Scalability of Replicated Metadata Services in Distributed File Systems

Dimokritos Stamatakis, Nikos Tsikoudis, Ourania Smyrnaki, Kostas Magoutis

Scalability of Replicated Metadata Services in Distributed File Systems

Dimokritos Stamatakis, Nikos Tsikoudis, Ourania Smyrnaki, Kostas Magoutis

Institute of Computer Science (ICS)
Foundation for Research and Technology Hellas (FORTH)
Heraklion, GR-70013, Greece

Abstract. There has been considerable interest recently in the use of highly-available configuration management services based on the Paxos family of algorithms to address long-standing problems in the management of large-scale heterogeneous distributed systems. These problems include providing distributed locking services, determining group membership, electing a leader, managing configuration parameters, etc. While these services are finding their way into the management of distributed middleware systems and data centers in general, there are still areas of applicability that remain largely unexplored. One such area is the management of metadata in distributed file systems. In this paper we show that a Paxos-based approach to building metadata services in distributed file systems can achieve high availability without incurring a performance penalty. Moreover, we demonstrate that it is easy to retrofit such an approach to existing systems (such as PVFS and HDFS) that currently use different approaches to availability. Our overall approach is based on the use of a general-purpose Paxos-compatible component (the embedded Oracle Berkeley database) along with a methodology for making it interoperate with existing distributed file system metadata services.

1 Introduction

There has recently been a surge in research into the use of distributed consensus algorithms such as Paxos [11] and viewstamped replication [18] in building highly-available configuration management services. The usefulness of this approach for the construction of general-purpose highly-available systems has been highlighted in the past [12]. This line of research has culminated into real, practical services such as Chubby [3] and ZooKeeper [9], which expose file-system like APIs along with locking services. Other Paxos-compatible systems such as Oracle Berkeley DB [19, 21] expose a standard key-value API. While such systems have been used for configuration management of heterogeneous distributed middleware and data centers in general [4, 21, 22], their use in managing other types of metadata has not been sufficiently investigated. What is currently lacking is a study of Paxos-compatible systems under high-throughput scenarios typical of distributed file systems. The high rate of failures and changes in modern data centers call for a high degree of replication (5 or more replicas is not atypical today). In this paper we focus on the use of a Paxos-compatible replicated

key-value store as a metadata-server backend for two well-known and widely-deployed distributed file systems: The Parallel Virtual File System (PVFS) and the Hadoop File System (HDFS). We study the scalability of these two systems, which is defined as the ability to sustain performance as the number of replicas grows, under metadata-intensive workloads on Amazon's EC2 Cloud.

Consistent replication has been used in the domain of distributed file systems in the past. Petal [13] was an early storage system that used Paxos for replicating metadata. Although Petal featured an ambitious design where metadata were fully replicated across all system nodes (potentially tens of them), it was never evaluated for scalability under intense metadata updates at a large system size. Harp [15] used viewstamped replication for handling replica group reconfiguration under failures. Later Boxwood [16] proposed Paxos as a general-purpose primitive for maintaining global state as well as an underlying component of a lock service. Recently, Paxos has been used for configuration metadata management in systems such as Niobe [17]. Generally speaking, most distributed file systems that use Paxos today are reserving it for infrequently accessed/updated state (i.e., not file system metadata).

Modern distributed and parallel file systems such as pNFS [24], PVFS [14], HDFS [25], and GoogleFS [6] treat metadata services as an independent system component, separately from data servers. A reason behind this separation is to ensure that metadata access does not obstruct the data access path. Another reason is design simplicity and the ability to scale the two parts of the system independently. Given the overwhelming popularity of this paradigm we focus exclusively on it in this paper. Significant past research has improved the performance and reliability of data access in these systems, through data partitioning and replication, typically implemented in a primary-backup style [15, 17]. The metadata component however, has been traditionally dealt with separately via different techniques and often in an ad-hoc and non-scalable manner.

