local nodes to participate. Instead of creating one extra table, we attach the status of the connection to the primary as a new column in the *local* leader election instance. With such a failure detector in place, detecting a split brain only requires checking the status of the connection to the primary on all other nodes, as shown in Algorithm 2.

Algorithm 2 Split brain detection: secondary cluster

```

1: if currentNode.isConnectedToPrimary() then
2:   return ok  $\triangleright$  if the current node has a connection to the primary metadata
      cluster, there is no partition
3: end if  $\triangleright$  get failure detector from leader election
4: fd  $\leftarrow$  secondaryLeaderElection.getFD()
5: liveNodes  $\leftarrow$  fd.getLiveNNSet()  $\setminus \{currentNode\}$ 
6: for node in liveNodes do
7:   if node.isConnectedToPrimary() then
8:     return ok
9:   end if
10: end for
11: return detected

```

While namenodes are now capable of detecting a split brain independently, they do not have the capability of distinguishing between a network partition or a cluster crash. This capability can be provided by providing a system hosted in a third zone, independent from the first two, which will act as a tie-breaker and allow the systems to consistently know whether both zones are still live (network partition) or if the remaining cluster is the only one currently running. Such a system could be implemented in a variety of ways, for example by configuring a Zookeeper [15] cluster with three nodes: one in the primary zone, one in the secondary zone and the tie-breaker in the third zone. In case of split brain, both clusters would query the tie-breaking system which would yield one of the following outcomes:

1. the cluster is unable to get a quorum of nodes; it is isolated both from the secondary and tie-breaker. In this case the cluster goes into read-only mode as it is the only safe course of action
2. the cluster is able to get a quorum with the tie-breaker; the other cluster failed
3. both clusters are able to contact the tie-breaker; the cluster is experiencing a network partition

In case of cluster failure the remaining cluster can continue serving all requests from the clients. The reason for this is because, following the tie-breaking, we are

sure that the other cluster is either not live or in read mode, therefore there will be no conflicts upon restoring the asynchronous replication of metadata. If the failed cluster is the primary, the secondary cluster namenodes need to switch over to the local cluster for write, as well as read operations, until such time where connectivity between the two clusters is restored and all necessary procedures to safely resume metadata replication have been executed.

While a cluster failure is relatively straightforward, network partition must be handled with extreme care to maintain the single-writer semantics of HDFS. Should the two cluster be allowed to continue without any restrictions on the operations that they are allowed to perform, they could cause conflicts in such a way that required human intervention to merge.

4.1.3 Conflict handling for network partitions

Conflicts on file metadata can only happen in three classes of tables:

- the inode table,
- the block and replica tables, and
- the lease table.

While conflict on leases can be avoided by clearing the leases upon both detecting a partition and resolving the partition, forcing clients to retake the lease and retry the operation, conflicts on the inode and block tables must be handled.

Conflicts on block and replica tables are particularly problematic as, after the partition is resolved, the system may be in a state where two disk blocks with different content have the same ID. Upon replication, the metadata for the blocks created on the primary cluster would “win” and on-disk replicas created on the secondary cluster would therefore be considered corrupt on the first block report due to having a different checksum. While it would be possible to devise a conflict resolution scheme to maintain both block versions, conflicts on blocks and replicas can be avoided altogether. Before introducing the solution it is necessary to understand how ID assignment for blocks (and other database objects) is handled in HopsFS. Given that *addBlock* is a frequent operation when writing files doing a round-trip to the database to request each new block ID would be prohibitively slow and would create a large amount of work on the metadata cluster, this operation is batched. At the first write operation, namenodes require a sequence (batch) of new IDs that they will use to fulfill subsequent *addBlock* operations. When all IDs in the batch have been assigned, the namenode just requests another batch. By configuring the namenodes on the primary cluster to only require batches of *even block IDs* and namenodes on the secondary cluster to require batches of *odd block IDs*, two blocks created on two different clusters will never have the same block ID and will therefore never cause a conflict.

