Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

GlobalFS: A Strongly Consistent Multi-Site File System

Leandro Pacheco[1], Raluca Halalai[2], Valerio Schiavoni[2], Fernando Pedone[1], Etienne Rivière[2], Pascal Felber[2]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] Institut d'informatique, Université de Neuchâtel, Switzerland

Abstract

This paper introduces GlobalFS, a POSIX-compliant geographically distributed file system. GlobalFS builds on two fundamental building blocks, an atomic multicast group communication abstraction and multiple instances of a single-site data store. We define four execution modes and show how all file system operations can be implemented with these modes while ensuring strong consistency and tolerating failures. We describe the GlobalFS prototype in detail and report on an extensive performance assessment. We have deployed GlobalFS across all EC2 regions and show that the system scales geographically, providing performance comparable to other state-of-the-art distributed file systems for local commands and allowing for strongly consistent operations over the whole system. The code of GlobalFS is available as open source.

Report Info

Published
April 2016
Number
USI-INF-TR-2016/01
Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
Online Access
www.inf.usi.ch/techreports

1 Introduction

Cloud infrastructures, composed of multiple interconnected datacenters, have become an essential part of modern computing systems. They provide an efficient and cost-effective solution to hosting web-accessible services, storing and processing data, or performing compute-intensive tasks. Large companies like Amazon or Google do not only use such architectures for their own needs, but they also rent them to external clients in a variety of flavors, e.g., infrastructure (IaaS), platform (PaaS), software (SaaS), or data (DaaS) as a service. Such global infrastructures rely on geographically distributed datacenters for fault-tolerance, scalability, and performance reasons.

We focus in this work on the design of a geographically distributed file system, accessible via a POSIX-compliant API. Most previous designs for geographically distributed file systems [27, 37] have provided weak consistency guarantees (e.g., eventual consistency [16]) to work around the limitations formalized by the CAP theorem [23], which states that distributed applications can fully support at most two of the following three properties simultaneously: consistency, availability, and tolerance to partitions. Our goal is to ensure strongly consistent file system operations despite node failures, at the price of possibly reduced availability in the event of a network partition. Weak consistency is suitable for domain-specific applications where programmers can anticipate and provide resolution methods for conflicts, or work with last-writer-wins resolution methods. Our rationale is that for general-purpose services such as a file system, strong consistency is more appropriate as it is both more intuitive for the users and does not require human intervention in case of conflicts.

Strong consistency requires ordering commands across replicas, which needs coordination among nodes at geographically distributed sites (i.e., regions). Designing strongly consistent distributed systems that provide good performance requires careful tradeoffs. The original approach we explore in this work is to trade the performance of global operations, spanning multiple regions, for the scalability of intra-region

operations. We capture this compromise with the notion of *geographical scalability*.

Geographical scalability is motivated by geo-distributed applications that wish to exploit locality without compromising consistency or reducing the scope of operations to a single region. This trend is becoming increasingly more important with the wide range of applications that are deployed over multiple datacenters spanning several regions, e.g., on Amazon EC2. Yet, achieving geographical scalability is notoriously difficult. For example, among the few existing file systems with support for geographical distribution, CalvinFS [60] totally orders requests. As a consequence, performance decreases with the number of regions in the system, even for operations that access objects in a single region.

This paper introduces GlobalFS, a file system that achieves geographical scalability by exploiting two abstractions. First, it relies on *data stores* located in geographically distributed datacenters. Files are replicated and stored as immutable blocks in the data stores, which are organized as distributed hash tables (DHTs). Second, GlobalFS uses an *atomic multicast* abstraction to maintain mutable file metadata and orchestrate multi-site operations. Atomic multicast provides strong order guarantees by partially ordering operations.

GlobalFS notably differs from other distributed file systems by defining a flexible partition model in which files and folders can be placed according to access patterns (e.g., in the same region as their most frequent users), as well as four execution modes corresponding to the operations that can be performed in the file system: (1) single-partition operations, (2) multi-partition uncoordinated operations, (3) multi-partition coordinated operations, and (4) read-only operations. While single-partition and read-only operations can be implemented efficiently by accessing a single region, the other two operations require interactions across multiple regions. By leveraging atomic multicast and distinguishing between these four modes of execution, GlobalFS can provide low latency for single-region commands while allowing for consistent operations across regions. GlobalFS can therefore exploit geographical locality in new ways to combine performance and consistency, and hence propose original contributions in the well-studied design space of distributed file systems.

We have implemented a complete prototype of GlobalFS and deployed it on Amazon's EC2 platform with nodes spread all over the world, across *all nine* available regions. We have conducted an in-depth study of its performance. Results show that GlobalFS outperforms other geographically distributed file systems that offer comparable guarantees and delivers good performance for single-site commands. The code of GlobalFS is freely available as open source.[1]

The rest of this paper is organized as follows. Sections 2 and 3 introduce GlobalFS's system model and architecture, respectively. Section 4 presents the protocol design. Section 5 describes the implementation of our prototype. Section 6 discusses results of experimental evaluation. Section 7 reviews related work and Section 8 concludes.

2 System model and definitions

We assume a distributed system composed of interconnected processes that communicate by message passing. There is an unbounded set of *client processes* and a bounded set of *server processes*. Processes may fail by crashing, but do not experience arbitrary behavior (i.e., no Byzantine failures).

Client and server processes are grouped within *datacenters* that are geographically distributed over different *regions*. Processes in the same region experience low-latency communication, while messages exchanged between processes located in different regions are subject to larger latencies. Links are quasi-reliable: if both the sender and the receiver are non-faulty, then every message sent is eventually received.

The system is *partially synchronous* [17]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *global stabilization time* (GST) and is unknown to the processes. Before the GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After the GST, such bounds exist but are unknown. In practice, "forever" means long enough for the atomic multicast protocol to make progress (i.e., deliver messages).

GlobalFS ensures sequential consistency for update operations and causal consistency for reads. A system is *sequentially consistent* if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands as defined in their sequential specification, and (ii) respects the ordering of commands as defined by each client [5]. A system is *causally consistent* if the result of read operations respect the causal ordering of events as defined by the "happens-before" relation [29].

[1] https://github.com/pacheco/GlobalFS

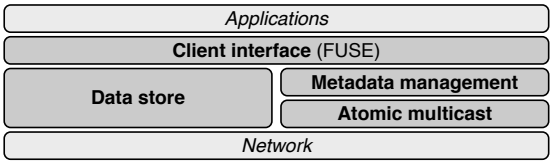

Figure 1: Overall architecture of GlobalFS.

3 System architecture

This section presents the overall architecture of GlobalFS and how the file system can be partitioned and replicated across datacenters.

3.1 Components

The architecture of GlobalFS consists of four components: the client interface, the data store, metadata management, and atomic multicast (see Figure 1).

