CFS: A Distributed File System for Large Scale Container Platforms

Haifeng Liu[§, †], Wei Ding[*], Yuan Chen[*], Weilong Guo[†], Shuoran Liu[†], Tianpeng Li[†], Mofei Zhang[†], Jianxing Zhao[†], Hongyin Zhu[†], Zhengyi Zhu[†]

[§]University of Science and Technology of China, Hefei, China

[†]JD.com, Beijing, China

[*]JD.com Silicon Valley R&D Center, Mountain View, CA, USA

ABSTRACT

We propose CFS, a highly scalable, general-purpose, and POSIX-compliant distributed file system for containers running at JD's Kubernetes platform. CFS is written in Go and has a few features that differentiate itself from the existing open source solutions, e.g., (1) it distributes the metadata of files across different storage nodes based on the memory usage to always have balance workloads; (2) it supports both sequential and random file accesses with optimized storage for both large files and small files; and (3) it adopts two strongly consistent replication protocols for different write patterns to improve the replication performance.

We performed a comprehensive comparison with Ceph, a widely-used distributed file system on container platforms. Our experimental results show that, in testing 7 commonly used metadata operations, CFS gives around 3 times performance boost on average. In addition, in the tests of different file access patterns, CFS exhibits better performance when the number of concurrent clients is large.

CCS CONCEPTS

• **Information systems → Distributed storage**;

KEYWORDS

distributed file system; container; cloud native;

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3314046

ACM Reference Format:
Haifeng Liu[§, †], Wei Ding[*], Yuan Chen[*], Weilong Guo[†], Shuoran Liu[†],, Tianpeng Li[†], Mofei Zhang[†], Jianxing Zhao[†], Hongyin Zhu[†], Zhengyi Zhu[†]. 2019. CFS: A Distributed File System for Large Scale Container Platforms. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3299869.3314046

1 INTRODUCTION

Containerization and microservices have revolutionized cloud environments and architectures over the past few years [1, 4, 18]. As applications can be built, deployed and managed faster through continuous delivery, more and more companies start to move legacy applications and core business functions to containerized environments, where the microservices running on each set of containers are usually independent from the local disk storage. Decoupling compute from storage allows the companies to scale the container resources in a more efficient way, but it also brings up the need of a separate storage because (1) containers may need to preserve the application data even after they are closed, (2) the same file may need to be accessed by different containers simultaneously, and (3) the storage resources may need to be shared by different services and applications. Without the ability to persist data, containers might have limited usage in many workloads, especially in stateful applications.

One option is to take the existing distributed file systems and bring them to the cloud native environment through storage orchestrators such as Rook[1] or through Container Storage Interface (CSI)[2] implemented in container orchestrators like Kubernetes [6], Mesos [14], etc. When seeking such a distributed file system, the engineering teams who own the applications and services running on JD's container platform provide many valuable feedbacks. However, in terms

[1]https://rook.io/
[2]https://github.com/container-storage-interface/spec

of *performance, scalability* and *usability*, these feedbacks also give us hard time to adopt any existing open source solution directly. For example, Ceph [27] is a widely-used distributed file system for container platforms. But many nice features provided by Ceph (e.g., multiple storage backends) also make it very complicated and difficult to learn and deploy, and these features may not even be needed in our case. In addition, in certain scenarios such as metadata operations and small file operations, its performance in a multi-client environment can be hard to optimize.

In this paper, we propose Chubao File System (CFS), a distributed file system designed for JD's container platform. The code is available at https://github.com/ChubaoFS/cfs. Some key features include:

Separate Metadata Cluster with Minimal Rebalancing. On JD's container platform, there could be heavy accesses to the files by a large number of clients simultaneously. Most file operations, such as creating, appending, or deleting a file would require updating the file metadata. Therefore, a single node that stores the file metadata could easily become the performance bottleneck due to the hardware limits. This motivates us to employ a separate cluster to store the file metadata. Different from most existing works [29], whose metadata placement mechanisms usually require rebalancing the storage nodes during capacity expansion, CFS distributes the file metadata across different storage nodes based on the memory usage so that the workloads are *always* balanced. Although similar idea has been used for chunk-server selection in MooseFS [24], to the best of knowledge, CFS is the first open source solution to employ this idea for metadata placement.

Multi-Tenancy and General-Purpose. To reduce the storage cost, we prefer different applications and services to be served from the same shared storage infrastructure. In this way, the provisioned storage for the combined workloads at any one point in time can be significantly less than if those workloads were run on their own dedicated hardware. CFS achieves this by providing different *volumes* to different set of containers, where each volume consists of a subset of the storage units on the same cluster (see the next section).

Because of this multi-tenancy feature, the size of files in the combined workloads can vary from a few kilobytes to hundreds of gigabytes, and these files can be accessed in a sequential or random fashion. CFS comes with a *general-purpose* storage engine to efficiently store both large and small files with excellent performance on different file access patterns. In particular, it utilizes the punch hole interface in Linux [22] to *asynchronously* free the disk space occupied by the deleted small files. This design avoids the need of implementing a garbage collection mechanism [3], which greatly simplifies the engineering work.

Strongly Consistent Storage Engine with Two Replication Protocols. Different from any existing open source solution, which usually only allows a single replication protocol at any time [23, 27, 28], CFS adopts two strongly consistent replication protocols based on different write scenarios (namely, *append* and *overwrite*) to improve the replication performance.

POSIX-Compliance with Relaxed Consistency Semantics. In a distributed file system with POSIX-compliant APIs, the behavior of serving multiple processes on multiple client nodes should be the same as the behavior of a local file system serving multiple processes on a single node with direct attached storage. CFS provides POSIX-compliant APIs to simplify the development of the upper level applications, and shorten the learning curve for new users. However, the POSIX consistency semantics have been selectively relaxed in order to better align with the needs of applications and to improve the system performance.

2 DESIGN AND IMPLEMENTATION

As shown in Figure 1, CFS consists of a *metadata subsystem*, a *data subsystem*, and a *resource manager*, and can be accessed by different *clients* (as a set of application processes) hosted on the containers through different file system instances called *volumes*.

The metadata subsystem stores the file metadata, and consists of a set of *meta nodes*. Each meta node consists of a set of *meta partitions*. The data subsystem stores the file contents, and consists of a set of *data nodes*. Each data node consists of a set of *data partitions*. We will give more details about these two subsystems in the following sections.

