Survey of Distributed File System Design Choices

PETER MACKO and JASON HENNESSEY, NetApp, Inc., USA

Decades of research on distributed file systems and storage systems exists. New researchers and engineers have a lot of literature to study, but only a comparatively small number of high-level design choices are available when creating a distributed file system. And within each aspect of the system, typically several common approaches are used. So, rather than surveying distributed file systems, this article presents a survey of important design decisions and, within those decisions, the most commonly used options. It also presents a qualitative exploration of their tradeoffs. We include several relatively recent designs and their variations that illustrate other tradeoff choices in the design space, despite being underexplored. In doing so, we provide a primer on distributed file systems, and we also show areas that are overexplored and underexplored, in the hopes of inspiring new research.

CCS Concepts: • **Information systems** → **Distributed storage**; Storage management; • **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → **File systems management**; *Distributed systems organizing principles*; *Cloud computing*

Additional Key Words and Phrases: Distributed file systems, distributed storage, design options, taxonomy, survey

ACM Reference format:
Peter Macko and Jason Hennessey. 2022. Survey of Distributed File System Design Choices. *ACM Trans. Storage* 18, 1, Article 4 (February 2022), 34 pages.
https://doi.org/10.1145/3465405

1 INTRODUCTION

Congratulations, you have just been asked to design the next distributed file system. But how should you go about it? You can find decades of research on distributed file systems and object stores. And the pace of research is accelerating as the world continues to move away from scale-up to scale-out solutions and as the desire increases to provide data management across multiple storage endpoints instead of treating them as isolated silos.

There are so many research and production systems to learn about—and to learn from. But comparably far fewer high-level design options are available when designing various aspects of a distributed file system, and each of these options results in a different set of tradeoffs. For example, there are only so many high-level approaches for stitching together a file system namespace that is spread over multiple servers, which then affects the system's performance and its ability to distribute load or to handle partitions. In this article, we identify several key design decisions,

Authors' address: P. Macko and J. Hennessey, NetApp, Inc., 1601 Trapelo Rd., Waltham, Massachusetts 02451; emails: {peter.macko, jason.hennessey}@netapp.com.

1553-3077/2022/02-ART4
https://doi.org/10.1145/3465405

examine the most commonly used options and some of their variations, and provide a qualitative analysis of the relevant tradeoffs. Quantitative evaluation is beyond the scope of this work.

When it comes to designing a new distributed file system, defining a new research project, or learning about these file systems for any other reason, it is important to be aware of the choices and their implications. None of the existing designs is obviously the best for every situation, so a file system designer should be aware of the various options and make careful choices that fit best with the requirements, such as read and write performance, consistency, partition tolerance, and redundancy. New researchers should also be well aware of this design space so that they can focus their efforts on underexplored areas or on eliminating undesirable properties of existing techniques, rather than simply designing a new file system that does not advance the state of the art.

Our aim is not to present an exhaustive survey of all the relevant literature but just to focus on the options for high-level approaches to designing three key parts of the distributed file system stack: locating an inode (or equivalent) given a path, locating data given an inode, and choosing various consistency and synchronization options, which can be—with some limitations—mixed and matched. Even though we do not claim to explore all the design choices and tradeoffs exhaustively, our goal is to survey the most important ones. We mention specific systems and publications only where they are relevant, drawing examples from both the old and the new and from both academia and industry. We do not aim to list or to examine all the important distributed file systems in history, of which a good starting point for exploration includes prior surveys of distributed file systems [56, 67], as well as topic-specific surveys on peer-to-peer file systems [22] and decentralized access control in distributed file systems [39].

For the purposes of this article—so that we can capture a wider set of interesting design options—we define a distributed file system quite broadly as a system that stores files across more than one node and supports basic operations such as read, write, create, lookup, and listing a directory. We also generally assume a hierarchical namespace, regardless of whether it has one or several roots. Many of the design options also apply to distributed object stores, which have several requirements and constraints in common with distributed file systems.

After we describe a high-level, archetypical file system architecture in Section 2, we survey the common types of pointers that are used in file systems, such as those that point from directory entries to inodes or from inodes to data, in Section 3. Sections 4, 5, and 6 then survey the key parts of the file system design: locating inodes given a path, locating data given an inode, and choosing options to handle coordination and to ensure file system consistency. Finally, we present several case studies and a design exercise in Section 7. In the case studies, we examine the tradeoffs or characteristics of several known file systems through the lens of those design choices. We then use these tradeoffs to design a hypothetical file system. We then conclude in Section 8.

2 INSIDE A DISTRIBUTED FILE SYSTEM

At a very abstract level, a distributed file system—just like any other storage system—maps pathnames to data so that it can perform reads and updates. However, its distributed nature makes it harder to coordinate operations across all nodes and to ensure enough redundancy for the system to sufficiently tolerate node failures and, if applicable, also to tolerate network partitions. In addition, use of distributed file system techniques can be advantageous when scaling out the performance of local file systems [28], even though the set of challenges for the system is smaller; for example, we would not expect a network partition between file system processes that run on different cores.

Figure 1 is an example of a distributed file system architecture (not of any specific file system) from a mash-up of several systems that, for didactic purposes, illustrates several important concepts, including:

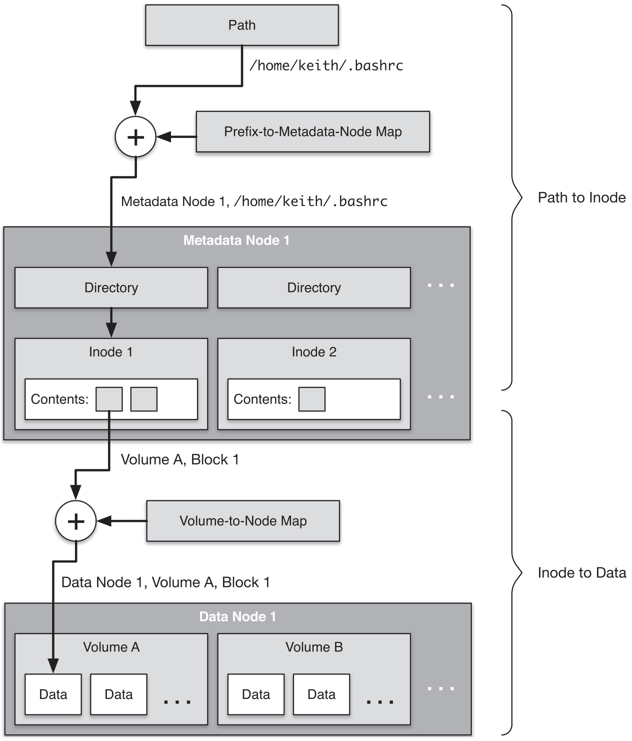

Fig. 1. An example of a distributed file system architecture data path, illustrating a path-to-inode translation, followed by inode-to-data translation.

- *Pointer types:* Distributed file systems use different kinds of what we call *pointers* to reference metadata and data. For example, the list of block locations in the inode in this figure uses an explicit, indirect pointer type, because the inode stores only volume IDs and block numbers, needing to consult an external *volume-to-node map* to determine the actual locations.
- *Data and metadata management granularity:* Data and metadata are managed at different granularities at the different levels. Figure 1 illustrates partitioning the metadata by subtrees (i.e., subtree granularity), while allowing different parts of files to be physically stored on different nodes (i.e., subfile granularity).

We discuss pointer types and granularities in more detail in Section 3.

And, at a high level, we can divide the data path into two parts:

- *Path-to-inode translation:* Starting from the pathname, how does a system locate the inode of the file or directory, which contains its key metadata, such as the location of the data? The example file system in Figure 1 uses prefix tables (Section 4.3) to determine which metadata server stores which part of the namespace.
- *Inode-to-data translation:* Starting from the inode, how does a system locate the data? In this example, the inodes reference data at subfile granularity, and the pointers specify volume IDs and offsets within volumes. The volume IDs are then translated to node IDs.

We discuss the various options for the above layers of the file system stack in Sections 4 and 5, respectively.

Table 1. Pointer Types

Type	Network Hops	Examples
Local	0	Inode to data in AFS [25]
Explicit, direct	1	Multiaddresses in libp2p [34]
Explicit, indirect	1	Remote links in NetApp® ONTAP® FlexGroup volumes [30]
Computed (implicit)	1	CephFS [73], distributed volumes in GlusterFS [21]
Overlay network	$O(\log n)$	Name to inode in IPFS [9]

And unlike local file systems, distributed file systems must also make additional design choices:

- *Redundancy and coordination:* How does a system ensure that there is enough redundancy—whether replication or erasure coding—to handle failures, and how does it then coordinate accesses and updates across all those replicas and/or erasure-coded fragments?

We discuss these aspects of the distributed system designs in Section 6.

For the purposes of this article, we use the terms *node* and *server* interchangeably to refer to a node that hosts a part of the file system, such as its data, metadata, or both. A file system *client* accesses the file system by sending requests to one or more of the servers, through either a standard protocol such as NFS, a client-side library, or an agent. A client does not permanently host any of the file system's data or metadata, though it may need to retain small amounts of state such as cryptographic keys or file handles for access. In some file systems, participants can be both clients and servers at the same time. *Cluster* refers to a collection of all the nodes that host the file system but not its clients unless stated otherwise.

3 POINTERS

Before we discuss the overall architecture of distributed file systems, we need to understand three important concepts: pointer types, pointer granularities, and multipointers.

3.1 Pointer Types

Distributed file systems use several kinds of pointers, or lookup mechanisms, for pointing to metadata or data. The types are summarized in Table 1 and are explained further in the rest of this section, from the simplest to the most complicated type.

3.1.1 Local Pointers. This is the simplest kind of pointer, which always references metadata or data on the same node of the file system. For example, this kind of pointer is used in inodes of file systems that store an inode and the corresponding data on the same node, such as AFS [25], Coda [57], and NetApp Data ONTAP® GX [17] and in some GlusterFS configurations [21]. In this case, these pointers are perfectly analogous to the local pointers that the underlying local file system uses to reference data from the inodes.

3.1.2 Explicit, Direct Pointers. The simplest distributed pointer explicitly names a node that it references, such as by specifying its IP address. The main disadvantage for these pointers is that they are inflexible. For example, if a node's IP address changes (e.g., due to the use of DHCP), then the pointers suddenly stop working. Or, more importantly, if a node fails and a different node assumes its role, then all the pointers have to be updated.

This kind of pointer is surprisingly rare in distributed systems, but one example of a well-designed usage is in libp2p [34], which is used by IPFS [9]: Instead of just using opaque peer IDs, it can also augment them with the peer's IP address and port in its multiaddress format. So, as long as the IP address does not change, others can contact the peer directly without needing to be routed through the overlay network.

3.1.3 Explicit, Indirect Pointers. This is perhaps the most common kind of pointer in distributed file systems, because it is both simple and flexible. Explicit, indirect pointers reference a node, a local volume, or a subdirectory on the node without actually naming the node, such as by referencing a virtual node ID or a local volume ID. These IDs are then mapped by using a table, which is typically replicated on all the nodes in a cluster, and—depending on the type of pointer and the implementation of the file system—is also cached or replicated on the clients. These tables are usually small and infrequently updated; updates are usually coordinated through a fault-tolerant mechanism such as Paxos.

A specific kind of this pointer type is a *prefix table* (Section 4.3), which maps path prefixes to nodes that manage the corresponding subtrees. An example of such usage is the volume location databases in AFS [25], Coda [57], and Data ONTAP GX [17].

3.1.4 Computed (Implicit) Pointers. Instead of storing the pointers explicitly, several distributed file systems compute the target of the pointer by using a key, such as a file ID or a content hash of the data, and a cluster membership list. Perhaps the best known example is the CRUSH [74] function used in CephFS [73] to place and to look up data on the nodes in the cluster. Computed (implicit) pointers are usually implemented by using *consistent hashing* [29] to minimize data movement when a node joins or leaves a cluster, but other approaches are possible. For example, Flat Datacenter Storage [45] uses a hash function to locate the first block (called the *tract*) of the file, but the rest of the blocks are located relative to it in a manner that distributes load and capacity more evenly across the cluster.

