Can a Decentralized Metadata Service Layer benefit Parallel Filesystems?

Vilobh Meshram, Xavier Besseron, Xiangyong Ouyang, Raghunath Rajachandrasekar,
Ravi Prakash Darbha and Dhabaleswar K. Panda
Department of Computer Science and Engineering, The Ohio State University
{meshram,besseron,ouyangx,rajachan,darbha,panda}@cse.ohio-state.edu

***Abstract*—The demand for scalable I/O continues to grow rapidly as computer clusters keep growing. Much of the research in storage systems has been focused on improving the scale and performance of I/O throughput. Scalable file systems do a good job of scaling large file access bandwidth by striping or sharing I/O resources across many servers or disks. However, the same cannot be said about scaling file metadata operation rates.**

Most existing parallel filesystems choose to concentrate all the metadata processing load on a single server. This centralized processing can guarantee the correctness, but it severely hampers scalability. This downside is becoming more and more unacceptable as metadata throughput is critical for large scale applications. Distributing metadata processing load is critical to improve metadata scalability when handling huge number of client nodes. However, a solution to speed up metadata operations has to address two challenges simultaneously, namely the scalability and reliability.

In this paper, we have designed a decentralized metadata service layer and evaluated its benefits and shortcomings that concern parallel filesystems. The main aim of this service layer is to maintain reliability and consistency in a distributed metadata environment. At the same time we also focus on improving the scalability of the metadata operations, and in turn, the scalability of the underlying parallel filesystem.

As demonstrated by experiments, the approach presented in this paper achieves significant improvements over native parallel filesystems by large margin for all the major metadata operations. With 256 client processes, our decentralized metadata service outperforms Lustre and PVFS2 by a factor of 1.9 and 23, respectively, to create directories. With respect to *stat()* operation on files, our approach is 1.3 and 3.0 times faster than Lustre and PVFS.

I. Introduction

Filesystem sizes grow exponentially as computing clusters advance into the exascale computing era. Data can be stored in several different forms and types, with Metadata (data about data) being the most universally accessed data type. Every time a file is opened, saved, closed, searched, backed up or replicated, some portion of metadata is accessed. As a result, metadata operations fall in the critical path of a broad spectrum of applications.

Modern distributed filesystems such as Lustre [1], PVFS [2], Google File System [3], separate metadata management from the actual storage of file data. This kind of architecture have proven to easily scale the storage capacity and bandwidth. However, the management of metadata remains a bottleneck. Studies [4], [5] show that over 75% of all filesystem calls require access to file metadata. Therefore, efficient management of metadata is crucial for the overall system performance.

The Lustre filesystem architecture has a single Metadata Server (MDS), which means that Lustre metadata operations can be processed only as quickly as what a single server and its backing filesystem can manage. To date, this has not been a critical limitation. However, as the number of Lustre clients grows, a single MDS becomes a performance bottleneck and constraints the throughput of the filesystem.

The Lustre community has proposed the concept of Clustered Metadata Server (CMD) [6] which is still in its early stages of design, and not production-ready yet. The CMD architecture allows for multiple active metadata servers which can share the metadata processing workload, in a single Lustre configuration. Clustered MDS design has some notable shortcomings. For instance, one metadata operation may need to update several different MDSs. To maintain the consistency of the filesystem, this update must be atomic. If the update on one MDS fails, all other servers must be rolled back to their original states. In order to take care of this problem, a global lock has to be in place to synchronize the updates and to maintain consistency [6]. This might hurt the throughput of metadata operations.

In this paper, we have designed a Decentralized Metadata Service Layer and evaluated its pros and cons in a Parallel File System environment. The proposed Service Layer, namely DUFS, is composed of a FUSE-based filesystem and a Distributed Coordination Service. Our goal is to handle a large amount of metadata operations per second while providing a guarantee for consistency and reliability. Specifically we want to answer several questions:

1) Can a distributed coordination service be incorporated into parallel filesystems to scale up the metadata processing throughput?
2) What will be the performance impact of such a decentralized metadata service layer compared to native parallel filesystems such as Lustre [1] and PVFS [2]?
3) Will such a decentralized metadata service excel in maintaining the consistency and reliability of the file system?

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CCF-0621484, #CCF-0833169, #CCF-0916302, #OCI-0926691 and #CCF-0937842; grant from Wright Center for Innovation #WCI04-010-OSU-0;

The paper is organized as follows. Section II presents some background on components in our design and evaluation. Section III provides motivation for this work and our design choices. In Section IV, we describe the design of our prototype. In section V, we present our experiments and evaluation. Related work is discussed in Section VI, and in section VII, we present the conclusion and future work.

II. Background

A. Lustre File System

Lustre is a POSIX-compliant, stateful, object-based parallel file system. It provides fine-grained parallel file services using distributed lock management. Lustre separates essential file system activities into three components: clients, metadata servers and storage servers. These three components are referred to as Object Storage Client (OSC), Meta-Data Server (MDS) and Object Storage Server (OSS), respectively. To access a file, a client first obtains its metadata from the primary MDS, including file attributes, file permissions and the layout of file objects termed as Extended Attributes (EA) in Lustre context. Subsequent file IO (storage) operations are performed directly between the client and the OSS. By decoupling metadata operations from IO operations, data IO can be carried out in a parallel fashion, which provides greater aggregated bandwidth.

B. Filesystem in Userspace (FUSE)

Filesystem in Userspace (FUSE) [7] is a software that allows to create a virtual filesystem in the user level. It relies on a kernel module to perform privileged operations at the kernel level, and provides a userspace library that ease communication with this kernel module. FUSE is widely used to create filesystems that do not really store the data itself but relies on other resources to effectively store the data. Then, a FUSE virtual filesystem is like a way to present and organize data to users through the classic filesystem interface.

C. ZooKeeper

ZooKeeper [8] is a distributed, open-source coordination service for distributed applications. It exposes a simple set of interfaces that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance and naming.

ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace which is organized similarly to a standard file system. The namespace consist of special nodes known as Znodes. Znodes do not store data but they store configuration information.

ZooKeeper can use multiple servers that replicate the whole namespace. The ZooKeeper coordination algorithms take care of maintaining the consistency between all the servers. Thus, all modifications on the namespace appear to be atomic and strictly ordered to all the clients [8].

