

MojaveFS: Providing Sequential Consistency in a Distributed Objects System

Cristian Țăpuș, David Noblet, Vlad Grama‡ and Jason Hickey

Computer Science Department
California Institute of Technology
{crt,dnoble,jyh}@cs.caltech.edu

‡ Computer Science Department
“Politehnica” University of Bucharest
vgrama@gmail.com

Abstract

This paper presents MojaveFS, a distributed file system with support for sequential consistency. It provides location transparency and makes use of replication for reliability and fault tolerance. We employ a hybrid hash-based and tree-based lookup mechanism for files that, combined with an efficient caching scheme, provides fast access to files in the system. MojaveFS uses a novel data storage strategy where files are split into smaller objects to increase data availability. It also uses a group communication protocol with guarantees for a total order on messages sent within the system, enabling MojaveFS to support the traditional sequential consistency model for concurrent access.

1 Introduction

Modern high-end computing platforms are massively parallel, with processor counts ranging from the tens-of-thousands to the hundreds-of-thousands. Fault-tolerance and fast, reliable access to data are critical in such environments. While storage capacity has grown tremendously, IO reliability, throughput and latency remain as principal inhibitors of platform performance.

NFS [13] is the de facto standard in distributed filesystems. Although several incarnations of this exist (industrial and otherwise), they all lack clear semantics in the presence of concurrent accesses. NFS does not even provide strong guarantees for non-concurrent causally-related accesses. Thus, NFS requires applications to implement complex synchronization mechanisms outside the filesystem. Moreover, the centralized organization of traditional NFS prevents location transparency and is less resilient to

hardware failures.

We propose a new filesystem, MojaveFS, that addresses the issues of reliability and scalability and that provides location transparency and sequential consistency with respect to data access. MojaveFS replicates the data for each file among multiple physical servers to increase reliability. Our filesystem achieves performance and scalability by using a novel hashing scheme over virtual servers that provides fast lookup and load balancing. Finally, we accomplish location transparency through a global namespace that maps files and directories to globally unique identifiers. The innovative model we present for this filesystem is attractive both to existing production systems and new emerging computation environments like mobile systems.

The next section of this paper describes the design of our filesystem. Section 3 discusses the architecture of the system and some of the implementation-specific details. Section 4 continues with a comparison our work with related projects. Finally, the paper concludes by proposing future directions of research.

2 Design overview

We assume that our distributed system contains at least the following two types of nodes. First, there are *data clients*; these are the consumers (and possibly producers) of data. Second, there are *data servers*; these nodes store and maintain the data and communicate it to the clients. Data servers are organized into *virtual server groups*¹. When clients try to access files, we employ a hashing mechanism to map filenames to virtual server groups. The high-level architecture is shown in Figure 1.

¹We use the terms virtual server group, virtual server and virtual group interchangeably throughout the text. They all refer to the same concept.

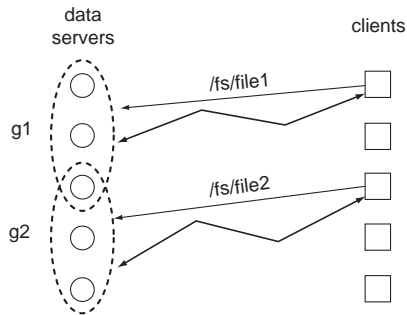


Figure 1. Each filename maps to a virtual data server group.

A virtual server group may contain several physical servers, and any physical server may belong to more than one virtual server (physical servers can be added to or removed from these virtual servers dynamically). Since the physical servers that belong to a virtual group may, in general, overlap with the servers of another group, the number of virtual servers can increase dramatically. This can significantly decrease the effect of hash collisions used by our hashing mechanism.

Also, the ability to dynamically configure virtual server groups is particularly attractive from a system administrator’s perspective. An individual machine can be hardened against failures by using redundant network cards, power supplies, RAID arrays, and so on, but if an operating system update needs to be applied then that machine will still have to be taken offline. With the ability to add or remove machines at will, however, taking a system offline to perform maintenance is not an issue. In addition, the need to harden individual machines against failures is also reduced, enabling the use of inexpensive off-the-shelf components.

All members of a virtual server group replicate the entire set of data that is mapped by our hashing function to that virtual server. Once one decides to distribute redundant copies of data objects among multiple servers, load balancing among those servers becomes a possibility. A simple way of distributing the load in our system is to program clients such that, when a client resolves a data object to a virtual server group, a client chooses a random server from that group to interact with. This simple approach helps to exploit the redundancy inherent in the system. However, more can be done.

If a given data object becomes a “hot spot”, the virtual server that serves it can dynamically recruit new machines to help carry the load. Once the load decreases, servers can resign from the group to reduce the synchronization overhead. Moreover, when a particular usage pattern results in one data group being used more than others, it can be split into several smaller groups. In some applications it may be

beneficial to take the network structure into account, migrating data objects to locations that are closer to the clients that access them most frequently; in some extreme cases it may even be beneficial to allow some of the clients to become data servers *themselves* in order to reduce latency. Note that the distinction we make between data servers and clients is merely conceptual; in practice, there is nothing in our approach that precludes a node from being both a client and a server simultaneously.