Existing approaches to high availability of metadata servers in PVFS and HDFS (as well as other similar file systems) are: (a) using a network-accessible disk device such as Amazon's Elastic Block Store (EBS) to store the underlying metadata, enabling an active-backup or active-active scenario (Figure 1-(a), in the case of PVFS); (b) using a checkpoint and roll-forward solution where the metadata server is periodically checkpointing its state and logging each mutation between checkpoints (Figure 1-(b), in the case of HDFS) to a shared store such as EBS or NFS. Drawbacks of option (a) are the need for special protocols and/or hardware for the shared storage-area network, as well as the bottleneck on the I/O path through a SAN server, limiting scalability. Drawbacks of option (b) are the limited amount of state the server can maintain (typically up to the size of main memory) and the need for a shared network file system, again limiting overall scalability.

Our proposed architecture (depicted in Figure 1-(c)) differs from (a) and (b) above by following a consistent-replication approach [11, 18] at the file system level, avoiding the SAN or centralized network file server bottleneck, requiring no special hardware, and avoiding limits to overall metadata size. Our approach is

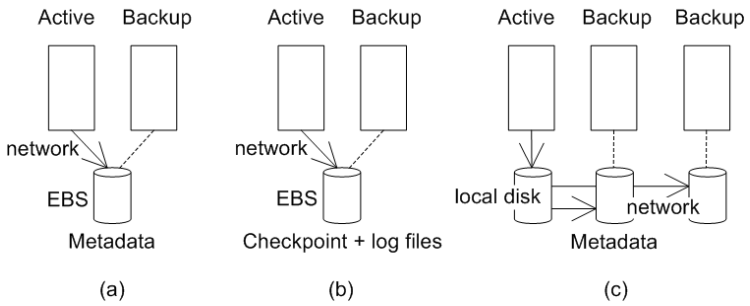


Fig. 1. Metadata server architecture options.

to extend a single-system metadata server into a replicated one via combination of three interoperating components: a highly-available (HA) 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 data store exposing a key-vale API (Oracle Berkeley DB or BDB). 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 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 without much complexity, achieving high availability as well as larger file system sizes than main-memory permits.

Our contributions in this paper are:

- A general methodology for building highly-available metadata services.
- Design and implementation of replicated metadata servers in the context of PVFS and HDFS.
- Evaluation of the above systems on Amazon Web Services' EC2 Cloud.

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

2 Design

Our design, depicted in Figure 2, consists of a number of metadata-serving nodes, one of which is designated as the master and the rest as followers. Clients are accessing the master node (accessible through a single client-visible network

address) for metadata reads and writes. Each node consists of three software layers: At the top is a network availability layer responsible for dynamically mapping a single client-visible IP address to the node that is currently elected master. At the intermediate layer is the metadata service adapted to map its file system state to a database key-value schema (examples of such a schema are shown in Figures 3 and 4). Finally, the bottom tier is a replicated database implementing consistent replication of tabular data exported via a generic key-value API. Coordination between the network availability layer and BDB is required upon master failure to ensure that only one layer holds elections and notifies the other of the outcome.

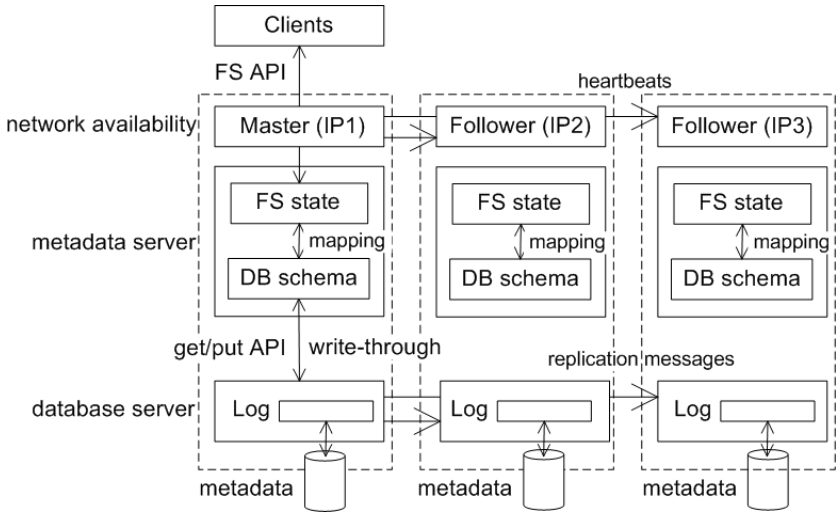


Fig. 2. Architecture of replicated metadata server

The metadata server cache (FS state in Figure 2) follows a write-through policy with clearly defined disk-synchronization points (typically at transaction boundaries) to ensure durability of metadata updates. In some cases however, commiting a transaction may be decoupled from synchronizing with the disk (e.g., when performing group commits), trading off durability with performance.

