Distributed Hierarchical File Systems strike back in the Cloud

Mahmoud Ismail*†, Salman Niazi†, Mauritz Sundell‡, Mikael Ronström‡, Seif Haridi*†, and Jim Dowling*†
* KTH - Royal Institute of Technology, † Logical Clocks AB, ‡ Oracle AB
{maism, haridi, jdowling}@kth.se, {mahmoud, salman, seif, jim}@logicalclocks.com,
{mauritz.sundell,mikael.ronstrom}@oracle.com

***Abstract*—Cloud service providers have aligned on availability zones as an important unit of failure and replication for storage systems. An availability zone (AZ) has independent power, networking, and cooling systems and consists of one or more data centers. Multiple AZs in close geographic proximity form a region that can support replicated low latency storage services that can survive the failure of one or more AZs. Recent reductions in inter-AZ latency have made synchronous replication protocols increasingly viable, instead of traditional quorum-based replication protocols. We introduce HopsFS-CL, a distributed hierarchical file system with support for high-availability (HA) across AZs, backed by AZ-aware synchronously replicated metadata and AZ-aware block replication. HopsFS-CL is a redesign of HopsFS, a version of HDFS with distributed metadata, and its design involved making replication protocols and block placement protocols AZ-aware at all layers of its stack: the metadata serving, the metadata storage, and block storage layers. In experiments on a real-world workload from Spotify, we show that HopsFS-CL, deployed in HA mode over 3 AZs, reaches 1.66 million ops/s, and has similar performance to HopsFS when deployed in a single AZ, while preserving the same semantics.**

I. Introduction

The journey to the cloud is fraught. Databases made the transition from strongly consistent single-host systems (relational databases) to highly available (HA), eventually consistent distributed systems (NoSQL systems), and then back to strongly consistent (but also distributed) systems (Spanner [1], CockroachDB [2]). In this paper, we show that distributed hierarchical file systems are completing a similar journey, going from strongly consistent POSIX-compliant file systems to object stores (with their weaker consistency models, but high availability across data centers), and back to distributed hierarchical file systems that are HA across data centers, without any loss in performance.

Over the past decade, advances in networking for public clouds have reduced network latencies between geographically co-located data centers to close to where local area network latencies were a decade ago [3]–[5]. Public cloud providers such as Amazon [6], Google [7], and Microsoft [8] offer cloud infrastructures that are built around regions and availability zones (AZ). Regions are places spread across the world with large geographic distances between regions. A single region consists of one or more AZs (typically 3 AZs) that are physically separate data centers with independent power sources, cooling, and networking. AZs in the same region are connected through low-latency, high throughput network links. Storage services that are HA within a region (they can survive the failure of one or more AZs) can assume much lower and more reliable inter data center network latencies ($< 1ms$) than storage storage services that are HA across different regions ($\approx 10 - 100ms$) [9], [10].

Despite having weaker semantics than POSIX-like file systems, the need for a region-level HA file system in the cloud has made object stores the de facto file system in the cloud. For example, Amazon's S3 object store has become the de facto storage platform for AWS, but its deficiencies compared to POSIX-like file systems include eventually consistent directory listings [11], [12], eventually consistent read-after-update for objects, and the lack of atomic rename. Other object stores, such as Google cloud storage (GCS) and Azure blob store have strengthened S3's semantics to provide strongly consistent directory listings through the use of horizontally scalable strongly consistent metadata [13]–[15]. However, none of these object stores support atomic directory rename, instead, they require the copying of data over the network [16]–[18], which is problematic for both modern data lake frameworks [19]–[21] and SQL-on-Hadoop frameworks that both use atomic directory rename to provide ACID transaction support for updates [22]. What all of these object stores have in common is that they provide HA across AZs, by replicating files (objects) between AZs. Currently, however, there are no distributed hierarchical file systems that we are aware of, including HopsFS [23], that provide high availability over AZs.

In this paper, we introduce HopsFS-CL, a distributed hierarchical file system with integrated support for high availability across AZs. HopsFS-CL is a redesign of HopsFS, an open source, next-generation version of Apache HDFS [24] with strongly consistent horizontally scalable metadata that provides higher throughput and enables larger clusters than HDFS [23]. HopsFS provides POSIX-like semantics and supports consistent directory listings, and rename. HopsFS is composed of three main layers; the metadata storage layer (by default provided by an in-memory open-source database, NDB [25]), the metadata serving layer, and the block storage layer. To make HopsFS HA over AZs, we redesigned all three layers to make them AZ-aware, introducing new AZ-aware synchronous replication protocols for the metadata storage layer (NDB) and block storage layers (HopsFS). We also redesigned the file system metadata operations, such as read and fstat, to be AZ-aware, preferring reading replicas local to the client's AZ - enabled

by synchronous replication protocols, where a full replica of metadata is available on all AZs. In experiments based on a real-world workload from Spotify, we show that deploying HopsFS-CL across three AZs incurs no extra overhead, and delivers similar performance to deploying HopsFS in one AZ, delivering up to 1.66 million ops/sec. We also show that HopsFS-CL outperforms a leading file system backed by an object store, CephFS [26], by up to $2.14X$.

II. Background

In this section, we describe the basic components of HopsFS-CL, introducing both HopsFS [23] and NDB [25].

A. HopsFS

HopsFS [23] is an open-source next-generation distribution of HDFS, that replaces the HDFS single metadata service with a distributed metadata service where the metadata is stored fully normalized in a shared-nothing, in-memory distributed database. HopsFS consists of three main layers; the metadata storage layer, the metadata serving layer, and the block storage layer, as shown in Figure 1.

1) The Metadata Storage Layer

The default and recommended metadata storage layer for HopsFS is NDB which is the MySQL Cluster storage engine [25]. HopsFS provides a pluggable architecture to support, in principle, any database with support for transactions and row-level locking [23]. Other features such as application defined partitioning (ADP) and distribution aware transactions (DAT) are desirable to achieve high performance on HopsFS [27], see Section II-B. The file system metadata is stored as tables in NDB, and the file system operations are implemented as transactions on the stored metadata.

2) The Metadata Serving Layer

HopsFS supports multiple stateless metadata servers (NNs) that can concurrently access the file system metadata stored in the metadata storage layer (NDB) through the use of transactions. The metadata servers use a combination of row-level locks and application defined locks when manipulating the metadata in order to ensure strong consistency of the metadata [23]. To yield high performance, the metadata servers use hierarchical (implicit) locking, that is, the locks are only taken on the inode(s) and the rest of the associated metadata is read using read committed. The metadata servers are responsible for responding to file system requests from potentially thousands of concurrent clients. HopsFS provides the clients with a selection policy to ensure load balancing of the requests between the metadata servers. HopsFS clients select a random metadata server and stick with it until it fails, whereupon they select a random, surviving metadata server. Moreover, HopsFS implements a leader election protocol to elect one of the metadata servers as a leader to be responsible for internal housekeeping operations [28].

3) The Block Storage Layer

Whenever a client uploads a file into HopsFS, the file is split into multiple blocks (128 MB by default), then the blocks are replicated to different block storage nodes (DNs), 3 nodes by default. Users can define a topology for the block storage nodes which can then be used to do ensure fault tolerance in the case of rack-level failures. The block storage layer is responsible only for large files, $>$ 128 KB, while the small files,$<$ 128 KB, are stored with the files' metadata in the metadata storage layer (NDB) [29].