This approach can be combined with the explicit, indirect pointer type by letting the hash function reference a virtual node ID or a local volume ID, which is then resolved by using a table replicated on all the nodes in the system.

3.1.5 Overlay Networks. Many peer-to-peer systems use overlay networks based on *distributed hash tables (DHTs)*, for example, Chord-style DHTs [61]. One advantage of an overlay network is that a request can be routed to the appropriate node just by using a key, without knowing the node's address, location, or identity—and even without knowing the full membership list. In fact, only a small fraction of the membership nodes must be known for the routing to function, which makes it advantageous for large-scale clusters with high node churn. Such overlay networks are thus highly robust to node failures, and many (but not all) are also robust to partitions.

A disadvantage of DHT-based overlay networks is their cost in the number of network round-trips: Contacting a node based on the ID of an object that it hosts costs $O(\log n)$ round-trips, where n is the total number of nodes in the network. At each step, a node forwards the request to another node whose ID is closer to the requested ID by some metric, such as the number of shared least significant digits as illustrated in Figure 2. A common optimization is to cache seen node IDs and their IP addresses.

Examples of such networks are Kademlia [38] (a variant of which is used in IPFS [9]), Tapestry [23] (used in OceanStore [31]), and DHash (a part of CFS [13]). These networks should not be confused with Dynamo-style DHTs [14], which are also distributed hash tables. The difference is that Dynamo-style DHTs require each node to have full knowledge of node membership and are thus correspondingly faster, while DHT-based overlay networks do not require such knowledge and are correspondingly slower.

3.2 Pointer Granularities

A target that a pointer references can be one of the following four granularities:

- *Subtree* (sometimes called *volume*): An entire subtree.

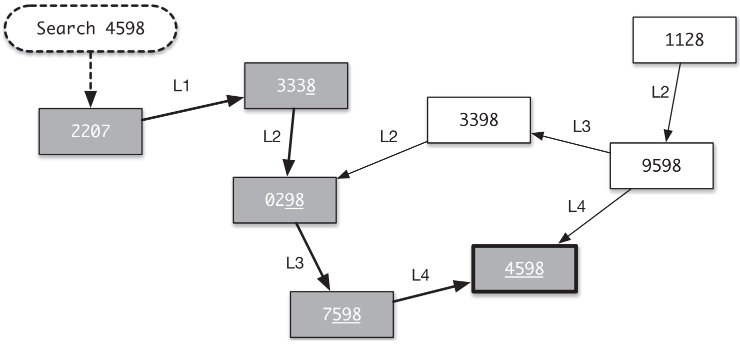

Fig. 2. An example of a DHT-style overlay network from OceanStore [31] showing a sample path of routing from node 2207 when searching for 4598. Each node routes the request to the next node that it knows about, which has at least one more shared least significant digit with the search key (underlined).

- *Bucket* (not to be confused with Amazon S3 buckets): A collection of potentially unrelated files or parts of files.
- *File:* An entire file, either its inode or all of its contents.
- *Subfile:* A part of a file, either fixed-length or variable-length.

These granularities often become the granularities of metadata or data management, because most file systems cannot switch between them. For example, prefix tables in AFS [25] have the subtree granularity, which means that it is impossible to change the location of one file without changing its name; we discuss this issue further in Section 4.3. IPFS [9] is a notable exception, which can switch between referencing file contents by using whole-file and subfile granularities.

3.3 Multipointers

Typically a pointer points to a single target; *multipointer* generalizes the concept of pointers to specify potentially multiple targets. It can be done implicitly or explicitly.

For example, the master server in Google File System (GFS) [20] has an explicit list of servers that store each chunk (even though it is just maintained in memory); another example is IPFS [9], which explicitly stores the list of replicas. An example of an implicit multipointer in CephFS [73] is the mapping of placement group IDs to data nodes (called Object Storage Devices), which determines the list of the nodes from a single key by using the CRUSH [74] hash function.

4 PATH-TO-INODE TRANSLATION

The first part of the data path is path-to-inode translation, in which the file system starts with a path, or alternatively a cached directory object and a file name, and finds the corresponding *inode*. This inode contains important metadata such as size, owner, permissions, checksum, and location of data. Many distributed file systems call this set of information a *metadata object*; in this article, we refer to it as an *inode* and *metadata* interchangeably.

Most distributed file systems resolve paths one component at a time, just like their single-node counterparts. Only a few file systems, such as CalvinFS [68], employ whole-path indexing. Resolving one component at a time makes many operations on directories faster, such as renaming or changing permissions. However, certain other operations are more complicated; for example, moving a directory must be properly synchronized to avoid creating directory loops [7].

Table 2. Path-to-Inode Translation

Method	Lookup Latency (Round-trips)	Flexible Inode Location	Metadata Granularity	Partition Tolerance	Examples
Single Node	$O(1)$	No	Subtree	No	HDFS 1.x [60], Orion [78]
Replicated Namespace	$O(1)$	No	Subtree	Depends	GlusterFS [21]
Replicated Prefix Table	$O(1)$	No	Subtree	Depends	AFS [25], Data ONTAP GX [17]
Remote Links	$O(1)$	Yes	Subtree or File	No	Farsite [7], FlexGroup volumes [30]
Shared-Disk Like	Usually $O(1)$	No	File	No	GPFS [58], Oracle FSS [32]
Computed	$O(1)$	No	File	Depends	GlusterFS [21], Tahoe-LAFS [64]
Overlay Network	$O(\log n)$	No	File	Depends	IPFS [9], OceanStore [31, 54]
Distributed Database	Depends	No	Depends	Depends	ADLS [51], CalvinFS [68]

Table 2 summarizes the common design options and assesses them based on the following characteristics:

- *Lookup latency (round-trips):* The latency of path component lookups and updates, measured as the number of nodes that need to be contacted via a network round-trip (n refers to the number of nodes in the cluster).
- *Flexible inode location:* Whether it is possible to change the location of the inode (not necessarily data) in the network without changing either the pathname or an internal file ID; that is, whether the file system supports *location independence of naming*. Even if an approach is marked as "No," it may be possible to achieve some flexibility through clever solutions, such as Coral Distributed Sloppy Hash Table (Coral DSHT) [19] for overlay networks (we briefly discuss Coral in Section 4.7).
- *Metadata granularity:* The granularity of metadata management; that is, whether the metadata for files is grouped by subtrees, or whether each file's metadata are distributed independently.
- *Partition tolerance:* Whether the approach (not necessarily all specific implementations) supports file system availability in minority network partitions; that is, when less than half of the nodes are reachable. There are several possible definitions of availability, but in this article, we generally concern ourselves with the stronger definition, which describes the system as available only if it can continue to perform updates even in the minority partition. Many systems support the weaker form of availability, in which nodes can read any data that they can navigate to on their side of the partition but may not be able to continue to write.

We will now discuss each of these approaches. In the following subsections, we first explain the approach by using an exemplar system, then we discuss tradeoffs, and then—for interested readers—we discuss variations and other notable technical details. We start by discussing the simplest but less common approaches and then proceed to the more common approaches.

Several systems combine two or more approaches, and we point out several such examples throughout this section. Finally, in Section 4.9, we present several other important design approaches. We conclude Section 4 with a brief discussion in Section 4.10 of several interesting, underexplored problems.

4.1 Single Node

The simplest case of path-to-inode translation is to store the entire namespace (both directories and files) on a single node of a cluster, potentially being replicated on another node in an active-passive manner. This approach may be sufficient for small clusters, but because of its scalability problems, it is not common. It is, however, important historically; for example, HDFS [60] used to support

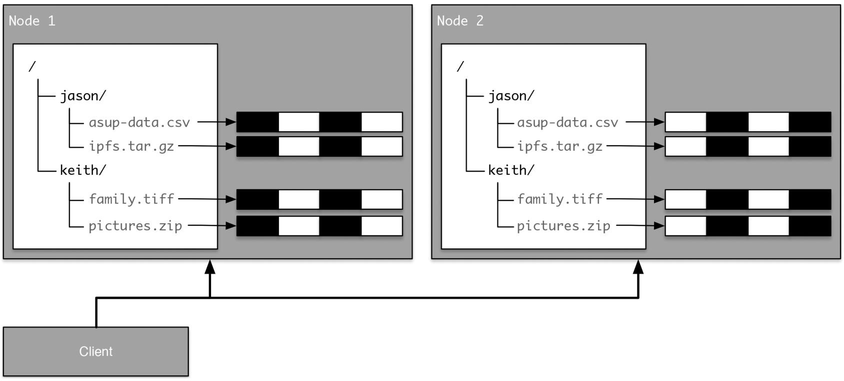

Fig. 3. A simplified example of a striped volume in GlusterFS [21], which replicates the entire namespace on all nodes, striping the data across them. In this example, the distribution is entirely driven by the client software.

only one active name server before HDFS Federation was introduced in version 2.0. Since version 2.0, HDFS supports multiple name nodes that manage independent namespaces, even though they still share the same set of data nodes. More recent examples include Orion [78] and MooseFS [40], which manage the metadata of the entire namespace on a single metadata server, which then can be replicated.

One advantage of storing the entire namespace on a single node is that resolving a path requires only a single round-trip to that node and back. However, the node can easily become the bottleneck, so this approach is viable only for small clusters or for cases of large files. One option to alleviate this problem is to use *shadow* masters, which can serve read-only traffic, such as in GFS [20]. Depending on the file system, the version of the metadata that is served by the shadow master may or may not slightly lag behind the version that is served by the master. Another advantage of this approach is that the single server becomes the central point of coordination.

4.2 Replicated Namespace

Perhaps the second simplest case of path-to-inode translation is to have the file system replicate the directory structure or even the entire namespace (including directory entries for files) on all nodes in the cluster. This approach can scale better than the single-node method but is still limited to smaller clusters, because, although metadata reads can be performed quickly, updates are slower. This approach, although not common, is used by one prominent distributed file system—GlusterFS [21].

The cleanest examples of replicated namespace are the *striped* and *dispersed* volumes in GlusterFS, illustrated in Figure 3, which replicate the entire namespace on all nodes of the cluster. In the striped case, the file system stripes the file data across the cluster; in the dispersed case, it uses erasure coding. Unlike many other file systems, this process is driven by software running on the clients. Metadata are smaller than data, so it is feasible for the entire namespace to be stored on each node, especially in large-file workloads.

In the *distributed* configuration, GlusterFS replicates only the directory structure onto all nodes, which allows it to scale to a larger number of nodes but stores each file on a single node or replicates it onto a subset of nodes. Files are placed according to the consistent hash of their name by using a mechanism that is reminiscent of a Dynamo-style DHT [14]. It can be thought of as a combination of the replicated namespace and hash-based file placement, which we describe in Section 4.6. This approach is more scalable, because there are usually many fewer directories than files.

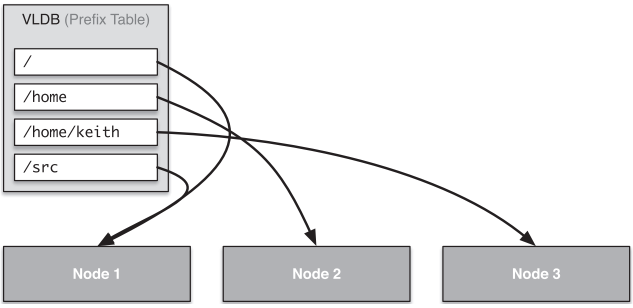

Fig. 4. An example of a replicated prefix table, similarly to the VLDB in AFS, which maps path prefixes to volumes or nodes.

An advantage of this approach is that a path lookup can be satisfied by using a single network round-trip if the entire namespace is replicated. At most, it can take two round-trips if only the directories are replicated (one to get the containing directory, the other to get the file if it is on a different node). The obvious drawbacks of this approach are the cost of managing replication across a potentially large number of replicas and the resulting impact on scalability, storage overhead, and the latency of metadata updates.

Partition tolerance depends on the implementation and/or the configuration of the system. For example, GlusterFS can be configured to turn replicas on the minority side of the network partition into read-only replicas, but it is also possible to let the two sides of the partition operate independently, creating a *split-brain* problem, and heal the differences later. However, many kinds of differences may need to be resolved manually.