In the process of commiting a transaction, the master expects a configurable number of acknowledgements from followers. Typical choices for the *ack policy* are : (a) master must receive acks from all followers; (b) master must receive acks from a weighted majority of electable peers; (c) master commits after receiving a single ack; or (d) master commits immediately without waiting for any ack. Luckily, the interplay between ack policy and group commit can achieve better durability in a replicated system than is possible in a single-node system. For example, when using group commit, a node failure may require undoing some number of the most recently committed transactions on that node during

recovery. However, the existence of replicas in other nodes ensures that these transactions can be recovered from the surviving replicas. In this case it is important to choose the right acknowledgement policy and to ensure that replica nodes fail independently of each other. Finally, our design can support multiple masters and thus increase overall throughput by statically partitioning metadata across servers [2]. Dynamic partition of metadata across servers is another possibility that has been explored in past research [1,28] but is outside the scope of this paper.

3 Implementation

In this section we describe the implementation of our metadata architecture on PVFS and HDFS, two systems that follow the prevailing trend of separating data from metadata services. Our PVFS implementation was straightforward to complete since the PVFS metadata server was already designed to use single-node BDB as its underlying store. The HDFS implementation required more involved re-design but turned out reasonably straightforward to carry out as well.

3.1 PVFS

We first describe the PVFS metadata schema and provide examples of the metadata operations performed when executing client requests. We then describe our transformation of the PVFS metadata server to use the replicated rather than the single-node version of BDB. The PVFS metadata schema (which in not modified by our implementation) is depicted in Figure 3. PVFS uses four types of metadata objects: directories, directory data objects, metafiles, and datafiles, which when combined make up logical objects such as files and directories. These objects are depicted in the schema of Figure 3.

A metafile object (T_METAFILE) represents a logical file. It stores metadata such as owner, group, permissions and change/access/modify times about the file. It also stores the datafile distribution (*md*), controling how data is striped on data servers. A metafile stores an array of data-file handles (*dh*) and their size (datafile count). Datafile objects (T_DATAFILE) store information about the actual content of files (such as their size) in data servers.

A directory object (T_DIRECTORY) represents a logical directory. It stores metadata such as owner, group, permissions and change/access/modify times about the directory. It also stores hints such as distribution name (*dist_name*), parameters (*dist_params*), and datafile count (*num_files*). Distribution name and parameters control how data for a new file are striped over the data servers. The datafile count sets the number of datafile objects for a file. The directory object stores the handle to a directory data object. A directory data object (T_DIRDATA) describes the contents of each directory object as pairs of the form (*file_name, handle to directory* or *metafile object*).

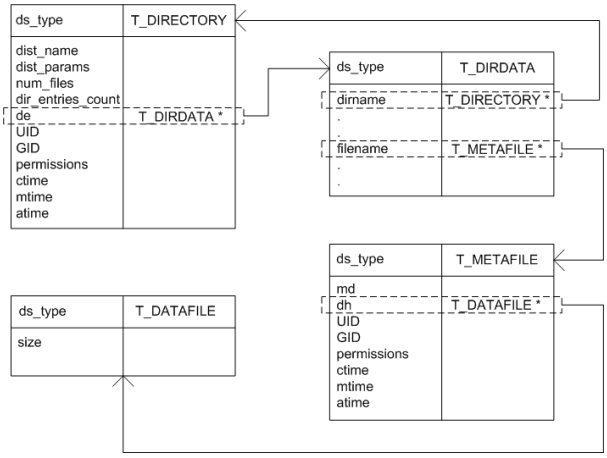


Fig. 3. The schema used in PVFS.