The volume is a logical concept in CFS and consists of one or multiple meta partitions and one or multiple data partitions. Each partition can only be assigned to a single volume. From a client's perspective, the volume can be viewed as a file system instance that contains data accessible by the containers. A volume can be mounted to multiple containers so that files can be shared among different clients simultaneously, and needs to be created at the very beginning before the any file operation.

The resource manager manages the meta nodes and data nodes by processing different types of tasks (such as creating and deleting partitions, creating new volumes, and adding/removing nodes) from a task manager in each meta/data node. It also keeps track of the status such as the memory and disk utilizations, and liveness of the meta and data nodes in the cluster. The resource manager has multiple replicas, among which the strong consistency is maintained by a consensus

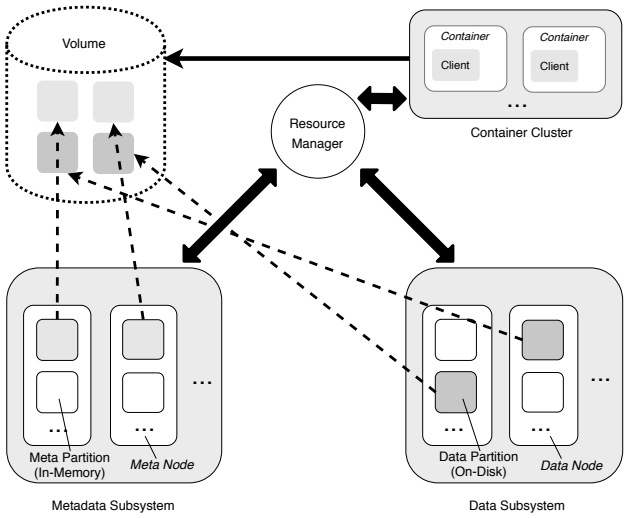

Figure 1: Architecture of CFS.

algorithm such as Raft [17], and persisted to a key value store such as RocksDB[3] for backup and recovery. Note that two replicas of the same partition never reside on the same node during replica selection.

2.1 Metadata Subsystem

The metadata operations could make up as much as half of typical file system workloads [19]. On JD's container platform, this becomes even more important as there could be heavy accesses to the metadata of files by a large number of clients simultaneously. A single node that stores the file metadata [23] could easily become the performance bottleneck. As a result, we employ a distributed metadata subsystem to store the file metadata and forward the metadata requests from different clients to different storage nodes, which greatly improves the scalability of the entire system.

2.1.1 Internal Structure. The metadata subsystem can be considered as an in-memory datastore of the file metadata. Each meta node which can have hundreds of meta partitions. Each meta partition on a meta node stores the file metadata in memory by maintaining a set of *inodes* and a set of *dentries* [2].

A meta partition can only store the inodes and dentries of the files from the same volume. We employ two b-trees called *inodeTree* and *dentryTree* for fast lookup of inodes and dentries in the memory. The *inodeTree* is indexed by the inode id, and the *dentryTree* is indexed by the dentry name and the parent inode id. We also maintain a range of the inode ids (denoted as *start* and *end*) stored on a meta partition for splitting, which will be discussed in the resource manager later. The code snippet below shows the definitions of the meta partition, the inode and the dentry in CFS.

```
type metaPartition struct {
   partitionId uint64 // Partition id
   start, end     uint64 // Lower and upper id bounds
   nodeId     uint64 // Meta node id
   dentryTree   *bTree // B-tree for dentries
   inodeTree    *bTree // B-tree for inodes
   ... // Other fields
}

type inode struct {
   inode     uint64 // Inode id
   type    uint32 // Inode type
   linkTarget []byte // SymLink target name
   nLink     uint32 // number of links
   flag uint32
   ...  // Other fields
}

type dentry struct {
   parentId uint64 // Parent inode id
   name    string // Name of the dentry
   inode   uint64 // Current inode id
   type    uint32 // Dentry type
}
```

2.1.2 Raft-based Replication. The replication during file write is performed in terms of meta partitions. The strong consistency among the replicas of each meta partition is ensured by a revision of the Raft consensus protocol [17] called the MultiRaft[4], which has the advantage of reduced heartbeat network traffic comparing to the original version.

2.1.3 Failure Recovery. The in-memory meta partitions are persisted to the local disk by snapshots and logs [17] for backup and recovery. Some techniques such as log compaction are used to reduce the log files sizes and shorten the recovery time. It is worth noting that, a failure that happens during a metadata operation could result an *orphan inode* with which has no dentry to be associated. The memory and disk space occupied by this inode can be hard to free. To minimize the chance of this case to happen, the client always issues a retry after a failure until the request succeeds or the maximum retry limit is reached.

2.2 Data Subsystem

The data subsystem is optimized for the storage of both large and small files, which can be accessed in a sequential or random fashion. It adopts two different replication protocols

[3]https://rocksdb.org/

[4]https://github.com/Zemnmez/cockroach/multiraft

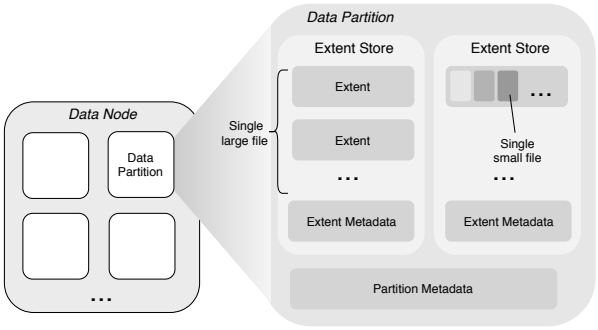

Figure 2: Internal structure of a data partition.

to ensure the strong consistency with some tradeoffs on the performance and code usability.

2.2.1 Internal Structure. The data subsystem consists of a set of data nodes. As illustrated in Figure 2, each data node has a set of data partitions, and a data partition consists of a storage engine called the *extent store*, which is constructed by a set of storage units called *extents*, and the partition metadata such as the partition id and the replica addresses. The quota of an extent is configurable at the cluster level. The CRC of each extent is cached in the memory to speed up the check for data integrity. The code snippet below shows the structure of data partition.

```
type dataPartition struct {
   volumeId      string // Volume id
   partitionId   uint32 // Partition id
   partitionSize int  // Size of the data partition
   replicaHosts  []string // Replicas Addresses
   path          string // Path of the partition
   extentStore   *storage.extentStore
   ... // Other fields
}
```

A threshold t (128 KB by default) is used to determine if the file should be considered as "small file", i.e., the file whose size is less than or equal to t is considered as "small file"; otherwise, it is considered as "large file". The value of t is configurable at the startup and usually aligned with the packet size during the data transfer to avoid the packet assembly or splitting. This is important as we store the large files and small files in different ways.