Given that our design makes use of replication in order to provide increased reliability and performance, it becomes necessary to consider a consistency model in order to ensure that the data remains uniform among the replicas in the system. However, consistency, replication, and performance are mutual antagonists. The issue is that, even though the data is replicated and multiple servers may be able to respond to a request, the consistency of a data item must be ensured before the request can be processed. In the worst case, this requires that a server contact the replica holders to obtain an updated data item before responding to a request. Furthermore, while failures may be infrequent, servers must be prepared to respond and recover from faults at any time.

To address this we use in our approach an efficient group communication protocol to enforce sequential consistency of replicated data items inside each virtual server. It is widely accepted that the simplest programming model for data consistency is sequential consistency [11], which preserves the order of accesses specified by each of the programs that concurrently access the shared data. In practice, the simple sequential consistency model has been reluctantly adopted due to concerns about its performance. However, recent work in compilers for parallel languages [10] has shown that sequential consistency is a feasible alternative to relaxed models.

3 Implementation overview

MojaveFS has two components. The first one lives in kernel space and acts as a layer that intercepts and interprets filesystem calls before passing them to the second component, a user-level daemon. The user-level daemon uses the existing underlying filesystem to store the data. Due to its open source nature, we chose Linux as the implementation platform for our filesystem (thus providing us with easy access to the internals of the operating system).

The internals of MojaveFS implement an effective mechanism that allows files to be replicated and distributed in an efficient manner. Normally, only part of a file will be used by any single process at a given time. Other parts of the file may be used by other processes in other locations. MojaveFS uses a novel data storage strategy where files are split into smaller *objects*. These objects are the indivisible data unit of our system and they are mapped to virtual data

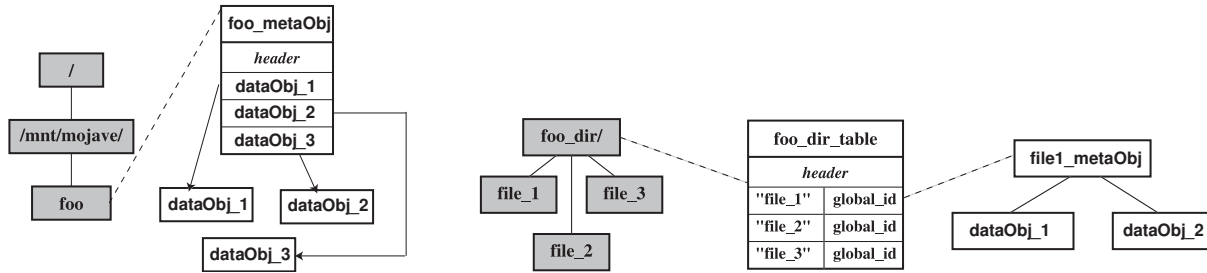


Figure 2. Representation of a file and of a directory in MojaveFS. Only the shaded areas are visible to the user and they represent the user’s perspective.

server groups through a hashing function. This scheme increases the availability of the data, improves file utilization, and decreases the access time.

Each user-visible file corresponds to a set of objects that collectively contain all of the data associated with the file. The metadata for each file is also contained in a separate object, and is comprised of a header describing properties of the entire file and a list of the objects of which the file is composed.

In Figure 2, we present the internal representation of a file in MojaveFS, named *foo*. The metadata object for *foo*, named *foo_metaObj*, contains the identifiers of the objects composing file *foo*: *dataObj_1*, *dataObj_2*, and *dataObj_3*.

A directory in MojaveFS is represented as a metadata object containing entries for each file that resides inside the directory. To access a file inside a given directory, the actual file can be resolved by looking up its identifier inside of the metadata object of the directory. This is illustrated on the right hand side of Figure 2. Here we have one directory, named *foo_dir*, containing the three files *file_1*, *file_2*, and *file_3*.

We have opted for a modular design for the implementation of MojaveFS, consisting of the following layers: Cap, Indirect I/O, Direct I/O, and Group Communication. The model we chose has an internal asynchronous behavior. Our design uses function calls to propagate requests and information from the top to the bottom layers, and events/callback functions to propagate information up the stack as it becomes available. Figure 3 shows the API exported by each layer on the left hand side of the layer block and the events/callback functions on the right hand side. The functionality of each of the layers is described in more detail below.

The Cap layer is a wrapper that provides applications with the standard filesystem interface. The exported API of the Cap layer, shown on the left side in Figure 3 includes calls for creating a new file (*create*), and for accessing data from a file (*read*, *write*). While the internal communication between the various layers of MojaveFS is asynchronous,

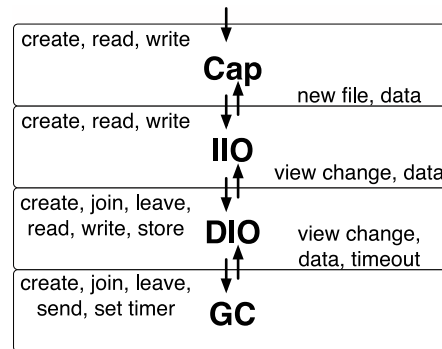


Figure 3. The layered architecture of MojaveFS.

the interface that the Cap layer exports to the application layer uses the traditional synchronous model.