Conflicts on inodes, on the other hand, are caused by both partitions creating a file or directory with the same name in the same parent directory. As previously discussed, inodes also have unique IDs, but conflict are detected on the primary key which is composed of the name and parent ID. These conflicts are therefore unavoidable but they can be resolved with ease. Upon detecting a conflict on the inode table, the inode created on the secondary cluster will be placed in the exception table due to the conflict resolution strategy. With the inode in the exception table, the namenode responsible for handling conflicts (which is a leader elected between nodes in the primary cluster), can create a new inode with a different name and place it back in the same directory. A possible example of such a naming scheme may be `<original name> + <sequential number>` such that if `myFolder/myFile` was created on both clusters, the conflicting file would be renamed as `myFolder/myFile1`. Allowing files to sometimes be renamed is a significant difference in behaviour compared to both HDFS and HopsFS in single zone mode, which is why clients of the system need to take this behaviour into account and react to it upon resolution of a network partition.

By using the *conflict avoidance and resolution* techniques developed, clients in both zones are allowed to continue all read and file creation operations with minor divergences in overall system behaviour. The techniques presented, however, are only sufficient to handle the file creation case, but not other operations that require modification of metadata. Given that in case of network partition the system doesn't have access to a consistent lease table, there is no way of knowing which existing files are being appended to, the only form of modification allowed on files in HDFS. Allowing clients to append data to a file, could therefore result in two clusters having two diverging versions of the same block, a conflict which cannot be resolved without either creating two different copies of the file with new blocks or implementing a way for the system to handle diverging copies of the same file. Due to the complexity both in terms of implementation and resulting behaviour of the proposed solutions, as well as the fact that in the Spotify synthetic workload shown in [18] append operations account for 0.0% of the total, the current course of action is to disallow them during network partition events. Subtree operations, due to their use of locking and the large amount of transactions they generate, are also disallowed. Deletes and moves are also not permitted due to the conflicts that they would generate. This solution allows the two data-centers to operate independently during network partitions, albeit with a subset of operations.

A possible alternative for workloads that require the full set of operations is to implement an arbitration strategy similar to that used in NDB. In this case, only one of the two sub-clusters would be allowed to continue performing write operations, while the other cluster would be free to continue in a read-only capacity.

4.1.4 Summary

In this section we describe a plan that allows the clusters to not only detect split brain situations, but to identify whether the situation is due to a network partition or a cluster crash and react accordingly. While many operations are disallowed during network partitions, this should be a rare and transient event. Furthermore this is only a plan for the initial implementation and restrictions may be lifted with further work on conflict resolution.

4.2 Block management

Aside from managing metadata in a replicated environment, a geographically replicated storage system also needs to manage file content in such a way that, during a split brain, the separated clusters are capable of serving all client requests. As previously mentioned, HDFS and HopsFS, store file content in *blocks* which are managed by data nodes. A file can span arbitrarily many blocks which have a configurable maximum size, by default 150 megabytes. Blocks are immutable once they are marked as finished and only the last block in a file can be modified. Adding content at the end of a block is the only modification allowed.

In order to maintain availability of blocks in the face of data node failure, HopsFS supports two different replication schemes: block replication and erasure coding.

In block replication, the system maintains multiple copies of the same block on different data nodes. The copies, called replicas, are distributed among data nodes according to a configurable placement policy, which aims to minimize the number of blocks which are unavailable as a result of component failure, be it machine or switch. In case one of the replicas is permanently lost, the leader namenode instructs data nodes to re-replicate the block, returning the amount of replicas to the specified number, three by default.

Erasure coding is a radically different concept than whole block replication. Instead of creating entire copies of the blocks, erasure coding computes new parity blocks from the original blocks. Both the number of source and output blocks are configurable, and the output blocks are called parity blocks. Assuming N source blocks, 10 for example, and M parity blocks, 5 for example, the 10 original blocks can be reconstructed using any combination of the $N + M$ blocks now available. The parity blocks form a new file, which is stored in a different directory than the original file. Block placement for erasure coding blocks is handled by the *erasure coding manager* which is described in [13]. In case one of the blocks for a erasure coded file fails, the system needs to regenerate either the original block or the parity block, which requires a full read of N of the blocks and it is accomplished through a mapreduce job.