III. Motivation

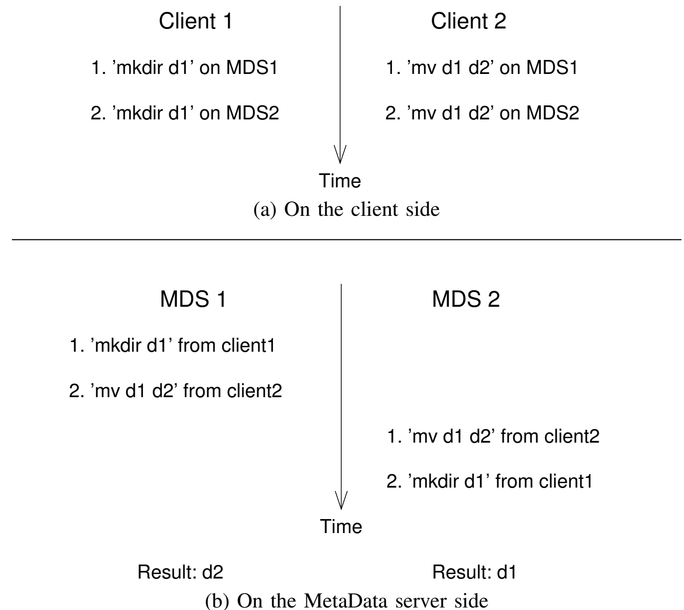
Parallel file systems can easily scale bandwidth and improve performance by operating on data in parallel using strategies such as data striping, sharing resources, etc. However, most parallel file systems do not provide the ability to scale and parallelize metadata operations as it is inherently more complex than scaling the performance of data operations [9]. PVFS provides some level of parallelism through distributed metadata servers that manage different ranges of metadata. The Lustre community has also proposed the idea of Clustered Metadata Server (CMD) to minimize the load on a single Metadata Server, wherein multiple metadata servers share the metadata processing workload.

A. Metadata Server (MDS) Bottlenecks

Most of the parallel file systems have a single MDS, with a fail-over MDS that becomes operational if the primary server becomes nonfunctional. Only one MDS is operational at a given point in time. This limitation poses a potential bottleneck as the number of clients and/or files increases. Often dismissed as negligible cost the metadata load can be a major bottleneck in the overall performance and scalability of the parallel file system [4], [10], [11].

B. Consistency management of Metadata

Majority of the distributed filesystems use a single metadata server. However, this is a bottleneck that limits the operation throughput. Managing multiple metadata servers brings many difficulties. Maintaining consistency between two copies of the same directory hierarchy is not straightforward. We illustrate such a difficulty in Figure 1.



(a) On the client side

(b) On the MetaData server side

Fig. 1. Example of consistency issue with 2 clients and 2 metadata servers. Client 1 creates the directory d1 and client 2 renames d1 to d2. If the two clients update the two metadata servers independently without any coordination, the resulting state may be inconsistent.

We have two metadata servers (MDS) and we consider two clients that perform an operation on the same directory at the same time. Client 1 creates the directory d1 and client 2 renames the directory d1 to d2. As shown in Figure 1a, each client performs its operation in the following order: first on the MDS1, then on the MDS2. From the MDS point of view, there is no guarantee on the execution order of the requests since they are coming from different clients. As shown in Figure 1b, the requests can be executed in a different order on each metadata server while still respecting the ordering that the clients demand. In this case, the resulting states of the two metadata servers are not consistent.

This small example highlights that distributed algorithms are required to maintain the consistency between multiple metadata servers. Each client operation must appear to be atomic and must be applied in the same order on all the metadata servers. For this reason, we decided to use a distributed coordination service like ZooKeeper in the proposed metadata service layer. Such a coordination service implements the required distributed algorithms in a reliable manner.

IV. Design and Implementation

A. Principle

The core principle of Distributed Union FileSystem (DUFS) is to distribute the load of the metadata operations across multiple distributed filesystems. DUFS provides a single POSIX-compliant filesystem abstraction to the user, without revealing the multiple underlying filesystem mounts. With such an abstraction, the single metadata server of the back-end distributed filesystem is not a bottleneck anymore. However, as described in section III, consistency has to be guaranteed across multiple clients which perform simultaneous metadata operations. This task is delegated to the distributed coordination service ZooKeeper [8].

DUFS maps each virtual filename, as seen by the user, to a physical path corresponding to one of the underlying filesystem mounts. A single-level indirection is introduced with the use of a File Identifier (FID), which uniquely identifies each file.

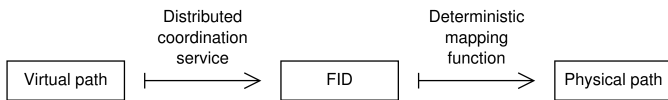


Fig. 2. DUFS mapping from the virtual path to the physical path using File Identifier (FID)

Figure 2 shows a schematic view of this indirection level in our design. The mapping between the FID and the physical path is carried out using a universally-known deterministic mapping function which every DUFS client is aware of. This mapping information is cached by ZooKeeper in a consistent manner. The second mapping step does not require any coordination between clients. Consistency management at the physical storage level is offloaded to the underlying filesystem.

This single-level indirection offers flexibility and allows to represent the contents of a file independently of its name. Indeed, a filename can represent two different data contents (after deletion and a new creation with the same name); and conversely, the data contents can correspond to any filename (for instance, a rename operation). This representation also makes rename operations and physical data relocation easier.

Finally, directories and directory-trees are considered as metadata only, so they are not physically created on the back-end storage. Instead, the directory-tree information is maintained in-memory by ZooKeeper.

B. Implementation Overview

The design of DUFS is broken down into three main components: the filesystem interface based on FUSE, the Metadata management based on ZooKeeper and the back-end storage provided by the underlying parallel filesystem. A DUFS client instance is only a local software that does not interact directly with other DUFS clients. Any necessary interaction is only made through ZooKeeper service or over the back-end storage.

Figure 3 shows the basic steps required to perform an *open()* operation on a file using DUFS.

A. The *open()* call is intercepted by FUSE which gives the virtual path of the file to DUFS.
B. DUFS queries ZooKeeper to get the Znode based on the filename and to retrieve the FID. If the file does not exist, ZooKeeper will return an error.
C. DUFS uses the deterministic mapping function to find the physical path associated to the FID.
D. Finally, DUFS opens the file based on its physical path. The result is returned to the application via FUSE.