4.3 Replicated Prefix Table

Arguably the most popular approach is to partition the file system namespace based on prefixes. The canonical example is AFS [25], which is illustrated in Figure 4. The file system maintains a small volume location database (VLDB), which maps prefixes to nodes that store the corresponding subtrees, which AFS calls volumes. More generically, we call such a database a *prefix table* for clarity and because it is a very similar concept to Sprite-style prefix tables [75]; the word "volume" is also often overused to name different concepts. The concept of prefix tables can be thought of as a special case of explicit, indirect pointers, as already mentioned in Section 3.1.3. Most file systems allow the path prefixes to nest, just as illustrated in the example in Figure 4. This approach is often combined with replicated namespaces for redundancy. For example, Coda [57] extends AFS to allow each volume (subtree) to be replicated on multiple nodes.

The prefix table is usually small, infrequently updated, and replicated on all nodes in the cluster. Most file systems update it by using a reliable, fault-tolerant mechanism, such as Paxos. The table is highly replicated, and updating it is expensive. This cost creates pressure to keep the table small and infrequently updated, which consequently means that the subtrees should be relatively large.

Many implementations also replicate the prefix table onto the clients, so that a client can determine which node stores the inode for which file and can access it in a single network round-trip. The file system may allow the copies of the table on the clients to become stale, in which case a client learns about the new version after attempting to access a file or a directory on a node that does not host it anymore.

Partitioning the namespace thus preserves low-latency access, while naturally resulting in a more scalable approach. Depending on the implementation, this approach can be tolerant to

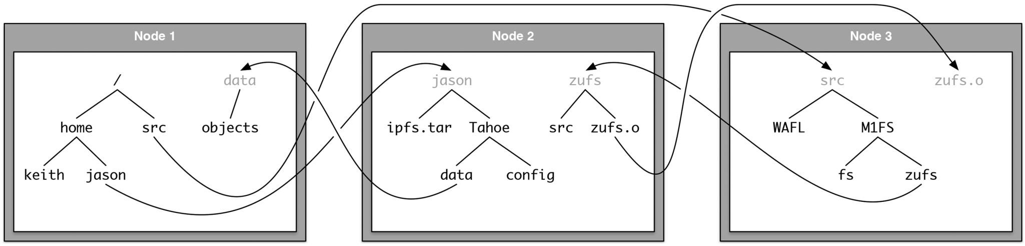

Fig. 5. A simplified example of remote links as used by FlexGroup volumes [30].

partitions as long as clients can continue to access the subtrees that are available on their side of the partition; however, updating the prefix table is generally not safe during partitions.

If data are colocated with the metadata, then a major disadvantage of prefix tables is that moving (renaming) files between prefixes results in data movement. Some systems, such as GlobalFS [46], solve this problem by creating temporary remote links. After a client requests a file or a directory to be moved, the system creates a remote link on the target volume to the original location of the file or directory, while the data movement completes in the background. File systems generally avoid making these links permanent, because it increases complexity and lowers performance when resolving paths. Conversely, it is generally not possible to change the location of a file without changing its name in replicated prefix table systems.

4.4 Remote Links

The most flexible method for partitioning the file system namespace across multiple nodes uses permanent *remote links* (as opposed to temporary remote links, as mentioned earlier), which are hard links that point from a directory entry on one node to an inode on a different node. An example of this approach is FlexGroup volumes [30], which is a distributed file system that is built on top of NetApp WAFL® [24] and is illustrated in Figure 5. Another example of a distributed file system with remote links is Slice [8].

Remote links thus avoid the need to share a prefix table across the nodes; the nodes need to agree only on which node hosts the root of the file system. These links are typically explicit, indirect pointers (see Section 3.1.3): They do not encode the IP addresses of the target nodes directly but instead specify them as node IDs. Or in the case of FlexGroup volumes, they encode local volume IDs, since a node can host more than one participating local volume, which allows coarse-grained rebalancing without the need to update the links.

Remote links enable fine-grained file placement, where any file or directory can be placed on any node, or in the case of FlexGroup volumes, on any local file system on any node. Data placement can be influenced by the potentially conflicting goals of balancing capacity and load while ideally also preserving locality [30]. And unlike the prefix tables, moving files and directories does not actually result in data movement. Fine-grained rebalancing may be possible, if supported by the implementation, by allowing the movement of files and directories across nodes and by updating the links accordingly.

Unlike partitioning the namespace via replicated prefix tables, traversing a path can result in several network round-trips, potentially even one or more round-trips per path component because of traversing remote links. This approach is particularly costly if the links cross WAN-level distances, especially if the system is laid out such that parsing a path encounters WAN-level latencies more than once.

Encountering several network round-trips in a single file system request is fortunately a rare occurrence, because most practical systems maintain caches that map prefixes to the locations where those path prefixes are actually stored, similar to Sprite-style prefix tables [75]. A client or a node typically starts with an empty cache, and it builds the cache over time. An efficient way to detect stale mappings is when a client sends a request to a node that does not store the file or a directory any longer, the node replies with an error [7]. The client then removes the mapping from the cache and restarts the path traversal by using the second-longest prefix match from its cache.

Another disadvantage of this approach is its lack of partition tolerance, in both its weaker and stronger forms. If part of the path traverses nodes that are hosted on the other side of a partition, then everything on that subtree may become inaccessible—even if large portions of that subtree are ultimately stored on the same side of the partition as the client. Partitions can further cause problems with making sure that remote links and their caches are consistent.

Retrieving attributes of multiple files or subdirectories in a given directory can incur high overhead, such as when running `ls -l`, which fetches the inode across every single remote link that is found in the directory. One way to address this issue is to cache remote inodes (i.e., the inodes on the other side of remote links) locally, which then incurs the overhead of making these caches consistent. Another source of overhead from remote links is their updates, which often require executing multinode transactions, potentially even up to three nodes in certain cases of atomically moving a file between directories on different nodes. Because of these overheads, systems like FlexGroup volumes choose to minimize both the number and the update frequency of remote links.

Farsite [7] uses remote links to stitch together a relatively small number of large subtrees. The namespace on each subtree is managed by a group of 7 to 10 nodes, in which the nodes replicate the same metadata and use Paxos to coordinate.

Lustre [37] is another example of a distributed file system that uses remote links to stitch together a relatively small number of large subtrees. Lustre Distributed Namespace Phase I [15] allows the administrator to create a new subdirectory as a remote directory on a different metadata node from its parent. All new files and directories under that new subdirectory are then created in the new location.

4.4.1 CephFS Implied Remote Links.

A unique variation of remote links is CephFS [73], which partitions subtrees across nodes, then uses an in-memory cache (backed by a journal) to get the distributed benefits of remote links without actually having remote links. Each CephFS metadata server manages a different set of *local subtrees* and caches information about adjacent directories (including which server hosts them authoritatively) on each of its local subtrees: both parents, recursively all the way to the root, and direct children.

Path resolution can thus start at any node, and at each step, if the node's cache does not have information about the requested subdirectory, then path traversal can continue at the appropriate authoritative node until it reaches the subtree boundary. At that point, the traversal switches to another node [72]. A path traversal can thus involve several nodes, just like with remote links. The metadata server's in-memory cache—and by extension, its journal contents for persistence—thus contains equivalent information to remote links, even though they are not stored explicitly.

Another difference between CephFS and file systems that use remote links is that the directory inode is stored with its parent directory, not on the node that manages its contents.

4.5 Shared-Disk-Like File Systems

An important class of distributed file systems is built around a *shared-disk* abstraction, where every node in the cluster has relatively low-level access—i.e., block- or chunk-level access—to all disks in the cluster. This typically assumes a form of networking in which optimizing for locality is not as important (e.g., flat data center networking [45]), and it also assumes that there are no network partitions, which limits the environments in which these file systems can be deployed. One of the best-known examples is GPFS [58], in which nodes access shared storage at a chunk granularity and coordinate among themselves by using a distributed locking manager.

Such file systems often resemble local file systems, except with distributed shared storage and distributed locking. Directory entries simply reference inode IDs as in a local file system. But the inodes may be actually stored on another node in the cluster, and the system generally does not make any effort to keep related items colocated. In fact, to maximize parallelism, it is common for chunks of the same file to be evenly spread throughout the cluster.

A more recent example of a shared-disk-like file system with a somewhat different architecture is Oracle FSS [32], which builds a file system on top of a distributed direct-access storage device that supports conditional multipage writes. FSS uses this feature to implement a B+ tree, which stores all of the file system's metadata.

Shared-disk-like file systems share some characteristics of remote links, because many directory entry-to-inode links are remote. However, because these "links" are so commonplace, these systems are designed so that updating them does not have to be as limited. Shared-disk-like file systems typically spread both data and metadata across a large number of nodes in a very scalable manner (e.g., consider the way that GPFS [58] uses extendible hashing for directories, discussed in Section 4.9.2), which provides good load distribution and allows large clusters. Striping data across multiple nodes allows a file system to offer high aggregate throughput even for a single file, which is why shared-disk-like file systems tend to also be *parallel* file systems. (We talk more about parallel file systems in Section 5.2.) As a result, many shared-disk-like file systems are used for serving high-performance computing (HPC) workloads.

These kinds of file systems are different from systems such as Lustre [37], which support shared (e.g., dual-ported) storage for reliability but normally allow shared storage to be accessed only by a single server at a time. (We describe this architecture in Section 6.1.4.) Similarly, these shared-disk-like systems should not be confused with file systems that use a shared-disk abstraction only for data but not for metadata. For example, Orion [78], which is a distributed nonvolatile memory file system based on NOVA [77], manages metadata on a separate metadata server but allows clients to directly access storage on all data nodes by using RDMA.

4.6 Computed (Using Cluster Membership List)

Instead of storing the location of file and directory inodes explicitly (or even not so explicitly through prefix tables), several distributed file systems compute the location based on the file name or an internal inode or file ID, in a manner similar to computed (implicit) pointers (Section 3.1.4). Perhaps the simplest examples are distributed volumes in GlusterFS [21], illustrated in Figure 6. This mechanism replicates the directory structure on all nodes as already mentioned in Section 4.2 but places files on the individual nodes based on a hash, similarly to Dynamo-style DHTs [14]. Each such directory stores a map of hash ranges to nodes, and the file system places files according to the appropriate directories. GlusterFS can rebalance data as necessary by modifying the hash ranges and by moving data accordingly, such as in response to the addition and removal of nodes.

In the case of GlusterFS, this process is entirely client driven, which means that a client can determine and access the given node with a single network round-trip. The file system supports two mechanisms for dealing with stale mappings: a temporary remote link and a lookup request broadcast. If a user renames a file and the new file name hash falls within a range that is assigned to another node, then GlusterFS has to migrate the file to that other node. However, so that the operation can be acknowledged to the user quickly, GlusterFS creates a temporary remote link in the new location while the data migrates in the background. In the absence of a link file, such as

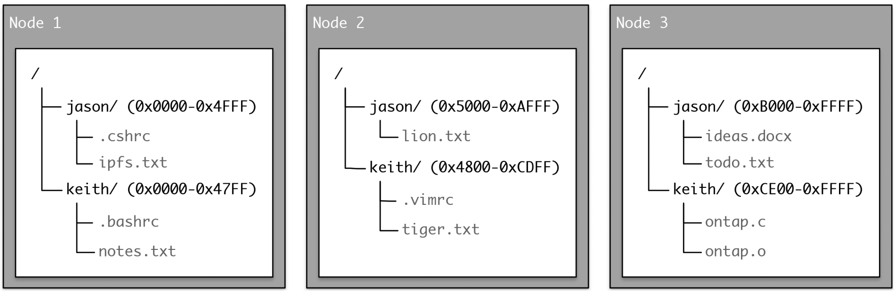

Fig. 6. A simplified example of a distributed volume in GlusterFS [21], which replicates the directory structure but places files on individual nodes based on a hash of the file name.

during rebalancing, if a client does not find the file where expected, then the client broadcasts the lookup request to all the nodes in the cluster.

Obviously, this approach depends on the file system knowing the list of members of the cluster; it does not deal gracefully with membership changes, and it is not partition tolerant.

