A General-Purpose Architecture for Replicated Metadata Services in Distributed File Systems

Dimokritos Stamatakis, Nikos Tsikoudis, Eirini Micheli, and Kostas Magoutis, *Member, IEEE*

Abstract—A large class of modern distributed file systems treat metadata services as an independent system component, separately from data servers. The availability of the metadata service is key to the availability of the overall system. Given the high rates of failures observed in large-scale data centers, distributed file systems usually incorporate high-availability (HA) features. A typical approach in the development of distributed file systems is to design and develop metadata services from the ground up, at significant cost in terms of complexity and time, often leading to functional shortcomings. Our motivation in this paper was to improve on this state of things by defining a general-purpose architecture for HA metadata services (which we call RMS) that can be easily incorporated and reused in new or existing file systems, reducing development time. Taking two prominent distributed file systems as case studies, PVFS and HDFS, we developed RMS variants that improve on functional shortcomings of the original HA solutions, while being easy to build and test. Our extensive evaluation of the RMS variant of HDFS shows that it does not incur an overall performance or availability penalty compared to the original implementation.

Index Terms—Distributed file systems, high availability, system recovery, metadata services

1 Introduction

Modern large-scale distributed and parallel file systems such as PVFS [2], HDFS [3], GoogleFS [4], pNFS [5], and Ceph [6] treat metadata services as an independent system component, separately from data servers (Fig. 1). Two reasons behind this separation are design simplicity and the ability to scale the two parts of the system independently. The availability of the metadata service is key to the availability of the overall system: If clients cannot contact the metadata service, they only have limited (e.g., over the duration of a lease) or no access to the entire data set. To ensure high availability of the metadata service, systems must be able to handle failure. In large-scale systems, failures tend to be the norm rather than the exception and can be caused by either planned or unplanned events including hardware failures, software bugs, reboots, software updates and maintenance [7], [8], [9], [10]. The high rate of failures in conjunction with the constant changes in modern data centers [11], [12] typically call for replication as a standard method to implement highly available metadata services.

Prominent distributed file systems, such as the Parallel Virtual File System[1] (PVFS) and the Hadoop File System (HDFS), already offer highly available (HA) metadata services. PVFS uses stateless replication with multiple metadata servers over a shared network-accessible storage service, such as NFS, for storing file system metadata. A drawback of such a solution is the single point of failure posed by the shared storage server. HDFS version 2.x avoids this problem by using stateful replication over quorum-based replicated storage. However, HDFS employs a checkpoint and roll-forward solution that limits the size of metadata state to fit within the main memory of metadata servers. Our initial motivation was to improve on the shortcomings of both systems while maintaining compatibility with existing applications, owing to large communities of users already using these systems. Our aim was to achieve that through a general-purpose metadata service, offering stateful metadata replication, designed to be easily retrofitted to both PVFS and HDFS.

The architecture (named RMS, for *replicated metadata services*) proposed in this paper offers a general-purpose solution that can be easily customized to the needs of a variety of distributed and parallel file systems. To achieve this, RMS separates the definition of file system metadata from the implementation of high availability for metadata via replication: Metadata is defined in terms of a set of relations that naturally map to a database API; high availability is achieved through standard data replication techniques at the database level. The RMS architecture avoids the centralized network file server single point of failure, as well as potential limits to overall metadata size.

Our implementation of the RMS architecture relies on three interoperating layered components: a highly-available networking layer, identical stateless replicas of the metadata server, and a replicated database. To make our approach practical, we base it on an existing replicated database exposing a key-value API, Oracle Berkeley DB (or BDB).

1. Recent releases of PVFS are also known as OrangeFS.

- *D. Stamatakis and N. Tsikoudis are with the Department of Computer Science, Brandeis University, Waltham, MA 02435. E-mail: {dimos, tsikudis}@brandeis.edu.*
- *E. Micheli and K. Magoutis are with the Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece. E-mail: {emicheli, magoutis}@cse.uoi.gr.*

Manuscript received 23 Sept. 2016; revised 16 Mar. 2017; accepted 20 Apr. 2017. Date of publication 2 May 2017; date of current version 13 Sept. 2017. Recommended for acceptance by M. Ripeanu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2700272

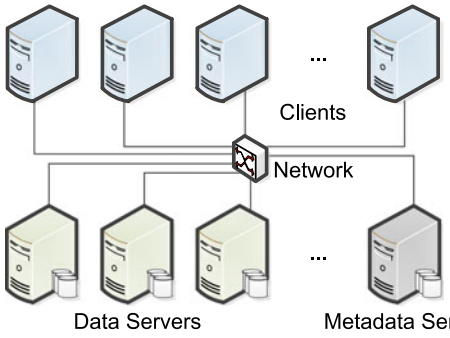

Fig. 1. Distributed file system architecture separating data from metadata servers.

For interoperability with that API, metadata servers should be designed with BDB as their underlying store or be retrofitted to it. We have experimented with both options: Our experience with PVFS, a system originally designed to use the BDB key-value API, shows that improving the availability of the metadata server through stateful replication can be straightforward in this case. Our experience with HDFS, a system that was not originally implemented over BDB, shows that it is possible to retrofit our solution into the metadata server with a reasonable level of complexity, achieving high availability as well as larger file system sizes than main-memory permits.

Our contributions in this paper are:

- A general-purpose architecture and methodology for building HA metadata services for distributed file systems.
- Design and implementation of two prototypes by extending the widely-used PVFS and HDFS systems.
- An extensive evaluation on microbenchmarks and application-level benchmarks in terms of performance and availability.

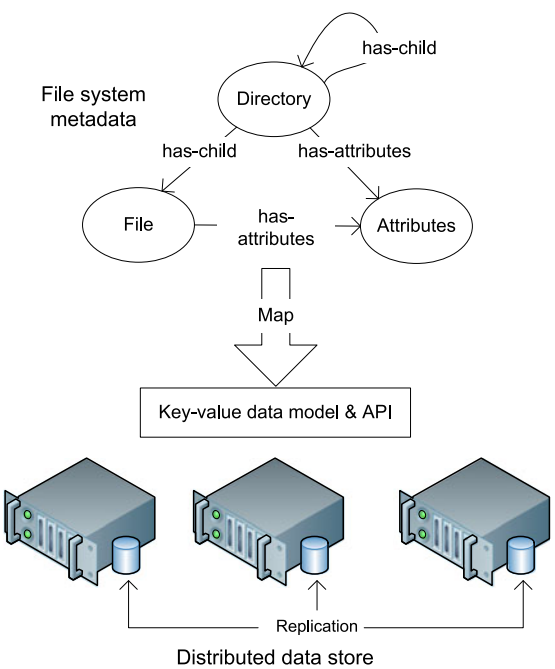

Fig. 2. Mapping metadata to an underlying data store.

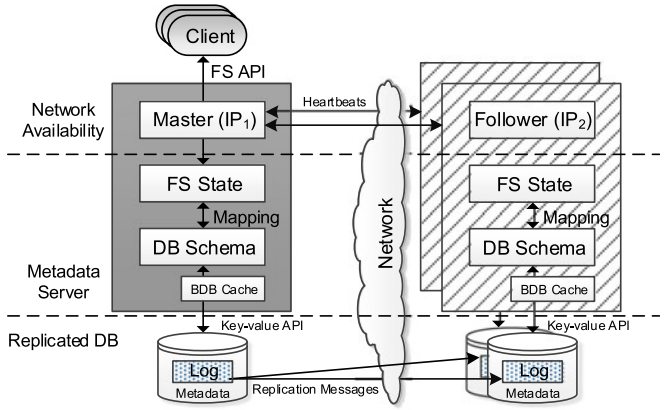

Fig. 3. Replicated metadata service (RMS).

The rest of the paper is organized as follows. In Section 2 we describe the overall design. In Section 3 we provide the details of our implementations and in Section 4 the evaluation of our systems. We describe related work in Section 5 and finally, in Section 6 we conclude.