Alternatively, directory operations take place only at the metadata level, so only ZooKeeper is involved and not the back-end storage. For example, the directory *stat()* operation is satisfied at the Zookeeper level itself since we maintain the entire directory hierarchy in Zookeeper and the back-end storage are not contacted. Along with the standard metadata attributes provided by Zookeeper, we store additional information in the data field provided for each Znode. Thus, only steps A and B are performed for directory specific operations.

The following subsections describe the functions of the primary elements comprised within DUFS.

C. FUSE-based Filesystem Interface

We use FUSE to provide a POSIX-compliant filesystem interface to the applications. Thus, our DUFS prototype appears like a classic mount-point of the standard filesystem.

In our DUFS prototype, we have implemented most of the basic file system operations like *mkdir*, *create*, *open*, *symlink*, *rename*, *stat*, *readdir*, *rmdir*, *unlink*, *truncate*, *chmod*, *access*, *read*, *write*. When an application wants to perform a filesystem operation, it will operate on the virtual path exposed to it by DUFS. The filesystem operations are translated into the FUSE specific operations, for example the *open()* call from application is translated into the *dufs_open()* in DUFS.

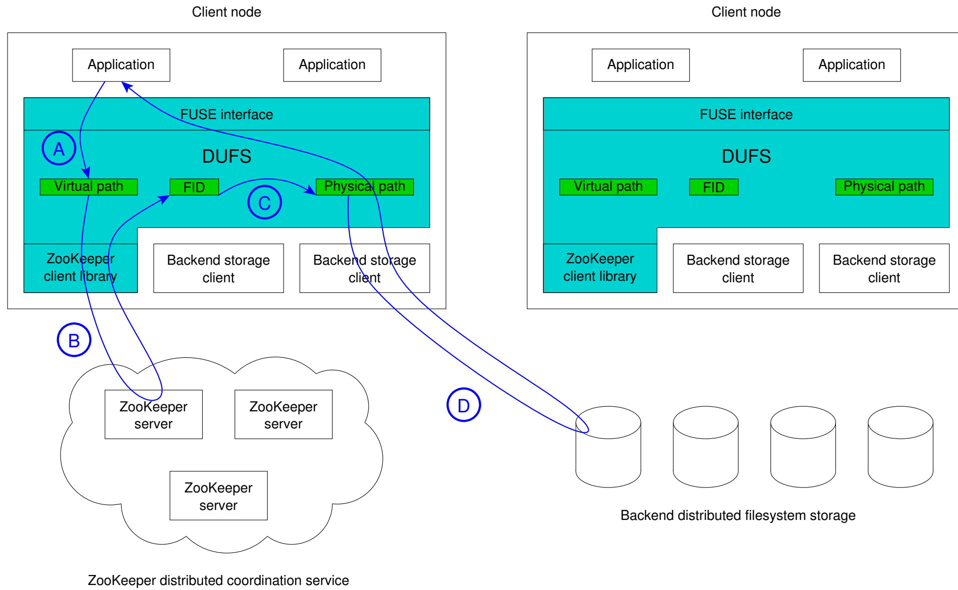


Fig. 3. DUFS overview. A, B, C and D show the steps required to perform an *open()* operation.

Finally, for each filesystem operation, DUFS can return the correct result after querying the ZooKeeper-based Metadata management service and the back-end storage as needed.

D. ZooKeeper-based Metadata Management

We use the ZooKeeper distributed coordination service to handle the consistency threats posed by the distributed accesses from several DUFS clients simultaneously. The synchronous ZooKeeper API were used for this purpose.

With our design, ZooKeeper will store a part of the virtual filesystem metadata. It keeps track of details of the directories and files that get created. A separate Znode is created in ZooKeeper for each directory or files created, and the virtual filesystem hierarchy is represented inside ZooKeeper using Znodes.

ZooKeeper has several information fields associated to each Znodes. Some of the standard fields include Znode creation time, list of children Znodes, etc. ZooKeeper also has the provision to add a custom data field to each Znode. In DUFS, this custom field is used to tell the Znode if it is representing a directory or a file. In the latter case, the FID of the file is also stored in this field.

As a consequence, all the metadata regarding a directory are stored in ZooKeeper, using the standard Znode fields and the custom data field. Regarding files, ZooKeeper only keeps tracks of existing filenames. The file metadata are maintained with the physical file on the back-end storage. Thanks to this, the access and modification times can transparently be updated when the physical file is accessed by the *read()* and *write()* operations after redirection by DUFS.

The ZooKeeper architecture uses multiple ZooKeeper servers. The data is replicated among all the servers. ZooKeeper uses coordination algorithms to ensure that the Znode hierarchy and their contents are consistent across the servers and that all the modifications are applied in the same order in all the servers [8].

All these information are kept in memory and ZooKeeper servers can be located close to DUFS clients. Thanks to this, ZooKeeper queries are fast and a large operation throughput can be performed. This raw throughput is studied in section V-A. However, the counterpart is that the ZooKeeper servers use a large amount of memory. We study this impact in memory usage in section V-E.

E. File Identifier

In our design, we use a File Identifier (FID) to uniquely represent the physical contents of a file. This FID is stored in the custom data field of the Znode corresponding to the virtual path of a file. The FID is designed to be unique for each newly created file. However, modifications to the file contents do not require changing the FID.

In DUFS, the FID is a 128-bit integer. We propose a simple approach to generate a unique FID at the DUFS client without requiring any coordination. The FID for a file is generated by the client who initially creates the file. It is a concatenation of a 64-bit client ID that uniquely represents that instance of DUFS client that created the file and a 64-bit file creation counter that records the number of file creations throughout the lifetime of that DUFS client. When a client is restarted, it acquires another unique 64-bit client ID and its creation counter is reset to 0.

The FID is used by DUFS to deduce the physical location of the file and the physical filename. Firstly, the physical location of the data in the underlying filesystem is generated using the deterministic mapping function. Secondly, the filename for the data contents on the physical storage is generated from the FID. In this manner, the contents of a file do not have to be renamed or moved between different physical mounts when

the virtual filename is changed.