The *client interface* provides a file system API supporting a subset of POSIX 1-2001 [1]. GlobalFS implements file system operations sufficient to manipulate files and directories. Some file system calls change the structure of the file system tree (i.e., the files and directories within each directory). Each file descriptor seen by a client when opening a file is mapped to a local file descriptor at each GlobalFS server. We support file-specific operations: `mknod`, `unlink`, `open`, `read`, `write`, `truncate`, `symlink`, `readlink`; directory-specific operations: `mkdir`, `rmdir`, `opendir`, `readdir`; and general purpose operations: `stat`, `chmod`, `chown`, `rename`, and `utime`. We support symbolic links, but not hard links.

Like most contemporary distributed file systems (e.g., [22, 53, 10, 51]), GlobalFS decouples metadata from data storage. Metadata in GlobalFS is handled by the *metadata management* layer. Each file has an associated *inode* block (`iblock`) containing the metadata information about the file (e.g., its size, owner, and access rights) and pointers for its data blocks. The actual content of a file is stored in data blocks (`dblocks`). The two types of blocks are handled differently and stored separately: `dblocks` are immutable and stored by the clients in the storage servers; `iblocks` are mutable and maintained by the metadata servers.

GlobalFS distinguishes updates (i.e. operations that modify the state of a file or directory) from read-only operations. Updates are sequentially consistent while reads are causally consistent (see Section 2). Every update operation is ordered by *atomic multicast* [34]. Atomic multicast is a one-to-many communication abstraction that implements the notion of *groups*. Servers subscribe to one or more groups and every message multicast to a group g will be delivered by processes that subscribe to g. Let relation $<$ be defined such that $m < m'$ iff there is a process that delivers message m before message m'. Atomic multicast ensures that (i) if a process delivers m, then all non-faulty processes that subscribe to the same group deliver m *(agreement)*; and (ii) relation $<$ is acyclic *(order)*. The (partial) order property implies that if processes p and q deliver messages m and m', then they deliver them in the same order.

It is important to understand the difference between atomic broadcast, as implemented by Paxos [30] and its variants (e.g., [35, 31, 40]), and atomic multicast. With atomic broadcast, for every pair of delivered messages m and m', either $m < m'$ or $m' < m$. With atomic multicast, it is possible that neither $m < m'$ nor $m' < m$. This is the case, for example, if m and m' are multicast to groups g and g', respectively, and no process subscribes to both groups. Partially ordering messages, as defined by atomic multicast, is a fundamental requirement for achieving scalable distributed systems.

The *data store* provides a linearizable key-value store with primitives to read (`get`) and create (`put`) data items. It is implemented as a collection of distributed hash tables (DHTs), with one instance of the data store per datacenter. Maintenance of the data in the DHT is simple and efficient given that data blocks are immutable. DHT-based data stores scale remarkably well horizontally [28, 16, 46, 14].

3.2 Partitioning and replication

Data partitioning and replication have an important impact on the performance and reliability of a data management system. Horizontal partitioning (sharding) is commonly used to scale distributed file systems. For example, hashing the pathname of each file is a straightforward way to distribute files across the system [60]. Hashing provides good load distribution of files but its lack of support for locality might place

Partition	Replication	Performance	Fault tolerance
Global	across regions	best for reads	disaster
Local	within region	best for reads & writes	datacenter crash

Table 1: Partitions in GlobalFS.

files far away from their most frequent or likely users. Although GlobalFS supports any partitioning scheme, including hashing, we explore a different approach to partitioning and replication, which takes locality into consideration, as we now explain.

The file system is partitioned and replicated according to the expected client access patterns and the degree of fault tolerance desired. Files that are mostly read and rarely modified (e.g., system and application programs) are placed in a single "global" partition, replicated across regions; files that experience locality of access (e.g., temporary files related to a client) are placed in "local" partitions, replicated in datacenters inside a single region, close to the clients most likely to access them. In this setup, a file in the global partition can be read from any region, resulting in high throughput and low latency for read operations. Updating a file in the global partition, however, involves all regions. Local partitions, on the other hand, can provide high throughput and low latency for both reads and updates, as long as the client is close to its location. Both local and global partitions can tolerate the failure of an entire datacenter. Moreover, the global partition can tolerate the failure of all datacenters in a region (i.e., a disaster). Table 1 summarizes the two partition types in GlobalFS.

To allow for flexible system deployment, GlobalFS decouples data from metadata. Although data and metadata are likely to be stored in the same servers, the system can cope with the case in which the metadata of a file is stored in a region and the file data is stored in a different region. This is useful, for example, to migrate large files from one region to another. The metadata, which is typically small, can be quickly moved from one region to another—hence completing the operation—while the data follows asynchronously.

3.3 Use of atomic multicast

In order to allow operations to be consistently propagated to the replicas, one multicast group is associated with each partition. Servers subscribe to two multicast groups: one, g_{all}, associated with all the servers in the system, and another associated with servers in the datacenters in the same region.[2] Commands that update files in the global partition or update files in multiple local partitions are multicast to g_{all}; commands that update files in a local partition are multicast to the group associated with the partition. The use of atomic multicast allows for independent local partitions while still providing consistent operations across them. Section 4 describes in detail how this is achieved by GlobalFS.

3.4 Example deployment

Consider a deployment involving three regions, R_1, R_2, and R_3, each with three datacenters. The file system is partitioned in four partitions, $P_0, ..., P_3$ (see Figure 2), such that P_0 is replicated in datacenters in all regions and partition P_i, $1 \leq i \leq 3$, is replicated in datacenters in region R_i. In this scenario, we have clients and servers (metadata and data store) distributed across the regions, that is, in addition to the metadata associated with the region's partition, each datacenter also hosts an instance of the data store. More precisely, the metadata for the directory `/1` and all its contents (recursively) are stored only in P_1. In the same manner, `/2` and `/3` are respectively mapped to P_2 and P_3. Files not contained in any of these directories (e.g., `/`, `/bin`, `/etc`) are in partition P_0.

4 Protocol design

GlobalFS differentiates four classes of operations and defines for each one a different execution mode. GlobalFS's execution modes provide the basis for the implementation of each file system operation. We start by going through the details of each execution mode. We then describe the execution `open`, `read` and `write` operations, from start to finish. Finally, we discuss how failures are handled in GlobalFS.

[2] Note that the setup of multicast groups is flexible and other configurations could be used to adapt for instance to the network topology, the workload, or specific performance/consistency requirements.

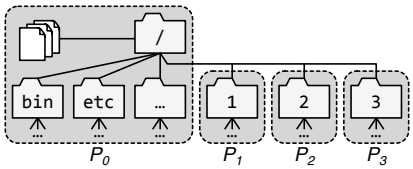

Figure 2: Illustrative deployment of GlobalFS with 4 partitions. Partition P_0 is replicated in all regions and each other partition is replicated in one different region.

4.1 Execution modes

Each operation in GlobalFS follows one of the following execution modes. Except for `read` and `write` operations, all file system operations access only the metadata servers.

Single-partition operations. A single-partition operation modifies metadata stored in a single partition. As a consequence, operations in this class are multicast to the group associated with the concerned partition and, when delivered, executed locally by the replicas. The execution of a single-partition operation follows state-machine replication [50]: each replica delivers a command and executes it deterministically. One of the replicas replies to the client.