2 Design

The goal of the general-purpose architecture described in this paper is to decouple the *definition* of distributed file system metadata from its *implementation*, including replication, in systems where metadata is implemented as a separate service. The key premise in the architecture is that file system metadata are naturally expressed as a set of entities and relationships and thus can be stored and managed by any type of database. While a standard relational database could be used, experience suggests that a lower-complexity solution, such as a transactional key-value data store, provides sufficient support for the type of operations accessing file system metadata, while it can exhibit better scalability in a distributed setting [13]. The upper part of Fig. 2 depicts a general data-model for file system metadata that can be customized to individual system needs as we demonstrate in this paper. The lower part of Fig. 2 depicts a data store exporting a key-value based data model and API, to which the file system metadata definitions map.

By decoupling the expression of metadata relationships from their implementation by the underlying store, we are able to use off-the-shelf replication-based solutions to provide highly available metadata services in existing or new distributed file systems. The key benefit of the architecture is reduction in the complexity of prototyping a full solution, compared to approaches implemented from the ground up.

The design principles depicted in Fig. 2 are fairly general and allow for a variety of design choices. Towards an implementation of this architecture we had to make concrete decisions that eventually led us to three stacked software layers shown in Fig. 3. The architecture can be mapped to a number of functionally-symmetric metadata-serving nodes.

The top layer is responsible for network availability of the overall metadata service, which is accessible to clients via a single IP address. This layer is typically implemented by dynamically mapping that client-visible IP address to the private IP address of one of the nodes that is designated master, while the rest are designated followers. While a master-follower node structure is not strictly a prerequisite

in our architecture, this choice reflects a requirement of the implementation of the replicated database layer.

At the intermediate layer is the metadata service adapted to map file-system state to a database key-value schema. This is the layer where an implementor needs to intervene in order to adapt an existing distributed file system into using our approach. The metadata server updates its in-memory state (box labeled "FS state" in Fig. 3) using a write-through policy, wrapping all metadata modifications of a file-system operation into a database transaction to ensure atomicity and durability of metadata updates.

The bottom layer is a replicated database implementing consistent replication of tabular data exported via a generic key-value API. Since the three layers are thought of as independent but collaborating, coordination is required, e.g., between the network availability and database layers to ensure that upon master failure, only one layer holds elections and notifies the other of the outcome. Since the top two layers are stateless, they need not necessarily be instantiated (for resource efficiency) on all system replicas; in other words, the replicated database layer is sufficient to maintain the state required to operate the system. This choice however can introduce a performance impact during recovery, to fully bring a new master to the level of performance of the previous master prior to its failure, as shown by our evaluation (Section 4.4.3). Maintaining a number of *hot-spare* metadata servers implementing all three layers can mitigate this issue. Finally, transaction processing performed by the replicated database may involve a number of performance/durability tradeoffs that are further discussed in Section 3.1.

In this paper we focus on full replication of metadata across metadata servers. Our design can accommodate static partitioning of metadata across servers [14], through a different replica group for each partition, as a way to increase overall throughput. Dynamic partition of metadata across servers is another possibility that has been explored in past research [6], [15] but is outside the scope of this paper.

3 Implementation

In this section we first discuss implementation choices made at the replicated database layer. We then describe the implementation of our metadata architecture on HDFS, calling the resulting system HDFS-RMS. We also carried out an implementation on PVFS (PVFS-RMS), which was straightforward to complete since the PVFS metadata server was already designed to use the single-node version of BDB as its underlying store. The details of the PVFS implementation are in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2017.2700272. HDFS-RMS required more involved re-design but turned out reasonably straightforward to carry out as well.

3.1 Replicated Database Layer

In this section we focus on the replicated database layer, the stateful layer in our design where configuration and tuning possibilities may lead to various performance/durability tradeoffs. Existing database engines that offer a key-value interface with support for data replication, such as Oracle Berkeley DB (BDB) [16] and LevelDB [17], can be used to realize this layer. As metadata partitioning across servers is not one of our main objectives in this paper, we do not consider scalable key-value data stores as alternatives. For concreteness and without loss of generality, we focus our discussion on BDB used in our implementations.

A BDB replica group features a master node that receives all client transactions, each transaction grouping one or more get and put operations. The master propagates each transaction to all followers. Master and followers log the transaction (writing it to their log buffer first and then -optionally- synchronizing with the commit-log on disk) and return acknowledgments to the master. After a master or follower failure, BDB uses a Paxos-compliant algorithm to reconfigure the replica group.

The BDB master considers a transaction committed when it receives a configurable number of acknowledgments. Commonly used policies include: (a) receive acks from all replicas (ALL); or (b) receive acks from a weighted majority of electable peers (QUORUM). These levels express the desired durability (ability to survive the last committed put under one or more node failures, decreasing from (a) to (b)), availability (ability to complete an operation under one or more node failures, improving from (a) to (b)), and delay to commit an operation, decreasing from (a) to (b). By default, BDB offers strong consistency for reads by performing get operations from the master node. While this is the mode we use in this paper, BDB can optionally be configured to provide applications with relaxed consistency by allowing get operations to access follower replicas [16].

Writes to a replica's log buffer may be synchronous or asynchronous with the commit log on disk. Synchronous writes can be a performance bottleneck [18], and therefore non-synchronous log writes are sometimes used for performance (although a failure may result in the loss of recently committed transactions). In a replicated system, the existence of replicas in other nodes allows the recovery of committed data from surviving replicas rather than from the failed-node's local disk. Non-synchronous log writes can thus be used without significantly reducing durability. Thus, it is important to choose the right acknowledgement policy so that enough replicas are guaranteed to store data before considering a write complete.

3.2 HDFS-RMS

3.2.1 Background

HDFS follows a main-memory database [19] approach in its metadata server, which is also called a NameNode. It keeps the entire namespace in main memory, while for recovery purposes it logs each metadata mutation to a write-ahead log (WAL) and occasionally takes checkpoints. Information about the organization of application data as files and directories and the list of blocks belonging to each file comprise the namespace *image* kept in main memory. The persistent record of the image stored on disk is called a *checkpoint*. The locations of block replicas, which are stored in servers referred to as DataNodes, may change over time, and are not included in the persistent checkpoint; instead DataNodes report which replicas they store via periodic messages sent to the NameNode. The log maintained by the NameNode to store metadata mutations is referred to as the *journal*.

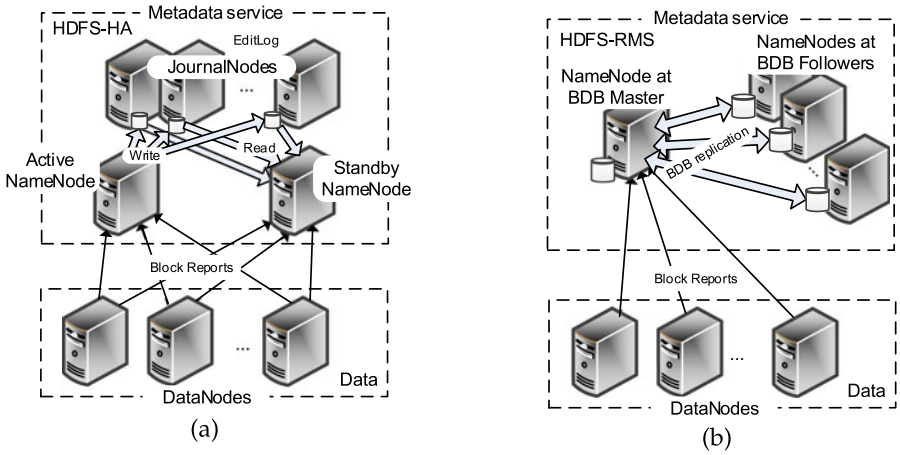

Fig. 4. HDFS high availability architecture: Original versus RMS.

The HDFS high availability architecture starting from version 2.0 and on, features two separate NameNodes in an active/standby configuration. The active NameNode is responsible for serving client load whereas the standby NameNode is a hot-spare maintaining metadata state and ready to take over in the event of a crash of the active. In one implementation of the HA architecture, both NameNodes have access to shared storage using NFS where the active NameNode stores updates to the log (or *edits*), such as namespace mutations. The standby NameNode checks for new edits and applies them to its own namespace.

A second implementation is based on a quorum-replicated store called the Quorum Journal Manager (QJM) [20]. It is the implementation used in this paper and shown in Fig. 4a. When a namespace modification is performed, the active NameNode logs the modification to a majority of replicas called JournalNodes. The standby NameNode watches the JournalNodes for new edits and applies them to its own namespace. DataNodes report block locations to both the active and standby NameNodes.