Next we give an example of the database operations (put/get) performed by the metadata server when executing an *mkdir* command. File/directory names and handle ids have been picked up randomly. The root directory ("/") of the filesystem maps to a directory object with handle 1 and a directory data object with handle 2. Note that the order of update operations is such as to ensure metadata consistency in the event of a metadata-server failure before the operation has been fully applied to stable storage. In case of failure, a consistency check (similar to the UNIX *fsck* [27]) is necessary to fully restore consistency.

mkdir /dir1/:

1. get *"de"* from handle 1 $\longrightarrow$ handle 2.
2. get *"dist_name"*, *"dist_params"*, *"num_dfiles"* from handle 1.
3. create new directory object with handle 3.
4. put T_DIRECTORY, id, gid, permissions, {a,c,m}time into *"ds_type"*, *"uid"*, *"gid"*, *"permissions"*, *"{a,c,m}time"* in handle 3.
5. create new directory data object with handle 4.
6. put T_DIRDATA into *"ds_type"* in handle 4
7. put handle 4 into *"de"* of handle 3
8. get *"de"* of handle 1 $\longrightarrow$ handle 2.
9. put handle 3 under key *"dir1"* in handle 2.
10. get *"directory_entries_count"* of handle 2 $\longrightarrow$ X.
11. put *"X+1"* into *directory_entries_count* of handle 2.

The original version of PVFS uses BDB without transactions or locking support. Our implementation modified the PVFS code to start BDB for replication, enabling the locking and transactional subsystems that are required under that mode. As part of the startup process we register the local and remote sites (the

replication group), set configuration parameters, and then call for a BDB election. After electing a master, BDB communicates its decision to PVFS via an asynchronous upcall. This upcall is also invoked any time the BDB node state (master or follower) changes. PVFS in turn instructs the network availability layer to switch the client-visible cluster IP address to the new master.

To operate under the transactional BDB mode while avoiding major changes to PVFS, we modified it so as to automatically wrap every database modification within a transaction. We did this by setting the DB_AUTO_COMMIT flag at server startup. While this method protects standard database accesses, it does not cover cursor operations. For this purpose we explicitly created transactions that protect cursor operations, ensuring that cursor handles are correctly deallocated to avoid running out of locks. Another important configuration setting was the use of master leases for consistent reads under network partitions.

3.2 HDFS

HDFS follows a main-memory database [5] approach in its metadata server (also called a NameNode), keeping the entire namespace in main memory while occasionally taking checkpoints and logging each metadata mutation to a log for recovery purposes. Information about the organization of application data as files and directories (their inodes [27]) and the list of blocks belonging to each file comprise the namespace *image* kept in main memory. The persistent record of the image typically stored on disk is called a *checkpoint*. The locations of block replicas may change over time and are not part of the persistent checkpoint. The NameNode stores metadata mutations in a write-ahead log called the *journal*.

Our implementation introduces BDB as a storage back-end of the NameNode creating a disk-resident representation of the image, using the schema depicted in Figure 4. HDFS now performs in-place updates to BDB during metadata updates. With this change, the memory-resident data structures of the NameNode can be thought of as a cache rather than a complete image of all metadata. In addition to providing the on-disk permanent locations of the image, BDB completely replaces the HDFS journal and checkpoint for recovery purposes.

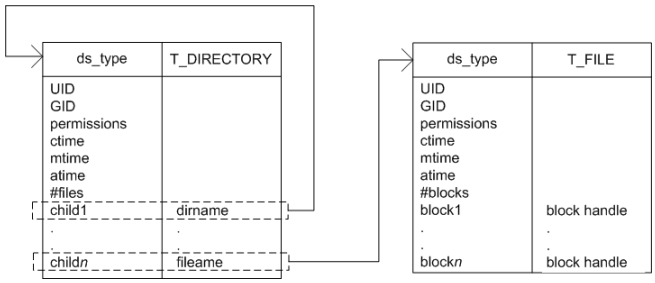


Fig. 4. The schema used in HDFS.

In the schema of Figure 4, each file and directory in the system maps to a table containing more specific file or directory attributes. In case of a file, a table of type T_FILE contains attributes of the file as well as the handles of all blocks in the file. For a directory, a table of type T_DIRECTORY contains attributes of the directory and the names of all files in the directory. Those names can be used as keys to further walk the filesystem namespace. The HDFS schema is simpler than that of PVFS but fully matches the semantics of HDFS.