F. Deterministic mapping function

The deterministic mapping function associates a physical location to each file contents based on its FID. This function takes as input a 128-bit integer representing the FID and returns a number between 1 and N, with N being the number of underlying back-end storages. It has to be deterministic so that any DUFS client can find the right location without coordination.

To achieve a good load-balancing between the different underlying storage mounts, the mapping function has to distribute the FIDs in a fair manner. For this reason, the mapping function of our current implementation is based on the MD5 hash function that has this property [12]. Our mapping function is:

$$fid \quad \longmapsto \quad \text{MD5}(fid) \mod N$$

G. Back-end storage

Once a particular physical filesystem is chosen using the deterministic mapping function, the data is accessed directly using the local mount-point of this distributed filesystem. The filename is deterministically interpreted from the FID. Thus, it is independent of any virtual filename and the DUFS client does not need to communicate with any other component to find the actual physical filename.

In DUFS, the physical filename used to store a file is the equivalent to the hexadecimal representation of the FID that was computed in the previous step. In order to avoid congestion due to file creation at a single directory level, the hexadecimal representation is divided into four parts to create multiple path components. The first component has the filename, while the other components are used for the path hierarchy. Figure 4 shows an example of the filename on the back-end storage for the FID `0123456789abcdef`.

FID:	`0123456789abcdef`
Physical filename:	`cdef` / `89ab` / `4567` / `0123`

Fig. 4. Sample physical filename generated from a given FID

This directory hierarchy is static and identical between all the back-end mount-points. This static structure avoids any potential conflict.

H. Algorithm examples for Metadata operations

In this section, we give some algorithms for some metadata operations in DUFS. Figure 5 shows the algorithm for the *mkdir()* operation; Figure 6 shows the algorithm for the *stat()* operation.

I. Reliability concerns

The DUFS client does not have any state. All the required information are stored either in ZooKeeper or in the back-end storage. So the DUFS reliability relies on the ZooKeeper and back-end distributed filesystems.

1: Get the virtual path of the directory
2: Look for the corresponding Znode
3: **if** Znode exists **then**
4:    **return** 'File exists' error code
5: **else**
6:    Generate the data field with type and metadata information
7:    Create the corresponding Znode with ZooKeeper
8:    **if** success **then**
9:       **return** Success
10:    **else**
11:       Handle error
12:    **end if**
13: **end if**

Fig. 5. Algorithm for the *mkdir()* operation

1: Get the virtual path of the file/directory
2: Get the corresponding Znode with ZooKeeper
3: **if** Znode does not exist **then**
4:    **return** 'No such file or directory' error code
5: **else**
6:    ZooKeeper returned the data field (type, FID, ...)
7:    **if** Znode type is directory **then**
8:       Fill the `struct stat` with information stored in ZooKeeper
9:       **return** `struct stat`
10:    **else**
11:       Compute the physical location
12:       Compute the physical path
13:       Perform *stat()* on the physical file
14:       **return** `struct stat`
15:    **end if**
16: **end if**

Fig. 6. Algorithm for the *stat()* operation

For ZooKeeper, all the information are duplicated among all the servers. Thanks to this, ZooKeeper is able to tolerate the failure of many servers. It needs to have the majority of the servers alive to maintain consistency of the data [8]. Furthermore, although each ZooKeeper server keeps all its data in memory, it is periodically checkpointed on disk. So, it can tolerate the failure of all servers by restarting them later.

Many distributed filesystems like Lustre provide fault tolerance. Data can be replicated among multiple data servers. If such filesystems are used as a back-end storage, it will benefit to the DUFS availability.

V. Performance Evaluation

In this section, we conduct experiments to evaluate the performance of metadata operations with our proposed design. These tests were performed on a Linux cluster. Each node has a dual Intel Xeon E5335 CPU (8 cores in total) and 6 GB memory. A SATA 250 GB hard drive is used as the storage device on each node. The nodes are connected with 1 GigE

for general purpose networking. Each node runs Linux 2.6.30.

We dedicate a set of nodes as Lustre MDS and OSS (version 1.8.3) to form multiple instances of Lustre filesystem. Another set of dedicated nodes work as PVFS2 servers (version 2.8.2) to export multiple instances of PVFS2 filesystem. Each client node mounts multiple instances of Lustre and PVFS2 filesystems and uses DUFS to merge these distinct physical partitions into one logically uniformed partition. ZooKeeper server runs along with the DUFS clients, and they provide distributed coordination services over 1 GigE.

We have used the mdtest benchmark [13] for our experiments. We carried out our experiments by creating a directory structure with a fan-out factor of 10 and directory depth of 5. As the number of processes increases, the number of files per directory also increases accordingly. We have also carried out experiments where many files are created in a single directory. We have used the same parameters and configuration while experimenting with different back-end parallel file systems like Lustre and PVFS2.

A. ZooKeeper throughput for basic operations

With DUFS design, each metadata operation has to go through the ZooKeeper service before it is actually issued to the corresponding physical back-end filesystem. In this section, we performed experiments in order to study the ZooKeeper throughput for basic operations like *zoo_create()*, *zoo_get()*, *zoo_set()* and *zoo_delete()* using the ZooKeeper synchronous API.

With a total of 8 DUFS clients in the experimental setup, we varied the number of ZooKeeper servers from 1 to 8. The results are shown on Figure 7. For the *zoo_create()*, *zoo_delete()* and *zoo_set()* operations, we can see that with more number of ZooKeeper Servers the overall throughput drops down. The is the expected behavior since this operation performs modifications on the Znodes. Thus, all the ZooKeeper servers have to coordinate to ensure the consistency of their replicated states. For the *zoo_get()* operation, the overall throughput increases with more number of ZooKeeper Servers. ZooKeeper performs very well in read dominant workloads [8]. Indeed, each ZooKeeper server can serve the request independently from each other.

B. Influence of the number of ZooKeeper Servers

In this section we performed experiments in order to study the outcome by varying the number of ZooKeeper servers. We used a set of 8 nodes with 8 DUFS clients, which use a number of ZooKeeper servers varying from 1 to 8. We measured the operation throughput and we compared it with the throughput of our basic Lustre configuration.

The results are presented on Figure 8. As expected, read operations like file *stat()* and directory *stat()* shows a significant performance improvement when the number of ZooKeeper servers increases. For the other operations, the effect of the number of ZooKeeper servers is lesser.