4.6.1 Probabilistic Computed Location. One unique approach for overcoming these problems is used by Tahoe-LAFS [64], a decentralized peer-to-peer file system designed specifically for high membership churn and low trust in nodes. It uses an algorithm similar to Rendezvous hashing, also known as Highest Random Weight hashing [65, 66], to probabilistically compute file locations. To determine where to store a file with the unique key g, a Tahoe-LAFS client first computes $\mathrm{Hash}(g, s)$ for every possible server node s, then it sorts the values. The client then stores the erasure-coded fragments on the first N nodes, but requires only $k < N$ fragments to reconstruct the file; typically $N = 10$ and $k = 3$. When looking up a file, it repeats this process, but checks the top $2N$ nodes in parallel, determines the highest version number of the file that it encounters, and fetches the corresponding fragments from those nodes. If not enough fragments are available, then it checks further down the list.

The system periodically checks each file to make sure that enough erasure-coded fragments are available within the top $2N$ nodes, and if not, it uploads additional fragments. This approach, although somewhat expensive, allows the system to handle high membership churn and network partitions. If a network partition occurs, then each file that is available within the partition is periodically republished, and Tahoe-LAFS can always create new files. Because of the codes that the system uses to achieve its high reliability, a particularly significant source of overhead in Tahoe-LAFS is the CPU cost of erasure coding.

4.7 Overlay Network

A more common approach that bases file or inode location on file names or IDs uses *overlay networks*, typically based on DHTs as explained in Section 3.1.5. Overlay networks are decentralized, robust, and often partition-tolerant mechanisms for finding nodes based on the IDs or hashes of data that they host. The nodes are not required to be aware of the entire membership list, albeit at the cost of $O(\log n)$ network round-trips, where n is the number of nodes, as explained previously.

A recent example of a distributed file system that uses an overlay network is IPFS [9], which stores a mapping from file IDs to the corresponding lists of peers in a DHT overlay network based on Kademlia [38]. When retrieving a file, a client retrieves the list of peers, after which it can contact any of them to read the contents. Many other peer-to-peer file systems, such as OceanStore [31, 54], address both inodes and file contents by using the overlay network.

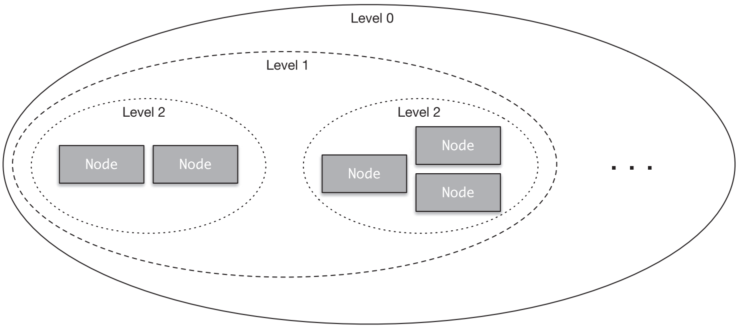

Fig. 7. Coral DSHT [19] organizes nodes into nested clusters based on pairwise ping latency. This method improves read latency for information that is stored on nodes closer in the network. IPFS [9] is an examle of Coral DSHT usage.

As an optimization, file systems such as IPFS that are designed to operate on a global scale sometimes use multiple nested clusters inspired by Coral DSHT [19], which makes retrieval of keys in nested clusters very efficient. Figure 7 illustrates Coral with three levels of nested clusters organized by ping latency; by default, level 2 requires round-trips between nodes to be less than 20 ms, level 1 must be less than 60 ms, and level 0 contains all the nodes. A node publishes information to all the clusters in which it participates, starting from the innermost one. During lookup, a node searches first within its innermost nested cluster, then progresses to the next level, until it ultimately reaches the global cluster, stopping as soon as it finds the key.

One example of how such file systems can be made partition tolerant (because not all of them are) is by letting nodes periodically republish the data they host to the DHT. The data are then accessible to other nodes on the same side of the partition. When the partition heals, the ID-to-peer mappings are stored on the nodes whose IDs are now closest to the file ID. Entries stored in such DHTs typically have expiration times (e.g., a Time to Live or TTL) to periodically eliminate stale entries.

4.8 Distributed Database

Several file systems use an existing distributed, sharded database management system (DBMS) to store the file system metadata, often both inodes and directories—in fact, the entire namespace. For example, ADLS [51] uses parts of Microsoft SQL Server's Hekaton engine [16]; CalvinFS [68] is built on top of Calvin [69] (a fast geo-distributed database); and Magic Pocket [12] heavily uses sharded MySQL [43]. Although most file systems are designed with a specific database engine in mind, HopsFS [44] can use any compatible NewSQL database engine, such as MySQL Cluster.

The use of existing, mature DBMS solutions greatly simplifies a file system's design by allowing it to reuse a lot of the DBMS' functionality, especially pertaining to consistency; coordination; replication; and, if applicable, transactions. Most of the properties of these systems, such as performance or partition tolerance, depend on the corresponding properties of the underlying database system.

4.9 Other (Orthogonal) Techniques

In addition to the aforementioned design options, several orthogonal techniques are also noteworthy.

4.9.1 File System Middleware. Some file systems, such as IndexFS [53], BatchFS [79], and DeltaFS [80], can be categorized as *file system middleware*. They do not store any data on their own, but they provide an abstraction or a value-add over one or more underlying distributed file systems or object stores.

For example, IndexFS [53] is a scalable metadata layer on top of existing storage, and BatchFS [79] extends it by allowing clients to check out a part of the namespace, modify it, and then efficiently check it back in. This optimization is useful for bulk-oriented workloads. DeltaFS [80] takes it a step further by providing decentralized namespace management with no global namespace; clients can almost arbitrarily import and combine existing namespaces and export their own namespaces.

4.9.2 Scalable Directory Implementations. If a large directory is stored on a single node, then it can quickly become a scalability bottleneck. This problem is particularly pronounced in workloads such as HPC, in which a large number of compute nodes generate and/or access many files in a single directory [53]. At the same time, it is important to make small directories efficient. Several systems, such as GPFS [58], GIGA+ [48], and IndexFS [53], provide directory implementations that are efficient for small directories and yet very scalable.

For example, directories in GPFS [58] use extendible hashing [18]: A directory starts as a hash table with one bucket, and if it exceeds a certain size, then GPFS splits it; also, buckets can be split recursively. A lookup of a specific directory entry can thus be accomplished by using a single network round-trip.

IndexFS [53] represents directories by using sorted string tables that are stored on underlying shared storage, but the basic principle is similar: A directory starts as a single partition, hosted by a single file system node, but when the partition size exceeds a threshold, IndexFS splits it by using hash partitioning and lets another node manage the other half. IndexFS can split partitions recursively as necessary, resulting in near-linear scaling. IndexFS also supports a very efficient batch import of changes to the directory.

4.10 Discussion

One of the underexplored areas in the research literature seems to be the ability to provide fine-grained inode placement that is flexible without changing the file paths. The only major approach that is flexible enough to do this is the use of remote links, but that approach has several major challenges, such as not being partition tolerant and not being suitable for WAN-level latencies.

Another example of an underexplored research area is how to overcome the disadvantages of decentralized path-to-inode translations. For example, it would be advantageous to reduce the lookup latencies in DHT-based overlay networks from $O(\log n)$ in the common case, or to make it easier to reason about data availability during partitions.

Other possible areas for research are novel combinations of the various approaches and development of new approaches for path-to-inode translation that are better than any of the existing solutions (or proving that such a thing does not exist).

5 INODE-TO-DATA TRANSLATION

After the file system translates a path to an inode and loads it, the file data can be stored locally on the same node as the inode, such as in AFS [25], Coda [57], and FlexGroup volumes [30]. Alternatively, the lookup process has to go through one or more levels of mapping—typically by using just one kind of pointer type or a combination of two pointer types from inodes—until the right data node is identified.

Unless the file system uses an overlay network (Section 3.1.5), identifying the node that houses the data is performed with only a single network round-trip. Even if the file system uses an explicit, indirect pointer or a computed pointer (Sections 3.1.3 and 3.1.4, respectively), the table or

Table 3. Separation or Colocation of Data and Metadata

Type	Latency	Throughput	Metadata Operations	Examples
Separate	Medium	Medium–Fast	Medium–Fast	ADLS [51], CalvinFS [68], Farsite [7]
Colocated	Fast	Medium	Slow–Medium	AFS [25], FlexGroup volumes [30]

membership list that is used to resolve the pointer is typically already replicated on all the nodes in the cluster. For example, CephFS uses a two-level mapping as follows:

(1) CephFS maps different parts of a file, called *objects*, to placement groups by using a hash function.
(2) Placement groups are then mapped to storage nodes by using the CRUSH function [74], which takes as an input the placement group ID and the membership list.

Because the membership list is already replicated on all participants, the data can be accessed by using a single network round-trip.

In the rest of this section, we discuss the relevant design decisions and tradeoffs.

5.1 Inode-to-Data Pointer Types

One of the key design choices is the mechanism that the file system uses in the inodes to point to data.

5.1.1 Separation or Colocation of Data and Metadata. The first consideration is whether to store metadata (inodes) and data separately or to colocate them; the tradeoffs for both are summarized in Table 3. Inodes and data are separated not only when the file system uses separate metadata nodes, such as in CephFS [73] or Farsite [7], but also when the system uses a distributed metadata service, such as in IPFS [9] or OceanStore [31].

Storing metadata on separate nodes can be advantageous especially in the following cases:

- When building a distributed file system on top of existing distributed storage, such as with CephFS [73] and HopsFS [44].
- When such architecture makes it possible to scale the metadata layer independently from the data layer, such as with IndexFS [53], DeltaFS [80], and CephFS [73].
- When an inode is allowed to reference parts or replicas of a file on different nodes, clients can achieve higher throughput by fetching data from the different nodes in parallel (analogous to striping in RAID), such as with Lustre [37] and HDFS [60].
- When the entire namespace can be stored on a single node, such as with GFS [20] and MooseFS [40].
- When reusing an existing DBMS or a DHT, such as with Magic Pocket [12] and IPFS [9].

Generally speaking, the latter two cases have the potential to implement multifile transactions more efficiently, or at least more simply, such as moving a file between directories that are stored on different nodes. The last case is particularly intriguing if the DBMS includes features such as distributed, atomic transactions.

In contrast, distributed systems that colocate metadata and data on the same nodes spread metadata across a larger number of nodes; systems that separate metadata and data can generally store more metadata on a single node, because metadata are smaller than data. Consequently, colocating data and metadata is likely to have a larger fraction of multinode transactions, with the corresponding negative effect on performance.

Table 4. Data Location (Separate Data and Metadata Only)

Type	Latency	Throughput	Data Location	Examples
Explicit	Fast	Fast	Flexible	Farsite [7], IPFS [9]
Hybrid	Slow–Medium	Medium	Inflexible	CalvinFS [68], Tahoe-LAFS [64]
Computed/Overlay	Slow–Fast	Medium–Fast	Inflexible	CephFS [73], OceanStore [31, 54]

Disadvantages of separating metadata from data include:

- The metadata server can become a scalability bottleneck.
- Even after the system has found and read an inode, it has to perform more network round-trips to read or to update the data, resulting in higher latency.

The latter issue can be overcome by using a hybrid approach (although it is not common): The first part of a file can be stored on the same node as the inode, so that no extra network round-trips are needed to start reading the data, while other parts of the file can be stored on different nodes to enable high parallel throughput.

5.1.2 Explicit, Hybrid, or Computed Data Location. If metadata and data are not colocated, then data can be referenced in one of the following three ways:

- *Explicit:* The data location is stored explicitly by using either a direct pointer (such as by specifying the node's IP address and the inode number) or an indirect pointer (such as by using a logical node ID or a volume ID instead of the IP address).
- *Hybrid:* The inode stores IDs of the data blocks, such as GUIDs (also called UUIDs) or content hashes, instead of their locations. The actual location is then determined from those IDs, either by using a computed pointer or by routing a request through an overlay network.
- *Computed/Overlay:* The location of data blocks is fully determined from the metadata, such as an internal file ID, or from the file name itself; the inode does not contain any information about the blocks. The final mapping can be performed with computed pointers or an overlay network.

Table 4 summarizes several tradeoffs, although the tradeoffs are somewhat more nuanced than can be expressed in a simple table. The main practical difference between the three approaches is their ability to evenly distribute load across different nodes in the cluster.