The standard HDFS recovery methodology (checkpointing plus log replay) is replaced by a method in which failure of the master causes a surviving NameNode to be elected new master and to resume operation by accessing a fresh BDB replica. Logging is not completely gone from the system: it still takes place within BDB. Our design improves recovery time over original HDFS by starting from a fresh replica rather than reconstructing metadata from a checkpoint and log replay.

Another key improvement of our design is extending the capacity of a NameNode beyond main-memory limits. By using a write-through cache (just as in the case of PVFS) we eliminate cache consistency issues across NameNodes. Our policy is to have a NameNode erase its cache when demoted to a follower and thus a new master HDFS NameNode always starts with a cold cache. The HDFS NameNode should now be extended with the following cache-management actions: During each metadata read, the server looks up its memory-resident data structures and reads from them in the case of a hit or otherwise fetches from BDB. For a metadata update, the server reads from BDB whatever is currently missing from the cache and then performs in-place writes of the modified BDB entries. The NameNode can simplify the creation of memory-resident data structures from table entries read from BDB by invoking high-level operations (such as mknod, mkdir, etc) rather than low-level data structure manipulations.

Durability/performance tradeoffs in a Cloud setup. BDB offers two ways to achieve durability. One way is the use of synchronous writes to a disk log at commit time. Executing in a Cloud environment however means that a virtual disk may be ephemeral (e.g., an Amazon EC2 instance store). In addition, synchronous file system writes on the guest operating system may not result in synchronous writes to an underlying physical disk. Synchronous commits by BDB therefore do not necessarily translate into strong durability guarantees. Another way to achieve durability is the use of BDB's distributed replication protocol where the master collects acks from a number of followers to reach agreement before committing a transaction (Section 2). Combining replication with synchronous commits in a Cloud environment may hurt performance (as our evaluation shows) without in fact achieving stronger durability.

Network high availability. We require a mechanism to be able to assign and relocate an IP address to the current master. We experimented with two approaches: In platforms where we control address allocation and assignment (such as our in-house Eucalyptus Cloud) we used Pacemaker [20] to manage the

floating IP address as a cluster resource. We disabled elections at that level (via setting negative election probabilities for all nodes) to avoid conflicts with BDB's independent election process. In a platform such as the Amazon EC2 Cloud that provides custom control over address assignment, we use EC2 *elastic addresses* to allocate, assign, and relocate a public address. Elastic addresses are Internet routable IP addresses that map to VM-private addresses.

4 Evaluation

Our experimental setup consists of Amazon EC2 VMs running Debian 5 32-bit Linux. Each VM has one virtual core, 1.7GB of memory, and a 168GB local virtual disk (EC2 instance store) with an ext3 file system. Our baseline software is PVFS version 2.8.1, HDFS version 0.20.205.0, and Berkeley DB version 5.30 configured with a 512MB cache. All PVFS and HDFS installations were configured with a single data-node collocated in the same VM with clients. Key parameters in our tests are: number of replicas; ack policy (ONE, QUORUM, ALL); and synchronous vs. asynchronous commits (SYNC vs. NOSYNC). NOSYNC transactions are considered committed as soon as their commit record is stored in the log memory buffer. The BDB replication protocol is configured to perform bulk transfers of log updates over the network.

