Efficient and Scalable Metadata Management in EB-scale File Systems

Quanqing Xu, *Member, IEEE,* Rajesh Vellore Arumugam, *Member, IEEE,*
Khai Leong Yong, *Member, IEEE,* and Sridhar Mahadevan

Abstract—Efficient and scalable distributed metadata management is critically important to overall system performance in large-scale distributed file systems, especially in the EB-scale era. Hash-based mapping and subtree partitioning are state-of-the-art distributed metadata management schemes. Hash-based mapping evenly distributes workload among metadata servers, but it eliminates all hierarchical locality of metadata. Subtree partitioning does not uniformly distribute workload among metadata servers, and metadata needs to be migrated to keep the load balanced roughly. Distributed metadata management is relatively difficult since it has to guarantee metadata consistency. Meanwhile, scaling metadata performance is more complicated than scaling raw I/O performance. The complexity further rises with distributed metadata. It results in a primary goal that is to improve metadata management scalability while paying attention to metadata consistency. In this paper, we present a ring-based metadata management mechanism named Dynamic Ring Online Partitioning (DROP). It can preserve metadata locality using locality-preserving hashing, keep metadata consistency, as well as dynamically distribute metadata among metadata server cluster to keep load balancing. By conducting performance evaluation through extensive trace-driven simulations and a prototype implementation, experimental results demonstrate the efficiency and scalability of DROP.

Index Terms—Metadata Management, Locality-preserving Hashing, Dynamic Load Balancing, EB-scale File Systems.

1 Introduction

Modern EB-scale (10^{18} or 2^{60} bytes) file systems [1] separate file data access and metadata transactions to achieve high performance and scalability. Data is stored on a storage cluster including many servers that are directly accessed by clients via the network, while metadata is managed separately by a metadata server (MDS) cluster consisting of a number of dedicated MDSs. The dedicated metadata server cluster manages the global namespace and the directory hierarchy of file system, the mapping from files to objects, and the permissions of files and directories. The MDS cluster just allows for concurrent data transfers between large numbers of clients and storage servers instead of being responsible for the storage and retrieval of data. Meanwhile, it provides efficient metadata service performance with specific workloads, such as renaming a large directory near the root of the hierarchy and thousands of clients updating to the same directory or accessing the same file.

The main problem of designing a MDS cluster is how to partition metadata efficiently among MDS cluster to provide high-performance metadata services [2], [3]. MDS cluster is involved in moving metadata to keep MDSs storage load balancing [4]. In order to keep good namespace locality, some MDSs are heavily overloaded in storage load, while other MDSs are lightly overloaded. A well-designed MDS cluster should be able to achieve satisfactory storage load balancing. In addition, we have to efficiently organize and maintain very large directories [5], each of which may contain billions of files. Internet applications such as Facebook [6] already have to manage hundreds of billions of photos. As there are millions of new files uploaded by users every day, the total number of files increases very rapidly and will soon be more than one trillion. Meanwhile, we have to provide high-performance metadata services for a large-scale file system with hundreds of billions or trillions of files. For example, Facebook serves over one million images per second at peak, and one billion new photos per week [6].

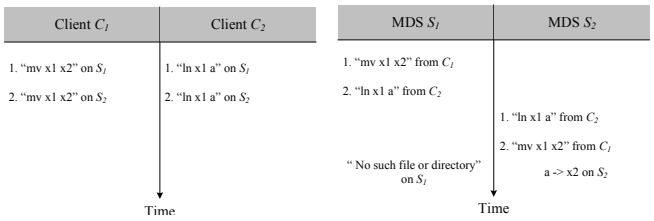
Compared to the overall data space, the size of metadata is relatively small, and it is typically 0.1% to 1% of data space [7], but it is still large in EB-scale file systems, e.g., 1PB to 10PB for 1EB data. Besides, 50% to 80% of all file system accesses are to metadata [8]. Therefore, in order to achieve high performance and scalability, a careful metadata server cluster architecture must be designed and implemented to avoid potential bottlenecks caused by metadata requests. To efficiently handle the workload generated by a large number of clients, metadata should be properly partitioned so as to evenly distribute metadata traffic by leveraging the MDS cluster efficiently. At the same time, to deal with the changing workload, a scalable metadata management mechanism is necessary to provide highly efficient metadata performance for mixed workloads generated by tens of thousands of concurrent clients [9]. The concurrent accesses from a large number of clients to large-scale distributed storage will cause request load imbalance among metadata servers and inefficient use of metadata cache. Caching is a popular technique to handle request load imbalance, and it is both orthogonal and complementary to the load balancing technique proposed in this paper.

Meanwhile, managing multiple MDSs brings many difficulties, in which maintaining consistency among multiple replicas

- *The author are with Data Storage Institute, A*STAR, Singapore, 138632. E-mail: {Xu_Quanqing, Rajesh_VA, YONG_Khai_Leong, Sridhar_M}@dsi.a-star.edu.sg*

Digital Object Indentifier 10.1109/TPDS.2013.293

of the same directory hierarchy is quite difficult. For example, as shown in Figure 1, there are two clients that simultaneously perform an operation on the same file in two MDSs: client C_1 renames a file *x1* to *x2* and client C_2 makes a hard link *a* for it by accessing the MDS S_1 followed by the MDS S_2. Since the requests come from different clients, there is no guarantee on the execution order from the MDS point of view, so the resulting states of the two MDSs are not consistent. Therefore, distributed algorithms are required to maintain the consistency among multiple MDSs. Maintaining consistency between two replicas of the same directory hierarchy is not straightforward. Each client operation must be atomic, and be executed in the same order on all the MDSs.



(a) Clients (b) Metadata Servers

Fig. 1. Metadata inconsistency. It is successful on S_2, while there is an error on S_1: "No such file or directory".

In this paper, we propose a novel metadata server cluster architecture named Dynamic Ring Online Partitioning (DROP). It is a highly scalable and available key-value store, and it provides a simple interface: *lookup(key)* under *put* and *get* operations. In DROP, we use locality-preserving hashing (LpH) to improve namespace locality, thus increasing put/get success rate depending on fewer MDSs and upgrading put/get performance involving fewer lookups. Maintaining metadata hierarchical locality improves availability and performance of metadata substantially, but it causes storage load imbalance in the MDSs. We explore an efficient Histogram-based Dynamic Load Balancing (HDLB) mechanism in DROP, and we also prove the convergence of the proposed mechanism. Meanwhile, it provides a linearizable consistency mechanism using ZooKeeper [10] to keep excellent metadata consistency.