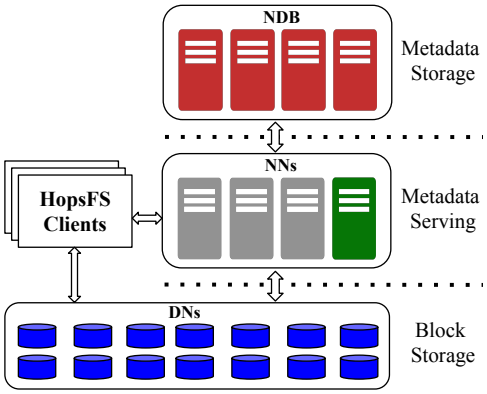

Fig. 1: The architecture diagram of HopsFS. HopsFS consists of three main layers, the metadata storage layer (NDB by default), the metadata serving layer where there is one metadata server that is elected as leader, and the block storage layer where the files' data reside except for small files $<$ 128 KB which reside on the metadata storage layer with their metadata.

B. NDB

NDB is the storage engine of MySQL Cluster [25], [30]. MySQL Cluster is a shared-nothing, in-memory distributed relational database that provides high throughput, high availability and real-time performance.

1) Architecture

A typical NDB setup consists of at least one management node (for configuration and arbitration during network partitions), along with multiple NDB datanodes for storing the table data and handling transactions that access/manipulate the stored data. NDB supports application defined partitioning (ADP), that is, the application controls how the tables are partitioned across NDB datanodes. Transactions are managed by transaction coordinators (TC), with one such TC per NDB datanode. NDB has performance optimizations such as distribution aware transactions (DAT), where a hint (partition key) based on the partitioning scheme can be specified to start a transaction on the NDB datanode containing the data accessed by the transaction. NDB transactions support only read committed isolation level, but row level locks can be used by the applications to provide stronger isolation guarantees to applications. HopsFS implements an application-level locking protocol that provides higher throughput than attainable by the database serializing conflicting transactions [23].

NDB datanodes are organized into node groups (replication groups). Given a cluster with N NDB datanodes, and R as the replication factor, the number of node groups is N/R. A partition is a fragment of data stored and replicated by a node group. Each NDB datanode in a node group stores a replica of the partition assigned to the node group. The default replication factor in NDB is 2, that is, each node group can tolerate the failure of 1 NDB datanode. For each partition, one

NDB datanode will be assigned the primary replica, while the other datanodes from the same node group will be assigned as backup replicas.

2) NDB Commit Protocol

NDB uses the Strict Two Phase Locking protocol [31] for concurrency control. The protocol consists of two phases. First, it acquires all the required locks, and then it releases the locks only once the transaction reaches the commit point. To avoid deadlocks, NDB always locks the row on primary replica first then the rows on backup replicas.

NDB implements a variant of Two Phase commit (2PC) protocol that is non-blocking and distributed [30]. The NDB commit protocol leverages the linear 2PC protocol [32] for each row in the transaction to decrease the number of messages exchanged compared to classic 2PC [30]. Figure 2 shows the NDB commit protocol in action while committing a transaction that writes two rows into two different partitions. The commit protocol starts at the transaction coordinator (TC) by sending the *Prepare* message to the primary replica, which in turns forwards the message to the backup replicas. Once the last backup replica finishes processing the *Prepare* message, it sends the *Prepared* message to the TC. Then, the TC sends the *Commit* message in the reverse order to the backup replicas and traverses back to the primary replica. Due to the reverse order of the *Commit* message, the TC is no longer the commit point. Instead, the primary replica commits the changes and releases the acquired locks, and then sends the *Committed* message to the TC. Once the TC receives all the *Committed* messages from all primary replicas, it acknowledges the client node that the transaction is committed, and it is safe to start reading your updates. In parallel, for performance, the TC sends the *Complete* message to the replicas to release the locks on the backup replicas and clean the memory used during the transaction. There is a short time window, at the end of the transaction, but before the backup replicas receive the *Complete* message, where the backup replicas might be out of date. Therefore, the default behaviour in NDB is to always redirect read committed reads to the primary replicas. For reads with shared or exclusive locks, it is always guaranteed to read the latest committed data since all locked reads go to the primary replica.

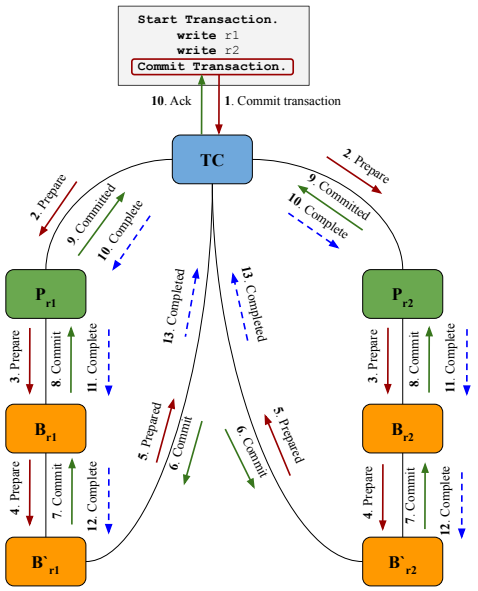

Fig. 2: A simple description of the NDB commit protocol to commit a transaction writing two rows (r1, r2) to two different partitions. P_{r1} is the primary replica for row $r1$ while B_{r1} and $B`_{r1}$ are the backup replicas. Similarly, $P_{r2}$ is the primary replica for row $r2$ while $B_{r2}$ and $B`_{r2}$ are the backup replicas.

In the case of failure of a TC (which typically blocks transactions in other 2PC implementations), NDB implements a take-over protocol that is used by a new TC to rebuild the transaction state of ongoing transactions that have lost their TC. Since all replicas reside in the same node group, a failure of an NDB datanode in the node group does not prevent the protocol from making progress. NDB implements node failure and heartbeat protocols to ensure agreement on which nodes have failed among surviving nodes in the cluster. To ensure liveness, NDB uses different timeouts such as *TransactionInactiveTimeout* to abort the transaction if the client abandoned it, as well as *TransactionDeadlockDetectionTimeout* to abort the transaction in cases of node failures, high load, and deadlocks. HopsFS uses these timeouts to implement a transaction retry mechanism providing backpressure to NDB. Moreover, NDB implements a global checkpoint protocol across node groups to allow system recovery in cases of cluster failures.

III. Challenges to deploy HopsFS in the cloud

The challenges to deploy HopsFS in the cloud are as follows:

a) ***C1****: HA deployments of HopsFS across AZs*

We cannot deploy HA HopsFS across AZs, since the three layers of HopsFS are not AZ aware. That is, we cannot guarantee the availability of HopsFS in case of AZ failure.

b) ***C2****: AZ local communications.*

The latency between two nodes in the same AZ is lower than across two different AZs in the same region, as shown in Table I for three AZs in **us-west1** region in Google Cloud Platform. Also, network traffic within the same AZ is typically free, whereas the cost of network traffic across AZs may not be insignificant [33]. For these reasons, it is desirable to minimize cross-AZ network traffic.

	us-wes1-a	**us-west1-b**	**us-west1-c**
us-west1-a	***0.247***	0.360	0.372
us-west1-b	0.360	***0.251***	0.399
us-west1-c	0.372	0.399	***0.249***

TABLE I: Measured latencies in milliseconds between two different virtual machines in Google Compute Engine (GCP) located on different AZs in **us-west1** region.

