2019 IEEE International Congress on Big Data (BigData Congress)

Scalable Block Reporting for HopsFS

Mahmoud Ismail*, August Bonds*, Salman Niazi†, Seif Haridi*, Jim Dowling*†
*KTH - Royal Institute of Technology, † Logical Clocks AB
{maism, bonds, haridi, jdowling}@kth.se, {salman, jim}@logicalclocks.com

***Abstract*—Distributed hierarchical file systems typically decouple the storage of the file system's metadata from the data (file system blocks) to enable the scalability of the file system. This decoupling, however, requires the introduction of a periodic synchronization protocol to ensure the consistency of the file system's metadata and its blocks. Apache HDFS and HopsFS implement a protocol, called block reporting, where each data server periodically sends ground truth information about all its file system blocks to the metadata servers, allowing the metadata to be synchronized with the actual state of the data blocks in the file system. The network and processing overhead of the existing block reporting protocol, however, increases with cluster size, ultimately limiting cluster scalability. In this paper, we introduce a new block reporting protocol for HopsFS that reduces the protocol bandwidth and processing overhead by up to three orders of magnitude, compared to HDFS/HopsFS' existing protocol. Our new protocol removes a major bottleneck that prevented HopsFS clusters scaling to tens of thousands of servers.**

***Keywords*-Distributed hierarchical file systems; Distributed database; Block reporting.**

I. Introduction

Distributed hierarchical file systems typically decouple the storage of the metadata from the data to allow the file system to scale, which in turns enables higher performance for the file system [1]–[4]. However, this decoupling comes at a price - the metadata and data can become inconsistent due to failures (disk, network, host failures). The file system needs a periodic synchronization protocol to ensure consistency between metadata and data. For example, HDFS [2] stores its file system metadata in memory in a single server called the *namenode*. The metadata contains information about directories and files, and files' data which is represented as blocks that are replicated (three replicas by default) and stored on servers called *datanodes*. HDFS addresses the metadata inconsistency problem by having datanodes periodically, every 6 hours by default, send a *block report* containing information about all replicas stored locally to the namenode which in turns cross-checks this list with its local metadata to ensure the consistency of the metadata and the blocks' data. If inconsistencies arise, the namenode takes actions to fix those inconsistencies - for example, if a block replica on a datanode becomes corrupt, then, eventually, the namenode will create a new block replica by issuing a re-replication command to one of the datanodes with a non-corrupt replica.

HDFS suffers from scalability bottlenecks due to the single namenode architecture. HopsFS [1] was developed to overcome these scalability bottlenecks. HopsFS [1] is a next-generation distribution of HDFS [2], that adds horizontal scalability at the metadata layer. It achieves this by decoupling the metadata storage from the metadata serving. HopsFS stores the file system metadata normalized in a highly available, in-memory, distributed, relational database called Network Database (NDB), a NewSQL storage engine for MySQL Cluster [5], [6]. This allows HopsFS to support multiple stateless namenodes to manipulate the metadata stored in NDB, in parallel, through the use of transactions and locking primitives to ensure the consistency of the file system [1]. However, HopsFS has the same block reporting protocol as HDFS and even though HopsFS has multiple namenodes, block reports generate a large amount of traffic on HopsFS' backend database - in [1], it was shown that HDFS can process up to 60 block reports per second with 150 datanodes simultaneously containing $100K$ blocks, while HopsFS, with 30 namenodes, can only process 30 block reports per second due to the load on the database. Moreover, given the default 6 hours block reporting interval, the namenode(s) has to process 694 blocks per second for block reporting alone.

The block reporting protocol in HDFS and HopsFS is a scalability bottleneck, preventing clusters scaling to tens of thousands of servers in size. According to Shvachko, "The internal load for block reports and heartbeat processing on a 10,000-node HDFS cluster with a total storage capacity of 60 PB will consume 30% of the total name-node processing

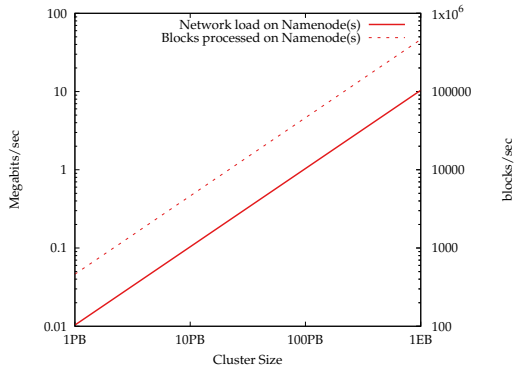

Figure 1: The block reporting processing load on the metadata server(s) grows with increasing cluster size, until it becomes the dominant workload in exabyte size clusters. The plot is in log-log scale with base 10.

978-1-7281-2772-9/19/$31.00 ©2019 IEEE
DOI 10.1109/BigDataCongress.2019.00035

capacity" [7]. Figure 1 shows the scalability problem in terms of the load on the namenode(s) attributable to the block-reporting protocol as a function of the cluster size.