We make the following contributions on the problem of distributed metadata management. First, we propose an effective locality-preserving hashing that keeps excellent namespace locality. Second, we present an efficient dynamic load balancing algorithm named HDLB to balance storage load in MDSs. Third, we give a linearizable consistency mechanism using ZooKeeper that keeps excellent metadata consistency among MDSs. Finally, we evaluate DROP and its competitors by simulations from multiple perspectives, and we demonstrate that DROP converges to load balancing quickly with different MDS cluster sizes. Our results based on trace-driven simulations and a prototype implementation demonstrate that DROP is more efficient than traditional state-of-the-art metadata management approaches. Compared with the conference version in [11], this paper presents an efficient linearizable consistency mechanism for the DROP MDS cluster, and it shows extensive experimental results. In addition, a proof-of-concept prototype of DROP has been implemented and empirically evaluated in terms of multiple metrics.

The rest of the paper is organized as follows. Section 2 presents the system architecture design of DROP. Section 3 describes the proposed mechanism of preserving namespace locality. The HDLB mechanism is presented in Section 4. Section 5 introduces metadata consistency using ZooKeeper in DROP. In Section 6 and Section 7 we present performance evaluation results of DROP, and prototype implementation and evaluation respectively. Section 8 describes related work. In Section 9 we conclude this paper.

2 DROP Design

Like hash-based mapping, DROP uses hashing to distribute the metadata across the MDS cluster. However, it still maintains hierarchical directories to support common directory hierarchy.

2.1 Goals

DROP is a distributed key-value store system, in which a key-value pair is shown in Figure 2. It is designed to meet the following four general goals: 1) high scalability of MDS cluster, 2) excellent namespace locality, 3) dynamic load balancing, and 4) metadata consistency. DROP is designed to scale to a large-scale distributed metadata server cluster for EB-scale file systems within a single global namespace. DROP uses pathname-based locality-preserving hashing explained in Section 3 for metadata distribution and location, avoiding the overhead of hierarchical directory traversal, and maintains hierarchical directories to provide directory operations such as renaming a directory. To access data, a client hashes the pathname of the file with the same locality-preserving hash function to locate which metadata server contains the metadata of the file, and then contacts the appropriate metadata server. The process is extremely efficient metadata access, typically involving a single message to a single MDS. Due to using locality-preserving hashing, the key distribution is no longer uniform in DROP, causing load balancing is a great challenge. We propose a simple but efficient dynamic load balancing algorithm to guarantee load balancing with losing negligible locality in metadata placement, as explained in Section 4. We discuss metadata consistency in DROP in Section 5.

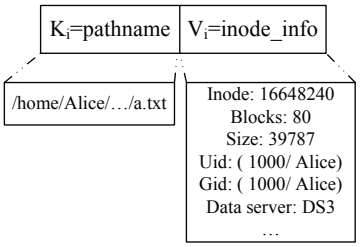


Fig. 2. Key-value data structure. *Key* is pathname, while *value* is its *inode* information.

2.2 System Architecture

The system architecture is shown in Figure 3, where a typical standard hash table evenly partitions the space of possible hash values. Current hash-based mapping does not evenly partition the address space into which keys get mapped, causing some metadata servers get a larger portion of it. To cope with this

problem, virtual nodes are used as a means of improving load balancing [12], each participating independently in the DROP network, thus its load is determined by summing over several virtual nodes'. Virtual nodes make not only re-distribution become easier, but also scaling out as data grows. When scaling out, more physical MDSs may be added and virtual nodes can be moved onto them seamlessly.

A physical MDS has to allocate storage space for each virtual node to store necessary data structures. It means that more virtual nodes need more space, leading to better load balancing. DROP can achieve better namespace locality and load balancing by allocating more virtual nodes per metadata server since metadata IDs are not uniformly distributed. The data structures are typically not so expensive from the perspective of space, thus it is not a serious problem. We have to consider a much more significant problem arising from network bandwidth. In general, to maintain connectivity of the network, every node frequently pings its neighbors to make sure them still alive, and replaces them with new neighbors if they are not alive any more. To maintain the DROP network, there is a multiplicative increase in network traffic because of running multiple virtual nodes in each MDS, but it is located in a data center with enough bandwidth.

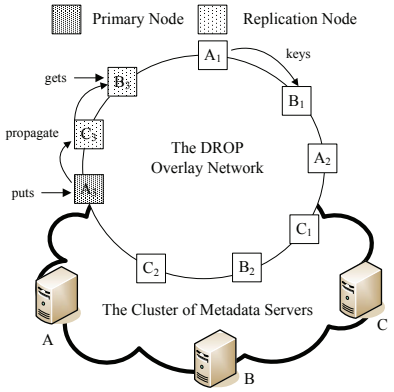


Fig. 3. System Architecture. Physical metadata servers compose a MDS cluster, while their virtual nodes form a DROP overlay network.

2.3 DROP-based DFS Architecture

Figure 4 is a distributed file system architecture based on DROP we design. There are two kinds of protocols: one is storage protocol between clients and storage nodes, and the other is management protocol between MDSs and storage nodes. *DFS server* is a distributed file system server daemon, such as *pnfsd* in pNFS. The component *DFS-DROP* provides interfaces between *DFS server* and DROP. Many *Back-end* nodes from MDSs as virtual nodes are organized into a DROP overlay network. Files and directories metadata items are published into the DROP network to be available to clients, like publish/subscribe in DHT networks [13], and they are maintained by the *Back-end* nodes.

3 Preserving Namespace Locality

In large-scale file systems, we can achieve near-optimal namespace locality by assigning keys that are consistent with the order of full pathnames.

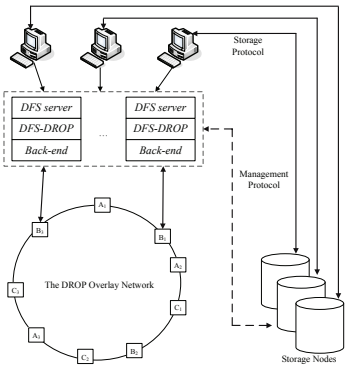


Fig. 4. DROP-based DFS Architecture

3.1 Traces Analyzed

There are three real traces we analyze as shown in Table 1. *Microsoft* means Microsoft Windows build server production traces [14] from BuildServer00 to BuildServer07 within 24 hours, and its data size is 223.7GB (including access pattern information). *Harvard* is a research and email NFS trace used by a large Harvard research group [15], and its data size is 158.6GB (including access pattern information). We implemented a metadata crawler that performs a recursive walk of the file system using *stat()* to extract file/directory metadata. By using the metadata crawler, the *Linux* trace is fetched from 22 Linux servers in our data center, and it is different from and much bigger than the *Linux* trace in [11]. Its file system metadata size is 4.53GB, and data size is 3.05TB. Based on the *Linux* trace, we perform two estimations: 1) storing 1 trillion files, the metadata size is 441TB and the data size is 290PB by computation, and 2) storing 1EB data, the metadata size is 1.56PB and the number of files is 3.53 trillion files by computation.

