Replichard: Towards Tradeoff between Consistency and Performance for Metadata

Zhiying Li, Ruini Xue, Lixiang Ao

School of Computer Science and Engineering

University of Electronic Science and Technology of China, Chengdu, China

zhiyingli@grace@gmail.com, xueruini@uestc.edu.cn, aolixiang@gmail.com

ABSTRACT

Metadata scalability is critical for distributed systems as the storage scale is growing rapidly. Because of the strict consistency requirement of metadata, many existing metadata services utilize a fundamentally unscalable design for the sake of easy management, while others provide improved scalability but lead to unacceptable latency and management complexity. Without delivering scalable performance, metadata will be the bottleneck of the entire system. Based on the observation that real file dependencies are few, and there are usually more idempotent than non-idempotent operations, we propose a practical strategy, REPLICHARD, allowing a tradeoff between metadata consistency and scalable performance. REPLICHARD provides metadata services through a cluster of metadata servers, in which a flexible consistency scheme is adopted: strict consistency for non-idempotent operations with dynamic write-lock sharding, and relaxed consistency with accuracy estimations of return values where consistency for idempotent requests is relaxed to achieve high throughput. Write-locks are dynamically created at subtree-level and designated to independent metadata servers in an application-oriented manner. A subtree metadata update that occurs on a particular server is replicated to all metadata servers conforming to the application “start-end” semantics, resulting in an eventually consistent namespace. An asynchronous notification mechanism is also devised to enable users to deal with potential stale reads from operations of relaxed consistency. A prototype was implemented based on HDFS, and the experimental results show promising scalability and performance for both micro benchmarks and various real-world applications written in Pig, Hive and MapReduce.

CCS Concepts

•Networks → Cloud computing; •Social and professional topics → File systems management; •Software and its engineering → Consistency; •Information systems → Cloud based storage;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '16, June 01-03, 2016, Istanbul, Turkey

© 2016 ACM. ISBN 978-1-4503-4361-9/16/06...\$15.00

DOI: <http://dx.doi.org/10.1145/2925426.2926292>

Keywords

Distributed Storage; File System; Metadata; High Performance; Hadoop; HDFS

1. INTRODUCTION

Distributed and parallel applications are capable of processing increasingly vast amounts of data, which is usually managed by file systems. However, file systems are facing tremendous challenges as a result of the complex application behaviors and dramatic data growth nowadays. Because metadata operations make up as many as half of typical workloads [25] of a file system, metadata management is considered to be one of the most critical performance challenges in both the cloud computing (e.g., GFS [10], HDFS [27]) and high-performance computing domains [19]. Efficient metadata management is essential to a file system and overall system performance.

Although existing file systems store hundreds of petabytes (or more) of data across thousands of machines, they usually maintain file metadata only on a single server (e.g., GFS [10], HDFS [27], PVFS [5]), or a federation of metadata servers that statically partition the namespace (e.g., HDFS Federation [28]). As a matter of fact, storage capacity and aggregated I/O rate can scale almost arbitrarily with the addition of more storage devices, but metadata operations make scalable consistency and coherence management not easy for they involve a greater degree of interdependence. It is difficult to balance *consistency* and *performance* for metadata. Generally, strict consistency indicates worse scalability and worse performance, while relaxed consistency leads to better scalability, better performance as well as potential outdated data. Without proper tradeoffs, metadata service will become the performance bottleneck of the entire storage system.

On the one hand, significant strides for metadata management have been made in providing vastly scalable solutions so that high performance can be achieved. For example, hash-based approaches utilize hashing of a unique file identifier, such as inode number or path name, to distribute the metadata to multiple nodes and perform lookups (e.g., CalvinFS [29], Lustre [16], GIGA+ [22]). Subtree-based partitioning (e.g., Ceph [31], PanFS [32]) designates a subtree of a directory to the same node till its subtree is designated again. However, the hash-based method eliminates the hierarchy locality and results in high overheads when upper directories are renamed while subtree partitioning results in too coarse-grained partitioning. Both of them are unable to re-balance properly when the metadata servers are

reconfigured.

On the other hand, NoSQL systems, such as Cassandra [14] and Dynamo [8], have managed to achieve both scalability and performance through reduced replica consistency and eventual guarantees. Reads are not guaranteed to be up to date, but updates are guaranteed to be propagated to all nodes eventually, albeit nobody knows the accuracy of return values in any time.

Unfortunately, file system metadata servers have not achieved both scalability and high performance. While many metadata servers of distributed file systems have been developed to scale to more than one machine, these systems do not provide low latency in a manner that allows high performance in the event of highly concurrent metadata accesses due to the maintenance of the consistency between different metadata servers becoming more costly.

Besides, modern file systems, even those known for scalability, utilize a fundamentally unscalable design for metadata management in order to avoid high yet necessary synchronization costs to maintain traditional file system semantics for file and directory metadata, including hierarchical access control and serializable writes.

To address these issues, we describe in this paper the design of a distributed file system metadata service, REPLICHARD, which is substantially different from any of the above file systems using either hashing or subtree partitioning to distribute file system namespace across a shared-nothing cluster of metadata servers. Based on the observations that there are few “*read-after-write*” dependencies in applications [35], REPLICHARD provides metadata service by a cluster of metadata servers, in which a flexible consistency scheme is adopted: strict consistency for non-idempotent operations with dynamic write-lock sharding, and relaxed consistency with accuracy estimations of return values where consistency for idempotent requests is relaxed to achieve low latency.

As far as we know, REPLICHARD is the first attempt to combine two consistency models for the tradeoff between consistency and performance. The major challenges lie in: *how to get rid of conflicts during replicating* and *how to evaluate the staleness before all replicas reach consistency*. REPLICHARD addresses them by separating namespaces with write-locks in application’s viewpoint and proposing a credibility model, respectively.