2.2.2 Large File Storage. For large files, the contents are stored as a sequence of one or multiple extents, which can be distributed across different data partitions on different data nodes. Writing a new file to the extent store always causes the data to be written at the zero-offset of a new extent, which eliminates the need for the offset within the extent. The last extent of a file does not need to fill up its size limit by padding (i.e., the extent does not have holes), and never stores the data from other files.

2.2.3 Small File Storage and Punch Holes. The contents of multiple small files are aggregated and stored in a single extent, and the physical offset of each file content in the extent is recorded in the corresponding meta node. CFS relies on the punch hole interface provided by the underlying file system, *fallocate()*[5], to *asynchronous* free the disk space occupied by the to-be-deleted file. The advantage of this design is to eliminate the need of implementing a garbage collection mechanism and therefore avoid to employ a mapping from logical offset to physical offset in an extent [3]. Note that this is different from deleting large files, where the extents of the file can be removed directly from the disk.

2.2.4 Two Strong Consistent Replication Protocols . Although one can simply apply a Raft-based replication to ensure the strong consistency, CFS employs two replication protocols for the data subsystem to achieve this with some tradeoffs between the performance and code reusability.

The replication is performed in terms of partitions during file writes. Depending on the file write pattern, CFS adopts different strongly consistent replication strategies. Specifically, for sequential write (i.e., *append*), we adopt a primary-backup replication protocol [26], and for overwrite, we employ a MultiRaft-based replication protocol similar to the one used in the metadata subsystem. The main reason for us to employ two replication protocols is that the primary-backup replication is not suitable for overwrites as either the data consistency cannot be guaranteed or the replication performance needs to be compromised, depending on how the overwrite is implemented.

In the case of the *in-place overwrite*, where the file contents are modified directly on the original extent, the file metadata will not change as the file offset/size does not change. Therefore, when failure happens, there could exist two replicas with the same offset/size but different contents. That is to say, the metadata stored at each replica can be unchanged as the largest offset of the data that has been committed by each replica is still the same, but the contents of these replicas are now different.

In the case of the *out-of-place overwrite*, at least one new extent will be created to store the new file content(s), and some of the original extents will be *logically* split into several fragmentations, which are usually linked together like a linked list. In this linked list, the pointer to the original fragmentations will be replaced by the ones associated with the newly created extent(s). As more and more file contents being overwritten, eventually there will be too many fragmentations

[5]http://man7.org/linux/man-pages/man2/fallocate.2.html

on the data partitions that requires *defragmentation*, which could significantly affect the replication performance.

Because of the above reason, we adopts the MultiRaft-based replication for overwrite. Although it is known to have the write amplification issue as it introduces extra IO of writing the log files that directly affects the read-after-write performance, this replication protocol ensures strong consistency regardless of the write patterns.

2.2.5 Failure Recovery. Because of the existence of two different replication protocols, when a failure on a replica is discovered, we first start the recovery process in the primary-backup-based replication by checking the length of each extent and making all extents aligned. Once this processed is finished, we then start the recovery process in our MultiRaft-based replication.

However, it should be noted that, during a sequential write, the stale data is allowed on the partitions, as long as it is never returned to the client. This is because that, in this case, the commitment of the data at an offset also indicates the commitment of all the data before that offset. As a result, we can use the offset to indicate the portion of the data that has been committed by all replicas, and the leader always returns the largest offset that has been committed by all the replicas to the client, who will then update this offset at the corresponding meta node. When the client requests a read, the meta node only provides the address of a replica with the offset of the data that has been committed by all replicas, regardless the existence of the stale data that has not been fully committed.

Because of this offset-based commitment, there is no need to recovery the inconsistent (stale) portion of the data on the replicas. If the client sends a write request of a file with size k MB, and the head only returns the commitment of the first p MB from all replicas, then the client will resend a write request for the remaining $k - p$ MB data to other extents in different data partitions. This design greatly simplifies our implementation of maintaining the strong consistency when a file is sequentially written to CFS.

2.3 Resource Manager

The resource manager manages the meta nodes and data nodes by asynchronously processing different types of tasks from a task manager in each meta/data node.

2.3.1 Utilization-Based Distribution. One important task for the resource manager is to distribute the file metadata and contents among different nodes. In CFS, we adopt a *utilization-based distribution*, where the file metadata and contents are distributed across the storage nodes based on the memory/disk usage. This greatly simplifies the resource allocation problem in a distributed file system. Although similar ideas have been used for chunk-server selection in MooseFS [24], to the best of knowledge, CFS is the first open source solution to employ this idea for metadata placement.

The commonly-used metadata placement schemes, such as *hashing* and *subtree partition* [5] usually require disproportionate amount of metadata to be moved when adding servers. Such data migration could be a headache in a container environment as capacity expansion usually needs to be finished in a short period of time. Other approaches such as *lazybird* [5] and *dynamic subtree partition* [29] move the data lazily or employ a proxy to hide the data migration latency. But these solutions also bring extra amount of engineering work for future development and maintenance. Different from these approaches, the utilization-based distribution, despite its simplicity, does not require any rebalancing of the data when adding new storage nodes during capacity expansion. In addition, because of the uniformed distribution, the chance of multiple clients accessing the data on the same storage node simultaneously can be reduced, which could potentially improve the performance stability of the file system. Specifically, our utilization-based distribution works as follows:

First, when creating a volume, the client asks the resource manager for a certain number of available meta and data partitions. These partitions are usually the ones on the nodes with the lowest memory/disk utilizations. However, it should be noted that, when writing a file, the client simply selects the meta and data partitions in a random fashion from the ones allocated by the resource manager. The reason that the client does not adopt similar utilization-based approach is to avoid the communication with the resource manager in order to obtain the update-to-date utilization information of each allocated node.

Second, when the resource manager finds that all the partitions in a volume is about to be full, it automatically adds a set of new partitions to this volume. These partitions are usually the ones on the nodes with the lowest memory/disk utilizations. Note that, when a partition is full, or a threshold (i.e., the number of files on a meta partition or the number of extents on a data partition) is reached, no new data can be stored on this partition, although it can still be modified or deleted.

2.3.2 Meta Partition Splitting. The resource manager also needs to handle a special requirement when splitting a meta partition. In particular, if a meta partition is about to reach its upper limit of the number of stored inodes and dentries, a splitting task needs to be performed with the requirement to ensure that the inode ids stored at the newly created partition are unique from the ones stored at the original partition.