TABLE 1
Traces

Trace	# of files	Path metadata	Max. length
Microsoft	7,725,928	416M	34
Harvard	7,936,109	176M	18
Linux	10,271,066	786M	21

3.2 Locality-preserving Hashing

In order to achieve near-optimal locality, the entire directory tree nested under a point has to reside on the same MDS if there is not an explicit subtree assignment, e.g., */sys/fs* may be assigned to one MDS, while */sys/fs/fuse* may be assigned to another one. Pathnames are directly used with fixed-size keys, where every lookup message should contain a key as large as the longest path. To limit message overhead without modifying routing mechanisms, we employ a more compact key encoding in DROP as shown in Figure 5.

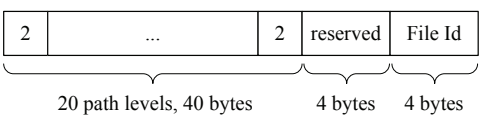


Fig. 5. Locality-preserving Hashing

The file path is encoded with the first 40 bytes, and each directory is encoded with 2 bytes. For longer paths, the next

4 bytes are reserved for the rest of path since 40 bytes are only sufficient for 20 path levels in terms of space. Although locality for files in longer paths will not be preserved, they make up 0.001%, 0.018% and 0.0% of the files in the *Linux*, *Microsoft* [14] and *Harvard* [15] traces, and there are an even smaller percentage of the requests based on the analysis of the last two traces. We plot the cumulative distribution function (CDF) of path length in the three analyzed traces in Figure 6, and we can see that the longer the path length is, the smaller the proportion is. The last 4 bytes are allocated for a file name, and they can represent 2^{32} files per directory in theory. Eventually, the 48-byte key enables up to many trillions files in count and many exabytes in size.

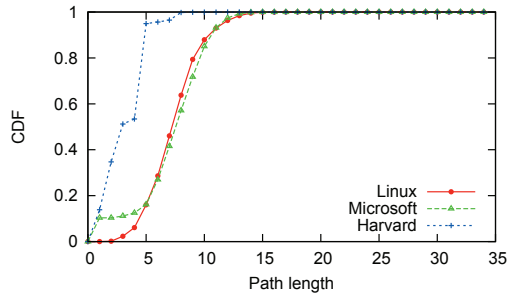


Fig. 6. CDF of path length in the three traces

The key encoding mechanism provides a good trade-off between key size and file count, and it enables naming of new files and directories. In addition, a file may be moved to a different directory, and its key can be quickly changed to reflect the new path using the encoding mechanism. Furthermore, related metadata items are organized into a group using it to preserve in-order traversal of file system, e.g., files in the same directory are related.

4 Dynamic Load Balancing

Since the key distribution is no longer uniform in DROP, load balancing is a great challenge that we have to face and address. We propose a simple but efficient dynamic load balancing algorithm to guarantee load balancing with losing negligible locality in metadata placement. The load balancing algorithm is simple, fully distributed, and converge quickly, which is similar to other dynamic load balancing methods [16], [17].

4.1 Metadata Histogram Maintenance

In order to achieve load balancing, a simple yet efficient metadata histogram maintenance mechanism is first proposed, and it is used by MDSs for maintaining histograms of metadata storage load. Its basic idea is to measure the range-density histogram locally, exchange these histograms throughout the system in a heartbeat protocol and determine if a MDS is overloaded or not. Let $\mathbb{N}$ denote the neighbor metadata server set of a MDS. Each MDS periodically samples MDSs in $\mathbb{N}$ and produces a local estimate of metadata items. Each of these MDSs reports back its local range-density. As time progresses, a metadata server builds a list of tuples of this form as shown in Definition 1.

Definition 1 (Range density). Range density is a five-tuple as follows: $\{virtual\_node, [min, max], load, timestamp\}$, where the $timestamp$ is used to age out old records.

If the DROP system needs to get an average histogram of node range-density, the collected range-densities can be used exactly as they are collected. When a MDS joins the cluster, it is required to have a duty for some ranges of locality-preserving hash values via its virtual nodes. However, files and directories in a particular range of values may exhibit a much greater popularity than other ranges, which would cause the MDS in charge of the popular range to become overloaded.

We leverage our metadata histogram maintenance mechanism to help implement load balancing in DROP. Firstly, each metadata server can get the average load $\overline{L}$ using histograms, thus determining if it is relatively heavily or lightly loaded in the system. Secondly, the histograms contain information about which parts of the DROP overlay are lightly loaded. Using this information, heavily loaded MDSs can send probes to lightly loaded ones. Once a probe encounters a lightly loaded MDS, it requests this lightly loaded MDS to gracefully take some virtual nodes from the heavily loaded MDSs in DROP, which effectively decreases the load of the heavily loaded MDSs. Note that this process is in parallel.

4.2 Histogram-based Dynamic Load Balancing

We present a simple but effective Histogram-based Dynamic Load Balancing (HDLB) approach. Each metadata server periodically contacts its neighbors in the system. The DROP system is said to be load-balanced when all the MDSs satisfy Definition 2, i.e., if the largest load is greater than t^2 times the smallest load, dynamic load balancing will be performed by reassigning virtual nodes from heavily loaded MDSs to lightly loaded MDSs.

Definition 2 (MDS_i is load balancing). MDS_i is load balancing if its load satisfies $1/t \leq L_i/\overline{L} \leq t$ ($t \leq 2$).

Given a set of m metadata servers $S = \{s_i, i = 1, \ldots, m\}$ and a set of n virtual nodes $V = \{v_j, j = 1, \ldots, n\}$, each virtual node v_j has a weight w_j that means how many files in a range are maintained by v_j, and each metadata server s_i has a remaining capacity (weight) W_i that means the difference between the average storage load (capacity) $\overline{W}$ and the existing weight in the metadata server s_i. The problem can be formulated as a 0-1 Multiple Knapsack Problem [18] (MKP) that is a NP-hard problem, i.e., it is to determine how to reassign n virtual nodes to m metadata servers in a way that minimizes the wasted space in the MDSs as follows:

$$\text{maximize} \qquad z = 1/\sum_{i=1}^{m} s_i \tag{1a}$$

$$\text{s.t.} \qquad \sum_{i=1}^{m} x_{ij} = 1, j \in N = \{1, \ldots, n\} \tag{1b}$$

$$\sum_{j=1}^{n} w_j x_{ij} + s_i = W_i y_i, i \in M = \{1, \ldots, m\} \tag{1c}$$

$$x_{ij} \in \{0, 1\}, y_i \in \{0, 1\}, i \in M, j \in N \tag{1d}$$

where

$$x_{ij} = \begin{cases} 1 & \text{if virtual node } j \text{ is reassigned to MDS } i \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{if MDS } i \text{ is used} \\ 0 & \text{otherwise} \end{cases}$$

$$s_i = \text{space left in MDS } i$$

Constraint (1b) makes sure that each virtual node is only assigned to a physical metadata server. Constraint (1c) ensures that the total number of files assigned to each metadata server is less than the capacity of metadata server. Constraint (1d) states it is a 0-1 knapsack problem.