In this paper, we introduce *hbr*, a scalable block reporting protocol, that both correctly synchronize the file system's metadata with the data, and reduces the network overhead compared to HDFS/HopsFS' block reporting protocol. Our solution introduces *buckets*, a logical collection of replicas in the file system, and three *hbr* functions:

1) an *assignment function* that dynamically assigns each replica in the file system to a specific bucket,
2) a *hash function* that hashes the replica information to a fixed size hash,
3) and a *hash combiner function* that combines the hashes of all replicas inside a bucket.

In experiments based on a real-world Hadoop workload from Spotify, ***hbr*** provides up to three orders of magnitude lower block processing overhead and up to three orders of magnitude reduction in the block report size.

II. Background

HopsFS [1] is an open source next-generation distribution of HDFS that mitigates HDFS scalability bottlenecks by replacing the single metadata storage layer with a distributed metadata storage layer. A typical HopsFS cluster consists of three main components, the data storage servers (datanodes), the metadata storage (NDB), and the metadata servers (namenodes), see Figure 2. HopsFS namenodes are stateless and access the metadata stored in NDB through the use of transactions. For internal housekeeping of the file system, HopsFS elects one of the namenodes as a leader using a leader election protocol [8]. NDB is the default database in HopsFS. However, HopsFS provides a pluggable data access layer (DAL) that allows using any other distributed databases with support for transactions and row-level locking. Files that are less than a configurable size, 64 KB by default, are called small files and are stored with the metadata in NDB to improve their access performance [9]. On the other hand, files bigger than the small files threshold are split into blocks, 128 MB by default. The files' blocks are then replicated, three times by default, into different datanodes on the file system to ensure high availability of the files.

Internally, HopsFS stores the file system metadata as rows in tables in NDB. The main three tables are inodes, blocks, and replicas. The inodes table contains information about files/directories such as parent, name, permission, size, etc.. The blocks table contains the list of blocks for each file in the file system. The replicas table contains the locations for each block in the file system. The file system operations are implemented as transactions on NDB and are guarded through the use of locking primitives in HopsFS [1]. HopsFS can be accessed using both HopsFS and HDFS clients. However, HopsFS clients are preferable since they can load balance their requests among all namenodes in the cluster. To write

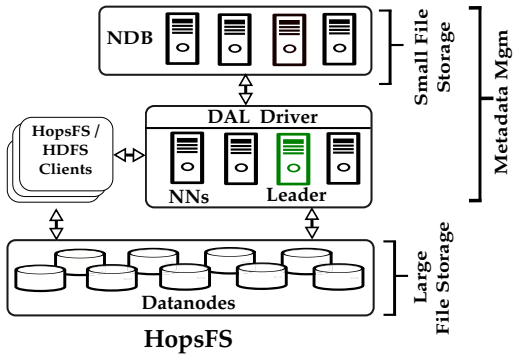

Figure 2: Architecture diagram of HopsFS. A typical cluster consists of a database cluster (NDB), a set of Namenodes (NNs), and a set of Datanodes (DNs). One Namenode is elected as leader for internal housekeeping of the file system. HopsFS and HDFS client can then be used to access the file system. HopsFS provides a data access layer (DAL) that allow using other distributed databases. Files are stored based on their size either in NDB if size is less than 64 KB, or in the datanodes if the size is larger than 64 KB.

a file to HopsFS, a client first sends a request to its selected namenode. The namenode then creates the file's metadata and stores it in NDB. Once acknowledged, the client requests a list of datanodes from the namenode to write the first block of the file. The list of datanodes is returned according to the file replication level, three by default. Then, the client setup a chain replication between the three selected datanodes, where it writes to the first datanode, and then the first datanode writes to the second and so on. While writing the datanodes continuously send both a status report and an incremental block report to the namenode(s) about the currently written, deleted, and completed block replicas. The client will repeat the same procedure for any additional blocks of the file. If a failure happens in one of the datanodes in the pipeline, then the client will request a new set of datanodes from the namenode to write the block.

A. Block and Replica States

Throughout the lifetime of a block, the block itself and its replicas go through a cycle of states. First, the block is *Under Construction* when first created. Then, the client sets up chain replication for the selected datanodes, and the block's replicas are now being written (*Replica Being Written*). Next, the replica is marked *Finalized* when all data bytes are received and written for that block. Once the client finishes writing to all replicas, the client closes the current block and asks for another block if needed to write more data. At that stage, the block state is *Committed*, and it will change to *Complete* only when the minimum replication level is reached. That is when the minimum number of datanodes have reported back *Finalized* replicas to the namenode(s).