In this work we only consider whole block replication but we plan to implement erasure coding schemes in later iterations of the project.

4.2.1 Placement policy

Given that the goal for the project is to allow both geographical regions to operate independently in case of split brain, each region requires a complete set of blocks from all locations. In order to obtain this, the block placement policy needs to be aware of the existence of multiple zones, which are considered separate failure domains. The existing hierarchy for failure domains only considers machine and rack but the modified version will also include a third level: *geographical zone*.

While this solves the issue of placing blocks in the correct datanodes, there remains the issue of the number of replicas to create. The default value of three creates imbalance, by assigning two replicas in the zone where the block was created and only one in the other zone. By using a replica value of four two replicas are assigned to every zone, ensuring that both zones have the same amount of blocks.

The final problem regarding the placement policy is the handling of split brain scenarios. Without further adaptations, a split brain scenario would lead the cluster to believe that half of all the replicas in the cluster are missing, forcing the leader namenode to re-replicate all the blocks an additional two times. Aside from creating a very large amount of network load between datanodes the result of such re-replication would be discarded as soon as the temporary split brain scenario is resolved. In order to avoid spurious re-replication, we modify the amount of replicas to two during split brain scenarios. By setting the value to two, we avoid any re-replication of existing blocks and we only create two replicas for new blocks. When the two clusters are merged, the replica value is once again increased to four, and the normal background re-replication tasks will create the necessary replicas in the other zone for blocks created during the split brain.

4.3 Adaptations

As previously described in Section 2.3.3, the data access layer (DAL), which provides the system with access to the metadata storage system, was structured around access to a globally available connector, which in turn assumed a connection to a single cluster. Due to these assumptions, the majority of the implementation work accomplished in the context of this thesis was to improve the DAL to allow multiple open connections to different databases. Multiple database connections are necessary for the namenodes in the secondary cluster to route some queries to the local metadata storage cluster and some others to the primary cluster. In addition to this, to be able to recover from network partitions, the DAL must be able to reconnect to the metadata storage cluster in case of failure and notify other components of this. Notifications of disconnections and reconnections are necessary to correctly manage state changes for the system, namely enter and exit partition

mode, both on the primary and on the secondary cluster.

Connection to multiple database As previously discussed, metadata accesses are not performed by directly accessing the connector but rather by using a request handler. When operating on multiple databases there needs to be a mechanism for a request handler to be executed either on the local or the primary metadata cluster. Note that a request for the local database still connects to the primary if the namenode requesting it is itself in the primary zone. While a first implementation required every request handler to explicitly provide the database to connect to as a parameter this required modification of all code locations where a transaction handler is created. This method is also extremely error prone as it disseminates the information on where to execute operations around the codebase. The better solution is to associate to every operation type the database where the operation is to be executed. This result is achieved by associating a constant to every member of the `OperationType` enumeration as shown in Listing 1. By extracting the database information from the `opType` the request handler can operate transparently without changes in signature and all the modifications are concentrated in one place, the operation type enumerator.

Listing 1: The `OperationType` enum

```
public interface OperationType {
    TransactionCluster getCluster();
}

public enum HDFSOperationType implements OperationType {
    INITIALIZE(TransactionCluster.PRIMARY),
    ACTIVATE(TransactionCluster.PRIMARY),
    META_SAVE(TransactionCluster.PRIMARY),
    SET_PERMISSION(TransactionCluster.PRIMARY),
    SET_OWNER(TransactionCluster.PRIMARY),
    SET_OWNER_SUBTREE(TransactionCluster.PRIMARY),
    GET_BLOCK_LOCATIONS(TransactionCluster.PRIMARY),
    GET_STATS(TransactionCluster.PRIMARY),
    CONCAT(TransactionCluster.PRIMARY),
    // many more

    private TransactionCluster cluster;

    HDFSOperationType(TransactionCluster c) {
        this.cluster = c;
    }

    private TransactionCluster getCluster() {
        return this.cluster;
    }
}
```

|}

Request handlers obtain a connector to a specific database using a *multizone storage connector*, shown in Listing 2 along with . This interface, which is implemented both in the primary cluster and in the secondary cluster, allows clients to obtain a concrete connector towards a single database. In future iterations of the project, the multizone connector will also modify its behaviour during network partitions, for instance by always returning the local connector on the secondary cluster during a cluster crash.