- Write-locks are dynamically created at namespace subtree-level and designated to independent metadata servers in an application-oriented manner. Metadata updates that occur in a particular server for a subtree are replicated to all metadata servers conforming to the application “start-end” semantics, resulting in an eventually consistent namespace. Additionally, REPLICHARD supports the single root namespace, renaming and hierarchical access in general.
- REPLICHARD supports two execution modes: in *baseline* mode (hereafter RH-BL), operations to the same file are dispatched to the same metadata server with the ownership; thus strict consistency is guaranteed. *High performance* mode (RH-HP) directs *non-idempotent* operations (e.g., *write*) to the same ownership metadata server as RH-BL, while allowing applications to make *idempotent* operations (e.g., *read*) to any metadata server. Inevitably, RH-HP might result in outdated data for idempotent requests, therefore, an online credibility system is devised to evaluate the accuracy

of returned value for different applications.

To summarize the contributions of this paper, we design and implement a two-mode metadata service with reasonable tradeoff between consistency and scalable performance. This is also the first design with accuracy estimations of return values where consistency for idempotent requests is relaxed to achieve low latency. The prototype of REPLICHARD was implemented based on Hadoop HDFS [27], and tested against both micro benchmarks and real-world MapReduce applications. The experimental results show promising scalability and performance of REPLICHARD: in contrast to HDFS HA [23], REPLICHARD can deliver metadata speedup ranging from 182% to 251% and 310% to 415% for idempotent and non-idempotent operations, respectively. Especially, RH-HP out-performs all other metadata services with speedup up to 162% for various metadata intensive applications written in Pig [20], Hive [30] and MapReduce [7] in terms of job execution time.

The rest part of the paper is organized as follows: Section 2 discusses the background and motivations. Section 3 presents the system design. The implementation of REPLICHARD and the Paxos emulation follow in Section 4. Then, Section 5 evaluates the REPLICHARD prototype against current solutions with real-world applications. Related work is illustrated in Section 6, and finally, the paper is concluded in Section 7.

2. MOTIVATIONS

Due to its inherent characteristic of strict consistency, metadata service is difficult to scale. Meanwhile, with the rapid development of massive storage systems, it is more and more important to match the performance of metadata service to the entire system, especially for distributed file systems. The goal of REPLICHARD is to deliver a tradeoff between consistency and scalable performance for metadata service and the solution proposed is based on the following observations.

First, there are few “*read-after-write*” dependencies among files in the same application. Applications usually split files explicitly into *input*, *output*, and *intermediate temp* files, and files belonging to the same category do not usually depend on each other. Input files are only for reading, output files are only for writing, while intermediate files can be read or written but not both simultaneously. For “*read-after-write*” dependencies, some of them are actually false positives in terms of a *fine-grained* instead of *coarse-grained* viewpoint. A coarse-grained dependency means that a file is read after being written, no matter whether the read part overlaps the written part, while a fine-grained dependency means the read part must overlap the written counterpart. After investigating the benchmarks in Section 5, we found that although there are some coarse-grained dependencies, very few are fine-grained.

Second, metadata of a particular application’s files is only active when the application is running. Therefore, it is possible to design an application-oriented metadata service like COMET [35], in which application-related metadata can be dynamically designated to a certain metadata server for load balance and fault isolation.

Third, there are more idempotent file operations than non-idempotent ones. For MapReduce workloads, idempotent operations account for more than 80% of the total file operations [3]. By dispatching idempotent operations to different metadata servers rather than a central one, metadata latency

Figure 1: Replichard Architecture. Each namenode server is denoted as a square box in NameNodes. The *master* namenode is the one in gray at the bottom right. The tree structure in each namenode means which subtrees this namenode owns. Namenodes can talk to each other as being connected altogether. *res* is the original result of the file request, and *cr* the *credibility*.

could be reduced drastically.

Hence, a central metadata service can be distributed to a group of metadata servers and managed in a relaxed consistent way, without hurting the correctness of the application while providing greatly improved performance. The principles of REPLICHARD are derived from these observations, and will be described in detail in Section 3.

3. SYSTEM DESIGN

It is difficult to design and implement a metadata service to address the issues mentioned above. So, it is important to clarify the principles and goals we have to align with before evaluating the design.

- **Improved metadata throughput.** Both metadata read and write throughput should scale well as the system scales up and scales out, which will speed up user applications subsequently.
- **Flexible consistency.** Strict consistency should be employed for write operations to avoid write conflicts, and relaxed consistency for read operations to speed up user applications.
- **Successful execution guarantee.** With flexible consistency in consideration, applications may suffer from outdated read data, and program errors in turn. So, there should be a mechanism to guarantee the correctness of applications.

The primary goal of REPLICHARD is to balance metadata consistency and performance, thus, namenode failure recovery is outside of the scope of this paper. This is reasonable because the failed namenodes could be recovered using existing mechanisms such as failover.

3.1 Architecture

REPLICHARD is designed to be an improved, decentralized metadata service in contrast to the original single NameNode

Table 1: File requests and their categories.

Category	Request API
Idempotent	<code>getPreferredBlockSize</code> , <code>getFile-Info</code> , <code>getDatanodeReport</code> , <code>getBlock-Locations</code> , <code>getListing</code> , <code>getStats</code>
Non-idempotent	<code>create</code> , <code>append</code> , <code>complete</code> , <code>rename</code> , <code>delete</code> , <code>renewLease</code> , <code>mkdirs</code> , <code>setOwner</code> , <code>addBlock</code> , <code>setPermission</code> , <code>abandonBlock</code> , <code>setReplication</code>

in HDFS. REPLICHARD can leverage all HDFS components by carefully intercepting file operations of user applications via a library, encapsulating all communication between user application and the NameNode cluster (or *NameNodes* in REPLICHARD). The interface between user applications and DataNodes remains unchanged.

Figure 1 illustrates the overall architecture of REPLICHARD. Metadata service is provided by *NameNodes*, a group of metadata servers (each one is named as a *namenode*), in a coordinated way. All namenodes are connected and communicate periodically to exchange local metadata updates to reach a consistent global state. Each namenode is a fully functional metadata service that extends the original HDFS NameNode. When a new namenode enters the NameNodes cluster, it has to register itself in the global registry. Any file request from user applications will be dispatched to a proper namenode by the global registry according to the request's type and target.