We introduce HopsFS-CL in order to address the aforementioned issues (**C1**-**C2**). HopsFS-CL tackles these challenges by efficiently deploying across AZs as well as introducing AZ-awareness at the three layers of HopsFS-CL; the metadata storage, see Section IV-A, the metadata serving, see Section IV-B, and the block storage layers, see Section IV-C. HopsFS-CL

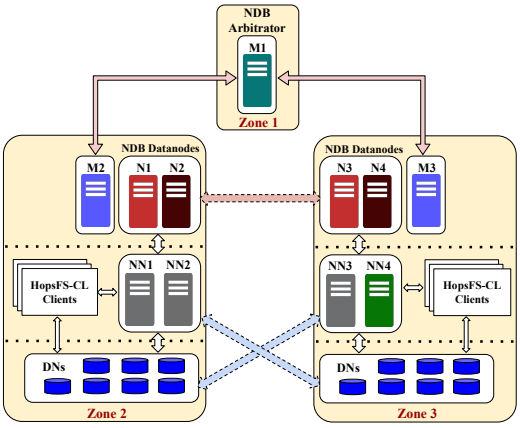

Fig. 3: A deployment diagram for HopsFS-CL. HopsFS-CL runs across three different AZs ($Zone1$, $Zone2$, and $Zone3$). Both $Zone2$ and $Zone3$ contain a replica of the metadata stored in the metadata storage layer, a management node, a set of metadata servers, a set of block storage servers (DNs), and a set of HopsFS-CL clients. The metadata storage layer has a replication factor 2, where $N1$ and $N3$ form a node group and $N2$ and $N4$ form another node group. A third management node runs on $Zone1$ and acts as an arbitrator in the case of a network partition between $Zone2$ and $Zone3$.

consistently uses synchronous replication protocols at all layers to ensure that a full replica of the file system resides at each AZ. By doing so, we address the challenge **C1**. As each AZ has a full copy of the file system, we are able to redesign metadata and file system protocols as AZ-aware, preferring AZ local operations, thus, addressing challenge **C2**.

IV. AZ-Awareness throughout the Stack

In this section, we present our approach to achieving *AZ-aware* high availability in HopsFS-CL by showing the different techniques and algorithms introduced across all the layers from the metadata storage layer to the metadata serving layer and the block storage layer.

A. Metadata Storage Layer

The default metadata storage layer is NDB. To add *AZ awareness* into NDB, we introduce three new features to NDB. Firstly, we introduce a new configuration parameter (*LocationDomainId*) that assigns each node in NDB to a specific AZ within the cloud. Secondly, we introduce a new table option *Read Backup* that allows transactions to read from the table's primary and backup replicas consistently using Read Committed, that required changes to the NDB commit protocol, see Section II-B2. Thirdly, we introduce another table option *Fully Replicated* that enables the table's partitions to be replicated on all NDB datanodes in the cluster which trades off slower writes for potentially faster reads. Moreover, we changed the ordering mechanisms used by NDB to account for different AZs. Also, we adjusted the transaction coordinators selection policy to take AZs and table options into consideration.

1) Deploying across Three AZs

With the use of the newly introduced *locationDomainId*, we can mark NDB datanodes from the same node group to be on different AZs to ensure high availability of their data. For example, as shown in Figure 3, $N1$ and $N3$ are part of the same node group but are deployed on two different AZs, $Zone2$ and $Zone3$, respectively. Similarly, $N2$ and $N4$ are placed on different AZs. With this configuration, the metadata of HopsFS-CL is replicated across $Zone2$ and $Zone3$. Each AZ $Zone2$ and $Zone3$ has an NDB management node. Also, a third management node added at $Zone1$ to act as arbitrator in case of network partition scenarios. Moreover, we can take advantage of all 3 AZs by increasing the NDB replication factor to 3, see Figure 4. $N1$, $N3$, and $N5$ form a node group while $N2$, $N4$, $N6$ form the other node group.That is, the HopsFS-CL metadata is replicated across the 3 AZs.

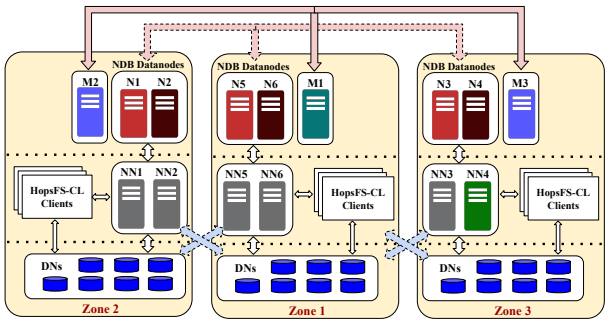

Fig. 4: A deployment diagram for HopsFS-CL. HopsFS-CL runs across three different AZs ($Zone1$, $Zone2$, and $Zone3$). Each zone contains a replica of the metadata stored in the metadata storage layer, a management node, a set of metadata servers, a set of block storage servers (DNs), and a set of HopsFS-CL clients. The metadata storage layer has a replication factor of 3, where $N1$, $N3$, and $N5$ form a node group and $N2$, $N4$, and $N6$ form another node group. The management node $M1$ acts as the arbitrator in the case of a network partition between AZs.

2) Failures

Failures could happen at the machine level or even more drastically at the AZ level. NDB guarantees that a failure in a node in a node group will not block the currently running transactions from proceeding and will not bring the cluster down as long as there is still other alive nodes in the node groups, see Section II-B2. The surviving nodes in the node group will upgrade their backup partitions to primary partitions to account for the failed primary partitions in the failed node. For example, in Figure 4, If $N1$ failed, then other nodes in the node group $N3$ and $N5$ will upgrade their backup partitions to account for the failed primary partitions in $N1$. Network partitions could arise between any two zones $Zone2$ and $Zone3$, see Figure 3, which would result in a split-brain scenario. To avoid such a scenario, we added a third management node on $Zone1$ that acts as an arbitrator in case of network partitions. During network partitions, the arbitrator accepts the first set of database nodes to contact it and tells the remaining set to shutdown. The database nodes will assume they are in a network partition and shutdown gracefully if they failed to contact the arbitrator. Similarly, in Figure 4, each AZ has a management node, $M1$ is elected as the arbitrator. If $M1$ failed, another management node will be elected as the arbitrator. The setup in Figure 3 can tolerate the failure of 1 AZ, while the setup in Figure 4 can tolerate up to 2 AZs failure given the existence of an external management node acting as arbitrator (fortunately, cloud providers have support for regions with four or more AZs).

3) Read Backup and Fully Replicated features

We introduce the *Read Backup* feature as a table option. Internally, when committing a transaction, we ensure that the changes on the primary replica and the backup replicas are completed before responding to the client ensuring that the client will be able to read his own updates on either the primary or the backup replicas. We changed the commit protocol described in Section II-B2 to delay sending the acknowledgement to the client until all the replicas are up to date. That is, the TC delays sending the *Ack* message to the client until receiving the *Completed* message from all the backup replicas. Therefore, in Figure 2, the *Ack* message will be sent after receiving all the *Completed* messages, and as such the *Ack* message number will be 14 instead of 10. Moreover, we introduce the *Fully Replicated* feature as a table option. When committing a transaction on a *Fully Replicated* table, we use the linear 2PC on all the primary replicas of the changed rows on all node groups. Similar to the Read Backup, we delay the transmission of the *Ack* message until the reception of all *Completed* messages.