The HDFS-RMS architecture consists of one master and any number of follower NameNodes, each using replicated BDB as its underlying storage (Fig. 4b). The BDB master replicates the metadata, including block information, to the followers' local storage. The DataNodes report block locations only to the master NameNode. In terms of recovery after a NameNode crash, HDFS-RMS can resume service by failing over to a new NameNode which accesses state from BDB replicas.

3.2.2 Implementation Details

To efficiently implement the RMS architecture within HDFS, three key issues had to be resolved. First, a BDB schema whose maintenance and updating introduce low overhead. Second, reimplementation of the HDFS operations to update the disk representation of metadata stored in BDB tables. Lastly, we had to extend the HDFS NameNode to read from disk any missing metadata after a missed memory access. Using the RMS architecture, HDFS is no longer constrained by the size of NameNode main memory.

Traditional UNIX file systems use the *i*-node (short for "index-node") structure to store metadata information such as size, protection, timestamps, etc. and the location of a file's data blocks. They use directories to store file (or directory) names and their mappings to *i*-nodes [21]. HDFS does away with the indirection between a file name and its metadata via the *i*-node number and maps a file name directly to the file (or directory) metadata structures, which are the INodeFile and INodeDirectory classes respectively. Both classes inherit from an INode superclass that contains the name of the file or directory, a pointer to its parent, the last-modification and last-access times, and the full pathname. The INodeFile class extends INode to contain the list of file blocks. The INodeDirectory class extends INode to contain the list of children INodes. In the remainder of this section, we will use the term INode to denote either an INodeDirectory or an INodeFile depending on the context.

HDFS-RMS creates a disk-resident representation of HDFS metadata using the BDB schema of Fig. 5. A design objective was to reduce the number of BDB tables storing metadata, motivated by the high cost of table open/close operations in BDB, a problem we detected in an early version of HDFS-RMS using a different schema. To store all filesystem related metadata, HDFS-RMS uses three basic tables: T_DATANODES, T_BLOCKS, and T_NAMESPACE which are updated during file-system metadata updates. The T_DATANODES table is a disk representation of information about each DataNode. The T_BLOCKS table is a disk representation of metadata about all file blocks, such as generation timestamp [3], degree of replication, and which DataNodes hold replicas of a given block. T_NAMESPACE is the disk representation of the INodeFile and INodeDirectory classes. It contains one entry for each file and directory in the filesystem. It includes two types of records: a file BDB record that corresponds to a file, and a directory BDB record that corresponds to a directory. The key of each entry is the full path (a string) of the file or directory. The content of a file BDB record includes the type of record, number of blocks of the file, attributes of the INodeFile class, such as permissions and modification time, and file block IDs. A directory BDB record includes the number of children, attributes of the INodeDirectory class, and name of each child of the directory.

Storing the name of all directory children within a single BDB record may lead to overly large records. This may happen either when a directory has a large number of children, or when a directory has children with long names, or both. Large directory records can cause increasingly costly update

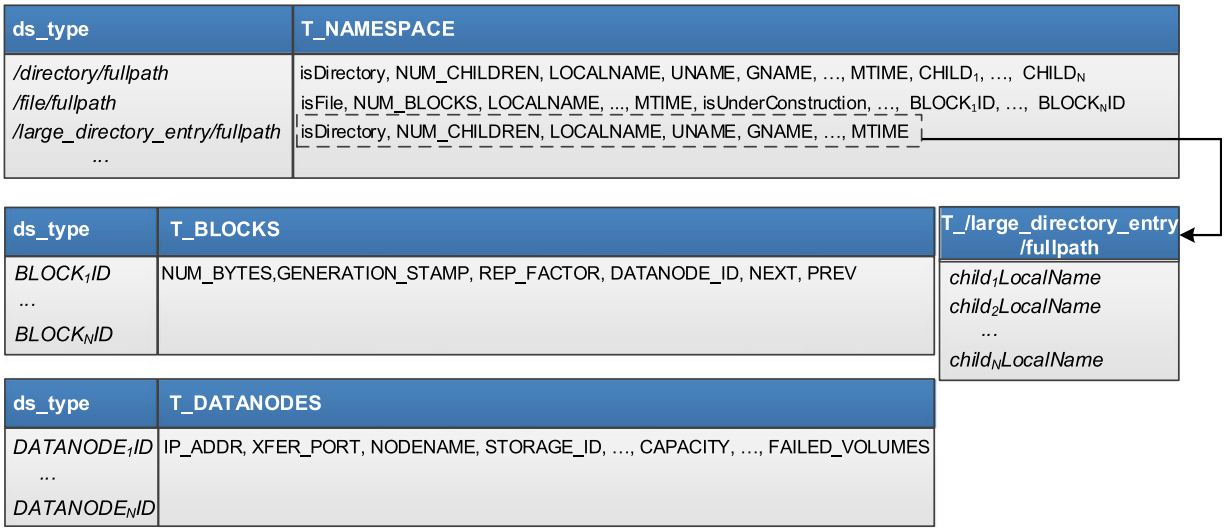

Fig. 5. The schema designed for HDFS.

operations as follows: Whenever a file or directory is added/renamed/deleted in the filesystem, there is a need to create/rename/delete its corresponding BDB record as well as to update the parent directory record. The update of a parent involves a Read-Modify-Write (RMW) operation to retrieve its value, modify it (e.g., replace child name for a rename operation), and then store it back to BDB. Our early experiments showed that operations that involve a RMW of parent directory records introduce a significant overhead when the size of those records grows uncontrollably. A similar situation occurs when appending blocks to a very large file.

To limit this overhead, we decided to treat large BDB records as follows: As soon as the size of a given BDB record exceeds a pre-defined threshold (at which point it is classified as large) a new BDB table is created. In the case of a large directory, all children names are moved to individual records of that table. The table is named after the full pathname of the large directory it corresponds to. The rest of the content of the BDB record remains stored at the T_NAMESPACE table. In such a case, to update the disk representation of a parent directory, we perform updates to more than one BDB records: the initial parent directory BDB record, plus one or more records at the new table. This is preferrable over time, compared to the cost of continuously updating a large (and growing) directory BDB record. We treat large file records similarly.

In HDFS-RMS we have reimplemented the HDFS operations that modify the filesystem metadata to update the disk representation stored in the BDB tables as well as operations that require fetching from BDB information not present in-memory. The former operations include file/directory creation, deletion and rename. The latter include file/directory open and list, when performing an `ls` command. More details can be found in the supplemental material, available online.

Standard HDFS assumes that all file/directory INodes are created in memory or instantiated in memory at system-boot time from the most recent checkpoint. In HDFS-RMS, metadata may be either in memory or on disk (BDB). HDFS-RMS thus extends the NameNode to perform two cache-management actions: During each metadata read, it looks up its memory-resident data structures and in the case of a hit it reads from them or otherwise fetches from BDB. For a metadata update, it reads from BDB whatever is currently missing from the cache and then modifies the BDB entries. The NameNode discovers out-of-core metadata during pathname resolution in the course of executing any file system operation.

In resolving a full pathname, standard HDFS retrieves all INodes on the file path by traversing its in-memory INode tree data structure. Any path components not found in that tree are explicitly added by creating new INodes for them and linking them to their parent. Standard HDFS performs these functions in an internal method called `mkdirs`.

In HDFS-RMS, INodes may be missing from memory and thus must be retrieved from the corresponding entries of T_* tables from disk. We have modified the internal HDFS `mkdirs` method to be able to detect when a file or directory is out of core and to retrieve it. When a directory brought in from BDB is the last component in the path name (e.g., when performing an `ls` on that directory), we bring in all its children INodes. This is necessary since we need the attributes of children INodes to fully carry out the operation. In our case the internal HDFS `mkdirs` method is invoked during each pathname resolution action as opposed to just during directory creation.

A remaining piece of NameNode metadata that we must be able to recover is write leases issued by the NameNode to ensure exclusive write access to files. This is necessary since HDFS does not support concurrent writers [3]. Instead of recovering lease state after a failure, we choose to let leases expire prior to resuming NameNode service. Clients that find that their leases have been revoked have to re-open files that were open for writing at the time of crash and write their contents again. Although this introduces a certain level of inconvenience for applications, it preserves lease consistency guarantees [22]. Fortunately, many important applications such as MapReduce and HBase are already designed to restart tasks whose file-writing activities have failed.