B. Block Reporting Protocol

There are two types of block reports in HopsFS and HDFS:

1) *incremental block reports:* the datanode informs the namenode once a change happens in the state of a block replica. It is used by the namenode to inform the clients about the block state. For example, a block is created and safe to read when it is complete.
2) *full block report:* the datanode periodically sends a block report including information about all the blocks' replicas to the namenode. It is used to synchronize the replicas and blocks view between the namenode and the datanodes (where the datanodes are the source of truth).

HopsFS load balances block reporting across all namenodes in the cluster. Before a datanode sends a block report, it asks the leader namenode where to send the next block report. All active block report requests are stored in NDB. The leader namenode assigns the block reports in a round robin fashion to namenodes, taking into consideration the load on each namenode.

III. Problem Definition

We need to maintain a consistent and synchronized view of the data blocks in the file system between the metadata layer (namenode(s) and NDB) and the data storage layer (datanodes). The existing protocol for both HDFS and HopsFS involves the datanodes periodically sending a list of the blocks it has to the namenode(s) in the cluster. The list contains information about the blocks such as id, size, generation timestamp, and replica state. The generation timestamp is mainly used during recovery, where the block generation timestamp is increased after failures. The existing full block reporting protocol has the following shortcomings:

P1: *Increasing network bandwidth consumption*

The block report size is directly dependent on the number of blocks in the datanode. A single block requires ≈ 28 bytes, that is, a datanode with 1 million blocks will send a block report of size 28 Megabytes every 6 hours (default block reporting interval).

P2: *Overloading the metadata storage (NDB)*

For each block report, the namenode has to validate the reported information of the blocks with the current metadata for those blocks (stored in NDB). That is, assuming a block report with 1 million blocks, the namenode has to read 1 million rows from NDB with locks to ensure a consistent view. Moreover, the number of datanodes in the cluster can potentially grow to thousands of datanodes trying to report their blocks status to the namenodes. This would overload NDB, and negatively affect the performance of the whole file system.

IV. *hbr*

In this section, we describe *hbr*, our efficient and scalable block reporting protocol that overcome the two aforementioned problems (***P1***, ***P2***).

A. System Model

We define B as the set of all blocks in the file system. Similarly, we define R as the set of all replicas. A block $b_i \in B$ is replicated to n replicas where n is the file replication factor. That is, $r_i = \{r_{i,1}, r_{i,2}, .., r_{i,n}\}$ where $r_i \in R$ is the set of replicas of b_i, and the replication factor of b_i is $n = |r_i|$. We define a bucket k as a logical collection of blocks' replicas, where a replica r can be part of only one bucket k. We define K as the set of buckets configured in the file system. We define an assignment function f_{assign} to map each replica to a single bucket, see Definition 1. The assignment function has to satisfy Property 1. For each block, we hash the block information including block id, size, generation timestamp, and replica state using a hash function, f_{hash}, as defined in Definition 2. The hash function has to satisfy Property 2. Then, for each bucket, we combine all the hashes for the replicas in this bucket using a bucket hash combiner function, $f_{combine}$, as defined in Definition 3. The bucket hash combiner function has to satisfy Property 3.

***Definition** 1 (Bucket assignment function):* Given a replica $r_{i,j} \in R$ where $r_{i,j}$ is the replica j of the block b_i, and the number of configured buckets in the system $|K|$ then $k = f_{assign}(r_{i,j}, |K|)$ where $k \in K$ and $|k| = 1$.

Property 1: Given a block $b_i \in B$, $r_i \in R$ where r_i is the set of replicas for block b_i, then $\forall r_{i,x}, r_{i,y} \in r_i$ $f_{assign}(r_{i,x}, |K|) = f_{assign}(r_{i,y}, |K|)$. That is, all replicas of the same block logically maps to the same bucket - even though the replicas are stored on different datanodes. That means that updates to any of the replicas of a block will be local to the same bucket.

***Definition** 2 (Replica hash function):* Given a replica $r_{i,j} \in R$ where $r_{i,j}$ is the replica j of the block b_i, then $h = f_{hash}(r_{i,j})$

Property 2: Given two replicas $r_{i,x}, r_{j,y} \in R$ where $r_{i,x}$ is the replica x of the block b_i and similarly $r_{j,y}$ is the replica y of the block b_j, then $\forall r_{i,x}, r_{j,y} \in R, f_{hash}(r_{i,x}) \neq f_{hash}(r_{j,y})$

***Definition** 3 (Bucket hash combiner function):* Given a bucket $k \in K$, $k = \{r_1, ..., r_{|k|}\}$ where r_1 is the first mapped replica to the bucket and $|k|$ is the number of replicas in the bucket k, and $kh_0 = \phi$ where kh_i is the combined hash for i replicas in the bucket k, then $\forall i \in \{1, .., |k|\}$. $kh_i = f_{combine}(kh_{i-1}, f_{hash}(r_i))$

Property 3: Given a bucket $k \in K$, $kh_0 = \phi$, and $\forall i \in \{1, .., |k|\}$. $kh_i = f_{combine}(kh_{i-1}, f_{hash}(r_i))$, then $kh_{i-1} = f_{combine}(kh_i, f_{hash}(r_i))$. That is the $f_{combine}$ function is invertible.