4) NDB datanodes ordering

NDB orders the database nodes for a transaction based on distribution awareness (DAT), using the hint (partition key) provided at the start of the transaction, then based on the proximity score. The proximity score between two nodes is a measure of the expected latency between them. We modified the proximity score order to take into account the addition of the AZs, through the *LocationDomainId*. We define the new proximity score in ascending order as follows:

1) Two nodes on the same host and the same AZ.
2) Two nodes on different hosts but within the same AZ.
3) Two nodes on different hosts but in two different AZs.

5) Transaction Coordinator Selection Policy

We adjusted the transaction coordinator selection policy to select the best transaction coordinator whenever we start a transaction to ensure that the transaction coordinator is located within the same AZ as the caller. In HopsFS, file system operations are implemented as transactions on the file system metadata stored in NDB. When a metadata server (NN) in HopsFS receives a request, it will start a transaction on NDB. NDB first uses the hint supplied by HopsFS (partition key) to locate the nodes that hold the partitions for the transaction's data. The nodes returned based on the hint are ordered with the primary replica first, followed by backup replicas. Then, we determine which node of these nodes should be selected as the transaction coordinator for that transaction based on the locationDomainId, table options, and the proximity score described in Section IV-A4. As a rule of thumb, we always select the transaction coordinator located on the same AZ as the metadata server. There are four different cases for selecting the transaction coordinator depending on the table options as follows:

a) Case 1: The table is read backup enabled.

We select the local replica to our AZ, it could be the primary or a backup replica.

b) Case 2: The table is fully replicated.

We use all the nodes for that table, not only the ones based on the partition key. For a fully replicated table we will have a replica of the table's partitions on every node, so we select a node in the same AZ based on the proximity score.

c) Case 3: The table is neither fully replicated nor read backup enabled, and there are nodes based on the partition key for that table.

This is the default behaviour. We select the node that reside in the same AZ. It could be the primary replica or a backup replica. However, if the backup replica was selected, all the reads will be rerouted to the node holding the primary replica unlike the read backup and fully replicated tables.

d) Case 4: There are no nodes based on the provided partition key.

This is the fallback mechanism where no nodes have been found for the provided partition key. In that case, we use all the NDB datanodes, and we select a node based on the proximity score. That is, preferring nodes in the same AZ. However, similar to $Case3$, if a backup replica was selected, all the reads will be rerouted to the nodes holding the primary replica.

After selecting the best transaction coordinator, NDB runs the transaction on the selected transaction coordinator. If the transaction coordinator doesn't have some/all of the requested data, then it will issue requests to other NDB datanodes while respecting the AZ awareness. The latency between AZs is usually higher, and inter-AZ bandwidth costs money, while intra-AZ bandwidth is usually free, see Table I. Thus, in HopsFS-CL, we ensure that all the tables are *Read Backup* enabled, to force NDB to route read transactions to both primary and backup replicas, which we show in Section V-E reduces network traffic between the AZs.

B. Metadata Serving Layer

The metadata servers (NNs) are responsible for serving requests from the file system clients. We introduce a configuration parameter *locationDomainId* to the metadata servers following the same approach as the metadata storage layer. Administrators can set the *locationDomainId* for each of their metadata servers. However, it has to be the same id that was set for the metadata storage to ensure AZ-local file system transactions.

1) Deploying across Three AZs

With the use of the newly introduced *locationDomainId*, we can tag metadata servers with their AZ, as well as HopsFS-Cl clients. For example, in Figure 3, $NN1$ and $NN2$ reside in $Zone2$, similarly $NN3$ and $NN4$ reside in $Zone3$. Also, we can set the same for HopsFS-CL clients as shown in Figure 3. Moreover, we can take advantage of the 3 AZs by deploying metadata servers and clients across all of them, see Figure 4. In both setups, we have the leader metadata server running on $NN4$ in $Zone3$.

2) Failures

The metadata servers are stateless servers. Therefore, HopsFS-CL can tolerate the failure of up to $N-1$ metadata servers for a cluster with N metadata servers. The leader metadata server can go down due to machine failure, network failure,

or AZ failure. A leader election protocol [28] ensures a new leader is elected.

3) Metadata server selection policy

We implemented a selection policy for the HopsFS-CL clients to ensure AZ-local metadata servers, that is, the clients can choose metadata servers running on the same AZ as them. The metadata servers are stateless with no communication between them except indirectly via their leader election protocol [28]. Therefore, we extended the leader election protocol to allow each metadata server to report its *locationDomainId* at every leader election round, by default every 2 seconds. When creating a new HopsFS-CL client, first it requests the list of active metadata servers from the leader metadata server. Then, the client chooses the metadata server with the same *locationDomainId*. Otherwise, a random metadata server is chosen. If the *locationDomainId* was set to 0, then we also fall back to a random metadata server. Systems running on top of HopsFS-CL can ensure that their applications run on the same AZ as the metadata servers by setting the *locationDomainId* to the same AZ.

C. Block Storage Layer

The Block storage layer is responsible for storing blocks of data for large files, $> 128KB$, in HopsFS-CL. The files are divided into blocks, typically $128MB$, and replicated across typically 3 block storage servers for high availability.

1) Deploying across Three AZs

We can use the same concept of the *locationDomainId*, and then devise a block placement algorithm to ensure that at least one of the three replicas resides on another AZ for fault tolerance. However, the rack-aware block placement policy in HopsFS (for on-premises installations) can be adapted to work with AZs instead of racks. Instead of introducing a new configuration parameter, we can use the existing topology configuration file to configure the block storage servers as if they are running on 2 or 3 racks, where the racks are in fact the AZs. With such a configuration, the current block placement policies will ensure that at least one replica resides on every AZ. Thus, ensuring high availability at the block storage layer, in the event of failure of an AZ. On the other hand, for small files, $< 128KB$, we ensure the locality and high availability of the data, since the NVMe disks that are used to store the small files' data are co-located with their NDB datanodes. The datanodes are configured such that the primary and the backup replicas reside on different AZs, see Figure 3 and Figure 4.

2) Failures

Block storage servers can fail due to machine failures or AZ wide failures. The block replication level is configurable while the default is 3. That is, HopsFS-CL tolerates the failure of up to 2 block storage servers. Once a failure is detected a re-replication event is triggered by the leader metadata server to maintain the block replication level. In case of small files, the failures are handled by the metadata storage layer, see Section IV-A2.

V. Evaluation

In this section, we evaluate the performance of HopsFS-CL (with AZ awareness) against vanilla HopsFS. Also, we benchmarked HopsFS-CL in comparison to a leading file system backed by an object store, CephFS. All our experiments were run on virtual machines (VMs) on Google Compute Engine in *us-west1* region with 32vCPU and 29 GB of memory. Depending on the system configuration, we ran our experiments either on one AZ *us-west1-b* or across three AZs *us-west1-a*, *us-west1-b*, and *us-west1-c*. We used HopsFS v2.8.2.5, NDB v7.6.8, and Ceph v13.2.4. To evaluate our system we used a real-world industrial workload from Spotify's Hadoop cluster [23]. We used the benchmarking tool introduced in [23]. In our experiments, we focus on the evaluation of the metadata storage and the metadata serving layer, the traditional metadata bottleneck in distributed hierarchical file systems. We did not evaluate the performance of the block layer as concurrency control is applied at the metadata layer, while the block layer scales linearly to tens of thousands of HopsFS clients and datanodes. As such, our experiments only used empty files, files of zero length.