Listing 2: The MultiZoneStorageConnector interface

```
/**
 * This class allows its clients to retrieve a connector
 * for the required cluster (primary or local).
 */
public interface MultiZoneStorageConnector {
    /**
     * This method returns a StorageConnector
     * for the appropriate cluster.
     * @param cluster whether to connect to
     * the local or primary cluster
     * @return the appropriate storage connector
     * @throws StorageException if a connector
     * cannot be returned
     */
    StorageConnector connectorFor(TransactionCluster cluster)
        throws StorageException;
}
```

The database connector was also modified to allow for re-connection capabilities and notifications of changes in state by implementing the **Reconnector** interface shown in Listing 4. The information on whether the connection is functioning or not is used on the secondary cluster by a **partition monitor** to perform split brain detection as shown in Algorithm 2 and Listing 3. When a split brain is detected by a partition monitor, a configurable action is executed and this action will, in the future, perform the state changes required by the system to handle the partition.

Listing 3: Implementation of the partition detection algorithm in the secondary cluster

```
/**
 * This methods performs partition detection
 * for the secondary cluster.
 * A partition is detected in the secondary cluster if all
 * the live namenodes lost the connection
 * to the primary cluster.
 * Additionally, this class updates the state
```

```

    * of the node's connection in the leader election procedure.
    */
@Override
protected PartitionEvent tick() {
    boolean connected = connector.isConnectedToPrimary();

    // update the state of the connection
    // in the leader election procedure
    leaderElection.setConnectedToPrimary(connected);
    // if connected to primary there is at least
    // one node connected (therefore no partition).
    if (connected) {
        return PartitionEvent.RESOLVED;
    }

    // if at least one of the other nodes is connected,
    // the partition is resolved.
    SortedActiveNodeList namenodes =
        leaderElection.getActiveNamenodes();
    // this can happen if run before the first leader
    // election round. unknown is ignored
    if (namenodes == null) {
        return PartitionEvent.UNKNOWN;
    }
    for (ActiveNode n: namenodes.getActiveNodes()) {
        if (n.isConnectedToPrimary()) {
            return PartitionEvent.RESOLVED;
        }
    }

    // if all the active namenodes aren't connected
    // to the database, detect a partition.
    return PartitionEvent.DETECTED;
}

```

Listing 4: Reconnector interface

```

/**
 * A reconnector can report whether the
 * connection is up and attempt reconnections.
 * Note that, if possible, checking
 * for connectivity should be cheap while
 * reconnection is expected to be more expensive.
 */
public interface Reconnector {
    /**
     * Checks whether the connector is connected to the remote.

```

```

    * @return whether the connection is up
    */
    boolean isConnected();

    /**
     * Attempts a reconnection.
     * If this method returns successfully,
     * the connection attempt was a success.
     * Should be called periodically
     * in the background to re-acquire connectivity
     */
    void reconnect() throws StorageException;
}

```

While the work performed so far is necessary to allow further progress towards the implementation of the theoretical framework described in this chapter, there is still much to do. The distinction between network partition and cluster crash is not implemented and will require an external system like ZooKeeper to perform arbitration. The routing of operations to the local database and all of the changes to the client to allow it to perform fully consistent reads are not implemented. Finally the behaviour of the system will need to be tested to make sure that it conforms with the expected behaviour described.

5 Summary

In this work we present a solution that allows HopsFS to transparently present multiple geographical areas as one cluster to clients. By leveraging the asynchronous replication built into MySQL cluster we perform metadata replication across geographical areas while still maintaining the same consistency guarantees as Apache HDFS and HopsFS when deployed in a single area. We also describe solutions for both network partitions and cluster crashes which allows clients to continue performing a safe subset of operations and allows the system to recover gracefully from such events. Furthermore, we detail the implementation work done to allow the inclusion of such changes into the HopsFS codebase. To the best of our knowledge, once complete, this would be the first HDFS implementation with such characteristics allowing it to reach the same levels of availability and data retention as cloud native storage systems such as Amazon S3, while still maintaining the consistent behaviour of a hierarchical file-system.