The pseudocode of our solution is given in Algorithm 1, in which, the resource manager cuts off the inode range

of the meta partition in advance at a upper bound *end*, a value greater than highest inode id used so far (denoted as *maxInodeID*), and sends a *split* request to the meta node to (1) specify the largest inode id as *end* for this meta partition, and (2) create a new meta partition with the inode range from *end* + 1 to ∞ for this volume. As a result, the inode range for these two meta partitions becomes $[1, end]$ and $[end + 1, \infty]$, respectively. If there is another file needs to be created, then its inode id will be chosen as *maxInodeID* + 1 in the original meta partition, or *end* + 1 in the newly created meta partition. The *maxInodeID* of each meta partition can be obtained by the periodical communication between the resource manager and the meta nodes.

When a request to a meta/data partition times out (e.g., due to network outage), the remaining replicas of this partition are marked as read-only. When a meta/data partition is no longer available (e.g., due to hardware failures), all the data on this partition will eventually be migrated to a new partition manually. This unavailability is identified by the multiple failures reported by the node.

2.4 Client

The client has been integrated with FUSE[6] to provide a file system interface in the user space. The client process runs entirely in the user space with its own cache.

To reduce the communication with the resource manager, the client caches the addresses of the available meta and data partitions assigned to the mounted volume, which can be obtained at the startup, and periodically synchronizes this available partitions with the resource manager.

To reduce the communication with the meta nodes, the client also caches the returned inodes and dentries when creating new files, as well as the data partition id, the extent id and the offset, after the file has been written to the data node successfully.When a file is opened for read, the client will force the cache metadata to be synchronous with the meta node. In this way, the metadata cached at the client side is always up-to-date before any read or write.

To reduce the communication with the data nodes, the client caches the most recently identified leader. Our observation is that, when reading a file, the client may not know which data node is the current leader because the leader could change after a failure recovery. As a result, the client may try to send the read request to each replica one by one until a leader is identified. However, since the leader does not change frequently, by caching the last identified leader, the client can minimize the number of retries in most cases.

[6]https://github.com/libfuse/libfuse

Algorithm 1 Splitting Meta Partition

```
1:  procedure PARTITIONING
2:      mp ← current meta partition
3:      c ← current cluster
4:      v ← cluster.getVolume(mp.volName);
5:      maxPartitionID ← v.getMaxPartitionID()
6:      if metaPartition.ID < maxPartitionID then return
7:      if mp.end == maxInodeID then
8:          end ← maxInodeID + Δ          ▷ curoff the inode range
9:          mp.end ← end
10:         task ← newSplitTask(c.Name, mp.partitionID, end)
11:
12:         c.addTask(task)               ▷ sync with the meta node
13:
14:         c.updateMetaPartition(mp.volName, mp)
15:         c.createMetaPartition(mp.volName, mp.end+1)
```

2.5 Optimizations

There are two main optimizations in CFS due to the extremely large scale of CFS clusters deployed in JD. The details are as explained as follows:

Minimizing Heartbeats. Because our production environment could have millions of partitions spread across many meta and data nodes, even with the MultiRaft-based protocol, the resource manager can still get an explosion of the heartbeats from these nodes, causing it to become a bottleneck. To alleviate this, we employ an extra layer of abstraction on the nodes called *Raft set* to further minimize the number of heartbeats to be exchanged among the Raft groups. Specifically, we divided all the nodes into several Raft set, each of which maintains its own Raft group. When creating a new partition, we prefer to select the replicas from the same Raft set. In this way, each node only needs to exchange heartbeats with the nodes in the same Raft set.

Non-Persistent Connections. There could be tens of thousands of clients accessing the same CFS cluster. As a result, a non-persistent connection is used between each client and the resource manager to reduce the network overheads.

3 METADATA OPERATIONS

In most modern POSIX-compliant file systems [27], the inode and dentry of a file usually reside on the same storage node in order to preserve the directory locality[7]. However, because of the utilization-based metadata distribution and placement, the inode and dentry of the same file may be distributed across different metadata nodes in CFS. As a result,

[7]Directory locality is a term referring to the phenomenon that the metadata of files under the same directory are likely to be accessed repeatedly, i.e., when a file metadata is accesses, it is likely that the metadata of the files under the same directory, or the directory metadata will also be accessed.

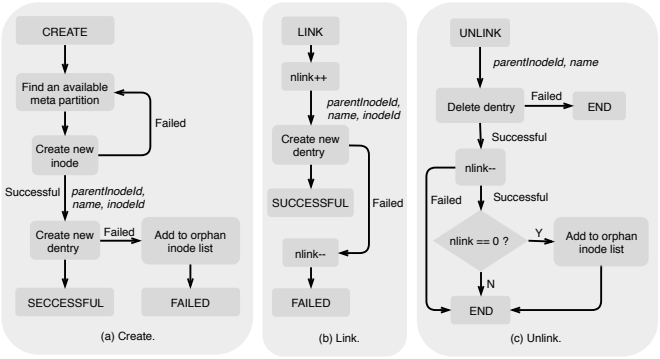

Figure 3: Workflows of three commonly used metadata operations.

operating a file's inode and dentry usually needs to run as distributed transactions, which brings the overheads that could significantly affect the performance.

Our tradeoff is to relax this atomicity requirement *as long as a dentry is always associated with at least one inode*. All the metadata operations in CFS are based on this design principle. The downside is the chance to create *orphan inodes*[8] on the meta node, which may be difficult to be freed from the memory. To mitigate this issue, each metadata-operation workflow in CFS has been carefully designed to *minimize the chance of an orphan inode to appear*. In practice, a meta node rarely has too many orphan inodes in the memory. But if this happens, tools like *fsck* can be used to repair the files by the administrator.

Figure 3 illustrates the workflows of three common metadata operations, which can be explained as follows.

Create. When creating a file, the client first asks an available meta node to create an inode. The meta node picks up the smallest inode id that has not been used so far in this partition for the newly created inode, and updates its largest inode id accordingly. Only when the inode has been successfully created, the client can ask for creating a corresponding dentry. Note that the inode and dentry of the same file do not need to be stored on the same meta node. If a failure happens when creating the dentry, the client will put the newly created inode into a local list of orphan inodes, who will be deleted when the meta node receives an evict request from the client.

Link. When linking a file, the client first asks the meta node of the inode to increase the value of *nlink* (the number of associated links) by one, and then asks the meta node of the target parent inode to create a dentry on the same meta partition. If a failure happens when creating the dentry, the value of *nlink* will be decreased by one.