A. Experiment Setup

a) HopsFS and HopsFS-CL setup:

We benchmarked different deployments of (vanilla) HopsFS to construct the baseline for the comparison against HopsFS-CL. For each setup, we deployed NDB cluster with 12 NDB datanodes where each NDB datanode is configured with 27 threads locked to separate CPUs, see Table II. The metadata replication factor is the NDB replication factor which controls the number of replicas for the file system metadata. We deployed HopsFS in four different setups. Each setup is identified as an ordered tuple where the first item is the NDB replication factor and the second item is the number of AZs used. For example, *HopsFS (2,1)* is a deployment of HopsFS in 1 AZ with the metadata replication factor set to 2. After constructing the baseline, we deployed HopsFS-CL in HA setup with 3 AZs while varying the NDB replication factor. Figure 3 shows the HA setup where metadata replication factor is set to 2. Similarly, Figure 4 shows the HA setup where the metadata replication factor is set to 3.

Type	Count	Responsibility
LDM	12	tables' data shards.
TC	7	on going transactions on the database nodes.
RECV	3	inbound network traffic.
SEND	2	outbound network traffic.
REP	1	replication across clusters.
IO	1	I/O operations.
MAIN	1	schema management.

TABLE II: The NDB CPU configuration. We used 27 CPUs. For each type, we locked the number of required CPUs.

b) CephFS:

A typical CephFS cluster consists of a monitor node (MON), a set of object storage daemons (OSD), and a set of metadata servers (MDS). The OSD nodes act as the storage layer for the metadata of the file system, as well as the data. For a fair comparison, we used 12 OSD nodes similar to the 12 NDB nodes in HopsFS and HopsFS-CL. In HopsFS/HopsFS-CL,

the metadata servers are stateless servers that are, in parallel, manipulating the file system's metadata stored in NDB. On the other hand, in CephFS, each metadata server is responsible for a subtree of the file system where it caches the file system metadata and periodically writes the list of operations to an operation log in the OSDs. CephFS uses a dynamic partitioning algorithm to partition the file system namespace between all metadata servers in the cluster [34]. We deployed CephFS in high availability setup across 3 AZs with the metadata replication factor set to 3. At first, we used the CephFS Hadoop plugin to test the performance of CephFS, however, the results were unreliable due to some unidentified bottlenecks in the plugin as it was not tested by the CephFS community for years. Therefore, we decided to use the CephFS native kernel driver to mount the file system to the experiment servers. We have three different setups for CephFS:

1) **CephFS:** This is the default and recommended setup for CephFS where the metadata servers (MDSs) use the default dynamic partitioning algorithm to load balance subtrees across the metadata servers.

2) **CephFS - DirPinned:** In this setup, to improve throughput, we manually assign each metadata server (MDS) to a subtree of the file system. Thus, we manually enforce the load balancing of the file system subtrees across the metadata servers - at the cost of location transparency for clients, overloading the MDS in case of hotspots, and increased failover time for metadata server failures.

3) **CephFS - SkipKCache:** In our experiments, inodes are created and then right after they are available for the file system clients to read/write. In CephFS, whenever a kernel client wants to operate on an inode, the MDS server responsible for that inode grants the corresponding capabilities to that client. Then, the kernel client caches the files' metadata in memory as long as they still have a valid capability from the MDS. However, that comes at a cost, that the MDSs have to keep track of all clients capabilities, so they can notify the clients' when a change happen to an inode capabilities which will potentially leads to higher failover time. In this setup, we always skip the kernel cache in order to evaluate the actual performance of the MDS.

B. Throughput

1) Industrial workload

We benchmarked all different setups of HopsFS, HopsFS-CL, and CephFS using a real-world workload based on operational traces from Spotify's Hadoop cluster [23]. Figure 5 shows the throughput of all the 9 setups introduced in Section V-A while varying the number of metadata servers in the cluster. HopsFS deployed in 1 AZ with a metadata replication factor set to 2 (*HopsFS (2, 1)*) delivers up to 1.62 million ops/sec with 60 metadata servers. These performance numbers are 30% higher than the last reported benchmarks on an on-premise cluster [23] and are mainly due to the performance improvements in HopsFS and the use of servers with 25% more CPUs. By increasing the metadata replication factor to 3 (*HopsFS (3, 1)*), the throughput drops to 1.56 million

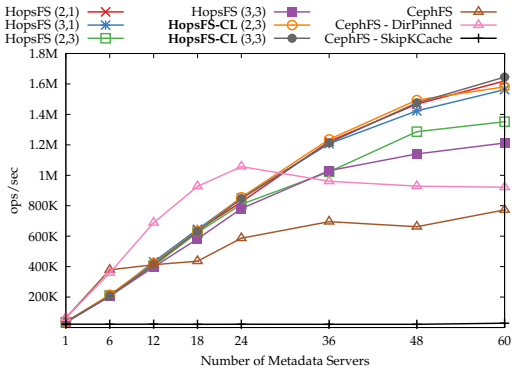

Fig. 5: The throughput of HopsFS, HopsFS-CL, and CephFS for Spotify's workload.

ops/sec. Deploying a highly available HopsFS cluster across 3 AZs drops the throughput by 17% in case of 2 metadata replicas (*HopsFS (2, 3)*) and by 22% in case of 3 metadata replicas (*HopsFS (3, 3)*). The main reason for the performance drop of HopsFS across 3 AZs is the higher network latency between AZs and the lack of AZ awareness in HopsFS. That is, inter-server connections cross more AZ boundaries than strictly neccessary - keeping more communication local within AZs should improve performance. HopsFS-CL provides AZ awareness and delivers similar throughput to HopsFS when deployed in one AZ, as expected. HopsFS-CL configured with 2 metadata replicas (*HopsFS-CL (2, 3)*) on 3 AZs delivers up to 17% higher throughput compared to HopsFS (*HopsFS (2, 3)*). For 3 metadata replicas on 3 different AZs, HopsFS-CL delivers even higher gains in throughput (up to 36% more than HopsFS). We see that the performance gap between HopsFS and HopsFS-CL keeps increasing for an increasing number of metadata servers in the cluster beyond 24. Our conclusion is that network I/O becomes a bottleneck, and adding AZ awareness helps improve throughput.

The default and recommended CephFS setup delivers up to 0.77 million ops/sec. For the same configuration, HopsFS-CL delivers $2.14X$ higher throughput than CephFS. However, the CephFS throughput does not increase linearly as one would expect for an increasing number of metadata servers. Therefore, we benchmarked another setup of CephFS (CephFS-DirPinned) where we manually assigned different subtrees to different metadata servers, as shown in Figure 5. However, the throughput drops after 24 metadata servers due to the single threaded nature of the MDS and the disk utilization, as shown in Section V-D1.