Referencing nodes (or virtual nodes or similar concepts that then directly map onto nodes) by using the explicit approach allows the system to distribute the load and capacity evenly, even taking into account up-to-date metrics, such as CPU utilization or used storage capacity [30]. To keep the load and capacity balanced, some file systems may even allow the ability to migrate data after the initial data placement. This level of flexibility makes the file system extremely well positioned to provide the best possible latency and throughput.

Computing the location of data from the file ID or file name (i.e., the third case) does not allow such flexibility, but it is usually still possible to achieve good throughput and—unless overlay networks are used—good latency. However, data cannot be placed naively. A common pitfall is to place different parts of a file (typically identified as a combination of file ID and a block number) by using a hash function. Even though the hash function produces a uniform distribution, ultimately one node receives a significantly higher number of blocks than others do [45]. This result is known as the *balls-into-bins problem.*

A possible approach to overcome this problem is to pick the first data node for the first block of the file in a sensible way, such as by using a uniform hash function, and then to place the rest

Table 5. Data Placement Granularities

Granularity	Throughput	Flexibility	Examples
Subtree/Bucket	Medium	Inflexible	AFS [25], Data ONTAP GX [17]
File	Medium	Flexible	Farsite [7], FlexGroup volumes [30]
Subfile	Fast	Flexible	CephFS [73], GFS [20], HDFS [60]

of the blocks relative to that. For example, if nodes are numbered $0 \ldots (n-1)$ and the file ID is g, then block i of the file should be placed on node $(\text{Hash}(g) + i) \bmod n$. And to further even out the load, Flat Datacenter Storage employs more techniques on top of this method [45]. However, because this approach requires knowing the full list of nodes, it does not work well with overlay networks.

One advantage of computed location is potentially simplified reader/writer coordination: Readers can immediately see the newly updated (overwritten) data without needing to reread the inode. Depending on the file system's implementation, this may or may not apply to data that are appended to (or otherwise written beyond the end of) the file. Another advantage of computed location is that writing data requires smaller updates to inodes, or possibly even no updates at all. CephFS [73] cites the latter as an important advantage, because it significantly decreases the load on its metadata servers.

The hybrid case, when the inode contains block IDs whose location is then computed or determined by an overlay network, results in a generally inflexible system. Even if the data are placed by using a uniform hash function, it may still overload a single node due to the aforementioned balls-into-bins problem, but without having an effective way to spread the load more evenly as in the computed case. However, other techniques may be able to mitigate the load on the hot nodes, such as caching data along common routing paths in the case of DHT-based overlay networks. The hybrid case is, however, particularly useful when implementing deduplication.

5.2 Data Placement Granularities

Another important consideration for a distributed file system is the granularity of data placement—how much data must be stored together on a single node. For example, flexible, fine-grained data placement is important for systems that dynamically move data or add replicas depending on the usage, such as IPFS [9], or for enabling fine-grained rebalancing. Table 5 summarizes the options and a few tradeoffs.

For distributed file systems that store inodes with their data, the granularity at which data are managed is identical to the granularity of metadata management. For example, AFS [25] employs prefix tables (Section 4.3), so it inherits subtree granularity. FlexGroup volumes [30] use remote links, so the whole system inherits file granularity. Storing inodes with their data generally implies that such systems can have only subtree, bucket, or file granularities.

File systems with *subtree* granularity store data under a subtree together, such as in AFS [25], and *bucket* granularity extends the concept by allowing the system to store potentially unrelated data together, such as in Magic Pocket [12]. The problem with subtree and bucket granularities is that they are inflexible, because a potentially large amount of file data must be stored together on a single node. This inflexibility can cause problems with imbalance in node capacity and load, without an easy way to rebalance the data to fix the problem. Fixing the problem likely involves significant data movement, such as splitting a subtree onto multiple nodes.

However, *file* granularity does not necessarily guarantee file-granular rebalancing either. For example, FlexGroup volumes [30] make data placement decisions only on ingest but do not move data afterward due to interplay with the NFSv3 protocol. However, an analysis of existing deployments showed that such imbalance is not a problem in practice.

In systems that colocate inodes and data as explained previously, managing data placement at a subtree/bucket level or a file level, despite being seemingly inferior in isolation, is often mandated by other parts of the system design, and the advantages of those other design decisions are often significant. For example, AFS [25] uses subtree granularity because of its use of prefix tables, which gives the system its low latency. Coarser data placement granularity, generally speaking, simplifies the system design. For many use cases, the improvement in throughput or flexibility that a finer granularity would provide is not important enough to warrant the extra complexity.

From the perspective of throughput, subtree and file granularities are similar, because the file's inode references data on a single node either way. In contrast, subfile granularity can improve the aggregate file throughput, because the data can be fetched from multiple storage nodes in parallel, possibly by multiple clients. Many such file systems also allow large files to be written concurrently by multiple clients in parallel, with different clients writing to different storage nodes to achieve high parallel throughput.

5.2.1 Parallel File Systems. File systems with subfile granularity match the definition of *parallel* file systems (sometimes called *parallel architecture* file systems). The definition specifies that a parallel file system is a file system in which "data blocks are striped, in parallel, across multiple storage devices on multiple storage servers" [67], typically to provide high performance. Informally, parallel file systems are often equated with file systems that are suitable for HPC environments, and in fact, most file systems that are used in HPC fall into that category.

A well-known example of a highly scalable parallel file system is Lustre [37], which stripes large files across a potentially large number of storage nodes. It is thus possible to achieve high write throughput if the clients are coordinated enough to write to different nodes. GPFS [58] is another example of a scalable parallel file system that is designed for HPC workloads.

However, the literal interpretation of the definition of parallel file systems allows for other file systems with subfile granularity, even if they are not typically used in HPC workloads, such as GFS [20] and even IPFS [9].

5.3 Mutable or Immutable Data

Another fundamental design decision is whether to treat data and possibly even metadata as immutable. Immutability has many benefits, such as simpler coordination, versioning, and snapshots, but it also has a significant negative consequence: The resultant high degree of data and metadata sharing makes data deletion extremely difficult, to the point that many such systems never delete old blocks of data. Examples of such systems are CYRUS [11] and Ivy [42]. Development of efficient, fault-tolerant, and partition-tolerant methods for garbage-collecting of unneeded immutable data would be a worthwhile area for research.

5.4 Client Library or Gateway

Another common difference between distributed file systems is whether the file system code lives in the client libraries, or whether a client accesses a storage node via some standard protocol, such as NFS or CIFS. An advantage of the former is an additional opportunity for optimization; for example, if a client knows the location of data, such as with CephFS [73], then it can access the data directly without going through a central gateway. This approach eliminates the need for gateway nodes, through which all requests must be processed and which can be scalability bottlenecks. By reducing the number of network round-trips, it also decreases latency and potentially increases throughput.

An advantage of the latter is a lower barrier to adoption, because many IT organizations have strict vetting of which code runs on the clients. An option in the middle is a protocol like pNFS [59]; among other features, pNFS allows the metadata server to directly specify which parts of the file are located where.

6 REDUNDANCY AND COORDINATION

Distributed file systems are trusted to store data and to make that data available upon request. Being distributed, however, means that there are more ways for things to go wrong than with a local file system; not only can disks fail, but so can nodes, networks, racks, and even data centers. How different systems account for these failures varies, reflecting diverse goals and tradeoffs based on the requirements of the file system. Some systems replicate data to increase availability (or use erasure coding to reduce storage overhead), and others offer eventual consistency to speed things up when it is not critical to have the most recent data. But complexity always comes at a price, and in distributed file systems, that price is some combination of additional overhead for storage, CPU, network bandwidth, and network latency; tolerance for stale data; and the durability of data.

The parameters discussed next need not be static for a particular system or data. They can be adjusted dynamically (e.g., by policy, API call, or machine learning), even on a per-I/O basis [47]. For example, highly read data and important metadata can have more copies for durability or performance reasons, or durability parameters can be tuned to match the measured reliability of the underlying media [27].

In this section, we delve into deeper detail on these techniques, discuss their overheads and offer some examples. Several of these techniques have been used in database products for some time. For a detailed examination, refer to Viotti and Vukolic [70].

6.1 Durability

6.1.1 Replication. Most distributed file systems use replication to ensure data availability and durability if failures occur. Often, we can characterize replication with three constants:

- n = the number of replicas
- r = the number of replicas that a reader contacts
- w = the number of replicas to which a writer writes synchronously (i.e., the number of replicas that must acknowledge the write before it is acknowledged to the client)

In general, if we have $r + w > n$, then the read set and the write set of each data item overlap, so the system offers strong consistency [71]. The reader thus sees all the writes, but it may have to resolve any conflicting updates that the file system has not yet resolved in a consistent manner.

A more common method of ensuring strong consistency is to have a designated primary node for each piece of data and to require it to be one of the w nodes that must acknowledge the write synchronously. Then the reader can contact the primary node whenever strong consistency is necessary, or it can contact any node if it is acceptable to read slightly stale data.

In most distributed file systems, $r = 1$ and $w = n$, which results in a strongly consistent file system. But in some cases, $w < n$ or even $w = 1$, which describes asynchronous replication when $n > 1$. Only a few systems have $r > 1$ or $n > w > 1$, despite its being a more common occurrence in NoSQL systems.

6.1.2 Erasure Coding. Replication meets the needs of many systems; however, a family of techniques called *erasure coding (EC)* or *forward error correction* can produce comparable protection (e.g., tolerating two disk or node failures) while significantly reducing the storage overhead. It comes at the expense of additional network and computational overhead.

XOR is the fastest and most basic EC technique; it computes a simple parity block from n blocks using their bitwise eXclusive OR. This technique offers a useful property: If one of the original n blocks is lost, then, to recompute the missing block, just XOR the remaining $n - 1$ blocks with the computed parity block. For example, to be able to withstand a single data center failure, Facebook's f4 [41] uses XOR to erasure-code data across data centers. Another example of a system that supports XOR erasure coding is MooseFS [40].

In practice, many distributed systems that implement EC use more complex Reed-Solomon codes [52] to allow an arbitrary number of parity blocks to be generated at the expense of higher computational overhead. $RS(N, k)$ refers to a Reed-Solomon code that encodes an object into N blocks, of which any k can produce the original object; the space overhead is N/k. As an example, because Tahoe-LAFS [64] is a distributed file system that is designed for use with less reliable servers, it implements $RS(10, 3)$ by default. Data can be recovered even if 7 of the original 10 servers are no longer available, while having a space cost of only $3.\overline{3}$ times the original data. Other examples of distributed systems that support such EC are LizardFS [35] and HDFS 3.x [60], which support both Reed-Solomon and XOR codes.

For high performance, many systems keep the first k EC blocks unmodified from the original data, so the blocks can be accessed without additional network requests or computation. The additional parity blocks are then needed only during failure/repair scenarios. EC algorithms that include the original data in the output blocks are *systematic*, an example of which is Reed-Solomon with appropriate parameters.

Other EC algorithms can improve upon Reed-Solomon's computational or I/O overheads [49]. For example, Microsoft Windows Azure Storage [26] and HDFS-Xorbas [55] use *Local Reconstruction Codes* or *Locally Repairable Codes*, which reduce disk and network I/O during reconstruction but are not as space efficient as Reed-Solomon codes are.

Some storage systems combine different erasure coding methods to capture the benefits of two or more methods. For example, HACFS [76] uses a fast but less space-efficient EC for hot data and uses a slower but more compact EC for cold data. Facebook's f4 [41] uses a Reed-Solomon code within a data center but uses XOR across data centers, which minimizes the number of data centers that are involved in writes and degraded reads.

6.1.3 Placement. The placement policies for n replicas—and in the case of $w < n$, also for where the various writes must land—often take into account failure domains so that a single network or power failure is less likely to affect the availability of data. Domains can include fate-shared components within a data center (such as nodes and racks) as well as data centers themselves. For example, HDFS [60] addresses node and rack availability by placing at least one replica on a different rack, and Dropbox's Magic Pocket [33] and Facebook's f4 [41] erasure-code data across data centers. We can generalize such data placement policies as having d failure domains (e.g., racks or data centers) with the three parameters separated into $n_{0\ldots d-1}$, $r_{0\ldots d-1}$, and $w_{0\ldots d-1}$.