[8]An orphan inode is an inode that has no dentry to be associated with.

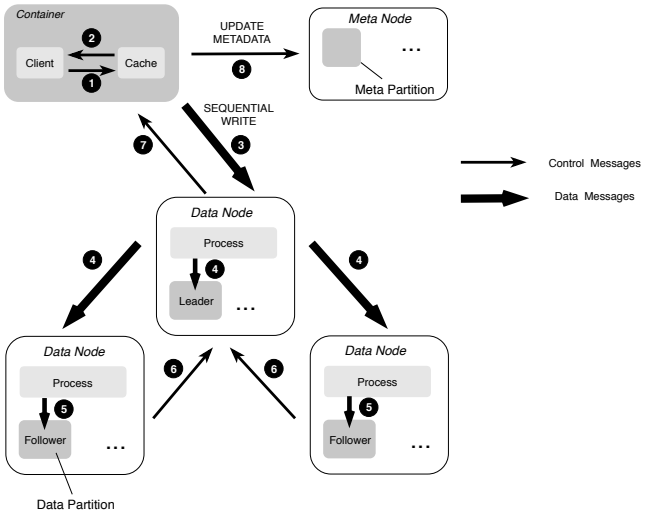

Figure 4: Workflow of sequential write.

Unlink. When unlinking a file, the client first asks the corresponding meta node to delete the dentry. Only when this operation succeeds, the client sends an *unlink* request to the meta node to decrease the value of *nlink* in the target inode by one. When it becomes zero, the client puts this inode into a local list of orphan inodes, who will be deleted when the meta node receives an evict request from the client. Note that, if decreasing the value of *nlink* fails, the client will perform several retries. If all the retries failed, this inode will eventually become an orphan inode, and the administrator may need to manually resolve the issue.

4 FILE OPERATIONS

CFS has relaxed POSIX consistency semantics, i.e., instead of providing strong consistency guarantees, it only ensures sequential consistency for file/directory operations, and does not have any leasing mechanism to prevent multiple clients writing to the same file/directory. It depends on the upper-level application to maintain a more restrict consistency level if necessary.

Sequential Write. As shown in Figure 4, to sequentially write a file, the client first randomly chooses the available data partitions from the cache, then continuously sends a number of fixed sized packets (e.g., 128 KB) to the leader, each of which includes the addresses of the replicas, the target extent id, the offset in the extent, and the file content. The addresses of the replicas are provided as an array by the resource manager and cached on the client side. The indices of the items in this array tell the order of the replication, i.e.,

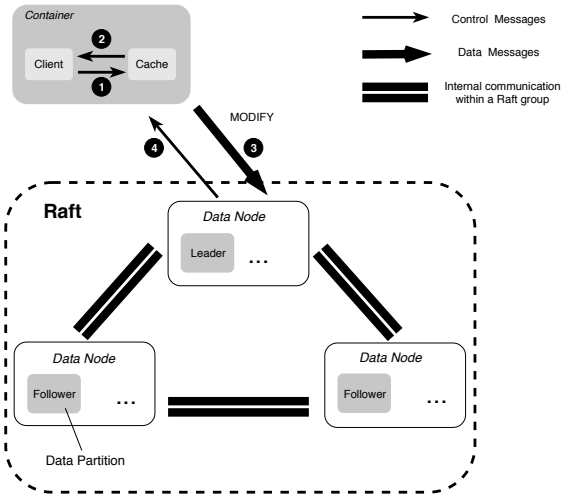

Figure 5: Workflow of overwriting an existing file (no appending).

the replica whose address at the index zero is the leader. For this reason, the client can always send a write request to the leader without introducing extra communication overhead, and as mentioned in the previous section, a primary-backup replication will be performed by following this order. Once the client receives the commit from the leader, it updates the local cache immediately, and synchronizes with meta node periodically or upon receiving a system call, *fsync()*[9], from the upper level application.

Random Write. The random write in CFS is in-place. To randomly write a file, the client first uses the offsets of the original data and the new data to figure out the portions of data to be appended and the portions of data to be overwritten, and then processes them separately. In the former case, the client sequentially writes the file as described previously. In the latter case, as shown in Figure 5, the offset of the file on the data partition does not change. The client continuously sends packets with fixed size (128 KB by default) to the leader, where each packet includes the target extent id, the offset in the extent, and the file content.

Delete. The delete operation is asynchronous. To delete a file, the client sends a delete request to the corresponding meta node. The meta node, once receives this request, marks the target inode as deleted (see the inode structure given in Section 2.1). Later on, there will be a separate process to clear up this inode and communicate with the data node to delete the file content.

[9]http://man7.org/linux/man-pages/man2/fdatasync.2.html

Read. A read can only happen at the Raft leader (the primary-backup group leader and raft group leader could be different). To read a file, the client sends a read request to the corresponding data node. This request is constructed by the data from the client cache, such as the data partition id, the extent id, the offset of the extent, etc.

5 DESIGN CHOICES

In this section, we highlight some of the design choices we have made when building CFS.

5.1 Centralization vs. Decentralization

Centralization and decentralization are two design paradigms for distributed systems [8]. Although the former one [11, 23] is relatively easy to implement, the single master could become a bottleneck in the sense of the scalability. In contrast, the latter [9] one is generally more reliable but also more complex to implement.

In designing CFS, we choose centralization over decentralization mainly for the reason of its simplicity. However, a single resource manager that stores all the file metadata limits the scalability of the metadata operations, which could make up as much as half of typical file system workloads [19]. For this reason, we employ a separate cluster to store the metadata, which drastically improves the scalability of the entire file system. From this perspective, CFS is designed in a way to minimize the involvements of the resource manager so that it has less chance to become a bottleneck. Admittedly, even with these efforts, the scalability of the resource manager could still be limited by its memory and disk space. But based on our experience, this never becomes an issue.

5.2 Separate Meta Node vs. Metadata on Data Node

In some distributed file systems, the file metadata and contents are stored on the same machine [16, 27]. There are also some distributed file systems where the metadata is managed separately by specialized metadata servers [12, 23].

In CFS, we choose to have a separate meta node and maintain all the file metadata in the memory for fast access. As a result, we can select memory-intensive machines for the meta nodes, and disk-intensive machines for the data nodes for cost-effectiveness. Another advantage of this design is the flexibility of deployment. In case a machine has both large memory and disk spaces, we can always deploy the meta node and data node physically together.