We use $t = 2$ so that MDS loads differ by at most a factor of 4 in steady state. Each MDS stores both primary and secondary replicas, but only the primary replica count is exploited as the load value for the purpose of this approach. When primary load on all the MDSs is balanced, then total load, including both primary and secondary replicas, will be balanced as well. For example, there is a metadata server A, which has three neighbors B, C and D. They include virtual nodes as shown in Table 2, where the number means the load of a virtual node. There are a set of virtual nodes $V = \{3, 2, 7, 6, 2\}$ that will be reassigned to light MDSs $S = \{C, D\}$, which have the remaining capacities 11 and 12 respectively. After solving the 0-1 MKP, we can see that there is a rough load balancing from Table 2.

TABLE 2
Example load

MDS	items	removed items	results
A	{3, 2, 7, 12}	{3, 2, 7}	{12}
B	{15, 6, 2}	{6, 2}	{15}
C	{1}	$\emptyset$	{1, 2, 7, 2}
D	$\emptyset$	$\emptyset$	{3, 6}

4.3 Traffic Control

During load balancing, a metadata item may be moved multiple times. It often occurs when some files in a large directory are renamed since the directory initially is assigned to a single MDS with a high probability. DROP uses metadata pointers to minimize metadata migration overhead. For a metadata pointer, a MDS retrieves the metadata when it has held the pointer for longer than the stabilization time of the pointer. Using metadata pointers only temporarily hurts data locality when balancing the load. Besides reducing load balancing overhead, pointers also enable writes to succeed even when the target MDS is at capacity, pointers can be used to divert metadata items from heavily loaded MDSs to lightly loaded MDSs. However, the MDS at capacity will eventually shed some load when balancing the load, just causing temporary additional indirection. Suppose that a MDS *X* is heavily loaded, and a MDS *Y* takes some virtual nodes of *X* to take some of *X*'s load. Now *X* must transfer some of its metadata items to *Y*. Instead of having *X* immediately transfer some of its metadata items to *Y* when *Y* gets some virtual nodes from *X*, *Y* will initially maintain metadata pointers to *X* and transfer the pointers to *Z*. Finally, *Z* will retrieve the actual metadata from *X* and delete the pointers.

5 Metadata Consistency

In file systems, metadata consistency means that it must have pre-defined metadata integrity constraints, e.g., inode numbers are unique and no directory entry (*dentry*) points to a non-existent inode. File systems can maintain a consistent on-disk state by wrapping related operations in transactions.

5.1 MDS Interaction with ZooKeeper

DROP as a distributed key-value store holding file/directory metadata comes across a set of metadata servers. It allows for all standard filesystem metadata operations, such as file/directory creation, renaming and deletion. We proceed to analyze the usage of ZooKeeper in DROP. ZooKeeper is vitally important to maintain virtual node distribution information by interacting with MDSs in DROP. When a MDS joins the DROP network, it connects to the ZooKeeper cluster for status synchronization with its local memory. The ZooKeeper service starts as an initial procedure instead of a normal start-up one if it is not initialized. After comfirming the existence of ZooKeeper, the MDS starts its metadata service, which tries to register itself to the ZooKeeper cluster by creating a ephemeral znode under the *MDS* znode. Also, it starts a number of threads for its virtual nodes and store them locally. If the mapping information between a MDS and its virtual nodes in ZooKeeper changes, the local metadata service needs to contact ZooKeeper to change the value of a znode. When a MDS fails, there is no heartbeat signal between the MDS and the ZooKeeper cluster, which makes the ZooKeeper service aware of the MDS's failure. DROP does not need to do anything, and recovery task is automatically started when reading or writing data in the MDS.

5.2 Metadata Synchronization

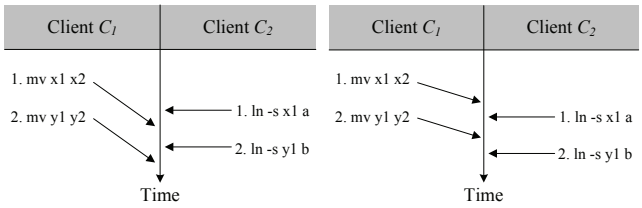
ZooKeeper provides node existence information for all virtual nodes, therefore its performance determines the performance of metadata management in DROP. ZooKeeper is much more suitable for reads than writes [10], so the MDSs in DROP mostly read the information from ZooKeeper instead of writing. There are two common situations that data in ZooKeeper is required to be modified. The first one is that the DROP cluster needs to create znodes in ZooKeeper where each znode represents a virtual node when it boots at the first time. Many creation operations take a long time when there are a large number of virtual nodes, but it only occurs once when the DROP cluster first starts up.

The second one is that DROP updates the information into ZooKeeper by setting znode's value whenever a MDS joins or leaves. Writes in ZooKeeper are much faster than the arrivals of new nodes, so it does not affect the performance of DROP. The read performance of ZooKeeper might be another potential bottleneck of DROP. To avoid this bottleneck, we use local cache in MDSs and ZooKeeper's watch mechanism. When cached data is invalid, i.e., "reject" or "timeout" is

returned, A MDS reads from ZooKeeper and updates its local cache. ZooKeeper uses a watch mechanism to enable MDSs to cache data without managing the local cache directly. With this mechanism, the MDS can watch for an update to a given metadata item, and receive a notification upon the update.

5.3 Linearizable Consistency

In DROP, we have to solve the consistency and availability problems, where requirements are from these conditions for metadata operations that modify either the set of virtual nodes or the key ranges among virtual nodes. Strict consistency relies on absolute global time, so it is impossible to be implemented in a distributed system. Linearizable consistency is weaker than strict consistency, but stronger than sequential consistency [19]. Operations in linearizable consistency are assumed to receive a timestamp with a global available clock that is loosely synchronized. Figure 7 illustrates the comparison of linearizable consistency and sequential consistency in distributed metadata management.