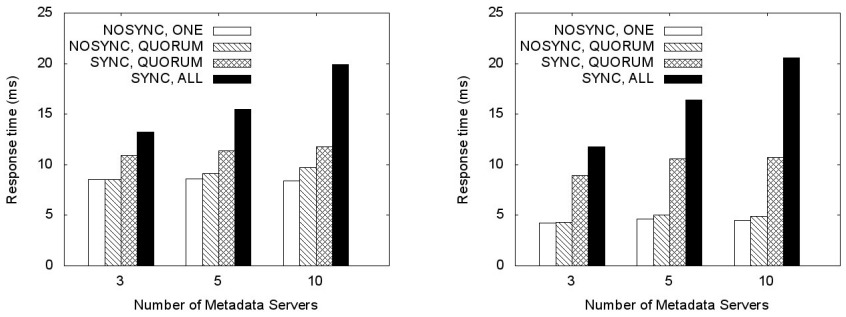
4.1 Microbenchmarks

We first evaluate performance of a single metadata server by measuring average response time of 2000 invocations of the mkdir command (a heavyweight metadata operation) using synchronous (SYNC) or asynchronous (NOSYNC) commits. Table 1 summarizes our results. Standard deviation was small (<3%) in all tests. Our implementation of PVFS is 15%-20% slower than the original PVFS due to the use of transactions and locking. PVFS generally has higher response time compared to HDFS due to its more complex schema and use of many put/get operations (Section 3.1). Our implementation of HDFS matches the performance of original HDFS when it uses a single file to back all database tables (which is also what PVFS does). We tested HDFS with multiple files (one file per BDB table) and found that table creation and open/close operations in the critical path slow down our version considerably (up to 3.3 times) compared to the original.

	PVFS		HDFS	
	SYNC	NOSYNC	SYNC	NOSYNC
Original	4.5	4.3	-	3.0
1-repl single file	5.1	5.1	3.7	3.0
1-repl one file per table	-	-	10.9	10.9

Table 1. Response time of mkdir on single metadata server (ms).

Next we evaluate performance of 3-, 5-, and 10-replicated metadata servers. Figure 5 depicts the average response time of 2000 mkdir operations under four setups: the strictest (SYNC, ALL), a Paxos-like configuration (SYNC, QUORUM), balancing performance with reliability (NOSYNC, QUORUM), and the most relaxed (NOSYNC, ONE). Focusing on HDFS results first, we observe a clear distinction between the NOSYNC setups on the one hand and the SYNC setups on the other. The former exhibit clearly scalable behavior; the case of (NOSYNC, QUORUM) is especially interesting since it provides sufficiently strong durability with little performance deterioration for up to 10 replicas. We attribute this to efficient network pipelining due to the use of bulk data transfers and the lack of synchronous commits. The PVFS results are similar except that (SYNC, QUORUM) results in only a 20-30% slowdown (rather than factor of two) compared to (NOSYNC, QUORUM).



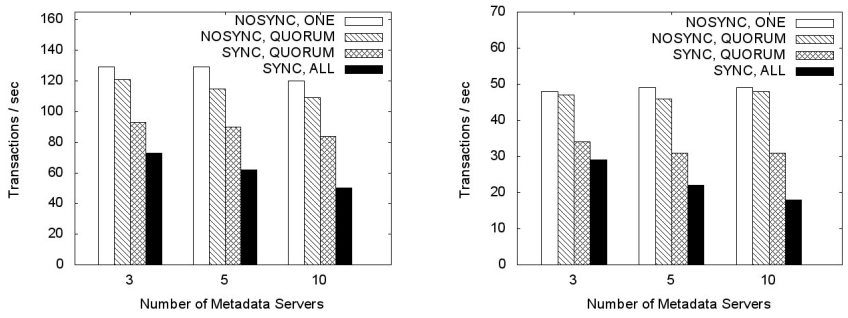
(a) PVFS (b) HDFS

Fig. 5. Response time of mkdir on 3, 5, 10 replicas (ms).

4.2 Postmark

To evaluate our systems under a more realistic workload we use Postmark [10], a synthetic benchmark aimed at measuring file system performance over a workload composed of many short-lived, relatively small files. Such a workload is typical of mail and netnews servers used by Internet Service Providers. Postmark workloads are characterized by a mix of metadata-intensive operations. The benchmark begins by creating a pool of 500 files with random sizes in the range 8-32KB. After creating the files, a sequence of 2000 transactions is performed. These transactions are chosen randomly from a file creation or deletion operation paired with a file read or write. A file creation operation creates and writes random text to a file in blocks of 4KB. File deletion removes a random file from the active set. File read reads a random file in its entirety and file write appends a random amount of data to a randomly chosen file. Each run involves two Postmark clients blasting the metadata server cluster.

Figure 6 depicts Postmark performance with 3, 5, and 10 metadata servers. Average transaction throughput with PVFS is higher than with HDFS due to differences in file I/O performance between the two systems (HDFS is not optimized for low-latency data access). HDFS results clearly highlight the gap between the NOSYNC and SYNC setups, similar to the gap observed in Figure 5-(b). The key observation is again that the (NOSYNC, QUORUM) setup provides strong durability guarantees with little performance degradation as the number of replicas grows. PVFS results indicate a somewhat higher drop of (NOSYNC, QUORUM) compared to (NOSYNC, ONE) but still within 5-10% of it.