3.2.3 Discussion

An improvement of HDFS-RMS implementation over original HDFS is the maintenance of a persistent representation for all block metadata (table T_BLOCKS in Fig. 5) rather than treating metadata as soft state reported by each

DataNode to both (active and standby) NameNodes. While the scheme used by HDFS ensures that the standby NameNode has a memory copy of all block replica locations, it requires network communication between each DataNode and all NameNodes, and thus does not easily scale to a higher number of NameNodes. HDFS-RMS avoids this problem by requiring that DataNodes communicate with a single NameNode, which then replicates the metadata information to BDB replicas.

The space overhead of persisting HDFS block information in HDFS-RMS amounts to about 100 bytes (a BDB entry) per file block. For a 1 TB fileset, assuming on average half-full 64 MB blocks, this sums up to about 3 MB. We believe that this is a reasonable space overhead to pay for the ability to have block information readily available following a NameNode crash.

3.3 Network Availability

Our network availability layer (Section 2) requires a mechanism to be able to assign and relocate an IP address to the current master. For this purpose we use Pacemaker [23] which allows to manage the floating IP address as a cluster resource. We have disabled elections at the Pacemaker level to avoid any conflict with BDB's independent election process.

4 Evaluation

In this section we present our experimental results. Our goal is to evaluate our general-purpose architecture for replicated metadata services in comparison with original implementations. Since PVFS-RMS inherits specific limitations from base PVFS (a schema producing large numbers of small tables, limited concurrency, etc.), neither PVFS nor PVFS-RMS can be driven to high levels of load in our experiments. As HDFS results were more insightful than those of PVFS, we decided to focus on HDFS in this paper.

4.1 Experimental Setup

Our experimental testbed is a cluster of 9 servers, each equipped with a dual-core AMD Opteron275 processor clocked at 2.2 GHz with 12 GB of main memory. All servers run Ubuntu 14.04 64-bit with a 3.14.1 Linux kernel and are interconnected via a 1 Gb/s Ethernet switch. Servers used as HDFS data nodes have a base 72 GB 10,000 RPM SCSI drive with an additional 300 GB 15,500 RPM SAS drive dedicated to storing data. All hard drives are formatted with ext4.

4.2 Software Versions

Our HDFS-RMS implementation is based on HDFS version 2.0.5, the latest version at the time we initiated our evaluation. HDFS 2.x is being rapidly developed however, and the latest version by the time of writing is HDFS 2.6. Despite continuous development, there are no changes to the high-availability architecture and the metadata server between HDFS 2.0.5 and HDFS 2.6, as testified by the release notes between the two versions. To experimentally compare the performance of HDFS 2.0.5 to 2.6 in HA mode, we use a client that executes a tight loop of 10,000 `mkdir` operations. The HDFS setup consists of one DataNode, one active and one standby NameNode, located on 3 different servers. We are using 3 JournalNodes hosted on the same servers.

TABLE 1
Performance of HDFS 2.0.5 versus HDFS 2.6.0

	Execution time (sec)	CPU utilization (%)	I/O requests per second
HDFS 2.0.5	124.0 ± 0.70	28.0 ± 0.55	161.4 ± 1.14
HDFS 2.6.0	126.8 ± 0.70	28.7 ± 0.74	157.8 ± 1.34

Table 1 reports the average and standard deviation of the execution time, CPU utilization, and number of I/Os per second on the active NameNode server over five runs. Our results show that the two HDFS versions perform similarly. This evidence along with a careful examination of release notes, led us to conclude that HDFS-RMS based on HDFS v2.0.5 is a solid reference point to base our experiments on. HDFS-RMS uses Java SE Runtime Environment version 1.7.0.

Early tests with HDFS-RMS, our more performant prototype, using the same BDB C Edition exhibited concurrency limitations. We were able to address them by switching to BDB Java Edition v6.4.25, featuring support for highly concurrent applications [24].

4.3 Objectives

Our research questions in this experimental study are:

1) What impact does the RMS architecture have on microbenchmark and application performance versus the original implementation of HDFS?
2) What impact does the RMS architecture have on recovery time versus original implementations?

To answer the first question, we perform metadata-intensive microbenchmarks using the standard NNBench microbenchmark over HDFS. To measure the end-to-end impact of our architecture versus the original HDFS implementation we selected the Yahoo! Cloud Serving Benchmark (YCSB) [25] application-level benchmark over HBase over HDFS. We answer the second question through a series of experiments measuring recovery time and the performance impact of bringing metadata from disk after a metadata server crash.

4.4 Experimental Results

The following conventions are used throughout the text: The high-availability configurations of the original implementations of HDFS are denoted HDFS-HA-#*JournalNodes*. The original implementation in non-HA mode is denoted HDFS-LOCAL.

We denote our own implementation as HDFS-RMS-#*Replicas*. For instance, HDFS-RMS-3R corresponds to HDFS-RMS with three metadata server replicas. We stress here that HDFS-RMS-1R and HDFS-LOCAL are non-HA configurations. HDFS-RMS can be further configured to operate in SYNC or NOSYNC mode, depending on whether writes to disk are forced at each transaction commit or periodically when a log buffer fills up (Section 3.1).

Our aim in this evaluation is to ensure that systems being compared provide comparable semantics. On the issue of durability, we determined through careful code inspection that HDFS-HA performs synchronous batched commits, with synchronous disk writes through the FileChannel.

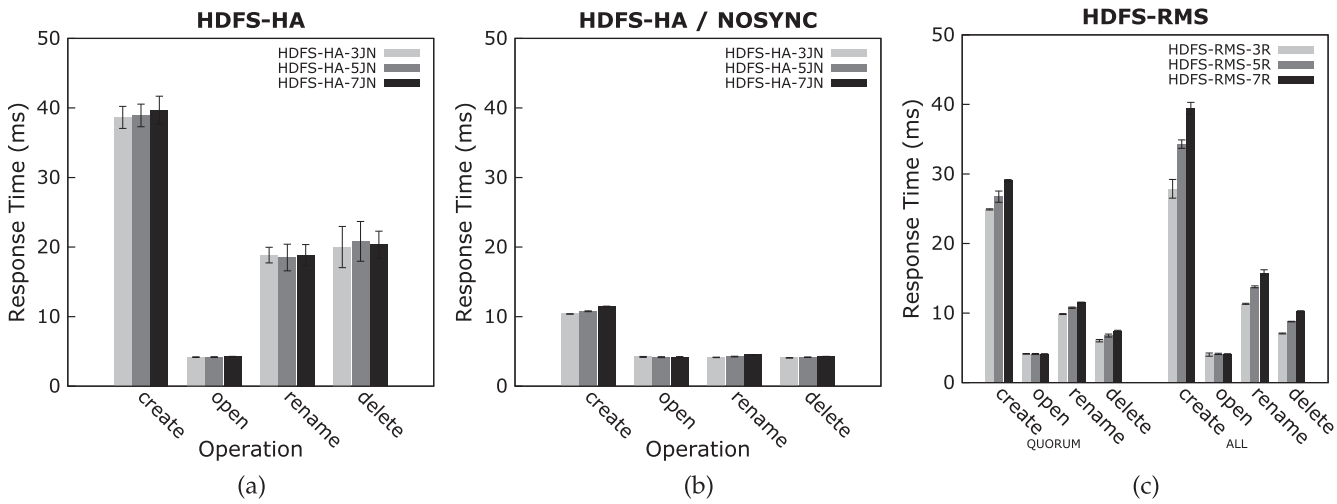

Fig. 6. Response time of NNBench operations for different HDFS and HDFS-RMS configurations.

force() Java API. Similar durability semantics can be achieved in BDB via use of the SYNC option (Section 3.1). However, early experiments showed that HDFS-RMS in SYNC mode incurs a significant performance penalty compared to HDFS-HA [1]. To detect the root cause of this, we isolated BDB transaction performance under a single replica repeatedly performing transactions encapsulating a single put operation, forcing writes to disk at commit time. The average commit time for such BDB transactions is 0.13 ms in NOSYNC[2] mode, growing to 17.41 ms under the SYNC mode. This result explains the increased response time of NNBench operations seen with HDFS-RMS in SYNC mode versus HDFS-HA. It also leads us to assume that the BDB implementation is more heavyweight compared to HDFS-HA, partly explained by the fact that BDB offers a general-purpose *fits-all* transactional API whereas HDFS-HA is a more lightweight system optimized for the filesystem metadata transactions used in HDFS. Based on these results we decided to take a pragmatic approach and configure HDFS-RMS in NOSYNC mode in our tests, offering slightly weaker durability semantics than HDFS-HA, however still sufficiently strong when used in conjunction with BDB's QUORUM and ALL replication policies (Section 3.1).