(a) Sequential Consistency (b) Linearizable Consistency

Fig. 7. Sequential Consistency vs. Linearizable Consistency. There is an error message in sequential consistency: "No such file or directory" for the links *a* and *b*.

In DROP, we structure virtual node updates as distributed transactions across nodes, which can provide a powerful framework for implementing the challenging multi-node operations such as renaming a large directory near the root. DROP implements distributed transactions across nodes using the ZooKeeper distributed consensus service. At a high level, nodes execute a two-phase commit (2PC) protocol before a node executes a step in the 2PC protocol and it uses ZooKeeper to replicate the decision for executing the step. Therefore, distributed replication plays a key role in the 2PC protocol for write-ahead logging to stable storage. The virtual node initiating a transaction is viewed as the coordinator and others are involved as the participants. We introduce a key concept: *Directory Metadata Group* as shown in Definition 3. There is an example in Figure 8 that shows the Zookeeper cluster is used to guarantee the structural integrity and consistency of metadata.

Definition 3 (Directory Metadata Group). Directory metadata group consists of one or more nodes for a given directory.

The overall structure of consensus is described as shown in Algorithm 1. The coordinator as the leader of a group initiates every action (line 1), and the number of messages is apparently reduced because broadcast and message batching are employed (lines 2,8). Concurrency is encouraged (lines 3-5, 6-7, 9-10), and the group continues to provide lookup

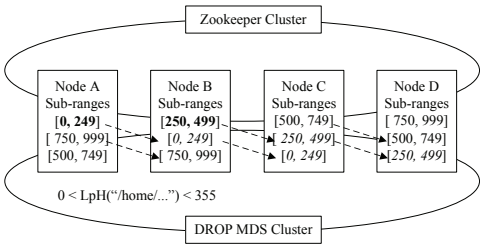


Fig. 8. Example of a DROP Cluster with ZooKeeper. For */home*, its directory metadata group includes A and B, its linearizable consistency comes across multiple node groups ({A, B}, {B, C}, {C, D}).

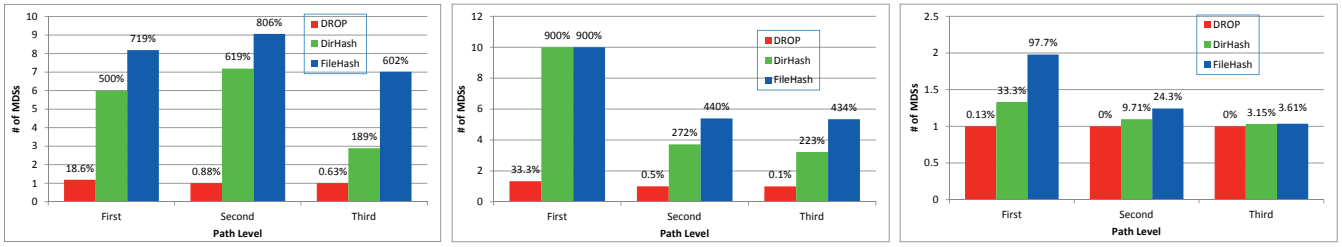
request services during transactions that change the the key-space partitioning. ZooKeeper is utilized to replicate the intermediate state needed for multi-node operations (lines 1,5,7). In order to improve throughput, the storage state of each group is partitioned among its group members. Each operation is forwarded to the node of the group assigned to the primary key. The group leader replicates information about the assignment of keys to primaries using ZooKeeper, as it handles the state for multi-node operations. Each primary node uses ZooKeeper to replicate operations on its key-range to all the other group members, which provides linearizable consistency. With ZooKeeper, a single message round is usually enough for replication, and it is unnecessary to synchronize operations on different primaries and keys.

Algorithm 1: Linearizable Consistency

Input: Coordinator N_c, Participants $\mathbb{P}$
1 N_c replicates the decision to initiate transaction T;
2 N_c broadcasts a *prepare* message m_p to $\mathbb{P}$;
3 **for** $p \in \mathbb{P}$ **do**
4 **if** p *receives* m_p **then**
5 p replicates its vote;
6 **if** N_c *receives the votes of* $\mathbb{P}$ **then**
7 N_c replicates its decision C;
8 N_c broadcasts the outcome of T to $\mathbb{P}$;
9 **if** C **then**
10 The steps of the transaction T is excuted;

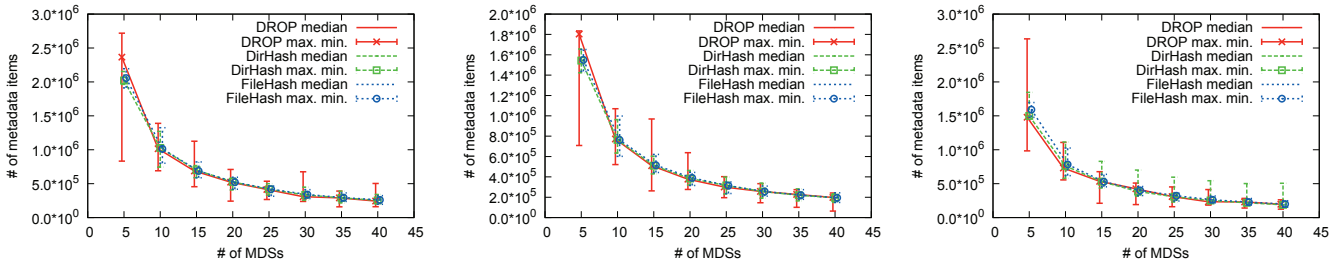
6 Performance Evaluation

In this section, we evaluate the performance of DROP using detailed trace-driven simulations. We have developed a detailed event-driven simulator to validate and evaluate our design decisions and choices. We first empirically evaluate the namespace locality effectiveness, and second measure the scalability of DROP. We evaluate DROP and compare its performance with other distributed metadata management schemes: 1) Subtree that is to manually partition directories and assign each subtree to a metadata server; 2) FileHash that is to randomly distribute files according to their pathnames, each of which is assigned to a metadata server; 3) DirHash that is to randomly distribute directories like FileHash. A virtual node's identifier is a 384-bit key obtained from the SHA-384