A higher number of data centers, and replicas in general, improves durability but increases the bandwidth consumption of writes. Requiring writes to be synchronously acknowledged from several failure domains has an adverse effect on write latency, and performing them asynchronously primarily affects the peak write bandwidth. However, a higher number of replicas and failure domains improves read latency, because reads can be satisfied from closer replicas, and/or (depending on the system) read bandwidth improves if the reads can be satisfied from several replicas in parallel.

6.1.4 An Alternative to Shared-Nothing Replication. Even though most distributed systems implement redundancy in a shared-nothing manner, where memory and storage of a node are not accessible to other nodes [62], several file systems support the use of shared, dual-ported storage

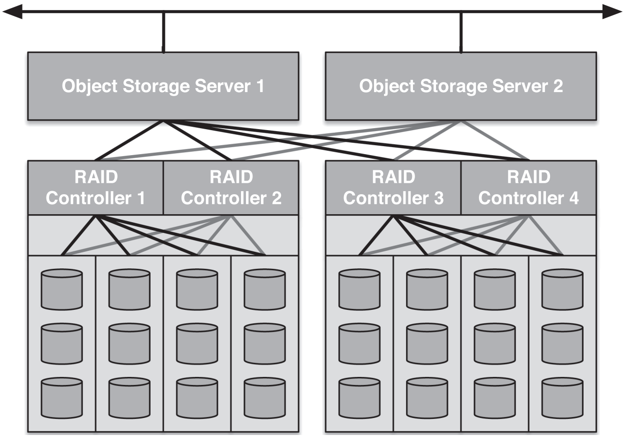

Fig. 8. An example of a redundant, highly available building block of Lustre's object storage service [37], which consists of two servers and two RAID arrays with redundant controllers.

to provide redundancy. Depending on the failure scenario, this approach obviates or lessens the need for redundancy above the storage layer.

Figure 8 illustrates an example of a building block in a highly available object storage configuration in Lustre [37], which consists of two dual-ported RAID arrays and two object storage servers, with both RAID arrays connected to both servers. The RAID arrays in this example also feature redundant controllers. In this way, Lustre is protected against individual server failures, individual disk failures, and individual RAID controller failures.

Another example with a similar architecture is FlexGroup volumes [30], which is built on top of a collection of high-availability pairs of storage controllers, where a controller can access and take over its partner's storage if the partner fails. This architecture provides data protection and high availability if disk and controller failures occur, and if more redundancy is required, then data can be replicated synchronously or asynchronously on another cluster.

6.2 Internal Consistency Mechanisms

File systems differ in how they manage their *internal* consistency; that is, the consistency of metadata and internal data structures. Examples are the consistency between replicas of the same data or metadata and the atomicity and consistency of operations that involve multiple files or directories. For many file systems, this mechanism is important to ensure consistency if failures occur, and they also use the same mechanism to ensure atomicity and consistency of concurrent operations by different clients. In Section 6.3, we survey approaches to multiclient coordination.

One of the most common methods for coordination is ensuring that a group of nodes always agrees on the same state, such as through the use of replicated state machines, Byzantine agreement protocols (such as Paxos), two-phase commit, three-phase commit, and their variants. The specific choice depends on the desired performance and failure tolerance tradeoffs. For example, CalvinFS [68] uses Paxos to create an ordered list of pending transactions. Pond [54] achieves high failure tolerance by using a variant of a Byzantine fault-tolerant protocol that was originally developed by Castro and Liskov [10]. And Oracle FSS [32] implements two-phase commit by using Paxos, which gives it better failure resilience properties than pure two-phase commit does. Two-phase commit by itself is a fast coordination protocol with just two network round-trips, but it is not resilient to coordinator failures.

Table 6. Coordination Taxonomy

Write to / Coordination by	Partition Tolerance	Write Perf.	Latest Version Is on	Examples
Single dedicated master	Consistent	Medium	The master	AFS [25], CephFS [73]
Quorum of *m* replicas	Consistent	Slow	*m* replicas	CalvinFS [68], Farsite [7]
Any replicas, resolve conflicts later	Available	Fast	A replica	Coda [57]
Any nodes, write to replicas later	Available	Fast	A replica, typically	WheelFS [63]
Any nodes, data has no home	Available	Fast	Any node	DeltaFS [80], IPFS [9]

Several other noteworthy approaches include the following:

- *No Coordination*: In some cases, the overhead or the complexity of coordination is simply not worth the benefits, so the systems instead opt for just a best-effort approach that may not be entirely fault tolerant. For example, Flat Datacenter Storage [45] forgoes coordinating data in replicas and synchronizes only the metadata.
- *Persistent Log*: File systems can also ensure crash-consistency through the use of persistent logs that survive individual node failures and by replaying those logs when a node fails; examples of such systems are GPFS [58], FlexGroup volumes [30], and IndexFS [53]. Ivy [42] takes it a step further and also uses logs to undo the operations of rogue users.
- *Distributed Locking*: Another approach is to lock all the required resources before they are modified or even just accessed, depending on the desired level of consistency. Locks typically govern entire objects or blocks, but byte-range locking can also be implemented. A noteworthy example of such a system is GPFS [58].
- *Optimistic Concurrency Control*: In contrast to pessimistically acquiring locks, it is often possible to optimistically assume that most file system objects are not being concurrently accessed. The file system keeps track of the objects that it read and wrote during an operation (called the *read sets* and *write sets*) and validates that there are no conflicts with other concurrent operations before committing them. Although this approach is common in the database literature, it is underexplored in the file system literature. A noteworthy example of such a system is CalvinFS [68], which traces its lineage to the database community. Pond [54] supports application-specific consistency for data objects, which is flexible enough to implement optimistic concurrency control.
- *Optimistic with Conflict Resolution*: Similarly to optimistic concurrency control, the system can proceed without acquiring any locks, but the difference is that there is no strict validation step at the end. A node instead accepts all changes that are valid according to its view of the state and resolves conflicts later. The main disadvantage of this approach is that many kinds of conflicting updates require manual intervention, although they are usually rare. A prominent example of such a system is Coda [57].

Combinations of some of these approaches are possible. In some file systems, one of the nodes is responsible for ensuring coordination, but in others, the client takes on this responsibility.

6.3 Multiclient Coordination

Table 6 summarizes five major mechanisms for coordinating concurrent file system operations and qualitatively examines the relevant tradeoffs: partition tolerance, write performance, and the location of the latest version of an object. These mechanisms are orthogonal to locks and leases through which clients can coordinate explicitly.

6.3.1 Consistent Mechanisms. The most common mechanisms that are found both in the literature and in practice favor file system consistency over availability during partitions. Out of these mechanisms, the most common approach designates one of the replicas as the primary replica for a given file, directory, or block, which then handles all writes and, if applicable, consistent reads. This solution is the simplest, because it creates a single point of coordination between writers in the common case of only one file system object being involved (e.g., writing to a file), but there is a risk that the given node could become a bottleneck. The master node can then update the remaining replicas synchronously or asynchronously by using one of the internal coordination mechanisms, such as two-phase commit. Consistent reads are typically implemented by contacting the master node.

The other mechanism that favors consistency over availability uses a quorum of replicas to determine the global order of concurrent operations, typically by using Paxos or a similar protocol. Performing a round of Paxos for every single operation would be expensive, so several systems, such as CalvinFS [68], first aggregate operations into batches and then get the quorum to agree on the total order of batches. The result is a more scalable and fault-tolerant approach but at the cost of extra write latency. Consistent reads can be performed by enqueuing them together with the writes to establish their global order [68].

Using n, r, and w, and defining m to be the quorum size, with $m = 1$ in the case of single-master systems, we can express the updating of all replicas synchronously as $w = n$ or asynchronously as $w = m$. Single-master systems that synchronously update $1 < w < n$ replicas or quorum-based systems that update $m < w < n$ replicas are uncommon. In the case of $w = m$, a reader can perform a consistent read either by contacting $r > n-w$ nodes and getting the latest version or, much more commonly, by contacting the master node in single-master systems or by being enqueued with the write operations in quorum systems. Design points with $1 < r < n$ are uncommon despite being used in NoSQL. It is possible to achieve intermediate forms of consistency as well, such as causal consistency, by using vector clocks as in GlobalFS [46].

In the PACELC classification [6], a file system that always performs consistent reads is PC/EC (consistent during partitions and during normal operation); a file system that performs fast, inconsistent reads is PC/EL (consistent during partitions but otherwise optimizing for latency). Some file systems, such as CalvinFS [68] and Pond [54], allow clients to specify whether they want to perform a consistent or an inconsistent read.

6.3.2 Available Mechanisms. Some file systems choose availability over consistency during partitions, and in some cases even during normal operation, at the expense of introducing conflicts, in which different clients can see different versions of the file system.

The most straightforward approach is to allow clients to write to any available replicas, such as in Coda [57], and to resolve conflicts later. This method allows a client to write as long as it can contact at least one replica, and it generally improves write performance, because the writes can be performed by any replica without the need for coordination. It is particularly useful in geo-distributed systems, where clients are allowed to write only to the closest replica or replicas.

Locating the most recent data is slower, however, because typically all n nodes must be contacted. For example, Coda ensures consistent reads during normal operation by contacting all available replicas and by determining which one has the latest version. Inconsistencies can be detected proactively, or in the case of Coda, lazily during reads, and they can be resolved automatically with file-type-specific routines or with manual intervention. Simple inconsistencies in the directory structure can often be resolved automatically, such as if clients on the different sides of the partition add differently named files to the same directory. Being able to automatically resolve more complicated kinds of inconsistencies would be a worthwhile area for future research.

A disadvantage of this approach is that a client cannot write to a file if all of the file's replicas are on the other side of a network partition. WheelFS [63] resolves this issue by allowing applications to create files on any node in the cluster and to later migrate them to the correct places. But in general, all writes to an existing file must be processed by the file's primary node, or if the user enables eventual consistency, either by the primary node or by one of its replicas.

Taking this approach to the extreme, in some file systems such as IPFS [9], data has no authoritative home. Any node can accept any write, but it also means that the file system either must do extra work to determine the location of replicas, or that information must be communicated externally to the file system. An example of the former is the use of a DHT in IPFS [9]; an example of the latter is DeltaFS [80].

The use of n, r, and w to analyze these systems is tricky at best, because the number of nodes that these systems contact during reads and writes depends on how many of them are reachable. Also, sometimes the system behavior changes if no replicas are available, such as in WheelFS. For systems in which data has no authoritative home, the number of replicas, n, may not even be defined. Many available systems choose to write only $w_{\min} = 1$ copy of the data before acknowledging the request, even if they would prefer to write more copies if more replica nodes were available. Similarly, they would be satisfied to contact only $r_{\min} = 1$ replica if only one were available, even if, for example, Coda would during the normal operation prefer to check all $r_{\max} = n$ replicas.

In the PACELC classification [6], Coda is PA/EC (available during partitions, but consistent during normal operation). File systems that read only one of the replicas that are not guaranteed to always return the latest version are PA/EL (available during partitions and optimizing for latency over consistency during normal operation).

6.4 Leases

Many file systems, such as AFS [25], CephFS [73], and Farsite [7], grant *leases* (sometimes called *capabilities*) to clients to allow them to cache reads and to buffer writes to improve latency if there are no other readers or writers. Leases in many systems are always exclusive, but in other systems, they can be shared. For example, Farsite [7] has an elaborate mechanism of leases and their compatibilities to emulate most of the semantics of NTFS.

Leases in most systems are long-lived and revocable. For example, a client releases a lease on a file only when the file is closed; sometimes clients hold on to leases even longer if they expect that the user may reopen the file soon. The file system revokes the lease when another client opens the file. In some systems, such as IndexFS [53], leases are short-lived, which greatly simplifies recovery: If a node that manages leases fails, then the node that assumes its place simply waits for the longest possible allowed period for a lease before granting any new leases.