Finally, these results show that using 8 ZooKeeper servers is a good compromise for our configuration.

C. Influence of the number of back-end storages

In this section, we performed experiments to study the influence of varying number of back-end storages to be combined by DUFS. For this experiment, we had an ensemble of 8 ZooKeeper servers. Since the directory operations do not touch the back-end distributed filesystems, we only focus on file operations for this experiment.

Figure 9 shows the throughput of file operations for 2 and 4 back-end storages and for different number of client processes. We also compare this throughput to the Basic Lustre case. Using 4 back-end storages instead of 2 provides a small improvement for file creation and removal. For file *stat()*, we can see an improvement of more than 37% with 256 client processes.

All file operations are redirected from the DUFS client to the back-end storage and are uniformly distributed among all the back-end storages. However, there is an indirection to a ZooKeeper server. File removal and creation require a Znode modification. The cost of this modification overtakes the benefit of multiple back-end storages. The file *stat()* operation only reads the content of the Znode, which is very fast. That is why we see a clear benefit when increasing the number of back-end storages in this case.

D. Comparison with Lustre and PVFS2

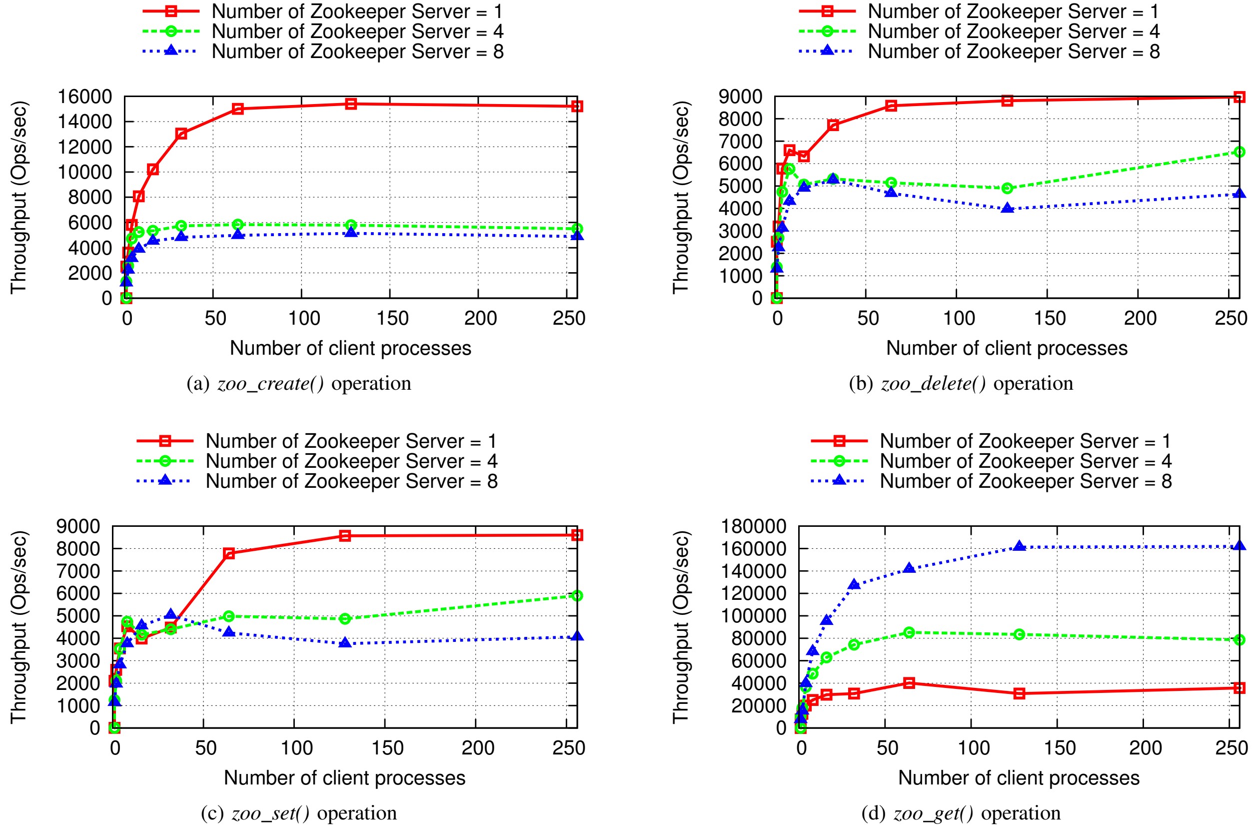
In this section, we study the performance of our DUFS prototype in comparison with two distributed filesystems: Lustre and PVFS2. To keep a fair comparison, we also use Lustre and PVFS2 as our back-end storages. We study the scalability by increasing the number of client processes.

In these experiments, we had 8 DUFS clients and 8 Zookeeper servers. The ZooKeeper servers and DUFS clients were running on same nodes.

From Figure 10, we can see that DUFS, with Lustre as a back-end physical filesystem, can outperform Basic Lustre. We can see similar results even in the PVFS2 case. One notable point is that for the directory operations, we see a similar trend for Lustre and PVFS2. This is expected because in DUFS, directory operations only rely on ZooKeeper. Also, for the file operations, DUFS with Lustre as back-end filesystem performs way better than DUFS with PVFS2 as back-end filesystem. This is because in that case, the back-end storage is actually used and thus, the throughput of these operations depend on the performance of this back-end filesystem .

From the scalability point of view, we see that Lustre and PVFS2 do not scale very well. When the number of client processes grows significantly, their performance drops down. Conversely, DUFS does not perform so well at small scale. However, it can outperform Lustre for all operations with 256 client processes. In all the cases, DUFS with PVFS2 back-end storage is clearly better than PVFS2 alone. For directory creation with 256 client processes, DUFS outperforms Lustre by a factor of 1.9, and PVFS2 by a factor of 23.

Finally, we can see that for directory/file stat the approach discussed in the paper performs exceedingly well as compared to its basic variant i.e. Lustre and PVFS2. With respect to file



(a) *zoo_create()* operation

(b) *zoo_delete()* operation

(c) *zoo_set()* operation

(d) *zoo_get()* operation

Fig. 7. ZooKeeper throughput for basic operations by varying the number of ZooKeeper Servers

stat() with 256 processes, our approach is 1.3 and 3.0 times faster than Lustre and PVFS2, respectively. This is mainly because ZooKeeper performs well in case of read dominant workloads.