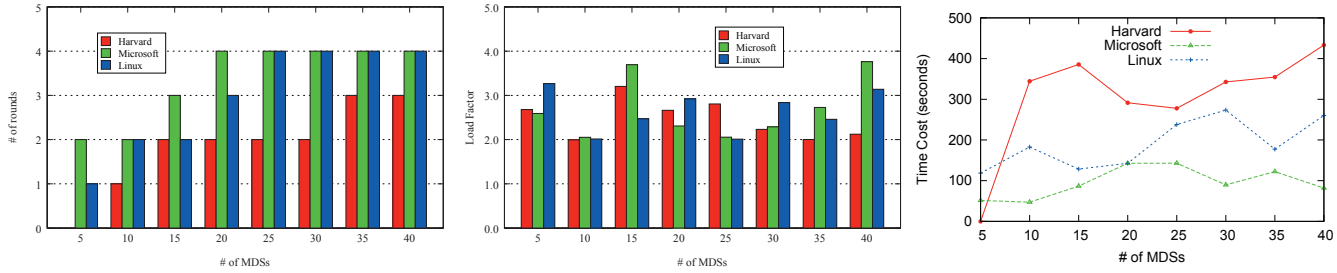
(a) Linux trace (b) Microsoft Windows trace (c) Harvard trace

Fig. 9. Locality Comparisons of Three Level Paths over Different Traces in the Cluster of Ten Metadata Servers



(a) Linux trace (b) Microsoft Windows trace (c) Harvard trace

Fig. 10. Load Distribution with Varying the Number of Metadata Servers



(a) Convergence Rate (b) Load Balancing (c) Time Cost

Fig. 11. Performance with Varying the Number of Metadata Servers

hash function. We demonstrate the effectiveness, performance and scalability over different MDS cluster sizes.

6.1 Namespace Locality

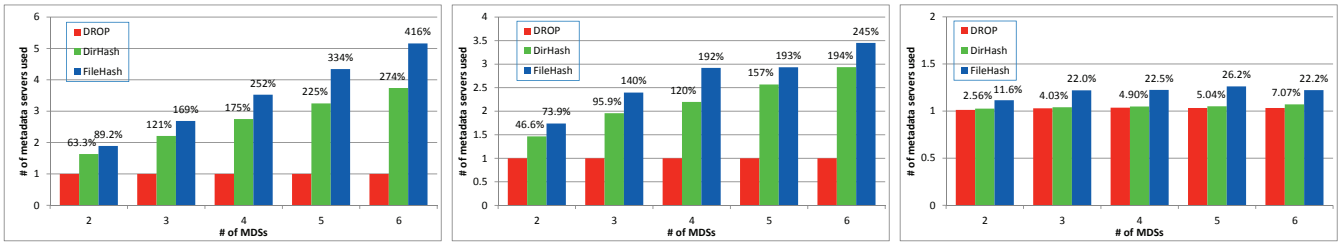
Locality-preserving hashing in DROP is clearly a suboptimal strategy to keep excellent namespace locality. Namespace locality is very important to large-scale distributed metadata management, and it is utilized to improve the performance of MDS cluster by reducing I/O requests. It can be measured: $locality = \sum_{j=1}^{m} p_{ij}$, where p_{ij} (0 or 1) represents if a subtree path p_i ($\in \mathbb{P}$) is located in MDS j. The metric represents how many metadata servers the path $\mathbb{P}$ is split across. Figure 9 presents namespace locality comparisons of three level paths on the three traces using three different distributed metadata management mechanisms. Note that we do not plot the results of static subtree partitioning since each path is maintained by only one metadata server according to its definition, but its load imbalance is severe.

Figure 9(a), Figure 9(b) and Figure 9(c) illustrate that DROP has much better namespace locality than DirHash and FileHash for the three traces. The percentage above a box is calculated as follows: $\frac{N-S}{S} \times 100\%$, where S is the number of MDSs using Subtree (S=1), N is the number of MDSs using one of other three approaches. DROP performs only negligibly worse than static subtree partitioning except the first level paths in both the *Linux* trace and the *Microsoft* Windows trace. The reason for this is that DROP can achieve suboptimal namespace locality using locality-preserving hashing, i.e., assigning keys that are consistent with the order of pathnames. For DirHash and FileHash, the order of pathnames is not considered so that namespace locality is lost thoroughly.

6.2 Scalability

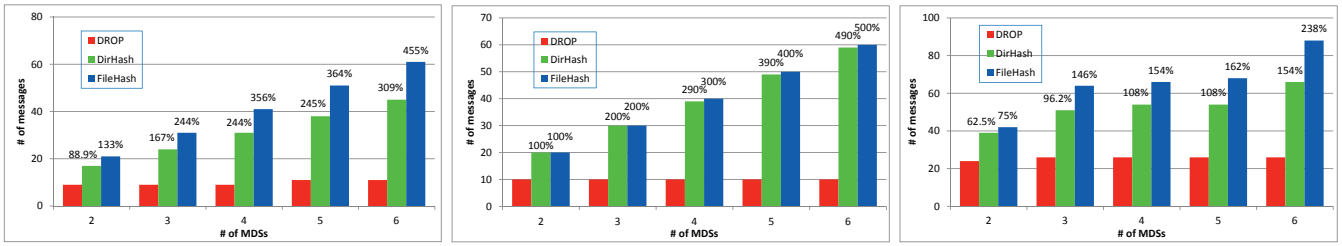
We first plot scalable load distribution with different sizes of MDS cluster as shown in Figure 10. We exploit three metrics: median load, maximum load and minimum load. Note that we use median load instead of average load, and we do not give the results of static subtree partitioning because its scalability is very bad in load distribution. From Figure 10(a), Figure 10(b) and Figure 10(c), we can see that DROP has somewhat worse load distribution than DirHash and FileHash when the MDS cluster size is small, while it has similar load distribution as DirHash and FileHash when the MDS cluster size is big enough. Therefore, DROP has as excellent scalability as DirHash and FileHash in load distribution.

The primary overhead of DROP's performance gains comes from active load balancing. We second evaluate the relative performance and scalability of DROP by scaling the number



(a) Linux trace (b) Microsoft Windows trace (c) Harvard trace

Fig. 12. Namespace Locality with Varying the Number of Metadata Servers



(a) Linux trace (b) Microsoft Windows trace (c) Harvard trace

Fig. 13. Metadata Request Communication Overhead with Varying the Number of Metadata Servers

of MDSs. Figure 11 presents the relative performance with varying the number of MDSs. Figure 11(a) demonstrates that histogram-based dynamic load balancing (HDLB) has excellent convergence rate. For the given three traces, it reaches a satisfactory load balancing state within four rounds even as the number of MDSs is 40. Therefore, HDLB can quickly converge to load balancing in fully distributed systems.

Figure 11(b) shows that HDLB has excellent load balancing performance with different MDS cluster sizes. Figure 11(c) illustrates that HDLB has excellent efficiency with different numbers of metadata servers. The histogram-based dynamic load balancing mechanism can efficiently assign and migrate loads among metadata servers. A large faction of loads are reassigned and migrated first within the same group and lastly within the entire network via hub MDSs, thus enabling fast and efficient load balancing.