REPLICHARD provides a user-level library, REPLICHARD *Client Library* (RCL), to be linked by user applications. RCL intercepts file operations and thus makes the communication to the global registry and NameNodes transparent to the user applications. Different file requests are handled in different ways. RCL classifies file requests from user applications into two categories: *idempotent* requests and *non-idempotent* requests, as illustrated in Table 1. RCL routes requests in a flexible consistent way as follows.

- For a non-idempotent request, RCL asks the global registry whether a namenode has been assigned as the owner of the write-lock for the request in terms of the file path. If not, the global registry will select a namenode from NameNodes, and then RCL will communicate with the selected namenode, the blue arrows in Figure 1 connecting **write** and a specific namenode. Thus, the namenode is the *owner* of the path's write-lock, and all non-idempotent requests to the same path will be routed to the same namenode to guarantee the single point of metadata updating for correctness and simplicity, and the updates will be replicated across all namenodes. The return value of the non-idempotent request is the same as it is in the original HDFS NameNode. Section 3.2 will explore the details of write-lock ownership management.
- Idempotent requests can be directed to any namenode server, red arrows connecting **read** and the entire NameNodes box in Figure 1, and return values not only include the original return value, but also an additional field, *credibility*, describing the probability of to what extent the return value is the latest value. So, the return value of **read** is a tuple (**res**, **cr**). Credibility calculation will be discussed in Section 3.4.

Algorithm 1 Major RCL workflows. DoREAD and DoWRITE are handlers for idempotent and non-idempotent operations respectively.

Require: GlobalRegistry, *IdempotentSet*, *CreditThreshold*

```

1: function DISPATCHER(request, callback)
2:   if request.name ∈ IdempotentSet then
3:     return DoREAD(request, CreditThreshold, callback)
4:   else
5:     return DoWRITE(request)
6:   end if
7: end function

8: function DoWRITE(request)
9:   GLOBALREGISTRY.LOCK(request)
10:  nn ← GlobalRegistry.getNamenode(request)
11:  if !nn then
12:    nn ← GlobalRegistry.selectNamenode()
13:    GlobalRegistry.assignNamenode(path, nn)
14:  end if
15:  GLOBALREGISTRY.UNLOCK(request)
16:  return ORIGINALREQUEST(request, nn)
17: end function

18: function DoREAD(request, threshold, callback)
19:  nn ← GlobalRegistry.selectRandomNamenode()
20:  result ← RCLRead(request, nn)
21:  if result.cr < threshold then
22:    CALLBACK(result)
23:  end if
24:  return result.res
25: end function

```

Algorithm 1 describes the RCL workflow as illustrated in Figure 1. DISPATCHER dispatches the request, an opaque object presenting all information of a file operation (e.g., API name, path and data), to DoWRITE or DoREAD by checking whether the request is idempotent. DoWRITE will create a record in the global registry that binds *path* and the selected namenode *nn* in Line 13. Then, RCL sends the request to *nn*, and waits for the result that the original non-idempotent operations will return from *nn*. For DoREAD, a credibility is returned along with the original idempotent operation’s return value from any metadata server, and the user provided CALLBACK will be invoked if the credibility does not meet the application’s requirement. The flexibility of CALLBACK allows user applications to respect the NameNodes for its additional credibility value without changing the prototypes of idempotent operations.

A *master* namenode is elected from Namenodes to accept DataNode heartbeat messages and block reports, and then DataNodes status will be broadcast to all namenodes as part of the metadata replication process. The election of master namenode is implemented with ZooKeeper [12]. With this piggybacking design, DataNodes are not required to send heartbeat to all namenodes which otherwise will lead to interface modification and additional communication overhead.

3.2 Dynamic Write-Lock Sharding

3.2.1 Application-Oriented Subtree Partition

Much existing research utilizes subtree-based namespace partitioning, and subtrees might be merged or split *dynamically* according to the “hotspot-coldspot” access pattern. For REPLICHARD, metadata is replicated in the namenode clus-

Figure 2: Write-lock ownership designation and bookkeeping update for MapReduce jobs. Only directories to be updated, */result* and *hadoop.tmp.dir/mapred/system*, are presented.

ter, so each subtree designates its write-lock to an *owner* namenode rather than being partitioned physically. Moreover, REPLICHARD defines subtrees in an *application-oriented* way, instead of the “hotspot-coldspot” access pattern. As it is *application-oriented*, REPLICHARD supposes each application will define the directory boundaries for different usage and will not access files outside of these directories. With the first non-idempotent operation for a particular path, a namenode server is assigned as the *owner* of the file’s write-lock. All non-idempotent requests will be handled by this namenode, and it will broadcast local metadata updates of this path to other namenodes. A simple *lease* mechanism is incorporated into REPLICHARD: designate an owner when an application starts, refresh the ownership during application execution before the lease timeouts, and release the ownership when the application ends. The global registry can assign the ownership to another namenode if the application does not refresh the lease in time.

With this design, the ownership of the same subtree may change over time as applications start and finish, and owners of different subtrees may span over all namenodes, in which way, it seems that the entire namespace is adaptively partitioned into subtrees jumping from one namenode to another, but it is unnecessary to detect hotspots and coldspots as existing systems. In addition, REPLICHARD bookkeeps the number of owners in every namenode, and queries the numbers upon new owner designation for the purpose of load balance. This write-lock sharding mechanism provides adaptive and dynamic load balance without data access pattern detection overhead.