The following operations are single-partition in GlobalFS, where the terms *child* and *parent* are used to refer to a node and the directory that contains it.

- `chmod`, `chown`, `truncate`, `open`, and `write`;
- `mknod`, `unlink`, `symlink`, and `mkdir` when the parent and child are in the same partition; and
- `rename`, when the origin, origin's parent, destination, and destination's parent are in the same partition.

Note that while a single-partition operation in a local partition involves only servers in one region, a single-partition operation in the global partition (multicast to group g_{all}) involves servers in all regions of the system.

Uncoordinated multi-partition operations. An uncoordinated multi-partition operation accesses metadata in more than one partition, but the operation's execution at each partition can complete without any input from the other partitions involved. The partial ordering of atomic multicast is sufficient to guarantee consistency: partitions will independently reach the same decision in regards to success or failure. This is similar to the notions of *independent transactions* in Granola [13] or *one-shot transactions* in H-Store [26].

To execute an operation that concerns multiple partitions $P_1, P_2, ..., P_n$, the operation is atomically multicast to all replicas of all involved partitions. Upon delivery, each replica P_i executes the operation and one of the replicas replies to the client. To reach replicas in multiple partitions, the operation is multicast to group g_{all}; if a replica delivers an operation it is not concerned about, the replica just discards the operation.

The following file system commands are implemented as uncoordinated multi-partition operations:

- `mknod`, `unlink`, `symlink`, `mkdir`, `rmdir` when the parent and child are in different partitions.

Coordinated multi-partition operations. Some operations require partitions to exchange information. In GlobalFS, this may happen in the case of a `rename` (i.e., moving the location of a file or directory). In this case, file metadata has to be moved from the origin's partition to the destination's partition. As a result, a rename may involve up to four partitions, given by the placement of the origin, origin's parent, destination, and destination's parent. Consequently, a `rename` operation might fail in one of the partitions (e.g., origin does not exist) but not in the other.

To execute a coordinated multi-partition operation, the client multicasts the operation to all concerned partitions (i.e., multicast group g_{all}). Upon delivery of the operation, the involved partitions exchange information about the command and whether it can or cannot be locally executed. In the case of a `rename`, the file's attributes and list of block identifiers need to be sent to the destination partition. Similarly to a

Operation	Partitions	Multicast	Performance
Read-only	one	not multicast	1^{st} (best)
Single-partition	one	g_{all} or g_i	2^{nd}
Uncoord. multi-partition	two or more	g_{all}	3^{rd}
Coord. multi-partition	two or more	g_{all}	4^{th} (worst)

Table 2: Operations in GlobalFS.

two-phase commit protocol, the command is only executed if all involved partitions agree that it can be executed.

Read-only operations. Read-only operations are executed by a single metadata replica and data store server.[3] For read-only operations, GlobalFS provides causal consistency. This is not obvious to ensure since a client may submit a write operation against a server and later issue a read operation against a different server or even read from two separate servers. When the second server is contacted, it may not have applied required updates yet. GlobalFS provides causal consistency for read operations by carefully synchronizing clients and replicas, as we explain in the following.

We use an approach inspired by vector clocks [20, 45] where clients and replicas keep a vector of counters, with one counter per system partition. In the example described in Section 3.4, clients and replicas keep a vector with four entries, associated with partitions $P_0, ..., P_4$. Every request sent by a client contains v_c, the client's current vector, and each reply from a replica includes the replica's vector, v_r. A read is executed by a replica only when $v[i]_r \geq v[i]_c$, i being the object's partition. The idea is that the replica knows whether it is running late, in which case it must wait to catch up before executing the request.

When a replica receives an update operation from a client, the client's vector v_c is atomically multicast together with the operation. Upon delivery of the command by a replica of P_i, entry $v[i]_r$ is incremented. Every other entry j in the replica's vector is updated according to the delivered v_c, whenever $v[j]_c > v[j]_r$. Clients update their vector on every reply, updating $v[i]_c$ if $v[i]_r > v[i]_c$, for each entry i.

The following file system commands are implemented as read-only operations: `read`, `getdir`, `readlink`, `open` (read-only), and `stat`.

Table 2 summarizes GlobalFS operations. Single-partition and read-only operations access a single partition. While a single-partition operation is multicast to the group associated with the partition, a read-only operation is not multicast but is executed by a single metadata replica (and a data store server). For example, according to the illustrative deployment described in Section 3.4, a write operation for partition P_0 is multicast to g_{all} and a write operation for any of the other partitions P_i is multicast to g_i. Uncoordinated multi-partition and coordinated multi-partition operations access multiple partitions. Such operations are multicast to group g_{all}. Since read-only operations only involve a single metadata server and are not multicast, we expect such operations to outperform any other operations in GlobalFS. Single-partition operations involve all replicas within a single partition, and therefore should perform better than the multi-partition operations. Finally, because uncoordinated multi-partition operations do not require servers in different partitions to interact during the execution of a command, they are expected to perform better than coordinated multi-partition operations.

4.2 The life of some file system operations

To `open` a file, the client uses the partitioning function to discover the partition replicating the provided path. With the partition, the client issues an `open` RPC to the closest replica. The response for this RPC is a file handle that the client uses to issue subsequent `read` and `write` operations. Upon receiving an `open` RPC from the client, a replica checks whether the file is being opened for reading or writing. If the file is open for reading, the replica creates a local file handle, valid only at this replica, and returns it to the client. If the file is open for writing, the file handle needs to be opened in all replicas as writes are replicated. The `open` command is multicast to the associated group (given by the partitioning function) and executed by all responsible replicas. Once a replica has finally delivered and executed the command, it directly replies to the client.

[3] GlobalFS does not implement *atime* (i.e., time of last access), as recording the time of the last access would essentially turn every read into a write operation to update the file's access time.

For a `read` operation, the client needs to execute two steps. First, it issues a `read` RPC to the replica holding the file handle. The replica, upon receiving the read, finds the requested file's metadata and looks for the blocks that match the offset and number of bytes requested. The reply from the RPC is a list of block identifiers and pointers. With the block identifiers, the client contacts the closest data store replicating the file to get the actual data for the blocks. Multiple blocks can be requested in parallel from different data store nodes. After that, the client can build the sequence of bytes that need to be returned by the read operation.

For a `write` operation, the client first creates one or more data blocks from the bytes that need to be written to the file. The client then contacts all the data stores that need to replicate the file (given the partitioning function), and inserts the blocks there, with unique identifiers generated at random. Insertion of multiple blocks can be done in parallel. If all inserts are successful, the client uses the partitioning function to get the partition replicating the file, chooses the closest replica and issues a `write` RPC with the block identifiers as parameters. The replica, upon receiving the `write` RPC, multicasts the command to the responsible group. Upon delivery of the command, a replica finds the medatada for this file and inserts the new blocks. The replica that received the initial RPC from the client replies. On success, the `write` returns the number of bytes written.