7 Prototype Implementation and Evaluation

In order not to reinvent the wheel, we have reused and extended FAWN [20] as the MDS cluster by introducing DROP on top of it. We first evaluate scalable namespace locality, and second present metadata request overhead comparison when renaming directories. We evaluate DROP and compare its performance with *FileHash* and *DirHash*. Each MDS with twenty virtual nodes is deployed in an m1.large instance with 4 cores and 7.5GB memory in Amazon Singapore.

7.1 Scalable Namespace Locality

Figure 12 presents scalability comparisons of namespace locality on three level paths in the three traces. Figure 12(a), Figure 12(b) and Figure 12(c) illustrate that DROP has much better scalability of namespace locality than DirHash and FileHash for the given three traces. The percentage above a box is calculated as follows: $\frac{N-D}{D} \times 100\%$, where D and N are the number of MDSs using DROP and FileHash/DirHash respectively. DROP performs increasingly better than the other two distributed metadata management schemes when the number of metadata servers rises. The reason for this is that DROP can achieve suboptimal namespace locality using locality-preserving hashing, i.e., assigning keys that are consistent with the order of pathnames, which is not taken into account in both DirHash and FileHash so that namespace locality is eliminated thoroughly. Note that DROP has somewhat better locality on the first-level paths in *Microsoft* than *Linux* and *Harvard*, while it has much worse locality on the non-first-level paths in *Microsoft* than *Linux* because of their different path naming mechanisms, so DROP has better scalability of namespace in *Linux* than *Microsoft*, as shown in Figure 12(a) and Figure 12(b).

7.2 Scalable Metadata Request

We measure metadata request overhead when renaming directories, which may come across multiple metadata servers using *get* and *put* operations. Packaging *get* or *put* operations into one message is a normal choice when renaming some file of a directory, which are maintained by one MDS. We randomly select sample directories from the three traces, and rename them. Figure 13 illustrates the metadata request overhead with excellent scalability when guaranteeing strong metadata consistency. Figure 13(a) and Figure 13(c) show that they have similar metadata request overhead because the *Linux* and *Harvard* traces are based on Linux namespace scheme, while Figure 13(b) illustrates that DROP has much better performance than DirHash and FileHash because there are fewer first-level paths in *Microsoft* than the other two traces.

8 Related Work

Distributed metadata management is the foundation of EB-scale file systems, which excellently support cloud-scale storage and backup, and even cloud-scale data management [21].

8.1 Metadata Server Cluster Scale

Single Metadata Server. The architecture of a single MDS vastly simplifies the design and enables the MDS to make data placement and replication decisions relatively easy, but there is a bottleneck in the single MDS, causing the single-point failure. Distributed file systems, e.g., Coda [22], partition their namespace statically among multiple storage servers, so most of the metadata operations are centralized. Other distributed file systems, e.g., GFS [23], have a single MDS, with a fail-over MDS that becomes operational if the primary server becomes unavailable. For example, file system metadata and application data are stored separately in GFS. File system metadata is stored on a dedicated server called *master*, while application data is stored on data servers called *chunkservers*. Only one MDS is operational at a given point in time, which is obviously a potential bottleneck as the number of clients and/or files increases.

Multiple Metadata Servers. Metadata server cluster can expand or contract, and it can rebalance the file system dynamically to distribute data evenly among MDSs. This ensures high performance and prevents heavy loads on specific MDSs within the cluster. Several distributed file systems have or are exploring truly distributed implementations of the single global namespace. Ceph [24] has a cluster of metadata servers and uses a dynamic subtree partitioning algorithm [3] to map the namespace tree to MDSs evenly. GFS [23] is also evolving into a distributed namespace implementation [25]. The new GFS will have hundreds of metadata servers with 100 million files per master. Lustre [26] has an implementation of clustered namespace on its roadmap for Lustre 2.2 release. The purpose is to stripe a directory over multiple metadata servers, each of which contains a disjoint portion of the namespace.

8.2 Metadata Organization

Subtree partitioning and hash-based mapping are two common techniques used for MDS cluster in EB-scale file systems, while Bloom-filter-based approaches [27], [28] provide probabilistic metadata lookups instead of metadata updates.

Hash-based mapping. Hash-based mapping [29], [30] applies hash function to a pathname or filename of a file to locate the file's metadata. It helps clients to locate and contact directly to the right metadata server. Client requests can be distributed evenly among a metadata server cluster, eliminating hot-spots consisting of popular directories. Vesta [29] and zFS [30] leverage pathname hashing to locate metadata. Hashing provides a better load balancing across metadata servers and gets rid of hot-spots e.g., popular directories. However, hashing is a random distribution, in which metadata updates may incur huge network overhead, e.g., the metadata of many files has to be migrated among MDS cluster after renaming a directory. In order to verify user access permissions, it results in high overhead from prefix directories cache or path traversal as the accessed files and their prefix directories are located on different MDSs. Furthermore, it eliminates the hierarchical locality and many benefits brought by the hierarchical locality. Lazy Hybrid [2] based on hashing exploits lazy update policies to defer and distribute update cost to address the update issue of metadata.

Subtree partitioning. Static subtree partitioning [31] provides a simple approach of distributing metadata operations among MDS cluster, which statically partitions the directory hierarchy and assigns each subtree to a particular MDS. It provides better locality of reference and greater MDS independence than hash-based mapping. Its major drawback is that the workload may not be evenly partitioned among MDS cluster, suffering from a system performance bottleneck. In order to adjust load imbalance, migrating subtrees is necessary in some cases (e.g., PanFS [31]). Static partitions fail to adapt to the growth or contraction of individual subtrees over time, often requiring intervention of system administrators to repartition or manually rebalance metadata storage load across MDSs. Dynamic subtree partitioning [3] uses dynamic load balancing mechanism to redistribute metadata dynamically among MDS cluster to handle the changing workload.

8.3 Dynamic Load Balancing

Many leave-join based load balancing mechanisms have been proposed concurrently in [16], [17], [32] and [33]. Mercury [16] works even when there are skewed node ranges since it utilizes an effective random sampling approach. The load balancing mechanisms in [17] do not cope with skewed node range distributions, and they dynamically balance load among servers without using multiple virtual nodes by reassigning lightly loaded servers to be neighbors of heavily loaded servers. However, it is not clear whether their approaches would be efficient in practice although they prove bounds on maximum node utilization and load movement. One-to-Many and Many-to-Many are extended to dynamic structured P2P systems [34], where One-to-Many is used for emergency load balancing of one particularly overloaded node, while Many-to-Many is used for periodic load balancing of all the nodes [33].