Figure 2 illustrates the write-lock ownership designation and bookkeeping updates in the global registry. Two top-level nodes, */namenodes* and */owners*, are created for bookkeeping information and owner assignments, respectively. Initially, no namenode is assigned to any path, depicted by the part above the dashed line, so there are no child nodes. When an application starts, e.g., a MapReduce job, RCL will figure out which part of the file path will be used to assign a lock. In Hadoop, every job has to configure explicitly its “input directory” for input, “output directory” for output, and “{hadoop.tmp.dir}/mapred/system” for MapReduce control files. All files accessed in the job should be located in one of the above directories. Therefore, REPLICHARD adopts a straightforward subtree partitioning approach: creating nodes for the top-level output directory and MapReduce controlled directory of the job. For example, two jobs, *job1* and *job2* are submitted, whose output directories are */result/r1*

and `/result/r2`, respectively. Thus, REPLICHARD will create two nodes, `r1` and `r2` right below `/owners/result`, and record the assigned namenodes, `nn1` and `nn2`, respectively. In the meantime, two nodes will be created in `/namenodes`, saving how many subtrees have been assigned to the namenode. When the jobs end, all created nodes below `/owners` and `/namenodes` will be updated or deleted accordingly.

Generally, different applications use different directories, so the subtree partitioning in REPLICHARD is not only simple and efficient but also effective. Furthermore, REPLICHARD supports multiple applications to write in the same folder, because writes are dispatched to the folder's write-lock owner, which will coordinate all the write requests from different applications.

3.2.2 Two Execution Modes

One critical issue that has to be addressed in REPLICHARD is: before the metadata updates are propagated to all namenodes, file operations directed to non-owner namenodes might return outdated data, and the application might encounter execution failure. REPLICHARD monitors idempotent operations since the last namenode synchronization, and once a previously returned file operation with outdated data is detected, REPLICHARD will notify applications immediately. With this asynchronous mechanism, applications are able to deal with possible errors introduced by relaxed consistency. Sometimes, it might be difficult for the applications to decide how to proceed upon error notification. In this case, REPLICHARD can be executed in *baseline* mode (hereafter RH-BL), which also designates subtree owner dynamically to different servers as the default mode but *all operations* to the same file are directed to the same server, thus strict consistency is guaranteed. Though it can not perform as well as the default mode (high-performance mode, RH-HP), RH-BL can still deliver better performance in contrast to existing metadata service, due to the application-oriented subtree partitioning and write-lock sharding.

3.3 Namenode Replication

Though providing flexible load balance, the application-oriented dynamic subtree write-lock sharding also leads to metadata divergence. In order to guarantee that metadata updates are visible to all namenodes, metadata updated in one namenode should be replicated to all other namenodes, eventually. REPLICHARD facilitates *instant-broadcast* and *subtree-barrier* techniques to reduce the interval for the namenodes to reach consistency.

For instant-broadcast, metadata updates will be broadcast once the corresponding local file operation is successfully performed. The destination namenode receives the updates and merges them into its local metadata. Since there is only one owner for a particular subtree, and different subtrees do not overlap, the merging process is usually simply replacing, without any conflict.

As mentioned in Section 3.2.1, an owner will release the ownership of its subtrees when related applications exit. However, this does ensure the application “start-end” semantic. The metadata of an undesignated directory should be consistent in all nodes upon applications starting and ending, because the current metadata of the subtree might not have been replicated to all namenodes in the case of temporary network glitches, though instant-broadcast can ensure most of them are updated. *Subtree-barrier* will force a barrier

to guarantee the subtree's metadata is replicated globally before the owner releases the ownership. It is simple and straightforward to package the entire subtree metadata and broadcast it; however, this is not efficient, because most of the metadata has been delivered by *instant-broadcast* already. REPLICHARD utilizes Merkle tree [17] for data volume optimization as follows: each namenode maintains a separate Merkle trees for its current metadata, one file for one Merkle tree leaf node, one directory for one Merkle tree non-leaf node. During subtree-barrier, the owner collects all hashes of the same subtree from all other namenodes, compares which part of the metadata should be transferred using the tree traversal scheme, and then transfers only the necessary part.

The replication information is critical for credibility model estimation; thus, REPLICHARD namenode marks every metadata transfer with two timestamps: one is “*major write*”, when a file operation is performed in the remote owner namenode, and “*local write*”, when the remote update arrives at the receiver namenode. For read operations that are not replicated, each read is marked with “*local read*”.

With *instant-broadcast* and *subtree-barrier*, REPLICHARD ensures the subtree's ownership can be safely assigned to any namenode when new applications start.

3.4 Credibility Formulation

For idempotent operations, REPLICHARD namenode will return to RCL the *credibility*, describing the probability of to what extent the return value is the latest value, along with the original operation result, as described in Algorithm 1 Line 20. Therefore, there are two major issues to deal with: “*how to formulate the credibility*” and “*what to do with the credibility*”.

3.4.1 Model Estimation

REPLICHARD formulates credibility with a *logistic regression model* (LRM). The output of LRM, thereby credibility, is between 0 and 1, and a bigger credibility indicates the returned value is more likely to be the latest value. The model is described as in Equation (1).

$$f(w, r, i) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 w + \beta_2 r + \beta_3 i)}} \quad (1)$$

Where f is the credibility, w the “*write count*” (for all non-idempotent operations), r the “*read count*” (for all idempotent operations), and i the elapsed time since the last write in a sliding window, consisting of the 50 most recent file requests. Applications based on HDFS usually issue sequential reads and writes, and most map and reduce tasks access files in a repetitive way, so the number of reads and writes are good indicators to demonstrate remote updates. Since all write requests are directed to the owner namenode, the *write count* for a non-owner namenode is obtained by counting the replicating messages in Section 3.3. Different namenodes usually observe different read patterns and apparently the last update intervals can hardly be the same across namenodes, so each namenode has to maintain an independent model for each directory as presented in Figure 2.

To estimate the coefficients, REPLICHARD has to figure out whether a read operation returns a correct result. If it does, f is 1, otherwise 0. Figure 3 demonstrates how to determine the value of f . Recall the definitions of *major write*, *local write* and *local read* in Section 3.3. Timestamps marked in triangle (t_1, t_5) are *major writes*, in circle (t_3, t_7) are *local*

Figure 3: File request history and LRM estimation.

writes, and in vertical solid line (t_2 , t_4 and t_6) are *local reads*. Generally, if a *local read* (e.g., t_2) appears between a *major write* and a *local write*, its return value should be wrong, since namenodes are in an inconsistent state; thus, f in t_2 is 0. Otherwise, the return value of a read between two *major writes* is absolutely correct, thus, f in t_4 is 1. The corresponding values of f and (w, r, i) for *local reads* are presented in Table 2.