4.3 Failure handling

Replicas use state machine replication to handle metadata within partitions. A replica only executes a command that has been successfully delivered by multicast. Thus, if a replica executes a command, other replicas in the same group will also execute the command. GlobalFS uses Multi-Ring Paxos as its atomic multicast (described in more detail in the next section). With Multi-Ring Paxos, as long as one replica and a quorum of acceptors are available in each of the groups, the whole file system is available for writing and reading.

The recovery of a metadata replica is handled by installing a replica checkpoint and replaying missing commands [8]. Coordinated multi-partition commands require one extra step. For coordinated multi-partition commands, replicas in the involved partitions need to exchange information before deciding whether the command can execute or not. A recovering replica, upon replaying a coordinated multi-partition command, requests this information from replicas in the other partitions. To allow for this, whenever a replica sends information out regarding a coordinated command, it also stores this information locally.

Each key-value pair in the data store is replicated in $f+1$ storage nodes. Hence, up to f storage nodes can fail concurrently without affecting data block availability. Datacenter failures and disasters can be handled by carefully replicating blocks in different datacenters or different regions.

Client failures during a `write` or a file delete operation can leave "dangling" `dblocks` inside the data store. `dblocks` without pointers in any `iblock` are unreachable and can be removed from the data store (the implementation of a garbage collector is part of our future work).

5 Implementation

In this section, we discuss the implementation of GlobalFS main components, as depicted in Figure 3.

5.1 Client

Files are accessed through a *file system in user space* (FUSE) implementation [21]. FUSE is a loadable kernel module that provides a file system API to user space programs, letting non-privileged users create and mount a file system without writing kernel code. According to [59], FUSE is a viable option in terms of performance for implementing distributed file systems. Clients know the partitioning function used by the system (currently hardcoded in the client) and use Zookeeper [24] to find the set of available replicas. When using FUSE, every system call directed at the file system is translated to one or more callbacks to the client implementation. In GlobalFS, most FUSE callbacks have an equivalent RPC (remote procedure call) available in the metadata servers. By using the partitioning function, a client can discover to which metadata replica or data store it needs to direct a given operation. Whenever a client has the option of directing a command to more than one destination, it chooses the closest one (with the lowest latency).

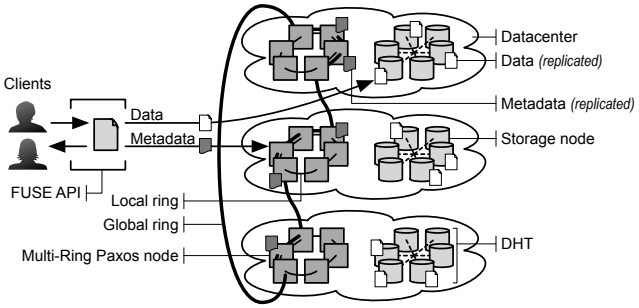

Figure 3: Components and interactions in GlobalFS.

5.2 Atomic multicast

We use URingPaxos,[4] a unicast implementation of Multi-Ring Paxos [34], which implements atomic multicast by composing multiple instances of Paxos to provide scalable performance. Each multicast group is mapped to one Paxos instance. A message is multicast to one group only. Processes that subscribe to multiple groups use a deterministic merge procedure to define the delivery order of the messages such that processes deliver common messages in the same relative order.

For each Paxos instance, Multi-Ring Paxos disposes proposers, learners, and a majority-quorum of acceptors in a logical directed ring in order to achieve high throughput. Processes in the ring can assume multiple roles and there is no restriction on the relative position of these processes in the ring, regardless of their roles. Each ring has a Paxos coordinator, typically the first acceptor in the ring.

In our setup we keep a global ring that includes all metadata replicas in the system, as illustrated in Figure 3. This ring implements the g_{all} group discussed in Section 4. Each other group is implemented by a ring that includes replicas in the same region.

5.3 Metadata replicas

Metadata in GlobalFS is kept by replicated servers, using state machine replication [50]. Replicas can be part of multiple multicast groups; in our prototype, each replica is a Multi-Ring Paxos learner. When a replica delivers a command, the replica checks whether it should execute the command by using the partitioning function. The file system metadata is kept in-memory by the replica and the sequence of commands is stored by Multi-Ring Paxos acceptors. Replicas can be configured to keep their state in memory or on disk, with asynchronous or synchronous disk writes.

The file system is represented as a tree of nodes. There are three node types: directory, file, and symbolic link. A directory node stores the directory properties (e.g., owner, permissions, times) and a hash table of its children nodes, stored by name. A file node keeps the file properties and a list of blocks representing its contents. Symbolic link nodes only need to store the node properties and the target path of the link.

The metadata replicas are implemented in Java and expose a remote interface to the clients via Thrift [4].

5.4 Data store

GlobalFS is designed to support any back-end data store that exposes a typical key-value store API and provides linearizability. Our data store is implemented in Go and uses LevelDB [32] as its storage backend. Depending on the application requirements and fault model, data may be stored persistently on disk or maintained in memory.

The data store is organized as a ring-based DHT and uses consistent hashing for data placement. Each server maintains a full membership of other servers on the ring, allowing one-hop lookups. This design, similar to Cassandra [28] or Dynamo [16], provides good horizontal scalability and stable performance.

Each block is assigned to the first server whose logical identifier follows the block identifier on the ring. A block is replicated as r copies, by copying it onto the $r-1$ successors (i.e., servers that immediately follow this first server on the ring). This ensures data availability with up to $r-1$ simultaneous failures. Servers periodically check for the availability of copies of their blocks onto their successors and create additional

[4] https://github.com/sambenz/URingPaxos

copies when necessary. Similarly, servers periodically check for their predecessor availability and take over the responsibility for their ranges upon failure, also creating additional copies. We note that the blocks stored in the DHT are only written once: there is no need to enforce write consistency between replicas.

Clients contact the DHT via any of its proxy servers. The proxy will create the r copies of the block, using the slower link from the client to send the block only once.

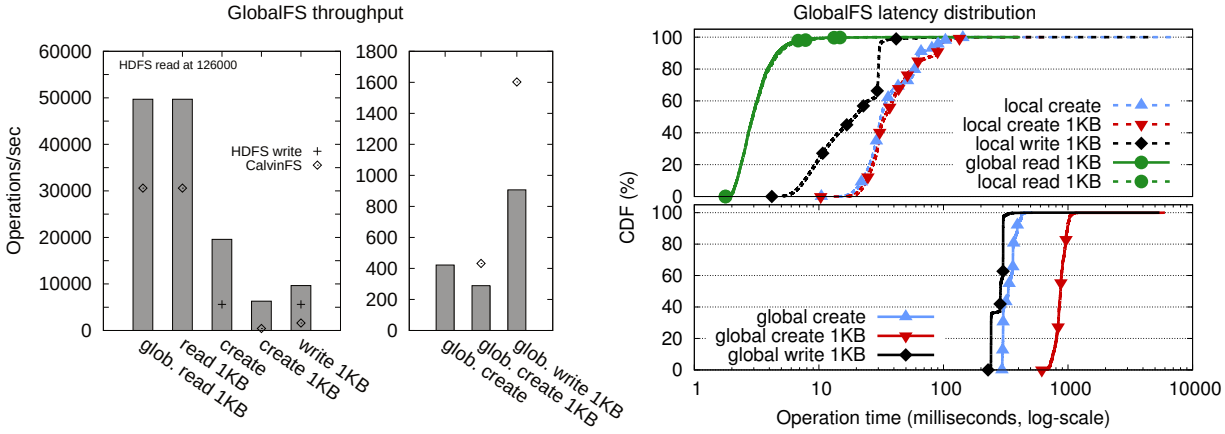

Figure 4: Maximum throughput and latency distribution for different GlobalFS operations with the baseline deployment of 3 partitions. Latencies measured at 50% of maximum throughput.