6.5 POSIX Semantics

Some file systems, such as CephFS [73], GPFS [58], and MooseFS [40], provide POSIX or near-POSIX semantics, but many others, such as GFS [20], HDFS [60], and IPFS [9], do not. POSIX not only prescribes specific APIs, metadata (e.g., the standard file attributes), and features (e.g., files with more than one hard link), but it is also fairly specific about the semantics of various file operations. Many difficulties can arise from trying to support some of these semantics, especially with respect to writes.

For example, one of the biggest challenges is the requirement for strong consistency: The read that immediately follows a write must return the most recently written data, even if the data were written by a different client [50]. It is generally not that hard to achieve if a file is opened by only one client at a time. Some file systems simply get around this problem by relaxing the POSIX semantics by providing a weaker consistency model, such as close-to-open consistency, which is the standard for NFS and is sufficient for many applications.

The two most common approaches for achieving strong consistency with multiple clients are as follows [36]:

- Clients commit writes immediately to the distributed file system without any write-back caching, and the file system immediately replicates the data by using one of the aforementioned strongly consistent mechanisms. For example, CephFS [73] normally uses leases, but it falls back to this behavior whenever a file is opened by multiple writers (or multiple clients with at least one writer)—unless they disable this behavior when they open the file by specifying the `O_LAZY` flag.
- Clients acquire byte-range locks on nonoverlapping parts of the file. For example, this approach is used in GPFS [58] and Lustre [37].

A file system can use the same two approaches to address another difficult part of POSIX semantics: the requirement that overlapping writes from multiple clients be applied serially without interleaving.

Another challenging requirement is for the file size and the last modification time to always be accurate. Updating these two values synchronously with the writes limits the system's scalability, so file systems more commonly just ensure that they return the correct values when they are read, such as by using the `stat` call. For example, CephFS [73] suspends all writers, collects relevant information from the writers, resolves the values, and then sends them back to the caller. In GPFS [58], write operations acquire a shared write lock on the file's inode, which conflicts only with operations that require the exact values, such as `stat`. Most file systems ignore—or at least provide a configuration option to ignore—updating the last access time (`atime`) on every read.

Yet another difficulty that stems from supporting POSIX semantics is the extra load that it places on the nodes due to tracking additional state per client or per open file. Other challenging parts of POSIX semantics include: atomic renames across directories, the ability to read and to write to an unlinked file until it is closed, and files with more than one hard link.

7 CASE STUDIES

Now that we have provided a taxonomy of various design choices for the different parts of the design stack, we will use it to briefly describe several noteworthy file systems. At the end of the section, we will use our taxonomy for a quick file system design exercise.

Table 7 summarizes the design options that have been selected by some of the file systems that we used as examples in this article.

7.1 AFS and Coda

Table 8 summarizes the key design choices in AFS [25] and in Coda [57], which is an evolution of AFS that adds replication and optimizes for availability and performance over consistency. We first discuss AFS and then Coda.

The path-to-inode translation in AFS uses a replicated prefix table, which maps path prefixes to volumes, which are then mapped to nodes. The inode-to-data translation is then very simple, because AFS colocates data and inodes on the same node. Moreover, no replication occurs, which significantly simplifies the system design.

The use of a replicated prefix table implies that metadata are partitioned by subtrees, and it takes a single network round-trip to access any file or directory. Inodes and data are colocated, so all data can be accessed within the one network round-trip, which results in low latency. However, moving files can result in data movement if the files are moved across volumes, and there is no easy or built-in method to rebalance data if one volume becomes too hot or starts to run low on free space.

Table 7. Key Design Decisions in Selected File Systems

System	Path to Inode	Separate Data and Metadata	Data Location Pointer	Data Location Granularity	Multiclient Coordination
AFS	Prefix table	No	N/A	Subtree	Single master
CalvinFS	Sharded database	Yes	Hybrid	Subfile	Quorum
CephFS	Remote links	Yes	Computed	Subfile	Single master
Coda	Prefix table	No	N/A	Subtree	Resolve conflicts later
Farsite	Remote links	Yes	Explicit	File	Quorum
FlexGroup volumes	Remote links	No	N/A	File	Single master
GFS	Single master	Yes	Explicit	Subfile	Single master
GPFS	Shared disk	No	N/A	Subfile	Single master
HDFS 1.x	Single master	Yes	Explicit	Subfile	Single master
HDFS 2.x and 3.x	Independent masters	Yes	Explicit	Subfile	Single master
IPFS	Overlay network	Yes	Explicit	File or subfile	Data has no home
Lustre	Remote links	Yes	Explicit	Subfile	Single master
OceanStore/Pond	Overlay network	Yes	Computed	Subfile	Quorum
Oracle FSS	Shared disk	No	N/A	Subfile	Quorum
Orion	Single master	Yes	Explicit	Subfile	Single master
Tahoe-LAFS	Probabilistically computed	No	N/A	File	No coordination
WheelFS	Remote links	No	N/A	File	Configurable

Table 8. AFS and Coda

System	(a) AFS	(b) Coda
Path to Inode		
Type	Replicated prefix table	Replicated prefix table
Inode to Data		
Separation of Data and Metadata?	No	No
Data Location Pointer	N/A	N/A
Data Placement Granularity	Subtree	Subtree
Mutable Data?	Yes	Yes
Client Library?	Yes	Yes
Redundancy and Coordination		
Redundancy	$n = 1, r = 1, w = 1$	$n > 1, r_{\min} = 1, r_{\max} = n, w_{\min} = 1$
Internal Consistency	N/A (no replication)	Optimistic with conflict resolution
Multiclient Coordination by	Single dedicated master	Any replicas, resolve conflicts later
Client-Server Coordination	Leases	Leases

Coda extends AFS by adding replication. It retains the replicated prefix tables and colocation of inodes and data from AFS, so it also inherits many of the AFS strengths and weaknesses. Coda uses replication with an optimistic coordination among nodes based on conflict resolution, where conflicts may sometimes need to be manually resolved. A consequence of this design decision is high partition tolerance, in which clients can continue to write even on the minority side of a partition.

These decisions thus align well with the use cases for which AFS and Coda have been designed, including home directories. The systems offer low latency, good colocation of data, and even disconnected operation. The limited degree of file sharing aligns well with the decisions about replication and coordination.

7.2 CephFS

Table 9, column (a), shows the design decisions of CephFS [73], which are quite different from AFS and focus on throughput and scalability over partition tolerance.

Table 9. CephFS and IPFS

System	(a) CephFS	(b) IPFS
Path to Inode		
Type	Remote links	Overlay network
Inode to Data		
Separation of Data and Metadata?	Yes	Yes
Data Location Pointer	Computed	Explicit
Data Placement Granularity	Subfile	File or subfile
Mutable Data?	Yes	No
Client Library?	Yes	Yes
Redundancy and Coordination		
Redundancy	$n > 1, r = 1, w = n$	$n \geq 1, r = 1, w = 1$
Internal Consistency	Persistent log	No coordination
Multiclient Coordination by	Single dedicated master	Any nodes, data has no home
Client-Server Coordination	Leases	None

In CephFS, the path-to-inode translation is accomplished through an approach that is reminiscent of remote links. This approach provides a very flexible mechanism for enabling its dynamic subtree partitioning, in which the namespace is dynamically rebalanced and distributed among the metadata servers according to the observed workload characteristics. Separation of data from metadata allows the system to rebalance the metadata without having to move any data. Storing metadata in a persistent log outside of the metadata servers simplifies failover and rebalancing.

One of the distinguishing characteristics of CephFS is that the location of data is computed instead of being stored explicitly, which significantly decreases the load on metadata servers. CephFS takes it further than most other systems by letting the computation determine the specific list of nodes for any particular object; most other systems let the computation only determine a “bucket” and then use a small replicated database that maps buckets to nodes.

The subfile data placement granularity enables high throughput, because a client can fetch different parts of a file from different servers in parallel. Separation of data and metadata costs CephFS some latency in the time to first byte when opening a file, but after the file is open, the client can locally determine which nodes to contact to read or to write bytes from/to any particular offset. For each underlying object, coordination is handled by a primary node, which simplifies the design.

One disadvantage of computed data locations that negatively affects CephFS is the high cost of adding and removing nodes due to rebalancing. CephFS is also not designed to be partition tolerant, but this aspect is not important for most of its intended use cases.

7.3 IPFS

Table 9, column (b), lists the design decisions in IPFS [9], which optimize for low trust, high resilience, moderately high node churn, scalability, and partition tolerance. The goal of IPFS is not only to be resilient during technical failures, but also to significantly reduce the ability for any administrative entity to significantly interfere with the file system operation. Those entities could be individuals, corporations, or even governments, who, for example, might have the power to add malicious nodes, shut down nodes, send malicious messages, and create network partitions (such as by interrupting a country’s connection to the internet). IPFS also enables very dynamic and fine-grained data movement, so that files are accessible close to where they are used, and more popular files are replicated more widely, which improves both resilience and performance.

The required high level of resilience and partition tolerance dictates the use of an overlay network for path-to-inode translations, because it can handle node churn and many implementations are also highly partition tolerant. A decentralized approach to computed path-to-inode translation (Section 4.6.1) might be the runner-up option, because it can handle high node churn. However, this method is not entirely appropriate, because it requires each node to know about most of the other nodes in the system, which limits both scalability and resilience. Data in IPFS are referenced by hash and are immutable, which both simplifies the system and makes it much harder for a rogue participant to corrupt files.

Two big disadvantages of overlay networks are low performance and the lack of control over where the data are stored. To overcome these issues and to provide both good performance and resilience, such as to prevent important files from being stuck on the other side of a network partition, IPFS addresses this problem in two ways. First, it uses Coral [19] to improve the locality of lookups in the overlay network, but this approach is mostly just an optimization (albeit an important one). The second way is perhaps more fundamental: IPFS stores file/subfile locations explicitly in the inodes, which gives it fine control over where data are stored, and it does not require data to have a home. Any file can be thus stored on any node, and in fact, an IPFS node by default creates a replica of all the files that it accesses. Given the assumption of geographical/network locality, this approach places files closer to where they may be needed next, creates more replicas of important files, and decreases the probability that a file could become inaccessible because of node failures and network partitions.

7.4 A Design Exercise

Now instead of analyzing existing file systems, let us pretend that we want to design a file system for home directory-like workloads for a medium-sized corporate customer. The workload requires low latency (even on time to first byte), good throughput (although not necessarily needing to optimize for high single-file throughput), low IT administration overhead, and easy expansion of the cluster. For this exercise, we assume reliable networking without partitions, and we assume the need for strongly consistent rather than available storage. Note that the following design may not be the only correct answer.

7.4.1 Path-to-Inode and Inode-to-Data Translation. We can easily eliminate several options for path-to-inode translation. Storing the entire namespace on a single node or replicating it on several nodes would not be scalable. The latter would be more scalable if we replicated only the directory structure, but it still would offer comparably low scalability over other options, especially if we wanted to keep low latency for directory operations. A shared-disk-like approach would require several network round-trips to parse a path and for nodes to coordinate. It would also separate data from metadata, which would increase latency further.

The computed approach by itself makes scaling more difficult, and latency would be higher, because descending into any directory would require contacting a different node and would make prefix caches much less effective.

Replicated prefix tables offer the best latency for path-to-inode translation. However, unless we separate data from metadata, there would be several disadvantages, including: expensive movement of files between certain subtrees, difficulty keeping free space balanced across nodes, and expensive cluster expansion. Separating data from metadata would solve such problems at the expense of extra latency in time to first byte, however.

Remote links are probably the best approach for path-to-inode translation. The time to first byte might require several network round-trips the first time that a file is accessed, but after the prefix cache has been populated, a file or a directory can be accessed with a single round-trip. Remote links make it easy to move files between any two directories, and they also make it easy to keep free space balanced and to nondisruptively expand the cluster, even when data and metadata are colocated.

And the use of remote links with colocated data and metadata is probably the best approach for inode-to-data translation. A runner-up might be replicated prefix tables with separation of data and metadata, which we can explore if we discover a problem with remote links later in the design exercise.

The next question is the type of pointers to use for remote links: explicit, computed, or hybrid. The "explicit, indirect" option might be best to give the system the most control over data placement so that it can balance free space well and so that cluster expansion does not result in any complications. Referring to nodes indirectly through an ID rather than through IP addresses is usually considered a good practice, because it is less fragile.