5.3 Consistency Model and Guarantees

In CFS, the storage layer and file system layer have different consistency models.

Table 1: System specification.

Processor Number	Xeon E5-2683V4
Number of Cores	16
Max Turbo Frequency	3.00 GHz
Processor Base Frequency	2.10 GHz
Network Bandwidth	1000Mbps
Memory	DDR4 2400MHZ, 8 × 32 GB
Disk	16 × 960 GB SSD
Operating System	Linux 4.17.12

The storage engine guarantees the strong consistency among the replicas through either primary-backup or Raft-based replication protocols. This design decision is based on the observations that the former one is not suitable for overwrite as either the data consistency cannot be guaranteed or the replication performance needs to be compromised, and the latter one has write amplification issue as it introduces extra IO of writing the log files.

The file system itself, although provides POSIX-compliant APIs, has selectively relaxed POSIX consistency semantics in order to better align with the needs of applications and to improve system performance. For example, the semantics of POSIX defines that writes must be strongly consistent, i.e, a write is required to block application execution until the system can guarantee that any other read will see the data that was just written. While this can be easily accomplished locally, ensuring such strong consistency on a distributed file system is very challenging due to the degraded performance associated with lock contentions/synchronizations. CFS relaxes the POSIX consistency in a way that provides consistency when different clients modify non-overlapping parts of a file, but it does not provide any consistency guarantee if two clients try to modify the same portion of the file. This design is based on the fact that in a containerized environment, there are many cases where the rigidity of POSIX semantics is not strictly necessary, i.e., the applications seldom rely on the file system to deliver full strong consistency, and two independent jobs rarely write to a common shared file on a multi-tenant system.

6 EVALUATION

In this section, we first evaluate and analyze the performance and scalability of CFS and Ceph, a distributed file system that has been widely used on the container platform.

6.1 Experiment Setup

For CFS, we deploy the meta nodes and data nodes on the same cluster of 10 machines, and a single resource manager with 3 replicas. For Ceph, we have similar setup, where the object storage devices (OSD) and metadata server (MDS) are deployed on the same cluster of 10 machines. The Ceph version is 12.2.9, and the storage engine is configured as the *bluestore* with the TCP/IP network stack. In both setups of CFS and Ceph, the client nodes are deployed separately with the page cache turned off. All these file systems are deployed on the machines with the same specification, which is given in Table 1.

Table 2: Explanation of Tested Metadata Operations.

DirCreation	Create a directory
DirStat	List all the files in the current directory
DirRemoval	Remove a directory
FileCreation	Create a file
FileRemoval	Remove the file attributes
TreeCreation	Create a directory with multiple files as a tree structure
TreeRemoval	Remove a directory with multiple files as a tree structure

Table 3: IOPS for the metadata operations with 8 clients where each client has 64 processes.

Test Name	CFS (multi)	Ceph (multi)	% of Improv.
DirCreation	83,729	16,627	404%
DirStat	875,867	91,050	862%
DirRemoval	94,235	23,807	296%
FileCreation	85,556	21,919	290%
FileRemoval	50,119	22,573	122%
TreeCreation	10	11	-9%
TreeRemoval	12	3	300%

6.2 Metadata Operations

In evaluating the performance and scalability of the metadata subsystem, we focus on the tests of 7 commonly used metadata operations from *mdtest*[10]. The description is given in Table 2. To align with Ceph, the cache expiration time at the client is set to 30 seconds in CFS.

Figure 6 plots the IOPS of these tests in a single-client environment with different number of processes, and Figure 7 plots the IOPS of the same set of tests in a multi-client environment, where each client runs 64 processes. Note that the Y-axis uses the *logarithmic scale* for better illustration with different test results.

It can be seen that, when there is only a single client with a single process, Ceph outperforms CFS in 5 out of 7 tests (except the *DirStat* and *TreeRemoval* tests), but as the number of clients and processes increase, CFS starts to catchup. When it comes to 8 clients, where each client runs 64 processes, CFS outperforms Ceph in 6 out of 7 tests (except the *TreeCreation* test). The detailed IOPS of the tests with 8 clients is given in

[10]https://github.com/hpc/ior

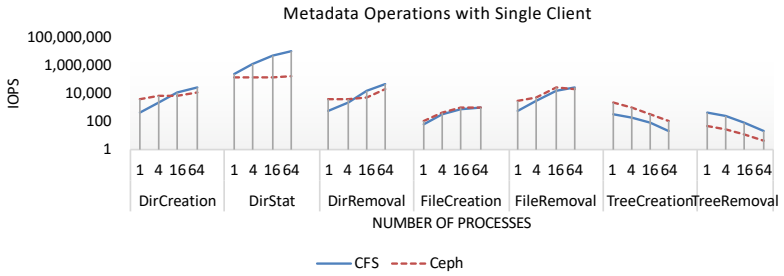

Figure 6: IOPS when a single client operates the file metadata.

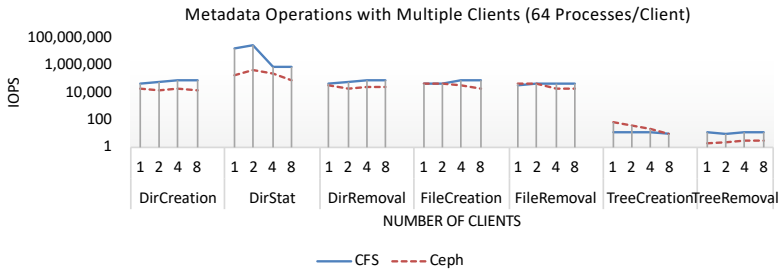

Figure 7: IOPS when multiple clients operate the file metadata.

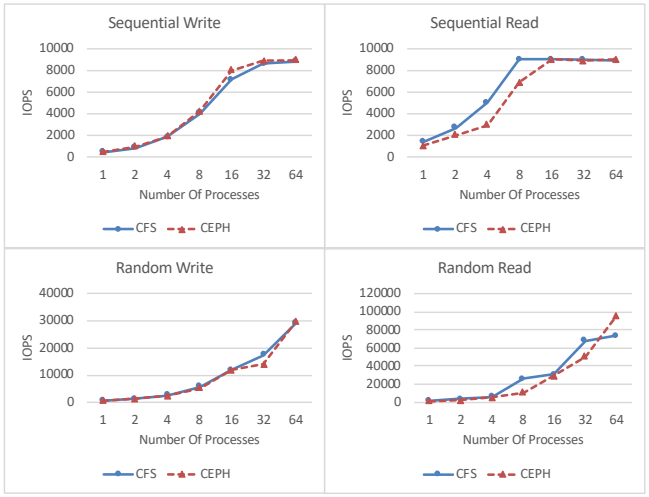

Figure 8: IOPS when different number of processes, in a single client, operate files with different access patterns (each process operates a 40 GB file).