Moreover, we benchmarked a third setup of CephFS (CephFS - SkipKCache) where we skipped the kernel cache of the mounted CephFS to evaluate the actual number of requests handled by the MDS. By skipping the kernel cache, CephFS handles only $28K$ ops/second with 60 metadata servers. Figure 6 shows the actual number of metadata requests handled per metadata server. CephFS-DirPinned can handle up to 4233 requests per second with 1 MDS and it drops to a 1178 requests per second with 60 MDSs which correlates with previously published benchmarks in the CephFS paper [26]. HopsFS-CL handles up to $23X$ more requests than the CephFS - DirPinned,

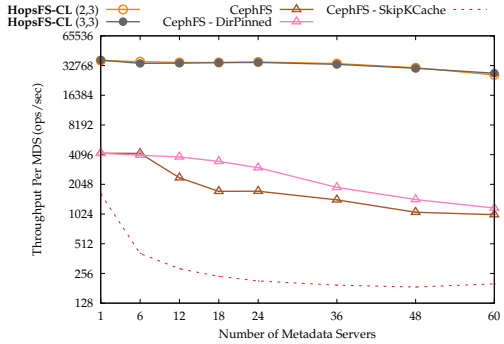

Fig. 6: The actual throughput per metadata server for HopsFS-CL and CephFS using Spotify's workload. The Y-axis is in log scale of base 2.

which is expected for CephFS, since most of the requests are handled locally by the kernel cache, however, for HopsFS-CL, all the requests go to the metadata servers where no client cache is involved.

2) Synthetic workload

In this experiment, we ran micro-benchmarks for the most popular file system operations; mkdir, createFile, readFile, and deleteFile using a cluster with 60 metadata servers. Figure 7 shows the throughput for the different systems: HopsFS, HopsFS-CL, and CephFS. First we notice that increasing the metadata replication factor from 2 to 3 drops the throughput of HopsFS and HopsFS-CL for file system operations that mutate the metadata, such as mkdir, createFile, and deleteFile. For instance, the throughput of HopsFS deployed in one AZ drops by up to 45%. Similarly, the throughput drops for three AZ deployments of HopsFS and HopsFS-CL by up to 23%. However, for read file operation, increasing the metadate replication factor from 2 to 3 accounts for up to 6% increase in the throughput. HopsFS-CL delivers up to $11.8X$ higher throughput than CephFS for file system operations that mutate the metadata (mkdir, createFile, and deleteFile). On the other hand, for read file operation, CephFS delivers up to $1.9X$ higher throughput of HopsFS-CL due to the use of the kernel cache in CephFS. By skipping the kernel cache, HopsFS-CL delivers up to $81X$ the throughput of CephFS.

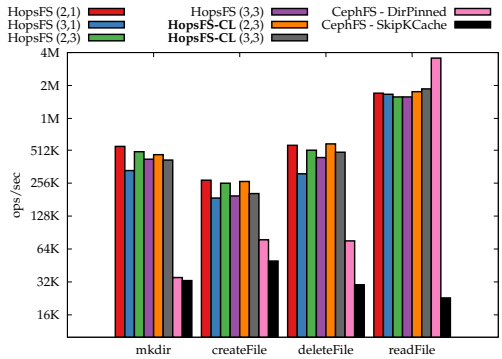

Fig. 7: The throughput of HopsFS, HopsFS-CL, and CephFS for the most common file system operations using 60 metadata servers. The Y-axis is in log scale of base 2.

C. End-to-End latency

In this experiment, we measured the end to end latency for file system operations while varying the number of metadata servers in the cluster. We benchmarked different setups for HopsFS, HopsFS-CL, and CephFS using the Spotify's workload. Figure 8 shows that HopsFS and HopsFS-CL deployments achieve almost constant average end-to-end latency, $\approx 8-14$ milliseconds, while varying the number of metadata servers in the cluster. HopsFS-CL achieves up to 35% lower latency compared to HopsFS deployments across three AZs (*HopsFS (2,3)* and *HopsFS (3,3)*) due to their lack of support for AZ awareness. HopsFS-CL achieves up to $9X$ lower average latency then CephFS. CephFS - DirPinned achieves up to $1.9X$ lower average latency than HopsFS-CL due to the use of the kernel cache. By skipping the kernel cache in CephFS, HopsFS-CL achieves up to $16X$ lower average latency then CephFS.

Moreover, we measured the latency taken to create, read, and delete a file in an unloaded cluster. An unloaded cluster is a cluster which delivers 50% of the full throughput of the cluster. We used 60 metadata servers for the different deployments of HopsFS, HopsFS-CL, and CephFS. Figure 9 shows the 50th, 90th, and 99th percentile for the latency of createFile, readFile, and deleteFile operations. The noticeable difference is that CephFS delivers significantly lower latency than HopsFS and HopsFS-CL. The main reason is that CephFS processes most of file system operations either locally within the kernel cache or in the main memory of the metadata servers. However, when increasing the load on metadata servers, the latency increases, as shown in Figure 8, due to the single threaded nature of the metadata server and the journal flushing time which reduces available resources for processing file system operations.

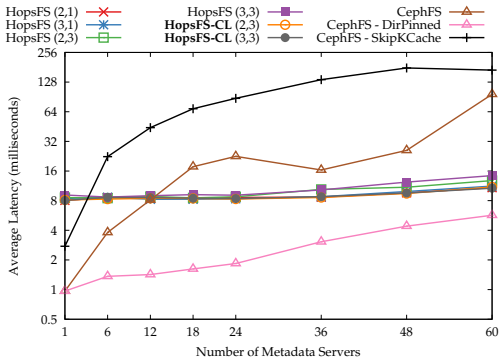

Fig. 8: The average end-to-end latency of file system operations for Spotify's workload. The Y-axis is in log scale of base 2.

D. Resource utilization

We collected resource utilization statistics while running the experiments in Section V-B1.

1) The metadata storage layer

In our experiments, HopsFS and HopsFS-CL use NDB as the storage layer for the file system metadata, while CephFS uses their object storage daemons (OSD) as the storage layer. Figure 10(a) shows the CPU utilization of the storage layer in HopsFS, HopsFS-CL, and CephFS. For HopsFS and HopsFS-CL, the CPU utilization of NDB increases for an increasing number of metadata servers in the cluster until it reaches a plateau after 12 metadata servers. On the other hand, for

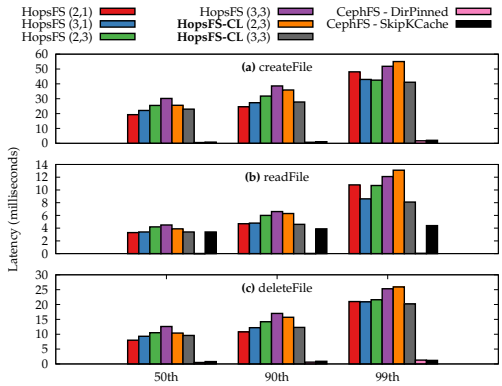

Fig. 9: The 50th, 90th, and 99th percentile of the latency of createFile, readFile, and deleteFile operation for HopsFS, HopsFS-CL, and CephFS. We used 60 metadata servers with 50% load.

CephFS, the CPU utilization of the OSD remains almost constant even when the number of metadata servers is increased. The NDB cluster datanodes were configured with 27 threads locked to 27 CPUs, see Table II. Figure 11 shows the average CPU utilization per thread type for the highly available HopsFS-CL (*HopsFS-CL (3,3)*). The CPU utilization of NDB reaches a peak after 24 metadata servers. However, HopsFS and HopsFS-CL still provide higher throughput while increasing the number of metadata servers, see Figure 5, and that is due to more batching of requests by NDB. Even though the REP thread saturates at almost 90%, we had no ongoing replication across clusters, and the reason for the high utilization is that idle threads will try to help other busy threads such as RECV and SEND to share their load.