Then, for each change to the state of the file blocks and replicas, see Section II-A, we recompute the hash and update the combined hash for the corresponding bucket. In practice, we track only the finalized replicas. When a datanode sends a block report, it sends its current combined hashes for its buckets to the assigned namenode. The namenode compares the received combined hashes with the stored bucket hashes.

If one or more of the bucket hashes does not match, the namenode asks the datanode to report back all the replicas within the mismatched buckets. For those buckets where the hashes match, the block report succeeds for the blocks in those buckets. For those blocks sent by datanodes for their mimatched buckets, we fall back to the original HDFS block reporting protocol - that is, the namenode validates all blocks received against those stored in NDB.

B. Implementation

We implemented the *hbr* block reporting protocol in HopsFS using the functions defined in Section IV-A. We devised an algorithm for *hbr* on the datanode and namenode, in Algorithm 1 and Algorithm 2, respectively. The datanode(s) and the namenode(s) communicate using an RPC model.

Algorithm 1 describes the *hbr* protocol from the datanode perspective. The datanode keeps a map of bucket hashes, hbr, line 1. We define the utility function UPDATEBUCKETDN that updates the bucket hash of any replica. It uses the assignment function, see Definition 1, to assign the replica to its corresponding bucket, line 18. If the map containing the bucket hashes, hbr, already contains a hash for that bucket, then, we use the hash combiner function, see Definition 3, to combine the hash of the new replica, line 22. Otherwise, if hbr contains no hash for bucket k, then we insert the hash of the replica as the combined hash, line 20. Whenever a replica is added to the datanode, we update the bucket where the replica logically resides, lines 2-4. Similarly, when a replica is deleted we use the same UPDATEBUCKETDN function to remove the replica from the current hash of the bucket, line 5-7. The UPDATEBUCKETDN works for addition and deletion since the hash combiner function $f_{combine}$ is invertible, see Property 3. The datanode will periodically send its buckets' hash map, hbr, to a selected namenode, lines 8-16. The datanode contacts the leader namenode to get the selected namenode for that block report. All active block reports on all namenodes are stored in the metadata storage (NDB). The leader namenode balances the block reporting handling across all namenodes. The datanode reconstructs the buckets' hash map if it was empty due to datanode restart before sending the block report, lines 9-13. Since collisions may arise due to the use of the *hash combiner function*, we perform a full block report once every configurable interval, every 24 hours by default, lines 25-28.

Algorithm 2 describes the *hbr* protocol from the namenode perspective. The buckets' hashes for all datanodes are stored in the metadata storage (NDB). Once the namenode receives a block report from a datanode, it iterates through the buckets and validates that their hashes match the stored hashes, line 1-8. If the reported hashes do not match, then, the namenode requests the datanode to resend the full block report for that specific bucket, line 5. The function SENDFULLREPORTFORBUCKET uses the vanilla block reporting protocol to report mismatched buckets and it is omitted for clarity. In order to keep the map of datanodes' bucket hashes up-to-date and to avoid unnecessary communication due to mismatch, we update the hashes for the buckets whenever there is an update to the replicas in the file system. We define the utility function UPDATEBUCKETNN that updates the bucket hash for a specific datanode dn with the hash of a finalized replica r, a replica where all its data has already been received, lines 30-43. The UPDATEBUCKETNN uses the same assignment function as used in the datanode to get the corresponding bucket for the supplied replica, line 32. Since the bucket hashes are stored in the database and are updated by all namenodes, we use the locking primitives as defined in HopsFS to ensure the serializability of the updates to the bucket hashes, lines 33-41.

Similar to the datanode side, we check if there is a stored hash for that datanode's bucket and update the bucket hash accordingly, lines 34-40. The datanodes inform the namenodes whenever there is an update in the state of a replica using the incremental block report. Therefore, we update the associated bucket hash for that replica, lines 9-11. Moreover, if a file got deleted, then all of its blocks will also be deleted, thus we update the hashes for the corresponding datanodes buckets, lines 12-16. So far, we have covered

Algorithm 1 Block reporting on datanode side

Require: $BRInterval$ ▷ Block report interval.
Require: $FBRInterval$ ▷ Full block report interval.
Require: LE ▷ leader namenode