Because individual file throughput is not important for the intended use cases, and especially if we do not intend to host many very large files, file data placement granularity should be sufficient.

7.4.2 Redundancy and Coordination. If the distributed file system is built on top of highly reliable local storage—which might combine, for example, RAID, dual-socket drives, and a redundant cluster network—then replication at the node level might not even be necessary. This design is in fact reminiscent of FlexGroup volumes [30].

However, if any node can fail together with its local storage, then we need to use replication with $n > 1$ and $w > 1$. The choice of consistency over availability dictates that we process writes through a single dedicated master or a quorum of nodes; the former usually results in lower latency unless the node becomes overloaded. The master could coordinate with replicas by using a protocol such as two-phase commit or one of its variants. We could achieve $r = 1$ by requiring clients to always coordinate with the master or by requiring $w = n$ and weakening the consistency requirements by a bit to allow clients to read from any replica. The former could, however, result in an overloaded master node; the extent to which this overload would be an issue depends on the workload and on how masters are selected.

Remote links work best if each remote link has one source and one destination, which obviously works well when there is no replication ($n = 1$). However, it is possible to make it work with $n > 1$ as demonstrated by Farsite [7]. Farsite gathers nodes that host metadata (Farsite separates metadata from data) into small groups of 7 to 10 nodes; nodes in a group replicate the same metadata and use Paxos to coordinate. A remote link then links from one group to another group, giving it effectively one virtual source and one virtual destination.

If the replication factor in our file system is small, such as $n = 3$, it is conceivable to have groups of n nodes that are replicas of each other and still have colocated data and metadata.

This design is actually not too distant from Farsite [7], which additionally assumes that a small fraction of nodes can be actively malicious. Farsite thus uses groups of $n_{\text{metadata}} \geq 7$ for replicated metadata that use a Byzantine fault-tolerant protocol for coordination. But because this replication factor is too high for data, Farsite separates data from metadata for all but the smallest files and stores the files on data nodes with a smaller replication factor.

8 CONCLUSION

Despite the vast research literature on distributed file systems, much of it explores combinations of existing high-level designs, while improving upon them and/or exploring other interesting problems, such as with novel storage devices, new kinds of networks, and different security requirements. Although this kind of work is very valuable and should be continued, there may be opportunities to develop new high-level designs or to drastically improve existing ones, significantly contributing to the state of the art. Throughout this article, we have mentioned several ideas for future research. As such, our hope is not only for this article to be a survey of existing design options and their characteristics, but perhaps also to be an inspiration for future research.

ACKNOWLEDGMENTS

We thank Keith Smith, David Slik, and the anonymous *ACM TOS* reviewers for their helpful comments and advice. We also thank Anna Giaconia for copyediting this article.

REFERENCES

[1] 2000. *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI'00).* USENIX Association.

[2] 2002. *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02).* USENIX Association.

[3] 2015. *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15).* USENIX Association.

[4] 2019. *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19).* USENIX Association.

[5] 2019. *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19).* USENIX Association.

[6] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Comput.* 45, 2 (2012), 37–42.

[7] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment, see Reference [2].

[8] Darrell C. Anderson, Jeffrey S. Chase, and Amin Vahdat. 2000. Interposed request routing for scalable network storage, See Reference [1], 259–272.

[9] Juan Benet. 2014. IPFS—Content Addressed, Versioned, P2P File System. arXiv preprint. arXiv:1407.356.

[10] Miguel Castro and Barbara Liskov. 2000. Proactive recovery in a byzantine-fault-tolerant system, see Reference [1], 273–288.

[11] Jae Yoon Chung, Carlee Joe-Wong, Sangtae Ha, James Won-Ki Hong, and Mung Chiang. 2015. CYRUS: Towards client-defined cloud storage. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15).* Association for Computing Machinery, 17:1–17:16.

[12] James Cowling. 2016. Inside the Magic Pocket. Retrieved from https://blogs.dropbox.com/tech/2016/05/inside-the-magic-pocket/.

[13] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Tappan Morris, and Ion Stoica. 2001. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01).* Association for Computing Machinery, 202–215.

[14] 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 the 21st ACM Symposium on Operating Systems Principles (SOSP'07).* Association for Computing Machinery, 205–220.

[15] Wang Di. 2012. Distributed Namespace Status Phase I—Remote Directories. Retrieved from https://wiki.lustre.org/images/4/41/LUG-2012-DNE_Phase_1-WangDi.pdf.

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'13).* Association for Computing Machinery, 1243–1254.

[17] Michael Eisler, Peter Corbett, Michael Kazar, Daniel S. Nydick, and J. Christopher Wagner. 2007. Data ONTAP GX: A scalable storage cluster. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07).* USENIX Association, 139–152.

[18] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Datab. Syst.* 4, 3 (1979), 315–344.

[19] Michael J. Freedman, Eric Freudenthal, and David Mazières. 2004. Democratizing content publication with coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04).* USENIX Association, 239–252.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03).* Association for Computing Machinery, 29–43.

[21] Gluster. Storage for Your Cloud. Retrieved from https://www.gluster.org/.

[22] Ragib Hasan, Zahid Anwar, William Yurcik, Larry Brumbaugh, and Roy Campbell. 2005. A survey of peer-to-peer storage techniques for distributed file systems. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, Vol. 2. IEEE Computer Society, 205–213.

[23] Kirsten Hildrum, John Kubiatowicz, Satish Rao, and Ben Y. Zhao. 2002. Distributed object location in a dynamic network. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*. Association for Computing Machinery, 41–52.

[24] Dave Hitz, James Lau, and Michael A. Malcolm. 1994. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, 235–246.

[25] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.* 6, 1 (1988), 51–81.

[26] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. USENIX Association, 15–26.

[27] Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger. 2019. Cluster storage systems gotta have HeART: Improving storage efficiency by exploiting disk-reliability heterogeneity, see Reference [4], 345–358.

[28] Junbin Kang, Benlong, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huae. 2015. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, 249–261.

[29] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC'97)*. Association for Computing Machinery, 654–663.

[30] Ram Kesavan, Jason Hennessey, Richard Jernigan, Peter Macko, Keith A. Smith, Daniel Tennant, and Bharadwaj V. R. 2019. FlexGroup volumes: A distributed WAFL file system, see Reference [5], 135–148.

[31] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. Association for Computing Machinery, 190–201.

[32] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. 2019. Everyone loves file: File storage service (FSS) in oracle cloud infrastructure, see Reference [5], 15–32.

[33] Preslav Le. 2019. How we optimized magic pocket for cold storage. Retrieved from https://blogs.dropbox.com/tech/2019/05/how-we-optimized-magic-pocket-for-cold-storage/.

[34] libp2p. A Modular Network Stack. Retrieved from https://libp2p.io/.

[35] LizardFS. LizardFS. Retrieved from https://lizardfs.com/.

[36] Glenn Lockwood. 2017. What's So Bad about POSIX I/O? Retrieved from https://www.nextplatform.com/2017/09/11/whats-bad-posix-io.

[37] Lustre 2017. Introduction to Lustre Architecture. Retrieved from https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf.

[38] Petar Maymounkov and David Mazières. 2002. Kademlia: A peer-to-peer information system based on the XOR metric. In *Peer-To-Peer Systems: First International Workshop (IPTPS'02)*. Springer, 53–65.

[39] Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos D. Keromytis, and Sotiris Ioannidis. 2008. Decentralized access control in distributed file systems. *ACM Comput. Surv.* 40, 3 (2008), 10:1–10:30.

[40] MooseFS. MooseFS. Retrieved from https://moosefs.com/.

[41] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's warm BLOB storage system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 383–398.

[42] Athicha Muthitacharoen, Robert Tappan Morris, Thomer M. Gil, and Benjie Chen. 2002. Ivy: A read/write peer-to-peer file system, see Reference [2].

[43] MySQL. MySQL. Retrieved from https://www.mysql.com/.

[44] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 89–104.

[45] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen S. Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat datacenter storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, 1–15.

[46] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Riviere, and Pascal Felber. 2016. GlobalFS: A strongly consistent multi-site file system. In *Proceedings of the IEEE 35th Symposium on Reliable Distributed Systems (SRDS'16)*. IEEE Computer Society, 147–156.

[47] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, 217–231.

[48] Swapnil Patil, Garth A. Gibson, Samuel Lang, and Milo Polte. 2007. GIGA+: Scalable directories for shared file systems. In *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW'07)*. Association for Computing Machinery, 26–29.

[49] James S. Plank. 2013. Erasure codes for storage systems: A brief primer. *;login: USENIX Mag.* 38, 6 (December 2013), 44–50.

[50] POSIX.1-2017. Portable Operating System Interface (POSIX™) Base Specifications. IEEE Std 1003.1™-2017 and the Open Group Technical Standard Base Specifications, Issue 7. Retrieved from https://pubs.opengroup.org/onlinepubs/9699919799/.

[51] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. 2017. Azure data lake store: A hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. Association for Computing Machinery, 51–63.

[52] Irving S. Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *J. Soc. Industr. Appl. Math.* 8, 2 (1960), 300–304.

[53] Kai Ren, Qing Zheng, Swapnil Patil, and Garth A. Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Computer Society, 237–248.

[54] Sean C. Rhea, Patrick R. Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y. Zhao, and John Kubiatowicz. 2003. Pond: The OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association.

[55] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris S. Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing elephants: Novel erasure codes for big data. *Proc. VLDB Endow.* 6, 5 (2013), 325–336.

[56] Mahadev Satyanarayanan. 1990. A survey of distributed file systems. *Annu. Rev. Comput. Sci.* 4, 1 (1990), 73–104.

[57] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers* 39, 4 (1990), 447–459.

[58] Frank B. Schmuck and Roger L. Haskin. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association, 231–244.

[59] Spencer Shepler, Mike Eisler, and Dave Noveck. 2010. *Network File System (NFS) Version 4 Minor Version 1 Protocol.* RFC 5661. Internet Engineering Task Force (IETF).

[60] 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'10)*. IEEE Computer Society.

[61] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*. Association for Computing Machinery, 149–160.

[62] Michael Stonebraker. 1986. The case for shared nothing. *IEEE Datab. Eng. Bull.* 9, 1 (1986), 4–9.

[63] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Tappan Morris. 2009. Flexible, wide-area storage for distributed systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*. USENIX Association, 43–58.

[64] Tahoe-LAFS. The Least-Authority File Store. Retrieved from https://www.tahoe-lafs.org/trac/tahoe-lafs.

[65] David G. Thaler and Chinya V. Ravishankar. 1996. *A Name-Based Mapping Scheme for Rendezvous.* Technical Report CSE-TR-316-96. University of Michigan.

[66] David G. Thaler and Chinya V. Ravishankar. 1998. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw.* 6, 1 (1998), 1–14.

[67] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, SangBum Kim, and Pilsung Kim. 2008. A taxonomy and survey on distributed file systems. In *Proceedings of the 4th International Conference on Networked Computing and Advanced Information Management (NCM'08)*, Vol. 1. IEEE Computer Society, 144–149.

[68] Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems, see Reference [3].

[69] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD'12)*. Association for Computing Machinery.

[70] Paolo Viotti and Marko Vukolic. 2016. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.* 49, 1 (2016), 19:1–19:34.

[71] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[72] Sage A. Weil. 2007. *Ceph: Reliable, Scalable, and High-Performance Distributed Storage.* Ph.D. Dissertation. University of California at Santa Cruz.

[73] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, 307–320.

[74] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*. Association for Computing Machinery.

[75] Brent B. Welch and John K. Ousterhout. 1986. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS'86)*. IEEE Computer Society, 184–189.

[76] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David Pease. 2015. A tale of two erasure codes in HDFS, see Reference [3], 213–226.

[77] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 323–338.

[78] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks, see Reference [4], 221–234.

[79] Qing Zheng, Kai Ren, and Garth A. Gibson. 2014. BatchFS: Scaling the file system control plane with client-funded metadata servers. In *Proceedings of the 9th Parallel Data Storage Workshop (PDSW'14)*. IEEE Computer Society.

[80] Qing Zheng, Kai Ren, Garth A. Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW'15)*. Association for Computing Machinery.

Received May 2020; revised November 2020; accepted May 2021