5.1 Future work

While the description of the basic solution presented in this thesis is complete, there is still much to be done both to implement the basic solution in the code and to further optimize it. Specifically, the implementation work done so far only covers the adaptation of the metadata access layer (DAL) to allow it to connect to multiple database clusters at the same time as well as being able to detect disconnections and perform re-connections. Furthermore, while the conflict detection functions used to merge the system after a network partition are provided by MySQL Cluster, no testing was performed regarding their impact on the performance of the database. There are also several areas where the proposed solution could be improved. First of, it would be interesting to study a way to execute some operations on the local cluster instead of routing them all to the primary cluster, while still maintaining the same consistency guarantees. By doing that we would further reduce the strain on the primary cluster and increase scalability of the overall system. Similarly, it would be beneficial to allow a greater set of operations when the cluster is experiencing a network partition to increase compatibility with applications that expect Apache HDFS and are therefore unaware of multiple zones. Finally, there are several key improvements to consider in the context of block storage and replication. Erasure coding techniques [14, 13] can reduce the block replication overhead allowing better utilization of space in the cluster, while improvements in block placement policies (as shown in [9]) can dramatically increase data retention in the presence of failures.

References

- [1] MySQL Cluster High Availability - Network Database. <https://www.mysql.com/products/cluster/availability.html>. Accessed: 2017-10-02.
- [2] MySQL Cluster Replication - Documentation. <https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster-replication.html>. Accessed: 2017-09-01.
- [3] C. L. Abad. *Big data storage workload characterization, modeling and synthetic generation*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.
- [4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison- Wesley, 1987.
- [5] E. A. Brewer. Towards robust distributed systems (abstract). In G. Neiger, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, page 7. ACM, 2000.
- [6] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In T. Wobber and P. Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSOP 2011, Cascais, Portugal, October 23-26, 2011*, pages 143–157. ACM, 2011.
- [7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *J. ACM*, 43(4):685–722, 1996.
- [8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In B. N. Bershad and J. C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218. USENIX Association, 2006.
- [9] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. K. Ousterhout, and M. Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In A. Birrell and E. G. Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 37–48. USENIX Association, 2013.
- [10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

- [11] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In M. L. Scott and L. L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43. ACM, 2003.
- [12] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In S. Keshav, J. Liebeherr, J. W. Byers, and J. C. Mogul, editors, *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 350–361. ACM, 2011.
- [13] Grohsschmiedt, Steffen. Making Big Data Smaller: Reducing the storage requirements for big data with erasure coding for Hadoop, 2014.
- [14] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In G. Heiser and W. C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 15–26. USENIX Association, 2012.
- [15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In P. Barham and T. Roscoe, editors, *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*. USENIX Association, 2010.
- [16] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [17] K. McKusick and S. Quinlan. GFS: Evolution on fast-forward. *Commun. ACM*, 53(3):42–49, 2010.
- [18] S. Niazi, M. Ismail, G. Berthou, and J. Dowling. Leader Election Using NewSQL Database Systems. In A. Bessani and S. Bouchenak, editors, *Distributed Applications and Interoperable Systems - 15th IFIP WG 6.1 International Conference, DAIS 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9038 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2015.
- [19] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In G. Kuenning and C. A. Waldspurger, editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 89–104. USENIX Association, 2017.