HDFS-RMS installations are configured to use a single data node, with all metadata server replicas running on separate machines. The original HDFS installation is configured with a single DataNode on a dedicated machine, one active and one standby NameNode, as well as with 3, 5, or 7 JournalNodes. One of the JournalNodes may be collocated with the active or standby NameNode. All results are averages of 10 runs, each run starting with a cold cache.

4.4.1 Microbenchmarks

We use NNBench, a standard MapReduce [26] benchmark in the Hadoop distribution that stresses the HDFS NameNode by performing sequences of create, open, rename, and finally, delete operations over a set of files. Each NNBench instance launches a single Map and Reduce task to perform the operations. We use 10,000 files per instance with a single HDFS replica per file. At file creation time we perform a zero-byte write to each file.

To sufficiently stress the NameNode, we experimentally determined that a load of 12 concurrent NNBench instances was enough to drive the least-performant of our systems (HDFS-HA) near saturation without exceed its capacity. For each file operation we report the average response time and the standard deviation (error bars).

In Fig. 6 we present performance results of three configurations: First, we consider HDFS-HA with increasing number of JournalNode replicas. Then, to isolate the impact of synchronous disk commits on HDFS-HA performance we consider a variant of HDFS-HA with asynchronous disk commits. Finally, we present performance of HDFS-RMS setups with a varying number of replicas and ack policies.

Fig. 6a depicts results for HDFS-HA. The HDFS-HA NameNode is I/O-bound in these experiments with moderate CPU utilization (≈ 10 percent in HDFS-HA-3JN). The key observations are that (1) the file open operation is significantly cheaper compared to file create, rename, and delete; (2) there is a noticeable variation in our measurements as evidenced by the errors bars in Fig. 6a; and (3) there is no observable performance impact when increasing the number of JournalNodes from 3 to 7. The first observation is explained by the fact that file open is a memory operation in HDFS-HA, whereas file create, rename, and delete operations involve quorum writes to the JournalNodes. The second and third obervations are not straightforward to explain at first sight. We hypothesized that the existence of synchronous group commits to disk were the primary cause of both.

To validate this hypothesis we modified the HDFS-HA source code by transforming synchronous group commits to asynchronous (removing the fileChannel.force() operation) and repeating the experiment. We call this modified configuration HDFS-HA NOSYNC. The results are shown in Fig. 6b, where we observe significantly better performance (a $3\times$ speedup), much lower error, and a measurable (though small) impact on performance from increasing the number of journal nodes.

Next we turn our focus to HDFS-RMS. Fig. 6c depicts performance with a different number of metadata server

2. Using the BDB write-nosync mode.

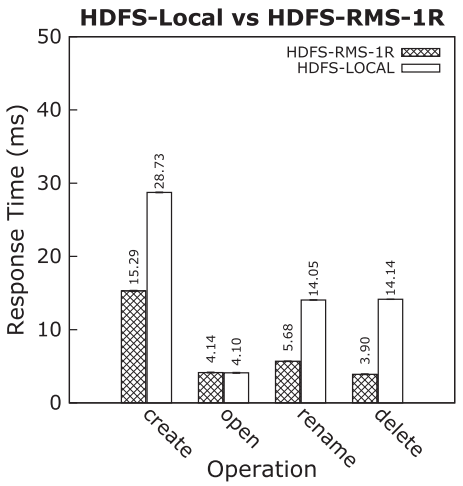

Fig. 7. Response time of NNBench operations for HDFS-LOCAL and HDFS-RMS-1R.

replicas and acknowledgment policies. We observe a performance hit when increasing the number of replicas, for both QUORUM and ALL ack policy, e.g., average response time for a create increases from 24.9 to 29.05 ms between 3 and 7 replicas for QUORUM policy. The performance hit is more pronounced under the ALL policy, where acks from all replicas are needed to complete an operation.

In comparison to HDFS-HA, HDFS-RMS performance on create, rename, and delete (operations involving the disk) is better by about 30 percent under QUORUM (or by about 10 percent under ALL) ack policy for 3 replicas. HDFS-RMS's performance advantage over HDFS-HA in create operation vanishes under the ALL policy and 7 replicas. HDFS-RMS exhibits about half (or worse) the performance of HDFS-HA NOSYNC.

We note that the impact of increasing replicas on performance is significantly more pronounced on HDFS-RMS than on HDFS-HA NOSYNC, pointing to HDFS-RMS's more heavyweight implementation. This impact is not measurable on HDFS-HA with synchronous group commits, masked by the high cost of disk writes in this case.

For open operations, performance is nearly the same across systems since all state is present in memory and no disk communication is required.

Fig. 7 compares HDFS-RMS in a non-HA configuration (HDFS-RMS-1R) to HDFS (HDFS-LOCAL). We observe that HDFS-RMS-1R outperforms HDFS-LOCAL on create, rename, and delete operations, while they perform similarly for open.

In summary, we find that HDFS-RMS performs better in file create, rename, and delete operations (albeit at the cost of slightly weaker durability semantics using the NOSYNC option versus HDFS-HA's synchronous batched disk writes), while they perform similarly for file open. Increasing the number of replicas introduces a small performance impact in HDFS-RMS (stronger under the ALL policy), none in HDFS-HA, which already pays a high cost on disk writes. Our results show that configurations such as HDFS-RMS-5R/7R QUORUM and HDFS-HA-7JN are comparable in terms of performance and offer sufficient durability, making them viable choices in performance-sensitive high availability environments.

4.4.2 Application-Level Benchmarks

To investigate end-to-end application performance we use the Yahoo! Cloud Servicing Benchmark [25] on top of the HBase [27] non-relational distributed datastore, an open source derivative of Google's BigTable [28]. HBase is widely used today for processing large amounts of data by companies such as Facebook [29], [30].

A typical HBase cluster consists of a master node that manages the cluster and several Region Servers that store user data. Data are logically organized into tables that are physically partitioned into regions stored by Region Servers. Each Region Server has a single Write-Ahead Log (HLog) and dedicates a memory buffer called Memstore for each region that it stores. Data updates are stored in the Memstore. When reaching a size threshold, they are flushed to a StoreFile. Both the HLog and StoreFiles are stored in HDFS. When the number of StoreFiles exceeds a threshold, HBase triggers a compaction to merge them into fewer, larger StoreFiles.

We configure YCSB to load 4 million records of 1 KB each into HBase using 16 threads. HBase operates with a single master and Region Server co-located with the HDFS Namenode and DataNode respectively. We dedicate 3 GB of memory to each of HDFS and HBase. We measure the average latency during the load phase of YCSB across different number of HDFS metadata server replicas/JournalNodes for HDFS-RMS and HDFS-HA, across different BDB ack policies for HDFS-RMS, and varying Memstore flush threshold for HBase and HDFS block size. Smaller Memstore sizes translate to more frequent flushes to StoreFiles in HDFS; smaller HDFS block size translates to more block allocations per file thus higher stress on the MDS.

Fig. 8a depicts average latency for HDFS-RMS-1R versus HDFS-LOCAL across different Memstore and HDFS block sizes. Performance of the two versions of HDFS is comparable for both 64 MB (default) and 8 MB Memstore/HDFS block size, observing higher latency as expected in the latter case.

We see similiar behavior in Fig. 8b exhibiting HDFS-RMS across different number of metadata server replicas and BDB ack policies. For Memstore/HDFS block size equal to 64 MB, HDFS-RMS and HDFS-HA have comparable latency across all configurations. The same trend holds for block size of 8 MB (Fig. 8c) at a higher latency. Latency is not impacted by the number of replicas in HDFS-RMS or HDFS-HA allowing for a higher degree of durability with no performance degradation.