E. Memory usage

Since ZooKeeper keeps all its data in memory, the memory usage can be a concern. In the following experiment, we study the memory usage of ZooKeeper (Java process), and DUFS as well, when the number of metadata information increases. We have designed a benchmark that creates a large number of directories and reports the resident process memory size. For this experiment, all the processes ran on the same node.

Additionally, in order to compare the memory usage of DUFS, we run the same benchmark for a dummy FUSE filesystem which just does nothing, except forwarding the requests to a local filesystem.

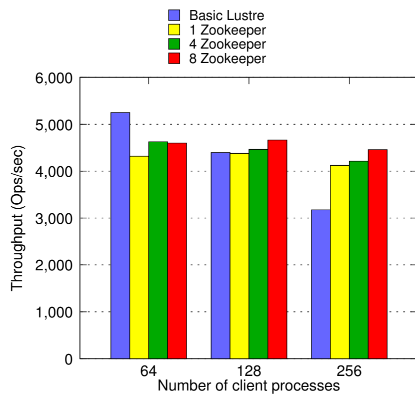
The results are shown on the Figure 11. We can see that the memory consumed by DUFS is bounded and similar to a normal FUSE based file system, which is what is expected. The ZooKeeper memory usage is proportional to the number of created directories or files (Znode data size is similar for file or directory). From these numbers, we can estimate that storing one million files or directory requires about 417 MB in memory. This drawback comes from the ZooKeeper design choice.

VI. Related Work

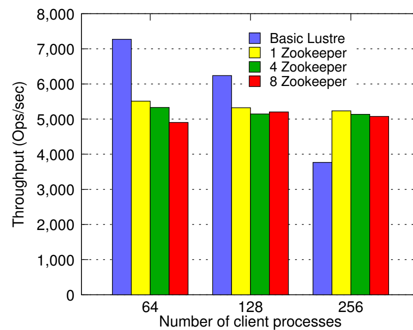
Distributed filesystems like NFS [14], Coda [15] and AFS [16] partition their namespace statically among multiple servers, so most of the metadata operations are centralized. pNFS [17] allows to distribute data but retains the concept of centralized metadata. Other parallel file systems like GPFS [18], GFS [19], Intermezzo [20] and Lustre [1] use directory locks for file creation, with the help of a distributed lock management (DLM) for better performance.

Lustre uses a single metadata server to manage the entire namespace. Lustre distributed lock management module handles locks between clients and servers and local locks between nodes. The Lustre community has also mentioned the fact of a single Metadata Server being a bottleneck in HPC environments. So they came up with the concept of Lustre Clustered Metadata Server (CMD). The original design for CMD was proposed in 2008. However, CMD is still a prototype and it has not been released yet. In CMD, files are identified by a global FID and are assigned to a metadata server. Once we know the FID, we can directly deal with the server. Getting this FID still requires a centralized/master metadata server and this information is not replicated. Thus, it represents a bottleneck. Also, the reliability and availability still rely on a single master node in this approach.

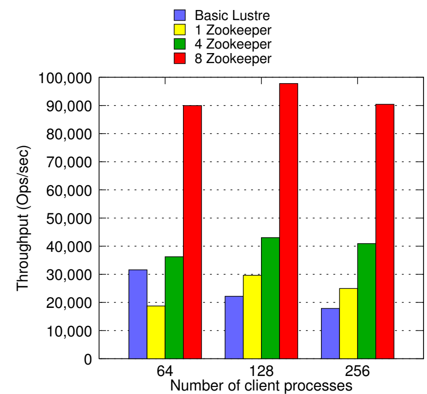
In order to improve the metadata mutation throughput, previous works aimed to mount more independent filesystems