1: $hbr \leftarrow \perp$ ▷ map of bucket hashes
2: **upon addition of a new replica** r
3: UPDATEBUCKETDN(r)
4: **end**
5: **upon deletion of a replica** r
6: UPDATEBUCKETDN(r)
7: **end**
8: **loop every** $BRInterval$
9: **if** $hbr = \perp$ **then**
10: **for** r **in** $replicas$ **do**
11: UPDATEBUCKEDN(r)
12: **end for**
13: **end if**
14: $nn \leftarrow LE.getNextNNToReportTo(thisDN)$
15: $nn.processReport(hbr)$
16: **end loop**
17: **function** UPDATEBUCKETDN(r)
18: $k \leftarrow f_{assign}(r)$
19: **if** $hbr[k] = \perp$ **then**
20: $hbr[k] \leftarrow f_{hash}(r)$
21: **else**
22: $hbr[k] \leftarrow f_{combine}(hbr[k], f_{hash}(r))$
23: **end if**
24: **end function**
25: **loop every** $FBRInterval$
26: $nn \leftarrow LE.getNextNNToReportTo(thisDN)$
27: $nn.processReport(replicas)$
28: **end loop**

most of the cases where a replica is updated. However, during failure, the client will request a newly updated writing pipeline from the namenode. In this case, the namenode needs to update the corresponding bucket hashes for the old datanodes that were used for writing before failure, line 17-21. Another operation that would require updates to the bucket hashes of the replicas is the append operation. If the last block size was less than the default block size of the file, then, upon append, we need to update the bucket hashes for the last block's replicas, lines 22-29. The function UPDATEBUCKETNN on the namenode side only updates the hashes for finalized replica in comparison to the function on the datanode side (UPDATEBUCKETDN) which considers all replicas when updating the hashes. The reason is that the datanode has the ground truth information about the replicas, therefore, it should always report the current state of the replicas, whatever that may be. Thus, during block reporting, if a file is being written with replicas that are not finalized yet, then, the buckets containing these replicas will be marked invalid and a full block report is required for those buckets.

Choosing the hbr functions: We chose the functions based on their definitions and their corresponding properties. First, the assignment function needs to have low computational complexity on both namenodes and datanodes, with only the knowledge of the block id and the number of buckets in the system. For that, the default assignment function is *modulus (%)*, where the bucket can be derived from the block id mod the number of buckets in the file system.

$$f_{assign}(r_{i,j}, |K|) = BlockId(r_{i,j}) \ \% \ |K| \tag{1}$$

The modulus function satisfies the Property 1, that is, the modulus will produce the same bucket id for all replicas of the same block. Since all replicas of the same block will have the same block id, then using the modulus of the block id will return the same bucket id for all replicas of the same block. Secondly, the hash function needs to produce a fixed size hash of the block information with low overhead and hard to compute collisions. The default hash function is *SHA1*, where we hash the replica information including block id, block size, generation stamp, and state.

$$\begin{aligned} f_{hash}(r_{i,j}) = SHA1(&BlockId(r_{i,j}),\\ &BlockSize(r_{i,j}), GenStamp(r_{i,j}), State(r_{i,j})) \end{aligned} \tag{2}$$

Thirdly, the hash combiner function needs to preserve the distribution of the input hashes and to be invertible. The invertibility is an important aspect since replica deletions require undoing the combined hash to remove that replica from the combined hash of the bucket. The default hash combiner function we use is *XOR* ($\oplus$).

$$kh_i = f_{combine}(kh_{i-1}, f_{hash}(r_i)) = kh_{i-1} \oplus f_{hash}(r_i) \tag{3}$$

The three functions can be overridden by any other functions as long as they satisfy the same properties as discussed for each function in Section IV-A.

Algorithm 2 Block reporting on namenode side

Require: MS ▷ Connection to the metadata storage.

1: **upon reception of block report** hbr **from** dn
2: **for** k, kh **in** hbr **do**
3: $storedkh \leftarrow MS.getBucketHash(dn, k)$
4: **if** $storedkh = \perp$ **or** $storedkh \neq kh$ **then**
5: $dn.sendFullReportForBucket(k)$
6: **end if**
7: **end for**
8: **end**

9: **upon reception of incr report for replica** r **from** dn
10: UPDATEBUCKETNN(dn, r)
11: **end**

12: **upon deletion of a block** b **with replicas** R
13: **for** dn, r **in** R **do**
14: UPDATEBUCKETNN(dn, r)
15: **end for**
16: **end**

17: **upon recovery of a block** b **with replicas** R
18: **for** dn, r **in** R **do**
19: UPDATEBUCKETNN(dn, r)
20: **end for**
21: **end**

22: **upon append of a file** f
23: $b \leftarrow f.getLastBlock()$
24: **if** $b.size \neq f.getDefaultBlockSize()$ **then**
25: **for** dn, r **in** $b.getReplicas()$ **do**
26: UPDATEBUCKETNN(dn, r)
27: **end for**
28: **end if**
29: **end**

30: **function** UPDATEBUCKETNN(dn, r)
31: **if** $r.state = FINALIZED$ **then**
32: $k \leftarrow f_{assign}(r)$
33: ***MS.lockBucket(k)***
34: $storedkh \leftarrow MS.getBucketHash(dn, k)$
35: **if** $storedkh = \perp$ **then**
36: $kh \leftarrow f_{hash}(r)$
37: **else**
38: $kh \leftarrow f_{combine}(storedkh, f_{hash}(r))$
39: **end if**
40: $MS.setBucketHash(dn, k, kh)$
41: ***MS.unlockBucket(k)***
42: **end if**
43: **end function**