9 Conclusions

In this paper, we present DROP, an efficient and scalable distributed metadata management architecture to serve EB-scale file systems. In order to keep excellent namespace locality, DROP exploits locality-preserving hashing to distribute metadata among MDSs. When storage load changes dynamically, it introduces the HDLB strategy to quickly adjust the metadata distribution. After the adjustment, DROP ensures that the namespace locality maintained by MDSs is still good. Besides, DROP can balance the metadata storage load as good as static hash-based mapping. When the size of the MDS cluster changes, DROP uses the HDLB strategy to move the minimal metadata to maintain the storage load balancing. It keeps excellent consistency of metadata replicas as well. Compared to other distributed metadata management techniques, DROP brings multiple advantages, such as balancing metadata storage load efficiently, high scalability and no bottlenecks with negligible additional overhead.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments that help improve this paper. This work is supported by A*STAR Thematic Strategic Research Programme (TSRP) Grant No. 1121720013.

REFERENCES

[1] I. Raicu, I. T. Foster, and P. Beckman, "Making a case for distributed file systems at exascale," in *LSAP*, 2011, pp. 11–18.
[2] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue, "Efficient metadata management in large distributed storage systems," in *IEEE Symposium on Mass Storage Systems*, 2003, pp. 290–298.
[3] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *SC*, 2004, p. 4.
[4] J. Xiong, Y. Hu, G. Li, R. Tang, and Z. Fan, "Metadata distribution and consistency techniques for large-scale cluster file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 5, pp. 803–816, 2011.
[5] S. Patil and G. A. Gibson, "Scale and concurrency of giga+: File system directories with millions of files," in *FAST*, 2011, pp. 177–190.
[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *OSDI*, 2010, pp. 47–60.
[7] E. L. Miller, K. Greenan, A. Leung, D. Long, and A. Wildani. (2008) Reliable and efficient metadata storage and indexing using nvram. [Online]. Available: dcslab.hanyang.ac.kr/nvramos08/EthanMiller.pdf
[8] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. D. Kupfer, and J. G. Thompson, "A trace-driven analysis of the unix 4.2 bsd file system," in *SOSP*, 1985, pp. 15–24.
[9] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *FAST*, 2008, pp. 17–33.
[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX Annual Technical Conference*, 2010.
[11] Q. Xu, R. V. Arumugam, K. L. Yang, and S. Mahadevan, "Drop: Facilitating distributed metadata management in eb-scale storage systems," in *MSST*, 2013, pp. 1–10.
[12] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, 2001, pp. 149–160.
[13] Q. Xu, X. Hou, B. Cui, H. T. Shen, and Y. Dai, "Facilitating effective resource publishing and searching in dht networks," *HKIE Transactions*, vol. 16, no. 3, pp. 32–41, 2009.
[14] S. Kavalanekar, B. L. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *IISWC*, 2008, pp. 119–128.
[15] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer, "Passive nfs tracing of email and research workloads," in *FAST*, 2003.
[16] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," in *SIGCOMM*, 2004, pp. 353–366.
[17] D. R. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in *SPAA*, 2004, pp. 36–43.
[18] C. Chekuri and S. Khanna, "A polynomial time approximation scheme for the multiple knapsack problem," *SIAM J. Comput.*, vol. 35, no. 3, pp. 713–728, 2005.
[19] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
[20] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *SOSP*, 2009, pp. 1–14.
[21] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "Es^2: A cloud data storage system for supporting both oltp and olap," in *ICDE*, 2011, pp. 291–302.
[22] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Trans. Computers*, vol. 39, no. 4, pp. 447–459, 1990.
[23] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*, 2003, pp. 29–43.
[24] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI*, 2006, pp. 307–320.
[25] M. K. McKusick and S. Quinlan, "Gfs: Evolution on fast-forward," *ACM Queue*, vol. 7, no. 7, p. 10, 2009.
[26] Lustre file system. [Online]. Available: http://www.lustre.org
[27] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "Hba: Distributed metadata management for large cluster-based storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 6, pp. 750–763, 2008.
[28] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Supporting scalable and adaptive metadata management in ultralarge-scale file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 580–593, 2011.
[29] P. F. Corbett and D. G. Feitelson, "The vesta parallel file system," *ACM Trans. Comput. Syst.*, vol. 14, no. 3, pp. 225–264, 1996.
[30] O. Rodeh and A. Teperman, "zfs - a scalable distributed file system using object disks," in *IEEE Symposium on Mass Storage Systems*, 2003, pp. 207–218.
[31] D. Nagle, D. Serenyi, and A. Matthews, "The panasas activescale storage cluster - delivering scalable high bandwidth storage," in *SC*, 2004, p. 53.
[32] J. Pang, P. B. Gibbons, M. Kaminsky, S. Seshan, and H. Yu, "Defragmenting dht-based distributed file systems," in *ICDCS*, 2007, p. 14.
[33] A. Rao, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica, "Load balancing in structured p2p systems," in *IPTPS*, 2003, pp. 68–79.
[34] B. Godfrey, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica, "Load balancing in dynamic structured p2p systems," in *INFOCOM*, 2004.

Quanqing Xu received the PhD degree in computer science from Peking University, Beijing, China. He obtained the award of Excellent PhD Graduate because of his excellent performance at Peking University. He is a Research Scientist of the Data Storage Institute (DSI), a research institute under the Agency for Science, Technology and Research (A*STAR), Singapore. He is leading research and development in the area of distributed metadata management of next generation large scale storage systems. His research interests mainly include distributed systems, P2P computing and cloud storage. He is a member of the IEEE and ACM.

Rajesh Arumugam is a Senior Researcher at the Data Storage Institute (DSI), a research institute under the Agency of Science, technology and research (A*STAR), Singapore. He is currently the Program Leader for the large scale hybrid storage system research program. He is leading a team of researchers/systems engineers in research, design and development of next generation Petabyte/Exabyte scale storage systems for next generation data centers. Rajesh holds a Master's degree in Electronics and Communication Engineering from Anna University, India. Currently, he is also a part-time PhD student in the School of Computer Engineering at Nanyang Technological University, Singapore.

Yong Khai Leong is a Division Manager of the Data Storage Institute (DSI), a research institute under the Agency for Science, Technology and Research (A*STAR), Singapore. In his role with DSI, Khai Leong leads a team of research scientists and engineers in developing data and storage technologies for next generation data centers. Khai Leong obtained his Electrical & Electronics Engineering degree from the National University of Singapore and hold a postgraduate degree in Communication Software and Networks.

Sridhar Mahadevan received his masters degree from National University of Singapore. He is a research engineer at the Data Storage Institute, a research institute under the Agency for Science Technology and Research (A*STAR) Singapore. He is part of the research and development in the area of distributed metadata management of next generation large scale storage systems. His research interests are in distributed file system.