6 Evaluation

We evaluate GlobalFS using Amazon's EC2 platform. We deploy VMs in all nine EC2 regions available at the time of our experiments. For each region, we distribute servers and clients in three separate availability zones to tolerate datacenter failures. More specifically, inside a single region, we place one server (metadata colocated with storage) and one client machine in each availability zone (six VMs per region). In regions where only two availability zones are present (e.g., `eu-central-1`) we compromise by placing two servers and clients in the same zone. We used `r3.large` (memory optimized) and `c3.large` (compute optimized) instance types, with 2 virtual CPUs, 32 GB SSD storage, and respectively 15.25 and 3.75 GiB memory [3]. We use `r3.large` instances for servers and `c3.large` instances for clients.

We configure the atomic multicast layer based on Multi-Ring Paxos to use in-memory storage. The data store nodes use LevelDB with asynchronous writes to persistent storage.

Our evaluation starts by assessing that the data store implementation in Go using LevelDB [32] can sustain enough throughput not to constitute a bottleneck in our GlobalFS microbenchmarks. We deploy five storage nodes inside a single region with a replication factor of 2 (i.e., each block has 2 copies). For blocks sizes of 1 KB, the data store achieves more than 8,000 `put` operations per second, i.e., around 0.06 Gb/s of aggregate traffic. With larger block sizes (32 KB), the same set-up could sustain around 6,500 `get` operations per second, or around 1.58 Gb/s. For the rest of our experiments, we use blocks of 1 KB.

6.1 Microbenchmarks

We use a custom microbenchmark to evaluate the performance and scalability of GlobalFS for the following types of operations:

▹ **read 1 KB:** each client reads sequentially from a small file (10 KB), in 1 KB chunks. Upon reaching the end of the file, a client wraps and continues reading from the beginning. We disable caching on the client side so that all reads go through the complete protocol.

▹ **write 1 KB:** each client writes sequentially to a file in 1 KB chunks.

▹ **create:** each client repeatedly creates empty files. This operation accesses only the metadata servers.

▹ **create 1 KB:** each client repeatedly creates a file and writes 1 KB to it. Each operation requires 3 sequential metadata operations: `mknod`, `open`, and `write`.

Each operation type is further divided into two categories: *local* operations target files located in the client's local partition and *global* operations target files located in the global partition.

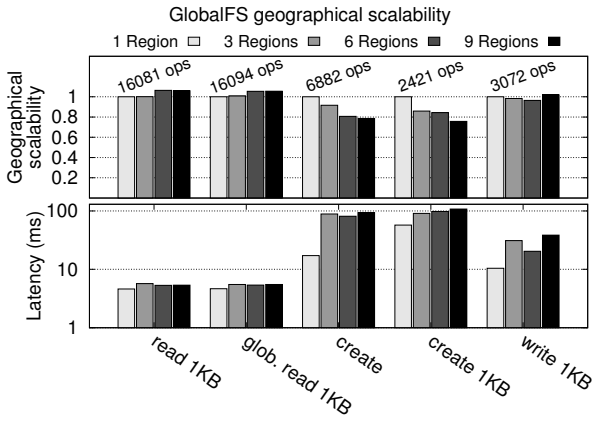

Figure 5: Geographical scalability and 95th percentile latencies for different GlobalFS operations, with increasing system size. Latencies measured at around 50% of maximum throughput.

6.1.1 Performance with 3 regions

For these experiments, we use 3 different geographically distributed regions: `us-west-2`, `us-east-1`, and `eu-west-1`. We deploy 1 local partition in each region. Each partition features 3 servers, each in a different datacenter (availability zone). Metadata and storage are co-located: each server holds a metadata replica and a storage node. Each datacenter also holds one client machine, thus there are 3 clients per region. Each client machine has one GlobalFS FUSE mount point. We then run multiple instances of our benchmark application on top of each client machine.

For comparison, we also show values reported by HDFS in [53] and CalvinFS in [60]. HDFS uses a centralized non-replicated metadata server. The values reported for HDFS consider only metadata performance, and thus represent an upper bound for the actual performance of HDFS. For CalvinFS, we report the approximate values with 9 servers. As the exact values for CalvinFS with 9 servers are not provided in [60], we approximate them by interpolating the values for 6 and 18 servers (we contacted the authors but could not obtain the source code). Due to the linear behavior exhibited by CalvinFS, our approximation should be fairly accurate.

Throughput. Figure 4 (left) shows the maximum throughput achieved for each operation. For read operations, GlobalFS achieved around 60% higher throughput than CalvinFS, for both local and global operations. HDFS achieves higher performance for reads, but it takes only metadata performance into account. Reads in GlobalFS scale linearly with the number of replicas (a single replica needs to be contacted).

For writes, GlobalFS was able to surpass the throughput of HDFS for local operations, even though HDFS considers only metadata. GlobalFS was able to achieve 6 times the throughput of CalvinFS for local writes. For global writes, CalvinFS's throughput was 1.7 times higher. In our setup for GlobalFS, the global partition is replicated by *all* servers in the system (thus it cannot scale).

For creating a file with content, by not complying to POSIX, CalvinFS is able to execute the operation using a single metadata access (by means of a custom transaction). Adhering to POSIX requires a sequence of *three* metadata operations: `create` the file, `open`, and `write`. The `close` is omitted as the write is synchronous. Even though GlobalFS needs the three operations in the same scenario, it can achieve throughput 14.5 times higher than CalvinFS using the faster local partitions. Considering global creates, CalvinFS achieves 1.5 times higher throughput. On the other hand, creating an empty file requires a single metadata operation. In this case, GlobalFS was able to surpass even the performance of HDFS when using the local partitions (3.5 times the throughput). Values for this operation are not reported in the paper that presents CalvinFS [60].

These results show the benefit of exploiting data locality. CalvinFS, while scaling throughput with the number of replicas within a datacenter, does not benefit from local, fast operations. In CalvinFS, all write operations need to go through the global log, thus introducing an overhead on latency. This problem is exacerbated in WAN deployments: either the log is disaster tolerant and all operations pay the cost, or the log is local to a region and clients in other regions need to pay the roundtrip latency. GlobalFS on the other

hand, allows for files to be either locally or globally replicated, thus providing the option for users to choose between availability (disaster tolerance) and performance (throughput and latency). Note that operations across the whole system are still strongly consistent in GlobalFS. The results also show that GlobalFS can deliver good performance while still providing a POSIX interface, thus allowing for existing applications to be used without modification.