V. EVALUATION

In this section, we present a comparative evaluation of the vanilla block reporting in HopsFS and *hbr*. (HDFS has the same block reporting protocol as HopsFS). All the experiments were run on PowerEdge R730xd servers(Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz, 256 GB RAM, 4 TB 7200 RPM HDDs) connected using a single 10 GbE network adapter. We deployed NDB, version 7.6.8, on 2 nodes and the data replication degree was set to default, 2. In all experiments, we used a fixed number of 100 datanodes. Also, we used a real-world hadoop workload from Spotify to

generate 1 million blocks on each datanode [1]. In production deployments, the datanodes send a block report every six hours by default, and the randomization in the protocol ensures that the datanodes do not send their block report to the namenodes at the same time. However, to determine the maximum throughput and capacity of *hbr*, we repeatedly send block reports to the HopsFS namenodes, that is, at any given time the namenodes will be processing 100 block reports from different datanodes.

A. Throughput of hbr

In this experiment, we benchmarked the throughput of the vanilla block reporting protocol and *hbr* with two different configurations of 1000 buckets and 2000 buckets, respectively, with no invalid buckets. Also, we varied the number of namenodes to show the scalability of both protocols. The vanilla block reporting can only process 0.25 block reports per second with one namenode, and the throughput increases to 0.75 block reports per second with three namenodes, see Figure 3. The performance of the vanilla block reporting suffers due to the increased load on the NDB to read the metadata required for processing the block reports. NDB is a real-time database which prioritizes short queries over large index scans returning many rows. More concretely, large queries that return a large number of rows (potentially millions of rows) can be frequently preempted by the NDB kernel to process other concurrent short queries.

Figure 3 shows that *hbr* delivers up to three orders of magnitude the throughput of the vanilla block reporting protocol. We performed two sets of experiments with different numbers of buckets per datanode. For 1000 buckets per datanode, *hbr* performs $\approx$ 1500 block reports per second using only one namenode, and the throughput increases up to $\approx$ 4600 reports per second using three namenodes. Similarly, for 2000 buckets per datanode, *hbr* performs around $\approx$ 800 block reports per namenode which linearly increases to $\approx$ 2300 block reports per second using three namenodes. The *hbr* performance directly depends on the number of buckets per datanode.

B. Latency of hbr

We compared the latency of both the vanilla block reporting, and *hbr* with two configurations 1000 buckets and 2000 buckets. The vanilla block reporting protocol takes on average $\approx$ 80 seconds to process a single block report containing 1 million blocks, see Figure 4. On the other hand, *hbr* takes on average $\approx$ 20 and $\approx$ 40 milliseconds to process a single block reports using 1000 and 2000 buckets per datanode respectively. Thus, increasing the number of buckets in the file system increases the latency and decreases the throughput. In this experiment, all the hashes on the datanode side and namenode side match. However, in real-world scenarios, depending on the workload some buckets

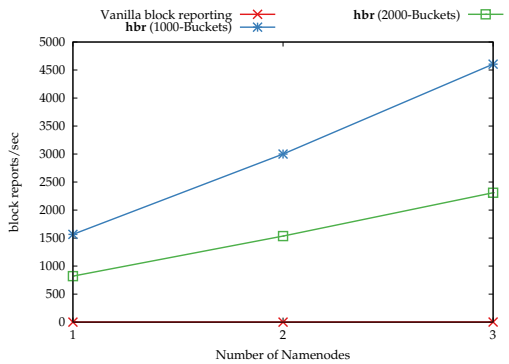

Figure 3: The throughput of the vanilla block reporting and *hbr* while varying the number of namenodes in the cluster. Each block report contains 1 millions blocks. For *hbr*, we use two configurations 1000 buckets and 2000 buckets.

on the datanodes and the namenodes may not match, see Section V-C.

C. Effect of invalid buckets

A mismatch between the bucket hashes on the namenode and datanode sides can happen due to different reasons such as failures of the datanode and file being written or deleted during the block report. Therefore, in this experiment, we investigated the performance of *hbr* in the presence of mismatching buckets. We did the experiment for two configurations of *hbr* with 1000 and 2000 buckets per datanode. Figure 5 shows the throughput of *hbr* protocol while varying the number of mismatching buckets per block report. Each block report contains 1 million blocks. That is 1000 blocks per bucket for the 1000 buckets setup, and 500 blocks per bucket for the 2000 buckets setup. The 2000 buckets setup delivers higher throughput in the presence of mismatching buckets since it has less number of blocks to process. A mismatched bucket requires a full block report for that bucket to be resent, thus dropping the throughput of the block reporting protocol. In real-world scenarios, the buckets mismatch will be minimal since industrial workloads are read