The model in Equation (1) could be estimated once all the fitting data is available as the application moves forward. Since the file access pattern may change as the application runs, the model should be updated over time. REPLICHARD re-estimates the coefficients when either of the following conditions is satisfied: a specific number of requests have been conducted, or the false positive rate is greater than a threshold.

3.4.2 Credibility Evaluation

Besides performing the original operations, a namenode will also calculate the credibility once a read operation is received based on the estimated model. And, both the result of the original operation and the credibility will be returned to RCL, despite whether or not the calculated credibility meets the application’s requirement. Before there is enough history request data to get the model fitted, REPLICHARD always sets credibility to 1. RCL, running in the application’s address space, will invoke the callback provided by the application when the credibility accompanied with the result is smaller than the application’s expectation, as in Algorithm 1 Line 22. Besides the credibility returned with the operation’s result, REPLICHARD will also notify RCL once a previous read with possible outdated data is detected, and it is the application that decides how to proceed. If the application can not accept either of these cases, it should re-execute the application in baseline mode to avoid potential errors. With this asynchronous mechanism, applications are able to deal with possible errors introduced by relaxed consistency.

4. IMPLEMENTATION

The prototype of REPLICHARD is implemented based on Hadoop-0.20, with about 12,000 lines of Java code. Popular open source tools and systems are used extensively. For example, the global registry is an instance of ZooKeeper [12], which manages the ownership and bookkeeping information in Figure 2, coordinates *subtree-barrier*, and deals with lock actions of a request in Algorithm 1 Line 9 and 15. Namenode communication involved in replication is handled by ZeroMQ [37], a lightweight but full-featured distributed messaging library, thus REPLICHARD can implement broadcasting efficiently with reasonable overheads. Several important issues are addressed with respect to implementation and optimization as follows.

- **Balanced owner selection.** To reduce the entry ownership skews of the namenodes, load balance suggestions from [18] are used for the initial server assignment. Us-

Table 2: Fitting data for history in Figure 3.

time	f	(w, r, i)
t_2	0	(0, 0, a)
t_4	1	(1, 1, b)
t_6	0	(1, 2, c)

ing the well-known “*power-of-two-choices*” principle, the proper owner namenode is selected by probing 2 random servers and placing the subtree on the one with fewer stored entries.

- **BlockID separation.** In REPLICHARD, blocks might be created independently by different namenodes simultaneously for they could be owners of different output directories; however, this decentralized way may lead to BlockID conflict because by default, namenodes randomly assign BlockIDs. REPLICHARD has to guarantee that the BlockIDs assigned by one namenode do not conflict with that of other namenodes. We adopt the same approach as Comet [35] by reserving the first 32 bits as “namenode identifier” assigned by the global registry for any active namenode. Namenode identifiers separate the “BlockID space” on the basis of the namenode to which the block belongs; thus, BlockID conflict can be prevented.
- **Paxos-based consensus namenode emulation.** In order to evaluate to what extent REPLICHARD could out-perform a strong-consistent distributed metadata cluster like *Consensus Namenode* [26], we obtain a performance upper bound with a Paxos-like system by simulating only the network overheads for *instant-broadcast* required in Paxos, i.e., accepting $(n + 1)/2$ approval messages before committing, where n is the number of namenodes. This simulation establishes a performance upper bound that avoids the complexity of implementing such a real system, while guaranteeing that it out-performs any real systems by ignoring the overheads of local running time and crash recovery.

5. EXPERIMENTAL EVALUATION

Several experiments were conducted to evaluate both the performance and prediction efficiency of REPLICHARD in terms of metadata requests and execution time of real-world applications. All experiments were conducted in a homogeneous cluster consisting of 10 nodes, in which all nodes are shared by user applications and metadata services. Each node has an Intel Celeron CPU running at 1.40GHz, with 4GB of RAM, 500GB of hard disk, and a 1Gb/s network adapter. They have Ubuntu 14.04 LTS (x86_64) installed and were connected through a 1Gb LAN switch. Though the hardware is relatively low-end compared to real-world configurations, it can behave similarly to the high-end hardware as it’s a proportional scale-down.

NNBench, a standard NameNode metadata benchmark, was executed to obtain the metadata performance, while real-world applications selected from Hadoop [1], a publicly available collection of MapReduce applications maintained by the University of Washington and Duke University, were used to evaluate the overall performance of REPLICHARD.

In order to squeeze all performance potentials for HDFS Federation, different output directories were configured to be placed in different namenodes, resembling RH-BL but in

Figure 4: Metadata performance comparison among HDFS HA, Hadoop Federation, Paxos, RH-BL and RH-HP. Every system is configured with 7 namenodes, except HA, only 2 are used, one in active and the other in standby.

Figure 5: Metadata performance comparison between RH-BL and RH-HP with different number of namenodes.

a static manner.

5.1 Metadata Performance

We examined the metadata opening, creation, and deletion throughput of HDFS HA, HDFS Federation, Paxos, RH-BL, and RH-HP as well as performance scalability of RH-BL and RH-HP with different namenode configurations.

Intensive file creation requests are sent by different numbers of *NNBench* clients to the original NameNode and REPLICARD with different numbers of namenodes. Figure 4 and Figure 5 present the experimental results in terms of aggregated metadata throughput.

5.1.1 Non-idempotent Operation Throughput

Create and **delete** are used to demonstrate non-idempotent file operations. Figure 4a and Figure 4b present throughputs of **create** and **delete** operations. HDFS HA’s performance is the worst because there is only **one** namenode, the active one, in service, while for all other systems, all seven namenodes can handle client requests. In contrast to HDFS HA, REPLICARD can deliver speedup ranging from 310% to 415% for non-idempotent operations. Federation scales metadata service horizontally using multiple namenodes, and delivers better throughput than the simulated Paxos. This is because Paxos spends quite a while on the two-phase protocol to reach strong consistency. Compared with Federation, to some extent, RH-BL performs similarly due to the load balance mechanism. Moreover, the throughput of RH-HP is 7% lower than that of RH-BL on account of message synchronization among namenodes.