Latency. Figure 4 (right) shows the latency distribution for the different types of operations. We measure latency with the system supporting around 50% of its maximum throughput. The results show that operations can be divided roughly in 3 groups in regards to latency: reads, local writes, and global writes (we group creates with writes). Read operations, global and local, observe the lowest latency values, an average of 3.5 ms. This is due to reads being executed by a single metadata replica and not having to go through atomic multicast. Clients can also obtain `dblock`s from the local data store. Local writes, which need to be multicast to servers in a single region, can achieve the second lowest latency, with averages around 20–40 ms. Finally, global writes observe the highest latency values. In our setup, global writes need to be multicast to all servers in the system, across all regions. Clients also need to insert `dblock`s in all data stores. Even so, latency values for writes and creating empty files on the global partition had an average of around 300 ms.

6.1.2 Geographical scalability

We introduce the notion of *geographical scalability* to assess the impact of geographical deployments on performance. Geographical scalability is defined as the ratio between the maximum throughput of local commands in a region in a system that spans multiple regions and the throughput of the region when deployed alone. A geographical scalability of 1 is ideal. Intuitively, it means that the throughput achieved in a single region is not affected by the other regions.

We compute geographical scalability as follows. We first measure the throughput achieved with GlobalFS in a single EC2 region, `eu-west-2`. Then, we consider multi-region deployments with 3, 6, and 9 regions:

- ▹ **3 regions:** `us-west-2`, `us-east-1`, `eu-west-1`.
- ▹ **6 regions:** + `us-west-1`, `eu-central-1`, `ap-northeast-1`.
- ▹ **9 regions:** + `ap-southeast-1`, `ap-southeast-2`, `sa-east-1`.

The reported value is the ratio between the multi-region and the single-region configurations.

Figure 5 (left) shows that GlobalFS scales almost perfectly for all local operations. For create operations, we see a drop in performance as regions are added, down to around 0.8 when all available regions are used. Maximum absolute throughput is shown above the single-region configuration. Figure 5 (right) shows the 95th-percentile of latency in each deployment, measured at around 50% of maximum load. Read commands suffer no impact in latency as they can be executed by a single replica (note that both lines are superimposed). For local writes and creates, the largest increase in latency happens when the system grows from 1 to 3 regions. While commands are executed by replicas inside a single region, Multi-Ring Paxos still needs to synchronize groups. Therefore, latency variations in the global ring can affect the performance of local commands [34].

6.2 Real-world applications

We now present results of an experimental evaluation conducted with real-world applications. We evaluate the performance of some real-world workloads when executed on global and local partitions of GlobalFS. We compare the results against three widely used distributed file systems: NFS (v4.1) [57], GlusterFS (v3.7) [15] and CephFS (v0.94) [63]. Our objective is to assess that, while providing stronger guarantees, GlobalFS compares favorably to *de-facto* industry implementations.

We configure NFS with one single shared directory mounted remotely by the same clients. The NFS server runs in the `us-west-2` region. We disable all caching features on GlobalFS, and the NFS clients mount the remote directory with `lookupcache=none,noac,sync` options. Note that NFS lacks native support for replication,[5] while GlobalFS is configured to always guarantee two copies per `dblock`.

[5] The `replicas` mount option of NFS is a client-side failover feature, but the replication of the shared data has to be handled independently from the NFS protocol.

Command	Operations breakdown	NFS	GlobalFS *global*	GlobalFS *local*	GlusterFS *global**	GlusterFS *local*	CephFS *local*
`tar xzvf bc-1.06.tgz`		1.94 s	47.09 ×	1.36 ×	149.05 ×	1.63 ×	0.17 ×
`configure`		5.32 s	44.66 ×	2.02 ×	45.67 ×	0.96 ×	0.56 ×
`make -j 10`		5.9 s	29.90 ×	2.38 ×	49.34 ×	1.17 ×	0.63 ×
`make`	*(same as above)*	13.14 s	20.73 ×	1.16 ×	55.20 ×	0.92 ×	0.30 ×
`gzip -d httpd-2.4.12.tgz`		3.87 s	117.12 ×	2.47 ×	284.75 ×	0.37 ×	0.11 ×
`tar xvf httpd-2.4.12.tar`		60.01 s	41.46 ×	1.08 ×	99.17 ×	0.12 ×	0.14 ×
`configure -prefix=/tmp`		29.32 s	49.35 ×	2.04 ×	56.53 ×	1.34 ×	0.33 ×
`make -j 10`		714.37 s	2.74 ×	0.52 ×	139.68 ×	0.87 ×	0.48 ×
`make`	*(same as above)*	3432.72 s	1.82 ×	0.36 ×	83.72 ×	0.50 ×	0.64 ×

Table 3: Execution times for several real-world benchmarks on GlobalFS with operations executed over global and local partitions. Execution times are given in seconds for NFS, and as relative times w.r.t. NFS for GlobalFS, GlusterFS and CephFS. *Note that GlusterFS does not support deployments with both global and local partitions; thus, we report results from two separate deployments.

We use FUSE-based bindings for GlobalFS, GlusterFS, and CephFS. We chose two well-known open-source projects as workload: the `bc` numeric processing language (v1.06), and the Apache `httpd` web-server (v2.4.12). These two projects differ in size of the compressed archives (278 kB and 6 MB), number of shipped files (94 and 2,452) and lines of ansi-C code to compile (8,510 and 157,575). They expose different workloads to the underlying file system and are often used as benchmarks [58]. Table 3 embeds the operations breakdown of the system calls issued by the different commands (decompress, configure, and compile) used for these experiments. We evaluate GlobalFS either within a global or a local partition, and compute the average over 3 distinct executions. All file systems are mounted by 9 clients spread equally across 3 regions, but the workload is executed on a single client. We use equivalent settings for GlusterFS,[6] and CephFS. For NFS, all clients mount a shared directory, and a client co-located with the service executes the commands. For GlusterFS we evaluate two different deployments, local (one region) and global (three regions). Each deployment consists of a *distributed/replicated* volume on top of regular *storage bricks*, one on each of the availability zones for the given EC2 regions. We deployed CephFS only at a single region (3 storage daemons, 1 metadata server, and 3 clients) because a deployment across regions would require forfeiting strong consistency [18]. We set the replication factor of GlusterFS, and CephFS to 3.

Table 3 presents our results. We observe that GlobalFS performs consistently better than GlusterFS when operating across regions. GlobalFS performs competitively against the other filesystems across the whole suite of benchmarks. Indeed, GlobalFS is up to 50.9× faster that GlusterFS in compiling Apache `httpd` over the global partition. Note that for the same benchmark on a local partition, GlobalFS is actually faster than NFS. When evaluating GlusterFS and CephFS we use their default, out-of-the-box configuration. Both are heavily optimized systems and some optimizations are on by default (e.g., clients in CephFS use write-back caching, which improves write performance by batching small writes). As expected, the performance penalty for accessing the global partition is higher for write-dominated workloads (extracting an archive, configuring the software package). For read-dominated or compute-intensive (`make`) operations, this overhead decreases because read operations can be completed locally. For comparison purposes, we also tested HDFS (v2.6) with FUSE bindings on a local partition with some of the benchmarks and observed performance in the order as GlobalFS and GlusterFS (e.g., 2.12× slower for the first command as compared to 1.36× and 1.63×, respectively).