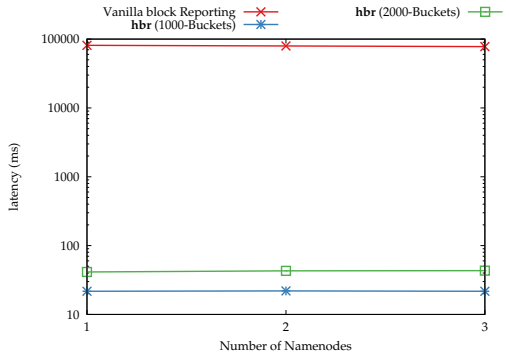

Figure 4: The average latency to process a block report with 1 million blocks of vanilla block reporting and *hbr*. For *hbr*, we use two configurations 1000 buckets and 2000 buckets. The Y-axis of the plot is in log scale with base 10.

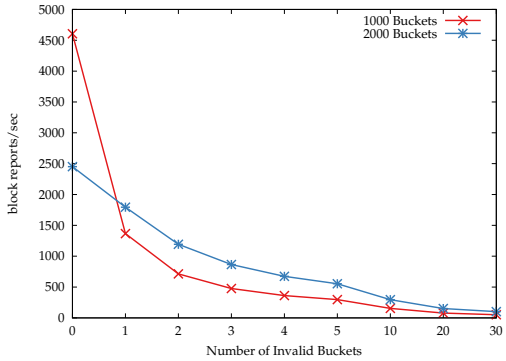

Figure 5: The throughput of *hbr* protocol while varying the number of mismatched buckets during the block report.

heavy where the percentage of create and delete operations is $\approx 3.45\%$ [1], and the *hbr* protocol ensures replicating the state of the blocks' replicas between the namenode(s) and the datanode(s).

D. Block Report Size

In this experiment, we compared the actual block report size for the vanilla block reporting protocol and *hbr* while varying the number of blocks in the block report. We used 1000 buckets for *hbr*. Also, we compared the size while changing the percentage of invalid buckets in *hbr*. Figure 6 shows that *hbr* has a constant block report size $\approx 20KB$ compared to the vanilla block reporting where the size goes up to $\approx 28MB$ for 1 million blocks. The *hbr* block report size depends only on the number of buckets configured in the file system and the size of the hash used in the f_{hash} function. The *hbr* block report size increase while increasing the percentage of the invalid bucket since we need to send all the information for the replicas in the invalid buckets as well.

E. Load on NDB and Namenodes

In this experiment, we show the effect of vanilla block reporting and *hbr* on the underlying metadata storage layer

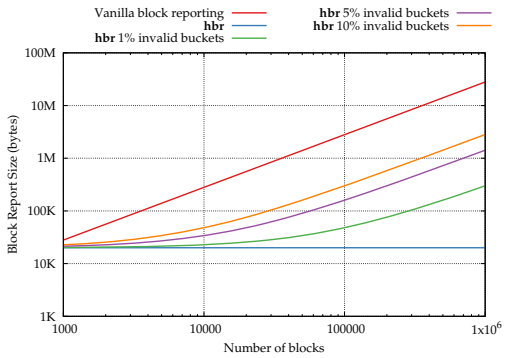

Figure 6: The block report size as a function of the number of blocks in the datanode. We compare *hbr* with 1000 buckets against the vanilla block reporting. Also, we show the *hbr* size in case of invalid buckets. The plot is in log-log scale with base 10.

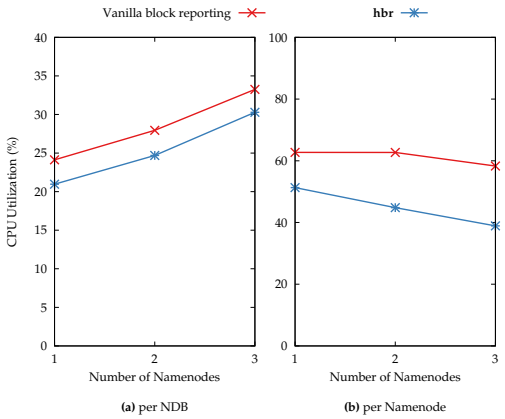

Figure 7: The average CPU utilization per NDB node and per Namenode that was recorded while running the vanilla block reporting and *hbr* with 1000 buckets.

(NDB) and the namenodes themselves. We collected the CPU and Network utilization while running a throughput experiment similar to the one introduced in Section V-A. Figure 7(a) shows the average CPU utilization per NDB node while varying the number of namenodes in the cluster. Similarly, Figure 7(b) shows the average CPU utilization per namenode while varying the number of namenodes in the cluster. The vanilla block reporting puts more load on NDB and the namenodes compared to the *hbr* protocol. This is due to the large block report size in the vanilla block reporting, that results in large index scan operations on NDB.