Figure 5a and Figure 5b illustrate write throughputs concerning **create** and **delete** operations in RH-BL and RH-HP ranging from 1 namenode to 8 namenodes. Both figures indicate that the metadata throughput increases as the namenode scales out. In addition, RH-HP performs worse than

RH-BL because there are only non-idempotent operations in these tests, and RH-HP can not leverage the benefit of relaxed consistency, while in the meantime, RH-HP has to afford the overhead of namenode replication. In addition, RH-HP scaled slightly less in **delete** throughput, that is, 1%, 3%, 7%, and 12% lower than the RH-BL, while for **create**, RH-HP is 5%, 7%, 10%, and 15% lower than RH-BL, for 1, 2, 4 and 8 namenodes, respectively. This is because the **delete** operation has a shorter response time and thus is greatly affected by the replication time. As discussed in Section 4, our Paxos simulation is a performance upper bound for systems with strict consistency, and Figure 4 indicates that REPLICARD can provide better metadata performance than Paxos.

5.1.2 Idempotent Operation Throughput

Open is used to demonstrate idempotent file operations. Figure 4c depicts the throughputs of **open** operations in different systems. Not surprisingly, HA delivers the worst throughput for idempotent requests as non-idempotent requests in Figure 4a and Figure 4b. In contrast to HDFS HA, REPLICARD can deliver speedups ranging from 182% to 251% for idempotent operations. RH-BL is 38.17% higher than Federation on account of skewed loads in the latter. Further more, RH-HP is 24.5% higher than RH-BL, and 72.0% higher than Federation due to the random distribution method.

Figure 5c shows that REPLICARD is able to achieve near-linear scalability for idempotent operations, as many as 4000 operations per second with 8 namenodes. In the single namenode configuration, RH-HP has lower throughput than RH-BL because of the overhead introduced by request history tracking and credibility model estimation, though they are unnecessary with only one namenode. When the number of namenode exceeds two, however, the throughput increases

Table 3: Descriptions on the selected real-world applications.

Application	Description
Cloudburst	A MapReduce implementation of the RMAP algorithm for short-read gene alignment.
PageRank	A link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively aggregating the weights of its inbound neighbors.
TF-IDF	Term Frequency-Inverse Document Frequency is a statistical measure used to evaluate how important a word is to a document in a collection or corpus, often used in information retrieval and text mining.
TPC-H-Pig	The Pig implementation of The TPC Benchmark TM H (TPC-H), a decision support benchmark [2].
TPC-H-Hive	Hive implementation of TPC-H.
PigMix	A set of queries used test Pig performance, including queries that test latency and queries that test scalability and a set of map reduce java programs to run equivalent map reduce jobs directly.
Wordcount	the wordcount example coming with Hadoop.

rapidly and RH-HP surpasses RH-BL thereafter.

5.2 Credibility Prediction

It is not adequate to adopt REPLICHARD in high-performance mode unless the following two issues are addressed concerning credibility:

- How accurate is the projected credibility?
- How probably could the application finish correctly in the case of credibility threshold violation?

Seven real-world applications including Pig, Hive, and MapReduce, were selected from Hadoop [1] as in Table 3. They were submitted to RH-HP with the credibility threshold ranging from 0 to 1. Both credibility smaller than the threshold and asynchronous notifications of outdated read described in Section 3.4.2 were ignored in the tests; thus, the benchmarks try best to finish the execution, and it is possible for us to evaluate whether the output results are correct. Fortunately, no job crash was observed in our experiments, indicating that all applications are unaware of the metadata accuracy.

The accuracy of projected credibility is measured in terms of *correct ratio*, the ratio of the correct credibility predictions. The ratio is computed as follows:

- For each read, if the predicted credibility is higher than the threshold, set the mark as 1, indicating the latest value is returned, otherwise 0. In this way, the reads generate a sequence of 1's and 0's.
- By checking the real file request history, it is easy to confirm whether a read returns its latest value, thus producing another sequence of 1's and 0's.
- Then, we count how many predictions are correct by comparing how many corresponding elements in the two sequences are the same, then we can calculate the correct ratio.

Figure 6 indicates that all applications share a similar curve shape, and in general, as the credibility threshold

(a) Correct results without waiting.

(b) Correct results by waiting for 2ms for lower credibilities than the threshold, otherwise results are incorrect.

Figure 6: Variations of correct ratios of predicted credibility along with the changing of credibility threshold.

increases, the correct ratio decreases because of the increase of false negatives. The only 100% correct prediction occurs in TF-IDF when the threshold is 0, and the correct ratio is never below 42%. It is interesting to note that the peak correct ratio appears at 0.1 instead of 0 in Figure 6b. This is because some true negatives at 0.1 are regarded as false positives when the threshold is 0.

And unsurprisingly, some benchmarks will generate incorrect output, such as TPC-H-Pig, TPC-H-Hive, PageRank and CloudBurst, while some can still deliver correct results even without 100% credibility as in Figure 6a. This is because the false positives of projected credibility stem from coarse-grained dependencies, which are actually not in terms of fine-grained dependency. For example, once a file is created, neither modification nor appending is allowed. Any following read will not raise real dependency even if the predicted credibility is smaller than the threshold. The failed applications could still generate correct results if waiting for quite a small period (2ms in our test, a heuristic time interval the model needs wait to return a predicted credibility bigger than the threshold from the last predicted credibility smaller than the threshold) for metadata replication with some thresholds as presented in Figure 6b, however, as the threshold grows, it is impossible for the estimation model to predict acceptable credibility in a limited waiting period. That is why Figure 6b does not provide the data for threshold greater than 0.8, in which case the application has to be executed in baseline mode.