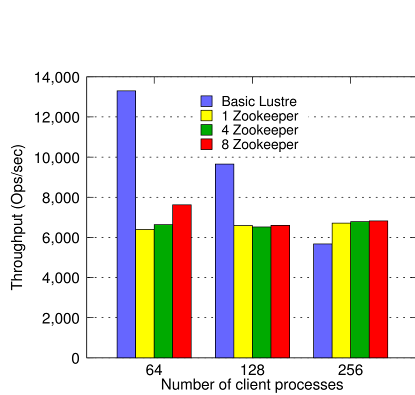
(a) Directory creation



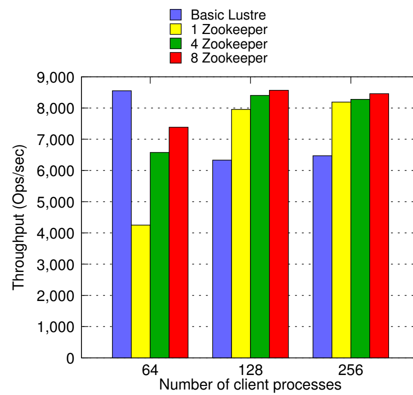
(b) Directory removal



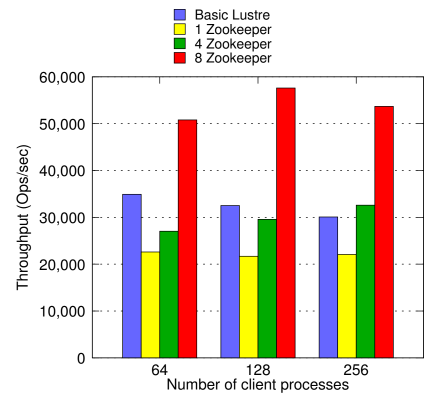
(c) Directory stat



(d) File creation

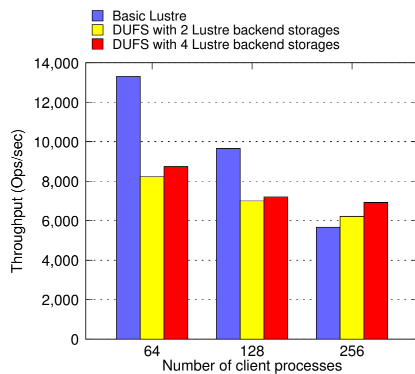


(e) File removal

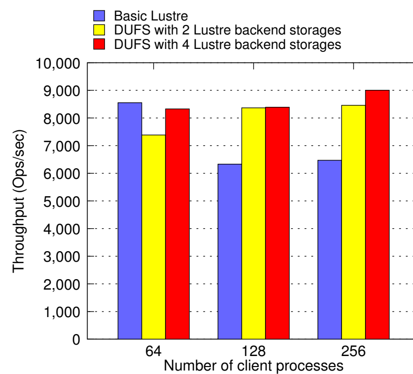


(f) File stat

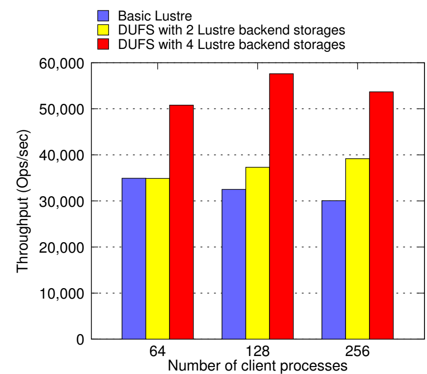
Fig. 8. Operation throughput by varying the number of Zookeeper Servers using 2 Lustre back-end storages, and compared to a basic Lustre configuration with one metadata server



(a) File creation

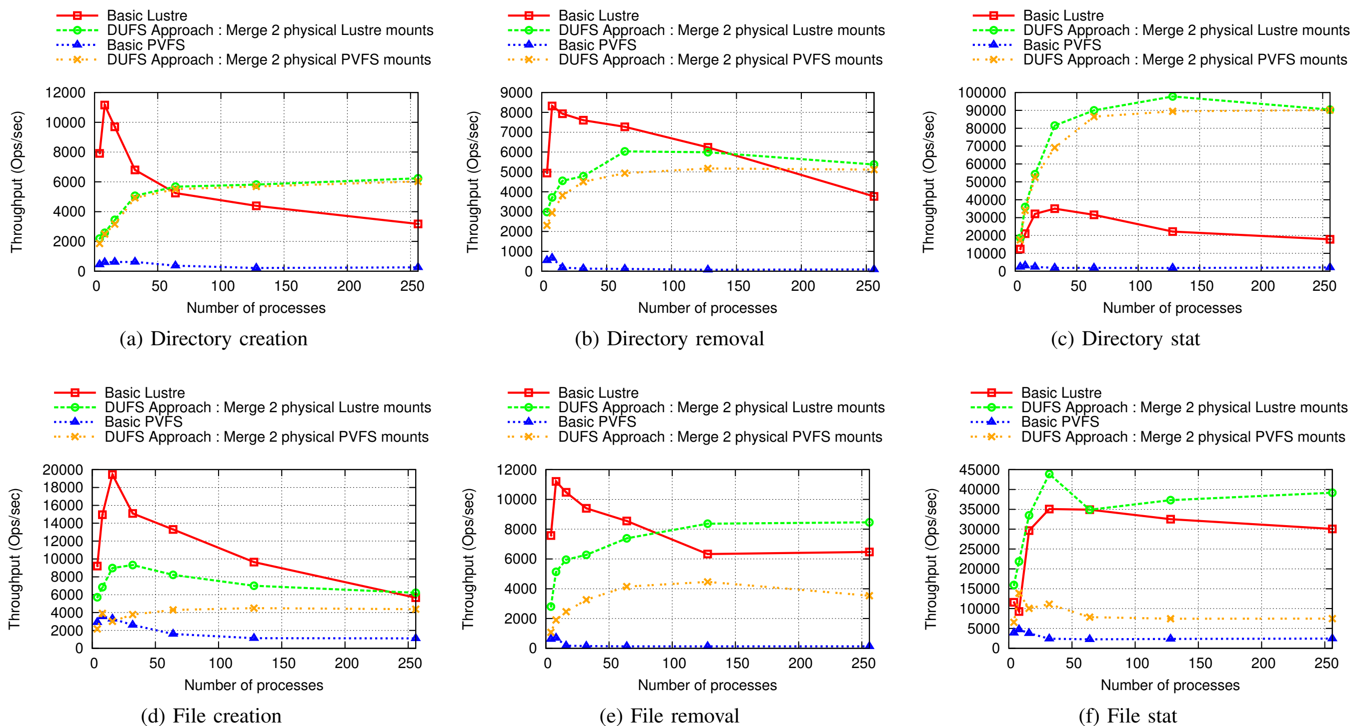


(b) File removal



(c) File stat

Fig. 9. File operation throughput for different numbers of back-end storages



(a) Directory creation (b) Directory removal (c) Directory stat

(d) File creation (e) File removal (f) File stat

Fig. 10. Operation throughput with respect to the number of clients for Lustre and PVFS2

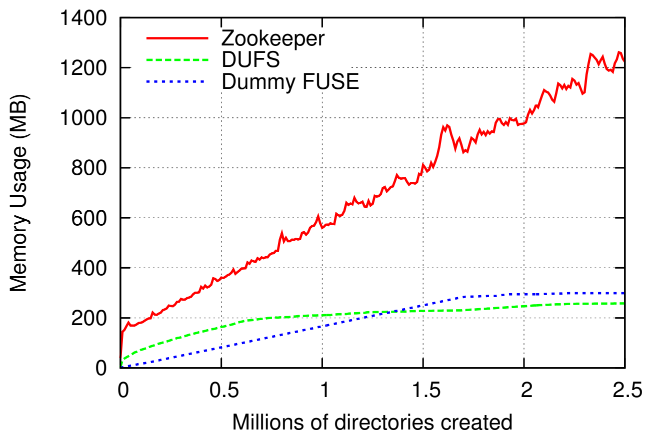


Fig. 11. Zookeeper memory usage and its comparison with DUFS and basic FUSE based file system memory usage

into a larger aggregate. But with this approach, each directory or directory sub-tree is still managed by one metadata server. Some systems use cluster metadata servers in pairs for failover. Alternatively, some systems allow any server to act as a proxy and forward requests to the appropriate server. However, these approaches do not increase the metadata mutation throughput [21].

Symmetric shared disk filesystems, that support concurrent updates to the same directory, use complex distributed locking and cache consistency semantics. These both techniques induce significant bottlenecks for concurrent create workloads, especially from many clients working on one single directory. Moreover, filesystems that support client caching of directory entries for faster read-only workloads, generally disable client caching during concurrent update workload to avoid excessive consistency overhead. [22] mainly focuses on the file creation strategies in distributed metadata filesystems, but it does not consider other metadata operations. Also this work targets PVFS. [23] describes a dynamic sub-tree partitioning and adaptive system schemes to manage metadata workloads.