Figure 8(a) shows the average network read throughput per NDB node while varying the number of namenodes. The vanilla block reporting incurs higher network read per NDB since it reads all the replicas from the database to validate. Even with a higher throughput of *hbr*, the load is still lower than the vanilla block reporting since we skip reading all the replicas and instead we read only the buckets hashes. Figure 8(b) shows the average network read throughput per namenode. The *hbr* protocol incurs a higher load on the network due to the higher throughput of *hbr* in comparison to the vanilla block reporting, see Figure 3. Similarly, Figure 8(c) and Figure 8(d) shows the average network write throughput per namenode and NDB respectively. The network write throughput per namenode in Figure 8(c) corresponds with the read throughput per NDB node in Figure 8(a). Similarly, The network write throughput per NDB node in Figure 8(d) corresponds with the read throughput per namenode in Figure 8(b).

VI. Related Work

Distributed hierarchical file systems such as GFS [4], HDFS [2] and HopsFS [1] use a simple block reporting protocol where data storage servers (datanodes) exchange their state with the metadata servers (namenodes) to synchronize the file system view of the blocks. Such a protocol will fail to scale under load when the number of datanodes and

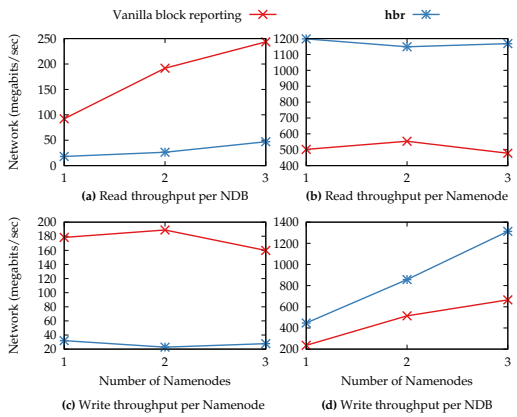

Figure 8: The average network read and write throughput per NDB node and per Namenode that was recorded while running the vanilla block reporting and *hbr* with 1000 buckets.

the number of blocks per datanode increase.

File systems such as SFS [10], BFS [11], and Tribler [12] uses merkle trees to reduce the amount of transferred data between servers. Key value stores such as Dynamo [13] also uses merkle trees to detect inconsistencies between replicas quickly and to reduce the amount of transferred data between servers. A merkle tree [14] is a balanced hash tree where the leaves are the hashes of data blocks (blocks in case of a file system). Every non-leaf node is the hash of its respective children's hashes. To check if a leaf node is consistent with the merkle tree, only the branch leading from this leaf node up to the root is required. However, if a change happens to any of the leaf nodes, we need to recompute the whole merkle tree. The design of *hbr* protocol is inspired by merkle trees. However, in *hbr* protocol we use only one level of buckets hashes, to avoid recalculation of the all buckets hashes when one replica changes its state.

VII. Conclusion

In this paper, we introduced *hbr*, an efficient and scalable block reporting protocol. We presented the concept of a bucket which is a logical collection of replicas in the file system. We defined three *hbr* functions to assign each replica to a bucket, to hash the replica information, and to combine all the replica hashes in the bucket. Also, we leveraged the incremental block reporting in HDFS/HopsFS to update the bucket hash whenever an update happens on the replica state. In experiments on HopsFS, we show that *hbr* scales up to three orders of magnitude better than the vanilla block reporting protocol. We also showed that *hbr* has up to three orders of magnitude lower block report size and latency than the vanilla block reporting protocol.

Acknowledgment

This work was funded by the Swedish Foundation for Strategic Research projects "Smart Intra-body network, RIT15-0119" and "Continuous Deep Analytics, BD15-0006", and by the EU Horizon 2020 projects "AEGIS, 732189" and "ExtremeEarth, 825258".

References

[1] 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)*. Santa Clara, CA: USENIX Association, 2017, pp. 89–104.

[2] 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.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43.

[5] M. Ronström, *MySQL Cluster 7.5 Inside and Out*, 2018.

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

[7] K. V. Shvachko, "Hdfs scalability: The limits to growth," *; login:: the magazine of USENIX & SAGE*, vol. 35, no. 2, pp. 6–16, 2010.

[8] S. Niazi, M. Ismail, G. Berthou, and J. Dowling, "Leader election using newsql database systems," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2015, pp. 158–172.

[9] 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*. ACM, 2018, pp. 26–39.

[10] K. Fu, M. F. Kaashoek, and D. Mazières, "Fast and secure distributed read-only file system," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, ser. OSDI'00. Berkeley, CA, USA: USENIX Association, 2000.

[11] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186.

[12] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. Epema, M. Reinders, M. R. Van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system," *Concurrency and computation: Practice and experience*, vol. 20, no. 2, pp. 127–138, 2008.

[13] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[14] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.