Our real-world benchmarks demonstrate that GlobalFS performs on par with widely adopted distributed file systems, it ensures a stronger consistency model, it supports replication, and allows users to benefit from locality thanks to its partitioning model.

7 Related work

In this section, we survey the literature on distributed file systems targeting datacenter deployments. All systems in this category separate the storage of data and metadata. The characteristics of all surveyed systems are provided in Table 4. We categorize file systems by their geographical scaling potential and identify three possible scenarios: file systems that work on LAN (*WoL*) mainly intended for cluster deployments; file systems that support but perform poorly in wide-area network deployments (*WoW*); and file systems that

[6] GlusterFS experiments over the global partition are executed only once due to the required AWS budget.

scale in WAN (*SoW*). GlobalFS is the only system to support data locality while at the same time providing strong consistency and geographical scalability.

Name	Consistency level	POSIX interface	Code available	Client type	Scaling potential
GlobalFS	*S*	√	√	*User*	*SoW*
AFS [48]	*W*,*CTO*	×	√	*User*	*WoW*
CalvinFS [60]	*S*	×	×	*User*	*SoW*
CephFS [63]	*S*	√	√	*Kernel*,*User*	*WoL*
CodaFS [49]	*E*	√	√	*Kernel*	*WoL*
Colossus [12]	*S*	–	×	–	*SoW*
BeeGFS [7]	*S**	√	√	*User*	*WoL*
GeoFS [33]	*S**,*CTO*	√	×	*User*	*WoW*
GFS/GFS2 [43]	–	√	√	*Kernel*	*WoL*
GIGA+ [42]	*E*	√	×	*User*	*WoL*
GlusterFS [15]	*S*	√	√	*User*	*WoW*
GoogleFS [22]	*S*	√	×	–	*SoW*
HDFS [53]	*S*, *CTO*	×	√	*User*	*SoW*
LOCUS [62]	*S*	√	×	*Kernel*	*WoL*
Lustre [51]	*CH*	√	√	*Kernel*	*WoL*
MooseFS [36]	*S**	√	√	*User*	*WoL*
NFS/pNFS [57]	*CTO*	√	√	*Kernel*	*WoW*
ObjectiveFS [38]	*RaW*	√	√	*User*	*WoW*
OCFS [2]	*CH*	×	√	*Kernel*	*WoL*
OCFS2 [2]	*CH*	√	√	*Kernel*	*WoL*
PVFS [10]	*RaW*	×	√	*User*	*WoL*
OrangeFS [44]	*RaW*	×	√	*User*	*WoL*
QuantcastFS [41]	*E*	×	√	*User*	*WoL*
SeaweedFS [52]	*S*	×	√	*Kernel*	*WoL*
XtreemFS [25]	*S**	√	√	*User*	*WoW*
WheelFS [56]	*S*,*CTO*	√	√	*User*	*WoW*

Table 4: Survey of distributed file systems along several criteria: consistency level (Strong=*S*, Weak=*W*, Eventual=*E*, Cache=*CH*, Close-To-Open=*CTO*, Read-after-Write=*RaW*), support of the POSIX standard, code availability, client type (user-space=*User*, kernel-space=*Kernel*), scaling potential (Works-on-LAN=*WoL*, Works-on-WAN=*WoW*, Scale-on-WAN=*SoW*). Some properties are unknown (–) or not by default (*).

7.1 File systems with strong consistency

CalvinFS [60] is a multi-site distributed file system built on top of Calvin [61], a transactional database. Metadata is stored in main memory across a shared-nothing cluster of machines. File operations that modify multiple metadata elements execute as distributed transactions. CalvinFS supports linearizable writes and reads using a single log service to totally order transactions, a mechanism known to scale throughput with the number of nodes within three regions [61]. Using more regions penalize all operations, implying lack of data locality support for CalvinFS. We note that CalvinFS relies on "custom transactions" that group multiple commands into a single operation to boost performance. For example, creating and writing a file, which in POSIX would require three sequential calls (i.e., create, open and write), can be executed as a single transaction in CalvinFS. As a consequence, POSIX compliance cannot benefit from these optimizations.

CephFS [63] is a file system implementation atop the distributed Ceph block storage [11]. It uses independent servers to manage metadata and link files and directories to blocks stored in the block storage. CephFS is able to scale up and down the metadata servers set and to change the file system partition at runtime for load balancing through its CRUSH [64] extension. Although CephFS supports geographical distribution, WAN deployment over Amazon's EC2 is discouraged by the CephFS developers [18].

The Google File System (GoogleFS) [22] stores data on a swarm of *slave* servers. It maintains metadata on a logically centralized master, replicated on several servers using state machine replication and total ordering of commands using Paxos [30]. GoogleFS is a flat storage system. It does not consider the case of a file system spread over multiple datacenters and the associated partitioning. MooseFS [36] is designed around a similar architecture and has the same limitations. Colossus [12], GoogleFS successor, provides the same strong consistency guarantees, but many of its internal details remain undisclosed.

FhGFS/BeeGFS [7] is distributed file system for high-performance computing clusters that targets read-dominated workloads.

GeoFS [33] is a POSIX-compliant file system for WAN deployments. It exploits user-defined timeouts

to invalidate cache entries. Clients pick the desired consistency for files and metadata, as in WheelFS's semantic cues [56].

Red Hat's GFS/GFS2 [43] and GlusterFS [15] support strong consistency by enforcing quorums for writes, which are fully synchronous. GlusterFS can be deployed across WAN links, but it scales poorly with the number of geographical locations, as it suffers from high-latency links for all write operations.

HDFS [53] is the distributed file system of the Hadoop framework. It is optimized for read-dominated workloads. Data is replicated and sharded across multiple data nodes. A name node is in charge of storing and handling metadata. As for GoogleFS, this node is replicated for availability. The HDFS interface is not POSIX-compliant and it only implements a subset of the specification via a FUSE interface. QuantcastFS [41] is a replacement for HDFS that adopts the same internal architecture. Instead of three-way replication, it exploits Reed-Solomon erasure coding to reduce space requirements while improving fault tolerance.

SeaweedFS [52] is a distributed file system that follows the design of Haystack [6]. It supports multiple master nodes and multiple metadata managers to locate files. It is optimized for (small) multimedia files and does not support the POSIX semantics.

XtreemFS [25] is a POSIX-compliant system that offers per-object strong-consistency guarantees on top of a set of independent volumes managed by a metadata server (MRC). To best of our understanding, it does not provide a global integrated file system. Further, it does not offer consistency guarantees for inter-volume operations.

PVFS [10] and HDFS can be adapted to support linearizability guarantees for metadata [54] by delegating the storage of the file system's metadata to Berkeley DB [39], which uses Paxos to totally order updates to its replicas.