(a) PVFS (b) HDFS

Fig. 6. Postmark throughput for different replication factors.

4.3 Recovery

In this section we compare the recovery of standard HDFS to our replicated version. The experimental setup for standard HDFS consists of two VMs, one hosting the NameNode and another hosting a helper process producing checkpoints of NameNode state, communicating with the NameNode via the HTTP protocol. We use an elastic block store (EBS) volume remotely-mount on the first VM to store the most recent checkpoint and operation log (Figure 1-(b)). Clients access the NameNode through an EC2 elastic address. Our experiments use a client that creates a number of files and then issues operations on them measuring response time (including retries). In case of failure of the NameNode VM, we restart the NameNode in the second VM, migrate the elastic address to it, and re-mount the elastic block store. Prior to resuming operations, the NameNode must recover its state by loading the last checkpoint and replaying the operation log. The recovery process for our replicated version of HDFS is described in Section 3.2.

Our experiments show that a large fraction of recovery time is spent in establishing the communication path between the client and the new NameNode

and (in the case of standard HDFS) remounting the EBS volume. Failover of the elastic IP takes on average 38 sec whereas remounting an EBS volume takes on average 35 sec (24 sec to unmount/detach and 11 sec to attach/mount again). Our experiments show that these steps cannot be parallelized. Table 2 presents our measurements with two standard HDFS setups parameterized by (checkpoint, log) sizes (MB), as well as a triply-replicated HDFS setup. In the former case, NameNode recovery is initiated manually by a script, right after a crash. In the replicated case, failure detection is performed by BDB. The replicated server does not have to wait for the failover of an EBS volume or for reconstruction of metadata state and thus provides significantly lower client outage.

	Network/disk failover	State reconstruction	Total client outage
HDFS (8, 1)	38+35	7.3	80.3
HDFS (0.8, 1)	38+35	1.9	74.9
HDFS 3-replicas	38	0	38

Table 2. Recovery time components and total outage (sec).

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 [23], process-pairs [8], and quorum systems [7]. A class of highly-available configuration management systems based on the replicated state machine approach using distributed asynchronous consensus algorithms such as Paxos at the core [11, 18] have recently emerged. Paxos-related approaches have also been used for maintaining configuration information within storage systems such as Petal [13] and Niobe [17]. They are also the underlying technology for systems such as Chubby [3] and Oracle Berkeley DB [19].

The issue of metadata placement and handling in distributed file systems has been a topic of intense research for a long time. Systems such as NFS opted for a centralized metadata server architecture, which allows for a simpler design at the expense of a scalability bottleneck. Other approaches that opted for distributing metadata symmetrically across the system nodes (such as Petal [13], Frangipani [26], and xFS [1]) resulted in complex designs that eventually restricted their applicability. Several modern distributed storage systems such as HDFS [25], PVFS [14], pNFS [24], and Ceph/RADOS [28], separate data and metadata paths avoiding the centalized metadata server bottleneck. Their popularity and impact motivates our focus on them in this paper.

6 Conclusions

We presented the adaptation of two modern, widely-deployed distributed file systems, PVFS and HDFS, to use a replicated Paxos-compatible key-value database (Oracle Berkeley DB) for achieving high availability of their file-related metadata. We found that high availability and data durability are achievable with minimal performance penalty under metadata-intensive workloads on the Amazon EC2 Cloud. Response time was found to be within 15-20% of the standard (non-replicated version) in the case of a single metadata server replica for both PVFS and HDFS. The replicated metadata server remains practically unaffected (0-5% drop) for higher number of replicas (3-10) when using asynchronous commits (a choice that does not sacrifice durability). Performance is noticeably impacted when turning to synchronous commits under the QUORUM or ALL acknowledgment policies. Overall our approach is straightforward, whether the distributed file system is designed for the target key-value API (as in the case of PVFS) or must be retrofitted to it (HDFS), and yields robust, performant distributed file systems that are easy to build and reason about.