In conclusion, HDFS-RMS performance is comparable to original HDFS when Memstore/HDFS block sizes are close to their default values; even when decreasing their sizes, leading to increased metadata stress, and using ALL ack policy the systems still perform comparably.

4.4.3 Failure Recovery

We evaluate recovery of HDFS-HA with 3 JournalNodes and HDFS-RMS with 3 NameNodes from a single replica crash. To enable automatic failover of the network path between clients and HDFS, HDFS-HA relies on ZooKeeper while HDFS-RMS uses Pacemaker to manage a virtual IP address through which the clients and DataNode access the master metadata server replica.

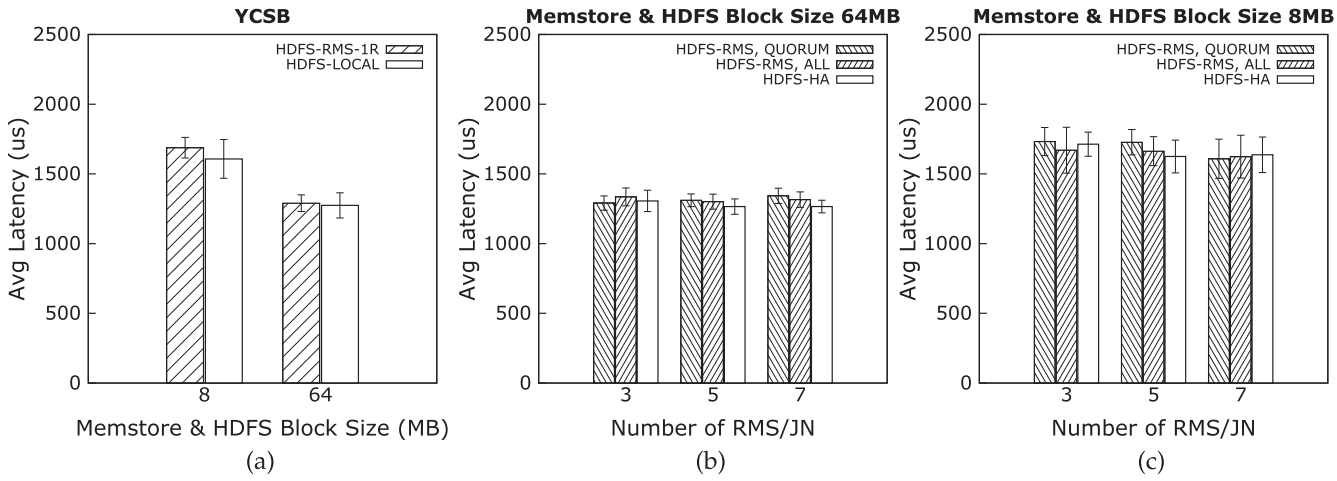

Fig. 8. YCSB latency HDFS-RMS versus HDFS-HA varying # of replicas, Memstore size, and HDFS block size.

HDFS-HA recovery proceeds as follows:

1) ZooKeeper detects crash of active NameNode
2) Election of standby as new active NameNode
3) Replay last journal edits made by old NameNode
4) Promote standby to active NameNode state

HDFS-RMS recovery consists of:

1) BDB detects crash of master replica, elects new master
2) Virtual IP re-assigned to new master by Pacemaker
3) Network path between DataNode/client and new master metadata server re-established

Recovery Time. We evaluate the recovery (outage) time as measured by a client or by a DataNode using NNBench. We set NNBench to create a predefined number of files and start file creation. At about 50 s into that phase we crash the master metadata server replica/active NameNode triggering the automatic failover mechanism. We then measure client and DataNode outage and report the average of 10 runs. We define DataNode outage as the time since the DataNode loses network connection with the master metadata server replica/active NameNode until it re-establishes connection with the new one. Client outage is defined as the time between issuing the last query before the crash, up to getting a response to that query.

Table 2 depicts our results. Focusing on DataNode outage first, we see that it is slightly higher in the case of HDFS-RMS due to the additional time needed to re-assign the virtual IP from the old to the new master metadata server and the time for the DataNode to re-establish network connection with the new NameNode. The DataNode in HDFS-HA is already connected with the standby NameNode, saving that step.

TABLE 2
Outage and Standard Deviation Measured at DataNode and Client

	Outage (s)	
	DataNode	Client
HDFS-HA	3.0 ± 0.00	6.7 ± 4.59
HDFS-RMS	3.6 ± 0.03	7.1 ± 0.60

Looking at client outage, we observe that HDFS-HA exhibits higher standard deviation than HDFS-RMS. A manifestation of this is occasional outliers in HDFS-HA client outage observations (we measured 14.3 and 16.4 s in our tests) substantially higher than the mean (6.7 s). We believe that this is due to two reasons: First, HDFS-RMS behaves differently than HDFS-HA in retrying a failed operation. With HDFS-RMS, NNBench repeatedly retries the operation upon receiving an exception, until it succeeds. HDFS-HA uses a backoff mechanism in trying to re-establish network connectivity with the active NameNode, which may at times lead to significantly higher client outage. A second factor is the fact that the HDFS-HA standby NameNode may in some cases not be fully up to date and thus need to apply the latest journal edits before promoting itself to active.

Post-Recovery Impact. As HDFS-RMS does not recover the full namespace information into memory, we expect to see a post-recovery performance hit for bringing metadata on demand from disk, in contrast to HDFS-HA, which maintains a hot-spare NameNode. The worst-case impact occurs when all filesystem state that the application needs to access after recovery must come from the disk. To quantify this impact, we perform the following two experiments:

In the first experiment we measure NNBench performance before and after a metadata server crash in HDFS-RMS-3R. NNBench is configured to create 50,000 files of one block each, then start performing open operations on the created files. After the system reaches steady state, we crash the master metadata server and allow the system to recover and resume execution. Fig. 9 depicts NNBench throughput of open operations sampled every 3 seconds. Before the master metadata server crashes, NNBench throughput for HDFS-RMS and HDFS-HA is comparable. NNBench stalls during recovery. After resuming execution, throughput with HDFS-RMS gradually increases, reaching 8 K ops per time interval. One might expect that the throughput would have ramped up to pre-failure levels after some time. However, this does not happen in this case because NNBench opens each file only once during the benchmark. Hence, all files that are read after crash have to be fetched from disk. In contrast, HDFS-HA serves read operations exclusively

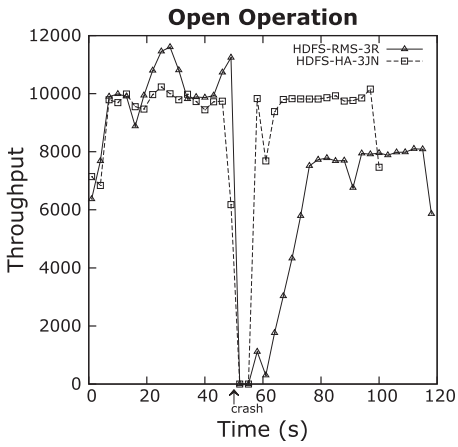

Fig. 9. Open operation before and after failure for HDFS-RMS-3R.

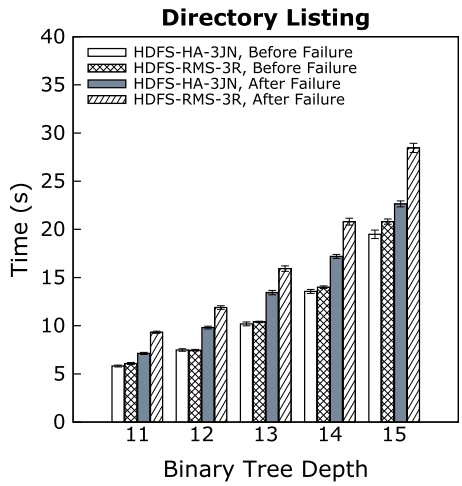

Fig. 10. Recursive directory listing for HDFS-RMS-3R and HDFS-HA-3JN before and after failure.

from memory before and after a potential crash performing better than HDFS-RMS. HDFS-RMS's advantage however is that it is not limited by memory on the size of metadata it can manage.