7.2 File systems with weak consistency

There are several distributed file systems for high-performance computing clusters, such as PVFS, PVFS2/OrangeFS [44], Lustre [51], and FhGFS/BeeGFS [7]. These systems have specific (e.g., MPI-based) interfaces and target read-dominated workloads. GIGA+ [42] implements eventual consistency and focus on the maintenance of very large directories. It complements the OrangeFS cluster-based file system.

ObjectiveFS [38] relies on a backing object store (typically Amazon S3) to provide a POSIX-compliant file system with read-after-write consistency guarantees. If deployed on a WAN, ObjectiveFS suffers from long round-trip times for operations such as `fsync` that need to wait until data has been safely committed to S3.

Close-to-open consistency (*CTO*) was introduced along with client-side caching mechanisms for the Andrew file system and implemented in its open-source implementation OpenAFS [48]. This was a response to previous distributed file systems designs such as LOCUS [62], which offered strict POSIX semantics but with poor performance. Close-to-open semantics are also used by NFS [57], HDFS [53], and WheelFS [56].

Oracle OCFS [2] is a distributed file system optimized for the Oracle ecosystem (e.g., database, application-server). It provides a cache consistency guarantee by exploiting Linux's `O_DIRECT`. Its successor OCFS2 [2] supports the POSIX standard while guaranteeing the same level of cache consistency.

7.3 Peer-to-peer file systems

Peer-to-peer file systems target deployments over a large number of independent servers rather than a collection of datacenters. We do not list these systems in Table 4 as they are less directly linked to our work, but discuss them below.

A common aspect of peer-to-peer file systems is that they store both metadata and data in the same storage substrate, unlike previously listed approaches and GlobalFS. This storage is typically a DHT. CFS [14] and PAST [46] are early examples of single-writer peer-to-peer file systems, using the Chord [55] and Pastry [47] DHTs. Ivy [37] is an evolution of CFS for multiple writers. The set of writers is static and each writer maintains its own log of modifications to the file system. A reader must causally parse through all writers' logs. Ivy supports eventual but *read-your-write* consistency. CFS and Ivy use immutable blocks, similarly to GlobalFS. Pastis [9] similarly extends PAST for multiple writers. It supports *read-your-write* semantics and *close-to-open* consistency. Finally, OceanStore [27] is a DHT-based peer-to-peer file system that offers both eventual consistency and linearizability. It leverages the eventually serializable data storage [19]: Weak operations may execute at any replica, while strong operations are totally ordered between writers.

8 Conclusion

This paper introduces GlobalFS, a geographically distributed file system that accommodates locality of access, scalable performance, and resiliency to failures without sacrificing strong consistency. GlobalFS builds on two abstractions: single-site linearizable data stores and an atomic multicast based on Multi-Ring Paxos. This modular design was crucial to handle the complexity of the development, testing, and assessment of GlobalFS. Our in-depth evaluation reveals that GlobalFS outperforms other geographically distributed file systems that offer comparable guarantees and delivers performance comparable to single-site networked file systems. We credit GlobalFS performance to its flexible partition model and four execution modes, which allow us to exploit common access patterns and optimize for the most frequent file system operations. These original features distinguish GlobalFS from other distributed file systems and are key to providing geographical scalability without compromising consistency.

References

[1] *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, 2001.

[2] Oracle Cluster File System (OCFS). In *Pro Oracle Database 10g RAC on Linux*, pages 171–200. Apress, 2006.

[3] http://aws.amazon.com/ec2/instance-types/.

[4] https://thrift.apache.org.

[5] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.

[7] BeeGFS. http://www.beegfs.com.

[8] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato. Building global and scalable systems with atomic multicast. In *15th ACM/IFIP/USENIX International Middleware Conference*, Middleware, 2014.

[9] J.-M. Busca, F. Picconi, and P. Sens. Pastis: a highly-scalable multi-user peer-to-peer file system. In *Euro-Par*, 2005.

[10] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *4th Annual Linux Showcase and Conference*, ALS, 2000.

[11] Ceph block storage. http://ceph.com/ceph-storage/block-storage/.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[13] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference*, ATC, 2012.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles*, SOSP, 2001.

[15] A. Davies and A. Orsaria. Scale out with GlusterFS. *Linux Journal*, 2013(235), Nov. 2013.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, 2007.

[17] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[18] Email exchange on CephFS mailing list. https://www.mail-archive.com/ceph-users@lists.ceph.com/msg23788.html.

[19] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220, 1999.

[20] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[21] File System in User Space (FUSE). http://fuse.sourceforge.net/.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, SOSP, 2003.

[23] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[24] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, ATC, 2010.

[25] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The XtreemFS architecture – a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

[26] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[27] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35(11):190–201, 2000.

[28] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), Apr. 2010.

[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[30] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[31] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[32] LevelDB. https://github.com/google/leveldb.

[33] G. Liu, L. Ma, P. Yan, S. Zhang, and L. Liu. Design and Implementation of GeoFS: A Wide-Area File System. In *9th IEEE International Conference on Networking, Architecture, and Storage*, NAS, 2014.

[34] P. J. Marandi, M. Primi, and F. Pedone. Multi-Ring Paxos. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2012.

[35] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A High-Throughput Atomic Broadcast Protocol. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2010.

[36] MooseFS. https://www.moosefs.org.

[37] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. In *5th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2002.

[38] ObjectiveFS. http://objectivefs.com.

[39] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference*, ATC, 1999.

[40] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, ATC, 2014.

[41] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. of the VLDB Endowment*, 6(11):1092–1101, 2013.

[42] S. Patil and G. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *9th USENIX Conference on File and Storage Technologies*, FAST, 2011.

[43] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, shared disk file system for linux. In *16th IEEE Symposium on Mass Storage Systems*, 1999.

[44] PVFS2. http://www.pvfs.org.

[45] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, Oct. 1991.

[46] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, SOSP, 2001.

[47] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Middleware Conference*, Middleware, 2001.

[48] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 1990.

[49] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput.*, 39(4):447–459, Apr. 1990.

[50] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[51] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Linux Symposium*, 2003.

[52] SeaweedFS. https://github.com/chrislusf/seaweedfs.

[53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *26th IEEE Symposium on Mass Storage Systems and Technologies*, MSST, 2010.

[54] D. Stamatakis, N. Tsikoudis, O. Smyrnaki, and K. Magoutis. Scalability of replicated metadata services in distributed file systems. In *12th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS, 2012.

[55] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, feb 2003.

[56] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, Wide-Area Storage for Distributed Systems with WheelFS. In *6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2009.

[57] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, Mar. 1989.

[58] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It *is* rocket science. In *13th USENIX Workshop on Hot Topics in Operating Systems*, HotOS, 2011.

[59] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra incognita: On the practicality of user-space file systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage, 2015.

[60] A. Thomson and D. J. Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies*, FAST, 2015.

[61] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2012.

[62] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *9th ACM Symposium on Operating Systems Principles*, SOSP, 1983.

[63] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[64] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE conference on Supercomputing*, SC, 2006.