There has been some work in the area to designing a distributed indexing scheme, GIGA+ [24], in order to build directories with millions/trillions of files with high degree of concurrency. This work is more relevant in workloads where the directories have a huge fan-out factor. In GIGA+, every server only keeps a local view of the partitions it manages, and this state is not shared. Hence, there are no synchronization and consistency bottlenecks. But, if the server or the partition goes down, or if the root level directory gets corrupted, then the files are not accessible anymore.

With our scheme, which is based on a distributed coordination service, we improve the reliability and availability. We can achieve a good scalability, but the size of our namespace is bounded by the actual size of the physical memory because ZooKeeper keeps all its information in memory.

Also, some approaches combine multiple partitions into a virtual mount point. UnionFS (Linux official union filesystem in kernel mainline) [25] has a lot of options but it does not support load-balancing between branches. Most of the filesystems which combine multiple partitions into a virtual

mount are limited to a single node. The consistency is not maintained if the underlying filesystem is modified remotely. Some union filesystems cannot extract the parallelism. Their default behavior is to use the first partition until it reaches a threshold (based on the free space). Thus, the metadata operation throughput is not improved even after combining multiple mount points.

VII. Conclusions and Future Work

In this paper, we have designed a Distributed Metadata Service Layer and evaluated its benefits to parallel file systems. Distributed metadata management is a hard problem since it involves taking care of various consistency and reliability aspects. Also, scaling metadata performance is more complex than scaling raw I/O performance. With distributed metadata, this complexity further increases. This leads to a primary goal while designing a Distributed Metadata Service Layer - to improve on the scalability aspect while taking care of consistency and reliability.

In order to study this topic, we have designed a FUSE-based filesystem, named Distributed Union File System (DUFS). DUFS can combine multiple mounts of a parallel filesystems into a single virtual filesystem which is exposed to the users. We have used ZooKeeper as a distributed coordination service to take care of metadata reliability and consistency. Finally, our ZooKeeper-based prototype shows the main trends that can be expected when using a distributed coordination service for metadata management.

From our experiments, we can see that for higher number of processes running on the client nodes and as the load on the client nodes increases, we can scale well with the approach proposed in the paper as compared to the other studied distributed filesystems Lustre and PVFS2. While Lustre performs very well for a small number of clients, its performance drops down when the number of clients increases. With our approach, we are able to maintain good performance even with a large number of clients. With 256 client processes, we are able to outperform Lustre for the 6 metadata operations studied.

One major drawback of our approach is the memory usage because the ZooKeeper servers keep all their data in memory. Future work will focus on addressing this issue. Additionally, we plan to replace our MD5-based mapping function with one based on consistent hashing [26]. This approach will allow to dynamically add and remove back-end storages while ensuring that the amount of data to relocate stays bounded.

References

[1] “Oracle Lustre File System,” http://wiki.lustre.org/index.php/MainPage.

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, “PVFS: A parallel file system for linux clusters,” in *Proceedings Of The 4th Annual Linux Showcase And Conference*. MIT Press, 2000.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP ’03. New York, NY, USA: ACM, 2003.

[4] D. Roselli, J. R. Lorch, and T. E. Anderson, “A comparison of file system workloads,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC ’00. Berkeley, CA, USA: USENIX Association, 2000.

[5] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, “Dynamic metadata management for petabyte-scale file systems,” in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, ser. SC ’04. Washington, DC, USA: IEEE Computer Society, 2004.

[6] “Clustered MetaData,” http://wiki.lustre.org/index.php/Clustered_Metadata.

[7] “File System in Userspace (FUSE),” http://fuse.sourceforge.net/.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: wait-free coordination for internet-scale systems,” in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC’10. Berkeley, CA, USA: USENIX Association, 2010.

[9] A. Saify, G. Kochhar, J. Hsieh, and O. Celebioglu, “Enhancing high-performance computing clusters with parallel file systems,” *Dell Power Solutions*, 2005.

[10] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long, “Obfs: A file system for object-based storage devices,” in *21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST2004)*, Apr. 2004.

[11] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. Mclarty, “File system workload analysis for large scale scientific computing applications,” in *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004, pp. 139–152.

[12] R. A. Rivest, “The md5 message digest algorithm,” Internet RFC 1321, 1992.

[13] “Mdtest Benchmark,” http://mdtest.sourceforge.net/.

[14] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, “Nfs version 3 - design and implementation,” in *In Proceedings of the Summer USENIX Conference*, 1994, pp. 137–152.

[15] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, and C. Steere, “Coda: A highly available file system for a distributed workstation environment,” *IEEE Transactions on Computers*, vol. 39, pp. 447–459, 1990.

[16] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, “Andrew: a distributed personal computing environment,” *Commun. ACM*, Mar. 1986.

[17] G.Goodson, B. Welch, B.Halevy, D.Black, and A.Adamson, “NFSv4 pNFS extensions,” Tech. Rep., Oct. 2005.

[18] F. Schmuck and R. Haskin, “Gpfs: A shared-disk file system for large computing clusters,” in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST ’02. USENIX Association, 2002.

[19] “RedHat Global File System,” http://www.redhat.com/gfs/.

[20] P. J. Braam, M. Callahan, and P. Schwan, “The intermezzo file system,” 1999.

[21] “Isilon Systems Inc.” http://www.isilon.com.

[22] A. Devulapalli and P. Wyckoff, “File creation strategies in a distributed metadata file system,” in *IPDPS*, 2007.

[23] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, “Dynamic metadata management for petabyte-scale file systems,” in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, ser. SC ’04. IEEE Computer Society, 2004.

[24] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte, “Giga+: scalable directories for shared file systems,” in *Proceedings of the 2nd international workshop on Petascale data storage*, ser. PDSW ’07. New York, NY, USA: ACM, 2007.

[25] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok, “Unionfs: User- and communityoriented development of a unification filesystem,” in *In Proceedings of the 2006 Linux Symposium*, 2006.

[26] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web,” in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC ’97, New York, NY, USA, 1997, pp. 654–663.