The network utilization of the NDB increases linearly for an increasing number of metadata servers, since more file system operations require more data to be read from NDB, see Figure 12(a) and Figure 12(b). On the other hand, CephFS serves most of the file system requests from either the kernel cache or the metadata server without requiring to contact the metadata storage layer (OSD). Therefore, network utilization on the OSD is quite low and doesn't change that much with the increased number of metadata servers. The OSD is neither

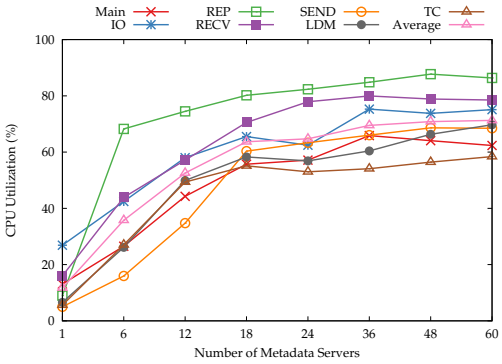

Fig. 11: The average CPU utilization per NDB thread type for HopsFS-CL (3,3).

CPU intensive nor network intensive, but, it is disk intensive, see Figure 12(d).

The disk utilization of the OSD increases linearly for an increasing number of metadata servers, until it reaches a plateau after 24 metadata servers, see Figure 12(d). The main reason for the increased load is the journalling done by the metadata server for each operation, where, the MDS periodically flush the journal to the OSDs. The journalling along with the MDS' single-threaded design explains the drop in throughput that can be seen after 24 metadata server for CephFS - DirPinned, see Figure 5. The disk utilization for NDB doesn't increase that much while increasing the number of metadata servers. As NDB is primarily an in-memory database, disk utilization only increases for the writing of the REDO log and checkpoints that are used for recovery.

2) The metadata serving layer

The metadata server in HopsFS and HopsFS-CL uses a granular locking mechanism allowing it to fully utilize all of the CPUs on its servers, see Figure 10(b). On the other hand, the metadata server in CephFS is single-threaded and uses a global lock preventing it from fully utilizing all available CPUs on its servers. In HopsFS and HopsFS-CL, the metadata servers process up to one order of magnitude more requests through the network than CephFS, see Figure 13(a) and Figure 13(b).

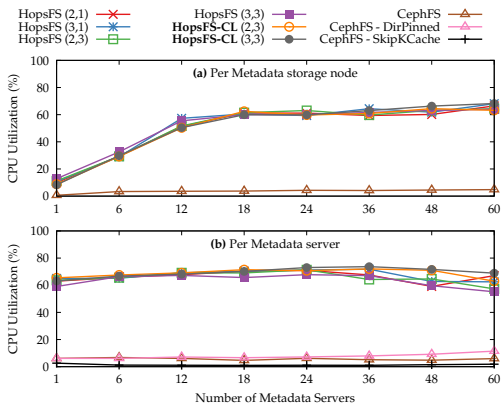

Fig. 10: The average CPU utilization per **(a)** metadata storage node and **(b)** metadata server. HopsFS and HopsFS-CL uses NDB as the metadata storage layer while CephFS uses a native object store (OSD).

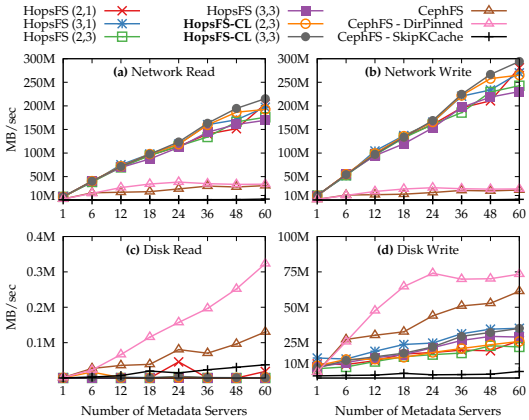

Fig. 12: The average network and disk utilizations of the metadata storage layer. HopsFS and HopsFS-CL uses NDB as the metadata storage layer while CephFS uses underling object store (OSD).

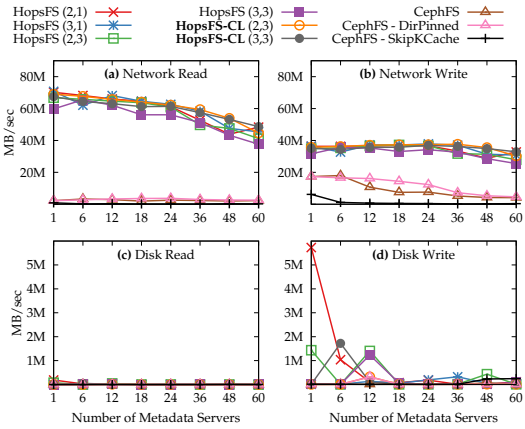

Fig. 13: The average network and disk utilizations per metadata server for HopsFS, HopsFS-CL, and CephFS.

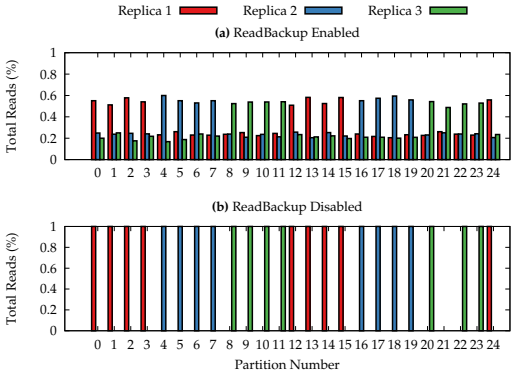

Fig. 14: As we can see for 24 partitions, without read backup all reads go the primary (which may not be AZ local), while with read backup enabled, we have the expected number of reads for the primary and the backup replicas 50% and 25%, respectively. This shows that reads are AZ-local.

The main reason is that in CephFS most of the file system metadata requests are handled directly by the kernel cache in the client side. HopsFS, HopsFS-CL, and CephFS do not use that much disk in the process since all work either is done in memory or through the network, see Figure 13(c) and Figure 13(d).

E. AZ-Local Reads

HopsFS-CL enables read backup feature on the metadata storage layer (NDB), enabling clients to read from the backup replicas (when shared or exclusive locks are not taken), instead of always going to the primary replica. That is, clients can read from a replica within the current AZ. We ran an experiment using the Spotify's workload for two setups read backup enabled and read backup disabled. Then, we collected the statistics from NDB for both setups. Figure 14 shows the effect of enabling and disabling read backup, for clarity we show only the first 24 partitions. Figure 14(b) shows that all requests go to the primary replicas in case of read backup being disabled, while Figure 14(a) shows that the requests are balanced between the backup replicas and the primary replica.