7 Acknowledgments

We thank Yiannis Kesapidis, Konstantinos Chasapis, and Yiannis Linardakis for contributions to the PVFS implementation. We thankfully acknowledge the support of the European ICT-FP7 program through the CUMULONIMBO (STREP 257993) project, and of Amazon Web Services through an Education Grant.

References

1. Anderson, T., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., and Wang, R. Serverless Network File Systems. In *Proc. of 15th Symposium on Operating Systems Principles*, Copper Mountain, CO, 1996.
2. Bhide, A.K., Elnozahy, E.N., Morgan, S.P. A Highly Available Network File Server. In *Proc. of the USENIX Winter Conference*, Nashville, TE, January 1991.
3. Burrows, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of OSDI 2006*, Seattle, WA, 2006.
4. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C. and Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
5. Garcia-Molina, H., Salem, K. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.
6. Ghemawat, S., Gobioff, H., Leung, S.-T. The Google File System. In *Proc. of 19th ACM Symposium on Operating Systems Principles (SOSP-19)*, Bolton Landing, New York, 2003.
7. Gifford, D. Weighted Voting for Replicated Data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, 1979.
8. Gray, J. Why Do Computers Stop and What Can be Done About it? Technical report, Tandem TR 85-7, 1985.

9. Junqueira, F., Reed, B. C., Serafini, M. Zab: High-performance Broadcast for Primary-Backup Systems. In *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks*, Hong Kong, China, 2011.
10. Katcher, J. PostMark: A New File System Benchmark. Technical report, Network Appliance TR-3022, October 1997.
11. Lamport, L. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
12. Lampson, B. W. How to Build a Highly Available System using Consensus. In *10th International Workshop on Distributed Algorithms (WDAG 96), LNCS*, volume 1151, pages 1–17, Berlin, Germany.
13. Lee, E., Thekkath, C. Petal: Distributed Virtual Disks. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, 1996.
14. Ligon, M., Ross, R. Overview of the Parallel Virtual Fle System. In *Proceedings of USENIX Extreme Linux Workshop*, Monterey, CA, 1999.
15. Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L. Replication in the Harp File System. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1991.
16. MacCormick, J., Murphy, N., Najork, M., Thekkath, C.A., Zhou, L. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of 6th Symposium on Operating Systems Design & Implementation (OSDI'04)*, San Francisco, CA, 2004.
17. MacCormick, J., Thekkath, C. A., Jager, M., Roomp, K., Zhou, L., Peterson, R. Niobe: A Practical Replication Protocol. *ACM Trans. on Storage*, 3(4):1–43, 2008.
18. Oki, B. M., Liskov, B. H. Viewstamped Replication: A New Primary Copy Method to Support Highly Available Distributed Systems. In *Proc. of the 7th ACM Symposium on Principles of Distributed Computing (PODC'88)*, Toronto, Canada, 1988.
19. Olson, M. A., Bostic, K., Seltzer, M. I. Berkeley DB. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, Monterey, CA, 1999.
20. Pacemaker. A Scalable High-Availability Cluster Resource Manager. http://clusterlabs.org.
21. Perl, S. E., Seltzer, M. I. Data Management for Internet-Scale Single-Sign-On. In *Proc. of USENIX WORLDS 2006*, Seattle, WA, 2006.
22. Redstone, J., Chandra, T., Griesemer, R. Paxos Made Live: An Engineering Perspective. In *Proc. of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*, Portland, OR, 2007.
23. Schneider, F. B. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
24. Shepler, S. et al. Parallel NFS, RFC 5661-5664. http://tools.ietf.org/html/rfc5661.
25. Shvachko, K., Kuang, H., Radia, S., Chansler, R. The Hadoop Distributed File System. In *Proc. of IEEE Conference on Mass Storage Systems and Technologies (MSST)*, Lake Tahoe, NV, 2010.
26. Thekkath, C. and Mann, T., Lee, E. Frangipani: a Scalable Distributed File System. In *Proc. of the 16th Symp. on Operating Systems Principles*, S. Malo, France, 1997.
27. Vahalia, U. *Unix Internals: The New Frontiers.* Prentice Hall, 2008.
28. Weil, S., Brandt, S., Miller, E.L., Long, D., Maltzahn, C. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, 2006.