In the second experiment, an application initially creates a binary directory tree of variable length. Following a master metadata server crash and subsequent recovery with a new master, the application issues a recursive list-directory command (the equivalent of the UNIX ls -R command). We perform this experiment with different data set sizes, expressed in terms of directory tree depth, over HDFS-HA-3JN and HDFS-RMS-3R.

Fig. 10 depicts in the x-axis the depth N of the binary directory tree (the total number of directories is thus $2^N - 1$) and in the y-axis the total time taken by the recursive directory listing. We observe that the performance of HDFS-RMS-3R is within 7 percent of HDFS-HA-3JN when both serve out of memory. HDFS-HA-3JN is slightly slowed down after recovery. This difference in performance is due to the caching effect at master metadata server before failure. In contrast, HDFS-RMS-3R is slowed down by 37 percent compared to its memory-only performance because it brings all state from disk.

Our results show that there would be a performance benefit if HDFS-RMS was able to maintain hot-spare NameNodes at backup BDB replica nodes. To achieve this, it would have to capture updates communicated from master to backup BDB replicas (for example, by inspecting the BDB write-ahead log) and apply them to HDFS in-memory structures. When the master fails, recovery code at a backup would have to ensure that log updates are fully reflected at NameNode state prior to appointing it a master. Implementation of this functionality within HDFS is beyond the scope of this paper and subject of future work.

5 Related Work

General fault-tolerance methods typically rely on replication to protect against failures. The main replication mechanisms for high-availability in distributed systems include state machine [31], [32], [33], process-pairs [34], and quorum systems [35]. A class of distributed storage systems use the replicated state machine approach to replicate data, metadata, as well as for configuration management [36], [37], [38], [39].

Mapping metadata to a database layer has been explored in previous research on strengthening file system semantics [40] and on rich-metadata management systems [41]. Several modern distributed storage systems such as the Hadoop File System [3], Google File System (GFS) [4], Parallel Virtual File System [2], Parallel Network File System (pNFS) [5], and Ceph/RADOS [6], separate data and metadata paths avoiding the centalized metadata server bottleneck. Their popularity and impact motivates our focus on them in this paper.

Several recent research projects have aimed at improving the availability of the original HDFS NameNode design. ContextWeb [42] proposed a highly-available HDFS setup with two replicated NameNodes as a master-backup pair. HDFS namespace updates are written to a commit log on a shared volume (DRBD). By maintaining the checkpoint-log architecture, ContextWeb maintains the metadata size and recovery-time limitations. Wang et al. [43] introduced a hot-spare solution to HDFS where the primary NameNode continuously sends update-state messages containing namespace and lease information to the backup. This approach does not rely on a shared storage solution and maintains the memory-size limitation of the NameNode. Clement et al. [44] developed UpRight, a general-purpose state-machine replication approach that protects against Byzantine failures and demonstrated it for HDFS. UpRight replicates namespace, lease and block information on any number of NameNode replicas. This approach required significant modifications to HDFS codebase to remove non-determinisms and to replace the entire communication layer. Oriani and Garcia [45] describe Hot Standby Node, an approach that reuses HDFS's standard BackupNode (a helper process that regularly creates checkpoints for the NameNode) and turns it into a hot-standby by duplicating DataNode messages to replicate block information, as in AvatarNodes [46]. Hot Standy Node was implemented in 1,373 lines of code, not counting AvatarNodes code. Our approach is implemented in 1,228 lines of code in a newly added file and 527 lines of code to existing HDFS files.

From HDFS release 2.x onwards, several of previously mentioned improvements have been adopted into the

mainstream HDFS codebase, including the AvatarNodes functionality and a hot standby NameNode named Backup. In addition, the reliability of the EditLog has been improved via an implementation of quorum replication named the Quorum Journal Manager [20]. The drawbacks of this scheme compared to RMS is that HDFS 2.x QJM still supports two NameNodes, whereas RMS can support any number of NameNodes, and is still subject to the metadata-size limitation, whereas RMS is not.

The problems of extending the NameNode's metadata-storing capacity and increasing its availability are orthogonal and in principle open to different solutions. RMS has the benefit of a single solution to both problems. A current approach to extending the HDFS NameNode capacity (without simultaneously addressing high availability) is Federated HDFS [47], which allows partitioning of the namespace and assigning each partition to a different NameNode. In Federated HDFS the NameNodes are independent and do not require coordination with each other. The DataNodes are used as common storage for blocks by all the NameNodes. Each DataNode registers with all NameNodes, sends periodic heartbeats and block reports to them, and handles commands from any of them. This approach requires manual partitioning of the file-system namespace, a challenging task for most system administrators.

Several modern distributed file systems apply metadata replication for high availability and performance. Ceph [6] is an object-based distributed file system that distributes the management of the file system namespace to a cluster of metadata servers using dynamic subtree partitioning [48]. Other distributed file systems aiming for scalable metadata management are IndexFS [49] and Giraffa [50]. IndexFS uses LevelDB [17], an efficient key-value store, to store metadata and Giraffa uses HBase [27], a non-relational database, for the same purpose. XtreemFS [51] is another object-based distributed file system that implements a metadata service that replicates metadata at the operation level. SCFS [52] is a cloud-backed file system storing metadata on a replicated coordination service for consistency and fault-tolerance.

A number of georeplicated filesystems put special focus on metadata management. GlobalFS [53] distributes and replicates file system metadata across different regions of multiple datacenters using the URingPaxos protocol [54]. GeoFS [55], a wide-area distributed file system gives users the ability to adjust the consistency and replication level of the file system to their needs. CalvinFS [56] is a wide-area distributed file system whose metadata is partitioned and replicated in a distributed database, with metadata operations performed as distributed transactions. CalvinFS, IndexFS, and Giraffa store metadata of each file or directory in a single row of a table, similar to HDFS-RMS. Besides Oracle BDB used in our prototype, RMS implementations could be based on other key/value stores with similar properties, such as LevelDB. Although scalable metadata access is not one of the main objectives in the work described in this paper, the RMS architecture could conceptually support it through the use of a scalable replicated, transactional key-value store [57] at the database layer.

6 Conclusion

In this paper we proposed a general-purpose architecture of replicated metadata services in distributed file systems, named RMS. Our evaluation shows that an RMS variant of HDFS performs comparably to native implementations when contrasted in microbenchmarks. However, this has to come at the expense of durability: Write transactions with synchronous commits to disk are more expensive in HDFS-RMS (SYNC mode) versus HDFS-HA with similar behavior, pointing to the efficiencies possible in special-purpose software (HDFS-HA) compared to general-purpose designs (HDFS-RMS); one can improve HDFS-RMS performance to the levels of HDFS-HA by offering somewhat weaker durability semantics using the NOSYNC mode, which we consider as an acceptable tradeoff in replicated setups. Our experience with tuning our HDFS-RMS prototype implementation provides important lessons for RMS implementers: Avoid a schema that results into spreading of metadata onto a large number of small tables, and use underlying database implementations that allow for sufficient concurrency. In most cases, including application-level benchmarks, RMS variants do not incur an end-to-end performance penalty.

In terms of availability, the RMS variant of HDFS matches the recovery characteristics of HDFS-HA v2.x, a state of the art implementation. HDFS-RMS is resilient to the loss of any number of NameNodes (assuming sufficient number of replicas), whereas HDFS-HA allows only up to two NameNodes. Given that the performance of RMS variants is not significantly affected when the degree of replication increases from 3 to 7 using a QUORUM ack policy, leads us to suggest configurations with 5 or 7 replicas as viable alternatives for real-world installations. Overall we have shown that use of RMS is straightforward and yields robust, performant distributed file systems that are easy to build and reason about. Based on our results, we advocate that future distributed file systems should use RMS rather than build a new metadata service solution from the ground up.

Acknowledgments

An early version of this paper appeared at IFIP DAIS 2012 [1]. The authors would like to thank the anonymous reviewers for their valuable comments that contributed to improving the final version of the paper.

References

[1] D. Stamatakis, N. Tsikoudis, O. Smyrnaki, and K. Magoutis, "Scalability of replicated metadata services in distributed file systems," in *Proc. 12th IFIP Int. Conf. Distrib. Appl. Interoperable Syst.*, 2012, pp. 31–44.

[2] M. Ligon and R. Ross, "Overview of the parallel virtual file system," in *Proc. USENIX Extreme Linux Workshop*, 1999.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE Conf. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 29–43.