F. Failures

The main motivation for supporting AZ awareness in HopsFS-CL is to tolerate failures of up to 2 AZs in a cloud region. The highly available HopsFS-CL uses an NDB cluster with replication factor set to 3. That is, HopsFS-CL can tolerate up to 2 failures of NDB datanodes within the same node group. Fortunately, with AZ awareness enabled, HopsFS-CL can tolerate the failure of up to 2 AZs given the existence of an external management node acting as arbitrator. HopsFS-CL metadata servers are stateless, and their failures shouldn't affect the whole file system. Therefore, a cluster with N metadata servers can tolerate failures of up to $N-1$ metadata servers. HopsFS-CL also supports AZ awareness at the block storage layer, where a block is guaranteed to be replicated across the 3 AZs. That is, a failure of 1 or 2 AZs won't bring the file system down. Moreover, in split brain cases where a network partition occurs between any two different AZs, the NDB management node in the third AZ will act as an arbitrator, shutting down servers in one of the network partitions. If an AZ has network connectivity with the arbitrator, then the file system within this AZ will continue to function as normal. Otherwise, if an AZ doesn't have network connectivity with the arbitrator, it will shut down the file system partition in that AZ.

VI. Related Work

Object stores are commonly used as cloud storage backends in public clouds, such as Amazon S3, Google Cloud Storage, and Azure Blob Storage. Public cloud providers offer different types of storage buckets allowing data objects to be replicated across AZs or regions within the same geographical area. Since these object stores are API-request rate-limited, we did not compare performance with them. Other Object stores, such as Ceph provide a file system interface. CephFS [26] is a POSIX-compliant distributed file system that stores its metadata and data as objects on the Ceph object storage nodes (OSD). On top of the OSDs, CephFS provides metadata servers (MDS) that manage the file systems' files and directories, and coordinate the security and consistency of the file system. File system operations on the MDS are single threaded due to the use of the MDS global lock which in turns limit the CPU utilization and performance of the MDS [35]. Ceph is popular for deployments on public or private cloud. Similar to HopsFS-CL, CephFS can be configured to be highly available across AZs by correctly placing the object storage nodes across AZs and then defining the unit of failure as the AZ.

File systems such as BlueSky [36] and SCFS [37] provide a file system backed by cloud storage. However, due to the weaker consistency semantics provided by the cloud storage, SCFS implements a coordination service on top to provide strong consistency independently of the guarantees provided by the storage clouds. On the other hand, HopsFS-CL provides strong consistency semantics based on locking primitives introduced in HopsFS, as well as, providing high availability across AZs. CalvinFS [38] is a distributed file system that uses a distributed shared-nothing database for file system metadata management. The file system operations are executed as deterministic transactions that are batched and logged into a global meta-log that is replicated across data centers. With the use of the meta-log, CalvinFS can survive the failure of a whole data center. However, CalvinFS is optimized for operations

on single files rather than subtree operations that might take considerably longer time compared to traditional file systems.

VII. Conclusions

In this paper, we introduced HopsFS-CL, a highly available distributed hierarchical file system with native support for AZ awareness using synchronous replication protocols. In experiments using a Spotify workload, we showed that our AZ aware optimizations for a HA HopsFS-CL cluster deployed across 3 AZs delivers up to 36% higher throughput than HA HopsFS. Also, we showed that HopsFS-CL delivers up to $2.14X$ times the throughput of a default CephFS setup. In future work, we will integrate HopFS-CL with native cloud storage as a block layer to make storage and inter-AZ networking costs competitive with native cloud object stores.

Acknowledgment

This work is supported by the ExtremeEarth project funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[2] "Cockroachdb," https://www.cockroachlabs.com/, [Online; accessed 10-Jun-2019].

[3] R. Urata, H. Liu, X. Zhou, and A. Vahdat, "Datacenter interconnect and networking: From evolution to holistic revolution," in *2017 Optical Fiber Communications Conference and Exhibition (OFC)*, March 2017, pp. 1–3.

[4] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *ACM SIGCOMM computer communication review*, vol. 45, no. 4, pp. 183–197, 2015.

[5] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 123–137.

[6] "Amazon ec2," https://aws.amazon.com/ec2/, [Online; accessed 5-Jan-2019].

[7] "Google compute engine," https://cloud.google.com/compute/, [Online; accessed 5-Jan-2019].

[8] "Microsoft azure," https://azure.microsoft.com, [Online; accessed 5-Jan-2019].

[9] "Aws inter-region latency," https://www.cloudping.co/, [Online; accessed 5-Jan-2019].

[10] "Google cloud platform: Geography and regions," https://cloud.google.com/docs/geography-and-regions, [Online; accessed 12-Sep-2019].

[11] "Amazon s3 consistency model," https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html, [Online; accessed 5-Jan-2019].

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[13] "Google cloud storage consistency," https://cloud.google.com/storage/docs/consistency, [Online; accessed 5-Jan-2019].

[14] "How google cloud storage offers strongly consistent object listing thanks to spanner," https://cloud.google.com/blog/products/gcp/how-google-cloud-storage-offers-strongly-consistent-object-listing-thanks-to-spanner, [Online; accessed 1-Jul-2019].

[15] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 143–157.

[16] "S3guard: Consistency and metadata caching for s3a," https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/s3guard.html, [Online; accessed 5-Jan-2019].

[17] "Hadoop azure support: Azure blob storage," https://hadoop.apache.org/docs/r3.1.2/hadoop-azure/index.html#Atomic_Folder_Rename, [Online; accessed 12-Sep-2019].

[18] "Google cloud storage: mv - move/rename objects," https://cloud.google.com/storage/docs/gsutil/commands/mv, [Online; accessed 12-Sep-2019].

[19] "Delta lake: Reliable data lakes at scale," https://delta.io/, [Online; accessed 12-Sep-2019].

[20] "Apache iceberg: open table format for huge analytic datasets," https://iceberg.incubator.apache.org/, [Online; accessed 12-Sep-2019].

[21] "Apache hudi: Upserts and incremental processing on big data," http://hudi.apache.org/, [Online; accessed 12-Sep-2019].

[22] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1235–1246.

[23] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "Hopsfs: Scaling hierarchical file system metadata using newsql databases," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 2017, pp. 89–104.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 2010, pp. 1–10.

[25] "MySQL Cluster CGE," http://www.mysql.com/products/cluster/, [Online; accessed 5-Jan-2018].

[26] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[27] M. Ismail, S. Niazi, M. Ronström, S. Haridi, and J. Dowling, "Scaling hdfs to more than 1 million operations per second with hopsfs," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 683–688.

[28] S. Niazi, M. Ismail, G. Berthou, and J. Dowling, "Leader election using newsql database systems," in *Proceedings of the 15th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 9038*, 2015, pp. 158–172.

[29] S. Niazi, M. Ronström, S. Haridi, and J. Dowling, "Size matters: Improving the performance of small files in hadoop," in *Proceedings of the 19th International Middleware Conference*, ser. Middleware '18, 2018, pp. 26–39.

[30] M. Ronström, *MySQL Cluster 7.5 Inside and Out*. Books on Demand, 2018.

[31] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[32] J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978, pp. 393–481.

[33] "Google compute engine pricing," https://cloud.google.com/compute/pricing, [Online; accessed 5-Jan-2019].

[34] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ser. SC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–.

[35] "Cephs," https://www.slideshare.net/XiaoxiChen3/cephfs-jewel-mds-performance-benchmark, [Online; accessed 5-Jan-2019].

[36] M. Vrable, S. Savage, and G. M. Voelker, "Bluesky: A cloud-backed file system for the enterprise," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012, pp. 19–19.

[37] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Verissimo, "Scfs: A shared cloud-backed file system." in *USENIX Annual Technical Conference*, 2014, pp. 169–180.

[38] A. Thomson and D. J. Abadi, "Calvinfs: Consistent WAN replication and scalable metadata management for distributed file systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 1–14.