Table 3, which shows about 3 times performance boost on the average. From this result we can see that, as the number of clients and processes gets increased, the performance benefits from the utilization-based metadata distribution in CFS can probably outweigh the advantage from the directory locality-aware metadata distribution in Ceph.

There are a few observations from the results of *DirStat*, *TreeCreation* and *TreeRemoval*, as explained below.

In the *DirStat* test, CFS exhibits better performance than Ceph in both two cases (i.e., single-client and multi-client). This is mainly because that they handle the *readdir* request in different ways. In Ceph, each *readdir* request is followed by a set of *inodeGet* requests to fetch all the inodes in the current directory from different MDS. The requested inodes are usually cache in the MDS for the fast access in the future. However, in CFS, these *inodeGet* requests are replaced by a *batchInodeGet* request in order to reduce the communication overheads, and the results are cached at the client side so that the successive requests can be quickly responded without further communications with the same set of meta nodes. The sudden performance drops in the multi-client case (see Figure 7) can be caused by the client cache misses in CFS.

In the *TreeCreation* test, Ceph performs better than CFS in the single-client case. But as more clients get involved, the performance gap between them gets closer. This can be caused by the fact that, when there are only have a few clients, the benefits from the directory locality such as reusing the cached dentries and inodes on the same MDS gives Ceph a better performance. However, as more clients get involved, the increasing pressure on certain MDS requires Ceph to dynamically place the metadata of files under the same directory into different MDSs and redirect the corresponding requests to the proxy MDSs [29], which incurs extra overheads and closes the gap between Ceph and CFS.

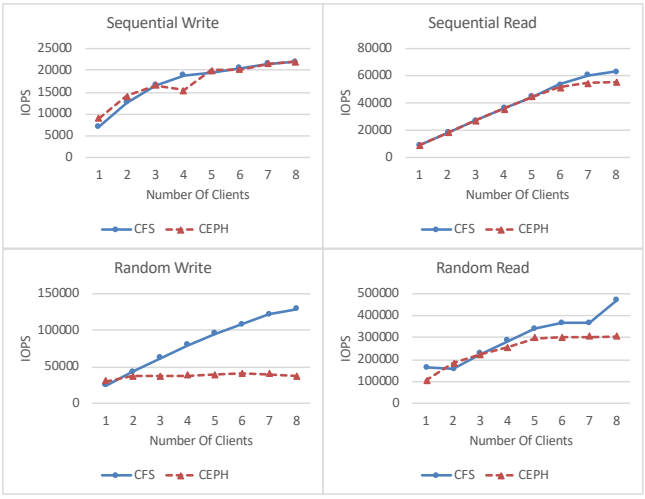

Figure 9: IOPS when different number of clients, each of which has 64 processes, operate files with different access patterns (each process operates a 40 GB file).

In the *TreeRemoval* test, CFS gives better performance than Ceph in both cases. Similar to the *DirStat* test, the way CFS handles the *readdir* request can be one of the reasons for such a result. In addition, when there are only a few clients, the requests of deleting the file metadata may need to be queued in Ceph as the metadata of the files under the same directory are usually stored on a single MDS; and when more clients get involved, the potential benefits from the directory locality in Ceph may be reduced since the file metadata may have been distributed across different MDSs because of the rebalancing triggered in the previous tests.

6.3 Large Files

To study the performance and scalability of CFS and Ceph when operating large files, let us first look at the results with different number of processes in a single-client environment, where each process operates a separate 40 GB file. We use *fio*[11] to generate various types of workloads. Here two parameters need to be tuned in Ceph in order to obtain the optimal performance, namely, the *osd_op_num_shards* and *osd_op_num_threads_per_shard*, which control the number of queues and the number of threads to process the queues. We set them to 6 and 4 respectively. Increasing any of these values further will cause degraded write performance due to the high CPU pressure.

As shown in Figure 8, the performance under different processes are quite similar, with the exception that in sequential read test, CFS has higher IOPS when the number of processes

[11]https://github.com/axboe/fio

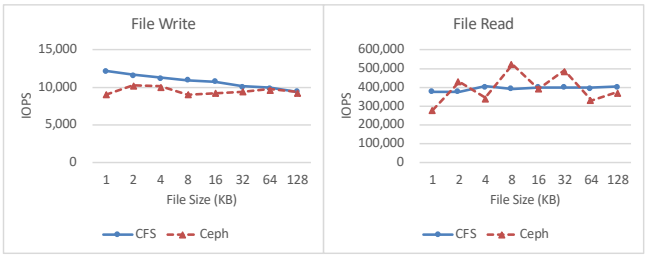

Figure 10: IOPS when 8 clients, each of which has 64 processes, operate small files with different sizes.

is less than 8. This is probably caused by the queuing mechanism in Ceph, where each process is usually associated with several queues. When there are limited number of processes, the number of queues that process the requests in parallel are also limited, which affects the sequential read performance.

Next, we study the performance and scalability of CFS and Ceph in a multi-client environment, where each client has 64 processes. Similarly, we use *fio* to perform this evaluation and each process operates a separate 40 GB file. As can be seen in Figure 9, in both sequential and random read tests, CFS outperforms Ceph as the number of clients increases. In addition, CFS has significant performance advantage over Ceph in the random write test. This is probably due to the following reasons. First, the overwrite in CFS is in-place, which does not need to update the file metadata. In contrast, the overwrite in Ceph usually needs to walk through multiple queues, and only after the data and metadata have been persisted and synchronized, the commit message can be returned to the client. Second, a latency breakdown in Ceph shows that *state_deferred_cleanup_lat* gives the costliest portion in these test, which is defined as the time spending on deleting the log files after the write-ahead log has been synchronized. In other words, clearing up the logs in Ceph during the random write directly affects write performance.

6.4 Small Files

In this section we study the performance of operating small files with various sizes from 1 KB to 128 KB in CFS and Ceph. Similar to the metadata test, the results are obtained from *mdtest*. This experiment simulates the use case when operating product images, which are usually never modified once created. In our CFS configuration, 128 KB is the threshold set to determine if the file should be aggregated in a single extent or not, i.e., if we should treat it as "small file" or not. As can be seen from Figure 10, CFS and Ceph have similar write performance in file write. However, in file read, CFS looks much stable than Ceph. This is probably due to the fact that, in Ceph's *bluestore* backend, RocksDB is used to persist the metadata and the log files, which employs several

layers of cache to speed up the data access. The fluctuation can be caused by the cache misses at certain cache layer.