[5] S. Shepler, et al., "Parallel NFS, RFC 5661–5664." (2010). [Online]. Available: http://tools.ietf.org/html/rfc5661, IETF.

[6] S. Weil, S. Brandt, E. L. Miller, D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th USENIX Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.

[7] D. Ford, et al., "Availability in globally distributed storage systems," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 1–7.

[8] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun. 25–28, 2006, pp. 249–258.

[9] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst.*, 2009, pp. 193–204.

[10] H. S. Gunawi, et al., "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.

[11] M. Theimer, "Some lessons learned from operating Amazon's web services," *Keynote talk at 3rd ACM SIGOPS Int. Workshop Large Scale Distributed Syst. Middleware*, Oct. 2009.

[12] J. Dean, "Designs, lessons and advice from building large distributed systems," *Keynote talk at 3rd ACM SIGOPS Int. Workshop Large Scale Distributed Syst. Middleware*, Oct. 2009.

[13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed data structures for internet service construction," in *Proc. 4th Conf. Symp. Operating Syst. Des. Implementation*, 2000, Art. no. 22.

[14] A. K. Bhide, E. N. Elnozahy, and S. P. Morgan, "A highly available network file server," in *Proc. USENIX Winter Conf.*, Jan. 1991, pp. 199–205.

[15] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless network file systems," in *Proc. 15th Symp. Operating Syst. Principles*, 1996, pp. 109–126.

[16] Oracle White Paper, "Oracle Berkeley DB Java edition high availability," March 2010. [Online]. Available: http://www.oracle.com/technetwork/products/berkeleydb/berkeleydb-jeha-whitepaper-132079.pdf

[17] Google, "LevelDB," (2017). [Online]. Available: https://github.com/google/leveldb/

[18] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," in *Proc. 11th ACM Symp. Operating Syst. Principles*, Nov. 1987, pp. 155–162.

[19] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 6, pp. 509–516, Dec. 1992.

[20] T. Lipcon, "Quorum-based journaling in cdh4.1," 2012. [Online]. Available: http://blog.cloudera.com/blog/2012/10/quorum-based-journaling-in-cdh4–1/

[21] U. Vahalia, *Unix Internals: The New Frontiers*. Englewood Cliffs, NJ, USA: Prentice Hall, 2008.

[22] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proc. 12th ACM Symp. Operating Syst. Principles*, Dec. 1989, pp. 202–210.

[23] Pacemaker, "A scalable high-availability cluster resource manager," (2017). [Online]. Available: http://clusterlabs.org.

[24] Oracle White Paper, "Oracle Berkeley DB Java Edition Architecture," Sep. 2006. [Online]. Available: http://www.oracle.com/technetwork/products/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdf

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[26] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[27] (2017). [Online]. Available: http://hbase.apache.org/

[28] F. Chang, et al., "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[29] T. Harter, et al., "Analysis of HDFS under HBase: A Facebook messages case study," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 199–212.

[30] (2017). [Online]. Available: https://www.facebook.com/

[31] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[32] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[33] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly available distributed systems," in *Proc. 7th ACM Symp. Principles Distrib. Comput.*, 1988, pp. 8–17.

[34] J. Gray, "Why do computers stop and what can be done about it?" Tandem TR 85–7, Tech. Rep. 85.7, 1985.

[35] D. Gifford, "Weighted voting for replicated data," in *Proc. 7th ACM Symp. Operating Syst. Principles*, 1979, pp. 150–162.

[36] E. Lee and C. Thekkath, "Petal: Distributed virtual disks," in *Proc. 7th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 1996, pp. 84–92.

[37] J. MacCormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson, "Niobe: A practical replication protocol," *ACM Trans. Storage*, vol. 3, no. 4, pp. 1–43, 2008.

[38] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. 7th USENIX Sump. Operating Syst. Des. Implementation*, 2006, pp. 335–350.

[39] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 1999, pp. 43–43.

[40] M. Olson, "The design and implementation of the inversion file system," in *Proc. Winter USENIX Tech. Conf.*, 1993, pp. 205–218.

[41] S. Ames, M. Gokhale, and C. Maltzahn, "QMDS: A file system metadata management service supporting a graph data model-based query language," in *Proc. 6th IEEE Int. Conf. Netw. Archit. Storage*, 2011, pp. 268–277.

[42] C. Bisciglia, "Hadoop HA configuration at ContextWeb," (2009). [Online]. Available: http://www.cloudera.com/blog/2009/07/hadoop-ha-configuration/

[43] F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, "Hadoop high availability through metadata replication," in *Proc. 1st Int. Workshop Cloud Data Manage.*, Nov. 2009, pp. 37–44.

[44] A. Clement, et al., "Upright cluster services," in *Proc. 22st ACM Symp. Operating Syst. Principles*, Oct. 2009, pp. 277–290.

[45] A. Oriani and I. C. Garcia, "From backup to hot standby: High availability for HDFS," in *Proc. 31st IEEE Int. Symp. Reliable Distrib. Syst.*, Oct. 2012, pp. 131–140.

[46] D. Borthakur, et al., "Apache hadoop goes realtime at Facebook," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2011, pp. 1071–1080.

[47] S. Srinivas, "An introduction to HDFS federation," (2011). [Online]. Available: http://hortonworks.com/blog/an-introduction-to-hdfs-federation/

[48] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for Petabyte-Scale file systems," in *Proc. ACM/IEEE Conf. Supercomputing*, 2004, Art. no. 4.

[49] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 237–248.

[50] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson, "ShardFS versus IndexFS: Replication versus caching strategies for distributed metadata management in cloud storage systems," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 236–249.

[51] F. Hupfeld, et al., "The XtreemFS architecture-a case for object-based file systems in grids," *Concurrency Comput.: Practice Experience*, vol. 20, no. 17, pp. 2049–2060, 2008.

[52] A. Bessani, et al., "SCFS: A shared cloud-backed file system," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 169–180.

[53] L. Pacheco, R. Halalai, V. Schiavoni, F. Pedone, E. Riviere, and P. Felber, "GlobalFS: A strongly consistent multi-site file system," in *Proc. 35th IEEE Symp. Reliable Distrib. Syst.*, 2016, pp. 147–156.

[54] (2017). [Online]. Available: https://github.com/sambenz/URingPaxos

[55] G. Liu, L. Ma, P. Yan, S. Zhang, and L. Liu, "Design and implementation of GeoFS: A wide-area file system," in *Proc. 9th IEEE Int. Conf. Netw. Archit. Storage*, 2014, pp. 108–112.

[56] A. Thomson and D. J. Abadi, "CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 1–14.

[57] J. C. Corbett, et al., "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

Dimokritos Stamatakis received the BSc and MSc degrees from University of Crete, Greece, in 2011 and 2012, respectively, while working as a research associate at the Computer Architecture and VLSI (CARV) Laboratory at ICS-FORTH. He is working toward the PhD degree and as a research associate at the Computer Science department of Brandeis University. His research interests include distributed storage systems and transactional memory systems.

Nikos Tsikoudis received the BSc and MSc degrees in computer science from the University of Crete, Greece, in 2010 and 2013, respectively, while working as research assistant in the Distributed Computing Systems Lab at FORTH-ICS. He is working toward the PhD degree and as a research assistant with the Computer Science Department at Brandeis University. His main research interests are in the area of databases and distributed systems.

Eirini Micheli received the BSc degree from the Department of Computer Science, University of Ioannina, Greece in 2011, and the MSc degree from the Department of Computer Science while being a research and teaching assistant with the Systems Research Group, in 2013. In 2015, she joined the Distributed Systems Group in the same department. Her main research interests are in the area of distributed systems.

Kostas Magoutis received the BS degree from the Aristotle University of Thessaloniki, Greece, in 1993, the MA degree from Boston University, in 1996, and PhD degree in Computer Science from Harvard University, in 2003. He is currently an assistant professor in the Department of Computer Science and Engineering, University of Ioannina, Greece. His research has received best paper awards at Usenix ATC 2002 and USENIX BSDCon 2002, and a best student paper award at IEEE SRDS 2014. His research interests include scalable and highly-available distributed systems and data-intensive services. He is a member of the IEEE.