The investigations on credibility projection reveal that many applications can leverage RH-HP to pursue improved performance without compromising correctness, while some applications have to be executed in RH-BL mode to ensure the application can output correct results without modifying the code to support callbacks in RCL.

5.3 Overall Performance

Figure 7: Execution time speedup compared to the original Hadoop HDFS architecture. Every system is configured with 7 namenodes, except HA, only 2 are used, one in active and the other in standby.

This section presents the overall performance improvement of REPLICHARD over existing metadata services in terms of applications in Table 3. The credibility threshold used by the clients in these tests was based on the peak correct ratios obtained in Figure 6. The experiment results are illustrated in Figure 7.

Performance speedup was observed in all metadata services except HA. With HA, all applications execute a little slower than the original single NameNode architecture. This is because all file requests are processed by the active namenode, which has to deal with synchronization with the standby namenode simultaneously; nevertheless, the standby namenode does not help to balance the system load. For a system of multiple namenodes that are in service at the same time, the more file operations there are in an application, the more speedup will be observed.

Figure 7 shows that PigMix, TPC-H-Pig, TPC-H-Hive, PageRank and CloudBurst gain remarkable speedup for Paxos and REPLICHARD, especially, for RH-HP, the speedup ranging from 103% to 162%. This could be attributed to the intensive metadata operations and especially the predominance of read accesses in these applications. Statistics [6] reveal that a major fraction of Hadoop jobs is small jobs whose duration is less than a minute, and some only last for seconds. Most of the small jobs are generated by query-like frameworks like Hive and Pig. For example, a complex Pig program may create a bunch of small MapReduce jobs and jobs may have data dependencies on each other. The intermediate result is stored on HDFS to be accessed by successor jobs. Inversely, for applications of fewer metadata operations, like TF-IDF, the speedup of RH-HP, RH-BL, and Paxos are 110.41%, 105.29%, and 101.45%, respectively, which is much slower than the aforementioned applications.

As described in Section 4, our Paxos emulation service demonstrates a performance upper bound for strict consistent metadata servers. Figure 7 indicates that REPLICHARD, both the baseline mode and high-performance mode, out-performs Paxos emulation due to the relaxed consistency.

Though it can not speedup all applications, REPLICHARD provides promising metadata throughput and performance improvement for a variety of real-world applications. There is sufficient space for the balancing of consistency and performance, and REPLICHARD demonstrates a case study of the balancing.

6. RELATED WORK

REPLICHARD builds upon a long history of research on the scalability and performance of distributed file systems. In this

section, we discuss prior work related to metadata services in modern cluster file systems, optimized techniques for high-performance metadata, and consistency relaxed systems.

Namespace Distribution. HDFS Namenode Federations [28] is based on the idea that multiple independent namespaces can share a common pool of DataNodes as a block storage layer. However, the federated approach provides static partitioning of the federated namespace. If one volume grows faster than the other and the corresponding NameNode reaches the limit, the metadata service will suffer from performance degradation. At the same time, it lacks flexibility. Partitioner [34] adopts near-equal subtree splitting and radix-tree-based lookup for better scalability and efficiency. IndexFS [24] uses the GIGA+ [22] binary splitting technique to distribute directory entries and uses coherent client caching of pathname lookup directory attributes. However, IndexFS’s clients observe higher latency due to cache misses in some directories. CalvinFS [29] file metadata is partitioned by the hash of the pathname and replicated across a (shared-nothing) cluster of independent servers. A directory’s files and subdirectories are stored as its value for `readdir`. This makes file creation a distributed transaction and hence does not scale well. Giraffa [33] stores file system metadata as entries in a table. Metadata distribution and load balance are delegated to the underlying table system, HBase [9]. The row key schema used by Giraffa only supports renaming of files in the same directory, because of the difficulty of supporting atomic renaming of directories. Ceph [31] uses an adaptive partitioning technique for distributing its metadata and directories on multiple metadata servers and hashing *hotspot* directories across multiple metadata servers. Nonetheless, the system is quite complex and needs additional cost to re-allocate data upon change of use access pattern or metadata cluster.

High Performance Metadata. *CNode*(Consensus Node) [26] enables replication of the namespace on multiple nodes of an HDFS cluster by means of a Coordination Engine. The Coordination Engine can be based on different algorithms, such as Paxos [15], 2PC [11], Raft [21] and Zab [13] (Zookeeper Atomic Broadcast), and can have different implementations depending on use cases, reliability, availability, scalability, and performance requirements. However, for metadata replication, all of them spend too much time to reach a consensus state, compared with one metadata server. Because of its complexity, *CNode* is still under development. ShardFS [33] scales distributed file system metadata performance by fully replicating directory lookup state across servers, which is based on pessimistic multi-server locking for

all metadata operations. However, only the scaling benefit that these operations get from namespace replication is the primary motivation for this technique. Other operations are distributed transactions and use a two-phase locking protocol, limiting overall scalability. IndexFS [24] uses two-phase distributed transactions for operations requiring a distributed transaction protocol such as directory splitting and rename operations, introducing a lot of latencies. In CalvinFS [29], directory renaming and permission changes need to recursively modify all entries under the affected subtree. NM²H [36] separates the storage and query of metadata by replacing the underlying storage of HDFS NameNode with NoSQL servers.

Flexible Consistency or Semantic. Dynamo [8] pioneered the idea of eventual consistency as a way to achieve higher availability and scalability: data fetched is not guaranteed to be up to date, but updates are guaranteed to be propagated to all nodes eventually. Applications can specify the version to be updated, for the put and delete operations. Nevertheless, it can not provide information about the probability of to what extent the return value is the latest value; thus, the application can not know the gap between their return value and latest value. An alternative approach taken by Giraffa [33] is to relax file system access control semantics and store the full `pathname` for each entry to reduce the number of lookups without checking the permission information of every ancestor directory. This reduces the number of client-server communications and helps with performance. Saturn [4] implements causal consistency for metadata service. It also classifies operations into “updates” and “reads”, similar to non-idempotent and idempotent requests in REPLICHARD. Saturn is different from REPLICHARD in the following aspects: (1) Saturn will issue a remote read if the local datanode can not provide the latest value, while REPLICHARD will just return the local copy as well as a credibility. (2) Saturn tries to minimize the size of metadata because it targets for geo-replication, while REPLICHARD is designed for a typical non-geo-distributed system. (3) Saturn is designed to be a pluggable component that can be used by distributed applications that manage replicated data, while REPLICHARD is a dedicated metadata service for file systems.