7 RELATED WORK

GFS [11] and its open source implementation HDFS [23] are designed for storing large files with sequential access. Both of them adopt the master-slave architecture [25], where the single master stores all the file metadata. Unlike GFS and HDFS, CFS employs a separate metadata subsystem to provide a scalable solution for the metadata storage so that the resource manager has less chance to become the bottleneck.

Haystack [3] takes after log-structured filesystems [20] to serve long tail of requests seen by sharing photos in a large social network. The key insight is to avoid disk operations when accessing metadata. CFS adopts similar ideas by putting the file metadata into the main memory. However, different from haystack, the actually physical offsets instead of logical indices of the file contents are stored in the memory, and deleting a file is achieved by the punch hole interface provided by the underlying file system instead of relying on the garbage collector to perform merging and compacting regularly for more efficient disk utilization. In addition, Haystack does not guarantee the strong consistency. Several works [10, 30] have proposed to manage small files and metadata more efficiently by grouping related files and metadata together intelligently. CFS takes a different design principle to separate the storage of file metadata and contents to have a more flexible and cost-effective deployment.

Windows Azure Storage (WAS) [7] is a cloud storage system that provides strong consistency and multi-tenancy to the clients. Different from CFS, it builds an extra partition layer to handle random writes before streaming data into the lower level. AWS EFS [21] is a cloud storage service that provides scalable and elastic file storage. We could not evaluate the performance of Azure and AWS EFS comparing with CFS as they are not open-sourced.

PolarFS [8] utilizes a lightweight network stack and I/O stack to take advantage of the emerging techniques like RDMA, NVMe, and SPDK. OctopusFS [15] is based on HDFS with automated data-driven policies for managing the placement and retrieval of data across the storage tiers of the cluster. These are orthogonal to our work as CFS can also adopt similar techniques to fully utilize the emerging hardware and storage hierarchies.

There are a few distributed file systems that have been integrated with Kubernetes [13, 24, 27]. Ceph [27] maximizes the separation between data and metadata management and employs a dynamic distributed metadata cluster to manage different workloads. We have performed a comprehensive comparison with Ceph through this paper. GlusterFS [13] is a scalable distributed file system that aggregates disk storage resources from multiple servers into a single global namespace. MooseFS [24] is a fault-tolerant, highly available, POSIX-compliant distributed file system that employs a single master to manage the file metadata.

MapR-FS[12] is a POSIX-compliant distributed file system that provides reliable, high performance, scalable, and full read/write data storage. Similar to Ceph, it stores the metadata in a distributed way alongside the data itself.

8 CONCLUSIONS AND FUTURE WORK

This paper describes CFS, a distributed file system designed for serving JD's e-commence business on one of the world's largest Kubernetes platform.

We have implemented most of the operations by following the POSIX standard, and are actively working on the remaining ones, such as *xattr, fcntl, ioctl, mknod* and *readdirplus*. In the future, we plan to take advantage of the Linux page cache to speed up the file operations, improve the file locking mechanism and cache coherency, and support emerging hardware standards such as RDMA. We also plan to develop our own POSIX-compliant file system interface in the kernel space to completely eliminate the overhead from FUSE.

REFERENCES

[1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33, 3 (May 2016), 42–52. https://doi.org/10.1109/MS.2016.64

[2] Moshe Bar. 2001. *Linux File Systems.* McGraw-Hill Professional.

[3] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10).* USENIX Association, Berkeley, CA, USA, 47–60. http://dl.acm.org/citation.cfm?id=1924943.1924947

[4] David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (Sept. 2014), 81–84. https://doi.org/10.1109/MCC.2014.51

[5] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. 2003. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03) (MSS '03).* IEEE Computer Society, Washington, DC, USA, 290–. http://dl.acm.org/citation.cfm?id=824467.825004

[6] Eric A. Brewer. 2015. Kubernetes and the Path to Cloud Native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15).* ACM, New York, NY, USA, 167–167. https://doi.org/10.1145/2806777.2809955

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas.

[12] https://mapr.com/products/mapr-fs/

2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571

[8] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1849–1862. https://doi.org/10.14778/3229863.3229872

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[10] Gregory R. Ganger and M. Frans Kaashoek. 1997. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '97)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1268680.1268681

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[12] Garth A. Gibson and Rodney Van Meter. 2000. Network Attached Storage Architecture. *Commun. ACM* 43, 11 (Nov. 2000), 37–45. https://doi.org/10.1145/353360.353362

[13] Red Hat. 2005. Gluster File System. "https://github.com/gluster/glusterfs".

[14] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[15] Elena Kakoulli and Herodotos Herodotou. 2017. OctopusFS: A Distributed File System with Tiered Storage Management. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 65–78. https://doi.org/10.1145/3035918.3064023

[16] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. 1986. Andrew: A Distributed Personal Computing Environment. *Commun. ACM* 29, 3 (March 1986), 184–201. https://doi.org/10.1145/5666.5671

[17] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[18] Claus Pahl. 2015. Containerization and the PaaS Cloud. *IEEE Cloud Computing* 2, 3 (May-June 2015), 24–31. https://doi.org/10.1109/MCC.2015.51

[19] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. 2000. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1267724.1267728

[20] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[21] Amazon Web Service. 2016. Amazon Elastic File System. "https://docs.aws.amazon.com/efs/latest/ug/efs-ug.pdf".

[22] Kai Shen, Stan Park, and Men Zhu. 2014. Journaling of Journal Is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX, Santa Clara, CA, 287–293. https://www.usenix.org/conference/fast14/technical-sessions/presentation/shen

[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[24] Core Technology. 2016. MooseFS 3.0 Storage Classes Manual. "https://moosefs.com/Content/Downloads/moosefs-storage-classes-manual.pdf".

[25] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. 1997. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 224–237. https://doi.org/10.1145/268998.266694

[26] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=1251254.1251261

[27] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 307–320. http://dl.acm.org/citation.cfm?id=1298455.1298485

[28] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07 (PDSW '07)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1374596.1374606

[29] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*. IEEE Computer Society, Washington, DC, USA, 4–. https://doi.org/10.1109/SC.2004.22

[30] Zhihui Zhang and Kanad Ghose. 2007. hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 175–187. https://doi.org/10.1145/1272996.1273016