- [20] K. Ren, Q. Zheng, S. Patil, and G. A. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In T. Damkroger and J. Dongarra, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 237–248. IEEE Computer Society, 2014.
- [21] M. Ronström and J. Orelund. Recovery Principles in MySQL Cluster 5.1. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, and B. C. Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1108–1115. ACM, 2005.
- [22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In M. G. Khatib, X. He, and M. Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.
- [23] K. V. Shvachko. HDFS Scalability: The limits to growth. ; *login:: the magazine of USENIX & SAGE*, 35(2):6–16, 2010.
- [24] A. Thomson and D. J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In J. Schindler and E. Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 1–14. USENIX Association, 2015.
- [25] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM, 2012.
- [26] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

A Code listings

Listing 5: Transactional request handler for the rename operation

```

OperationType opType
if(isUsingSubTreeLocks) {
    opType = HDFSOperationType.SUBTREE_RENAME;
} else {
    opType = HDFSOperationType.RENAME;
}
new HopsTransactionalRequestHandler(opType, src) {
    @Override
    public void acquireLock(TransactionLocks locks)
    throws IOException {
        LockFactory lf = LockFactory.getInstance();
        locks.add(lf.getRenameINodeLock(
            nameNode, INodeLockType.WRITE_ON_TARGET_AND_PARENT,
            INodeResolveType.PATH, true, src, dst))
        .add(lf.getBlockLock())
        .add(lf.getBlockRelated(
            BLK.RE, BLK.CR, BLK.UC,
            BLK.UR, BLK.IV, BLK.PE, BLK.ER));
        if (dir.isQuotaEnabled()) {
            locks.add(lf.getQuotaUpdateLock(
                true, src, dst));
        }
        if (!isUsingSubTreeLocks) {
            locks.add(lf.getLeaseLock(
                LockType.WRITE))
            .add(lf.getLeasePathLock(
                LockType.READ_COMMITTED));
        } else {
            locks.add(lf.getLeaseLock(
                LockType.WRITE))
            .add(lf.getLeasePathLock(
                LockType.WRITE, src));
        }
        if (erasureCodingEnabled) {
            locks.add(lf.getEncodingStatusLock(
                LockType.WRITE, dst));
        }
    }

    @Override
    public Object performTask(StorageConnector connector)
    throws IOException {

```

```

if (NameNode.stateChangeLog.isDebugEnabled()) {
    NameNode.stateChangeLog.debug(
        "DIR* NameSystem.renameTo: with options - " + src + " to
}

if (isInSafeMode()) {
    throw new SafeModeException("Cannot rename " + src, safeMode
}
if (!DFSUtil.isValidName(dst)) {
    throw new InvalidPathException("Invalid name: " + dst);
}
for (MetadataLogEntry logEntry: logEntries) {
    EntityManager.add(logEntry);
}

for (Options.Rename op: options) {
    if (op == Rename.KEEP_ENCODING_STATUS) {
        INode[] srcNodes = dir.getRootDir()
            .getExistingPathINodes(src, false);
        INode[] dstNodes = dir.getRootDir()
            .getExistingPathINodes(dst, false);
        INode srcNode =
            srcNodes[srcNodes.length - 1];
        INode dstNode =
            dstNodes[dstNodes.length - 1];
        EncodingStatus status = EntityManager.find(
            EncodingStatus.Finder.ByNodeId, dstNode.getId());
        EncodingStatus newStatus = new EncodingStatus(status);
        newStatus.setNodeId(srcNode.getId());
        EntityManager.add(newStatus);
        EntityManager.remove(status);
        break;
    }
}

removeSubTreeLocksForRenameInternal(
    src, isUsingSubTreeLocks, subTreeLockDst);

dir.renameTo(
    connector, src, dst, srcNsCount,
    srcDsCount, dstNsCount, dstDsCount, options);
return null;
}
}.handle(this);

```

Listing 6: Lightweight request handler

```
new LightweightRequestHandler(UsersOperationsType.GET_USER_GROUPS) {
    @Override
    public Object performTask(StorageConnector connector)
        throws IOException {
        boolean transactionActive = connector.isTransactionActive();

        if (!transactionActive) {
            connector.beginTransaction();
        }

        Integer userId = cache.getUserId(userName);

        User user;
        if(userId == null) {
            user = userDataAccess.getUser(userName);
        } else {
            user = userDataAccess.getUser(userId);
        }

        if (user == null) {
            return null;
        }

        List<Group> groups = userGroupDataAccess
            .getGroupsForUser(user.getId());

        if (!transactionActive) {
            connector.commit();
        }

        return new Pair<User, List<Group>>(user, groups);
    }.handle(this);
}
```