Client-oriented Metadata Management. Comet [35] proposes a novel client oriented metadata approach. The design of single central NameNode remains, but Comet will create RegionNodes dynamically and replicate metadata to RegionNodes upon application creating, get all metadata updates merged back to the NameNode when the application finishes and then destroy the RegionNodes. In contrast, REPLICHARD deploys multiple NameNodes and updates are conducted on corresponding NameNode; then the updates are replicated periodically among all namenodes. Additionally, Comet performs all file operations of one application in the same RegionNode, while REPLICHARD allows any namenode to server the application.

7. CONCLUSION AND FUTURE WORK

REPLICHARD is proposed to improve the performance of metadata service by exploiting the possibility to relax its consistency. REPLICHARD is designed based on the observation that real file dependencies are few, and there are usually more idempotent than non-idempotent operations. With *dynamic write-lock sharding* and *namenode replication*, REPLICHARD off-loads namespace subtrees in an application-oriented manner, supports flexible consistency for different operations for

the sake of scalable performance. The credibility model is estimated and calculated along with application execution, and the baseline mode is provided in case the client can not accept the projected credibility. A prototype of REPLICHARD was implemented based on HDFS and tested against both micro benchmarks and real-world applications. The experimental results demonstrate that REPLICHARD can deliver an acceptable balance between consistency and performance.

The lesson we learned from REPLICHARD is: it is possible to design and implement a scalable metadata service for distributed storage systems by carefully balancing consistency and performance without compromising correctness, especially for data/metadata intensive applications.

In the future, we will address namenode failures, support automatic execution mode switch and improve correct ratio for credibility estimation. In addition, we will figure out the criteria to find out applications suitable for REPLICHARD.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the paper. This work is supported by the Natural Science Foundation of China under grant numbered 61272528 and 61035004.

8. REFERENCES

- [1] Hadoop: Repository of mapreduce applications. <http://nuage.cs.washington.edu/repository.php/>.
- [2] The TPC Benchmark H. <http://www.tpc.org/tpch/>.
- [3] Cristina L Abad, Nathan Roberts, Yi Lu, and Roy H Campbell. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 100–109. IEEE, 2012.
- [4] Manuel Bravo, Luís E. T. Rodrigues, and Peter Van Roy. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In Ivan Beschastnikh and Wouter Joosen, editors, *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference, Middleware Doct Symposium 2015, Vancouver, BC, Canada, December 7-11, 2015*, pages 1–4. ACM, 2015.
- [5] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvf: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000.
- [6] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.
- [7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [8] John R Douceur and Jon Howell. Distributed directory service in the farsite file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 321–334. USENIX, 2006.
- [9] Lars George. *HBase: The Definitive Guide*. O’Reilly Media, 1 edition, 2011.

- [10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43. ACM, 2003.
- [11] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.
- [12] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, volume 8 of *USENIXATC’10*, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.
- [13] Flavio P. Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.
- [14] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- [15] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- [16] Lustre. Lustre file system. <http://www.lustre.org/>.
- [17] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology*, pages 369–378. Springer, 1988.
- [18] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.
- [19] Henry Newman. Hpcs mission partner file i/o scenarios, revision 3. <http://wiki.lustre.org/images/5/5a/NewmanMayLustreWorkshop.pdf>, 2008.
- [20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.
- [21] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.
- [22] Swapnil Patil and Garth A. Gibson. Scale and concurrency of GIGA+: file system directories with millions of files. In Gregory R. Ganger and John Wilkes, editors, *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 177–190. USENIX, 2011.
- [23] Sanjay Radia. High Availability Framework for HDFS NN. <https://issues.apache.org/jira/browse/HDFS-1623>, August 2012.
- [24] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: scaling file system metadata performance with stateless caching and bulk insertion. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 237–248. IEEE, 2014.
- [25] Drew S Roselli, Jacob R Lorch, Thomas E Anderson, et al. A comparison of file system workloads. In *USENIX annual technical conference, general track*, pages 41–54, 2000.
- [26] Konstantin Shvachko. Coordinated replication of the namespace using ConsensusNode. <https://issues.apache.org/jira/browse/HDFS-6469>, February 2015.
- [27] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.
- [28] Suresh Srinivas. Hdfs scalability with multiple namenodes. <https://issues.apache.org/jira/browse/HDFS-1052>, April 2011.
- [29] Alexander Thomson and Daniel J Abadi. Calvinfs: consistent wan replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2015.
- [30] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.
- [31] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.
- [32] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.
- [33] Lin Xiao, Kai Ren, Qing Zheng, and Garth A Gibson. Shardfs vs. indexfs: replication vs. caching strategies for distributed metadata management in cloud storage systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 236–249. ACM, 2015.
- [34] Ruini Xue, Lixiang Ao, Shengli Gao, Zhongyang Guan, and Lupeng Lian. Partitioner: A distributed hdfs metadata server cluster. In *2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 167–174. IEEE, 2014.
- [35] Ruini Xue, Lixiang Ao, and Zhongyang Guan. Comet: Client-oriented metadata service for highly available distributed file systems. In *2015 IEEE 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 157–164. IEEE, 2015.
- [36] Ruini Xue, Zhongyang Guan, Shengli Gao, and Lixiang Ao. NM2H: design and implementation of nosql extension for HDFS metadata management. In *17th IEEE International Conference on Computational Science and Engineering, CSE 2014, Chengdu, China, December 19-21, 2014*, pages 1282–1289. IEEE Computer Society, 2014.
- [37] ZeroMQ. 0MQ: Distributed Messaging. <http://zeromq.org/>.