The Indirect I/O Layer (IIO) handles data replication, and naming and localization of objects. It provides a one-to-one mapping to the API exported by the Cap layer, but its functionality is asynchronous. It is only when data is available that it uses the callback function provided by the Cap layer to send the *new file*, and *data* events.

The Direct I/O (DIO) layer handles data consistency across replicas, saves information to stable storage, and deals with certain fault-tolerance issues. Its API is closer to that of the bottom layer, the Group Communication (GC) layer. The events that the DIO layer propagates to the IIO layer are *view change* (when the membership of the data servers changes), *data* (when data becomes available for a previous request by IIO), and *timeout* (when an IIO expected event fails to occur in the specified time window).

Finally, the purpose of the Group Communication layer is to provide access to a network communication mechanism that enforces the sequential consistency guarantees that the upper layers rely upon.

3.1 The Indirect I/O layer

The I/O layer, which is the most complex layer in our stack, provides the following services: replication, naming and localization.

3.1.1 Reliability and Replication

In any filesystem it is important to keep the data sound. We propose to achieve data reliability through replication. We implement a simple k-copy replication mechanism in our prototype. As explained below, data replication helps improving the performance of the system by increasing its throughput.

One of the traditional approaches to distributed data allocation is to use a hash function to map some property of a file (such as its file name or full path) to the server which provides the data associated with the file [6]. We call the data associated with the same server after applying the hashing function a *data group*.

In a traditional hash-based distribution scheme, the hash function for a file's path indicates which server has the file. Adding a new server requires a change to the hash function. Since the hash function determines the physical location of each file, a change to the hash function will generally cause large amounts of metadata to be moved among the servers, which is a very expensive operation. Various approaches exist to try to amortize this cost over longer periods of time, but the expense cannot be avoided entirely.

In our approach, however, the hash function produces a virtual server group identifier. Each client has a lookup table that maps these identifiers to sets of physical servers, which is updated during the normal course of client-server interaction. This table acts as an extra layer of indirection between the hash value of a path and its physical location on the network, so adding a machine to a group does not require any change to the hash function. From the client's perspective, adding a new server to a group is transparent. Removing a server from a group can, in the worst case, cause a client to encounter a network timeout; the client will then proceed to the next server in the group.

A virtual data server acts as the replication group for its corresponding data group. Every node in a virtual data server possesses an up-to-date copy of the data corresponding to the data group that it serves. Note that we do not limit the number of different data groups that a data server can serve. Furthermore, each physical server can belong to an arbitrary set of virtual data servers without restriction.

Unlike traditional replication mechanisms where complete replicas are made of the entire information store from one physical node, our replication mechanism is more nimble. Specifically, we can exploit the relatively small cost of allocating additional virtual data servers in order to tune

the breadth of the responsibilities of each of the virtual data servers in our system.

3.1.2 Naming and localization

The naming service provides the following functionality. It generates system-wide unique object identifiers and it provides a global look-up service. The service distinguishes between two types of objects: data objects and metadata objects. The former type stores data contained in user files, while the latter type contains data pertaining to directories or file properties, as well as pointers to the data objects composing the files.

In our current design we use a hybrid hash-based and tree-based lookup mechanism for objects. This, combined with an efficient caching scheme for access privileges, provides fast access to files in the system.

The system uses a function that maps each potential file in the system to a unique virtual data server. The lookup mechanism for a file works as follows. First, the system identifies, using the mapping function, the virtual data server that is responsible for the metadata object associated with the file that is being looked up. Next, it contacts the virtual data server and retrieves the metadata object that contains information about the identifiers of the data objects composing the file.

In order to conserve resources it may be desirable for the system to consolidate the responsibilities of the virtual data servers corresponding to all of the files in some subtree of the filesystem directory structure. In this case, the lookup mechanism would perform a binary search in order to determine the most specific virtual data server that is responsible for the metadata of the file to look up. For example, suppose the directory "a/b/c" is responsible for all of the files in the subtree rooted at that directory and that a client is trying to locate the file "a/b/c/d/e.txt". In this case, the client will initially attempt to contact the virtual server corresponding to the file "a/b/c/d/e.txt". However, given that this virtual server is nonexistent, the lookup mechanism will then look up directory "a/b/c". Since this virtual server does exist, the client will then try to look up directory "a/b/c/d" in order to find a more specific virtual data server responsible for the file. However, since "a/b/c/d" is also nonexistent, this would conclude the search and the request would be directed towards the virtual data server serving "a/b/c". The advantage of this mechanism is that we don't need to create all possible virtual servers up front and, if we couple this with a caching mechanism, the scheme can perform similarly to the non-consolidated version.

While the mapping scheme is fast for file lookup, it is still convenient to preserve the hierarchical structure of the filesystem in order to determine access permissions. For example, given this structure, it is possible to traverse the

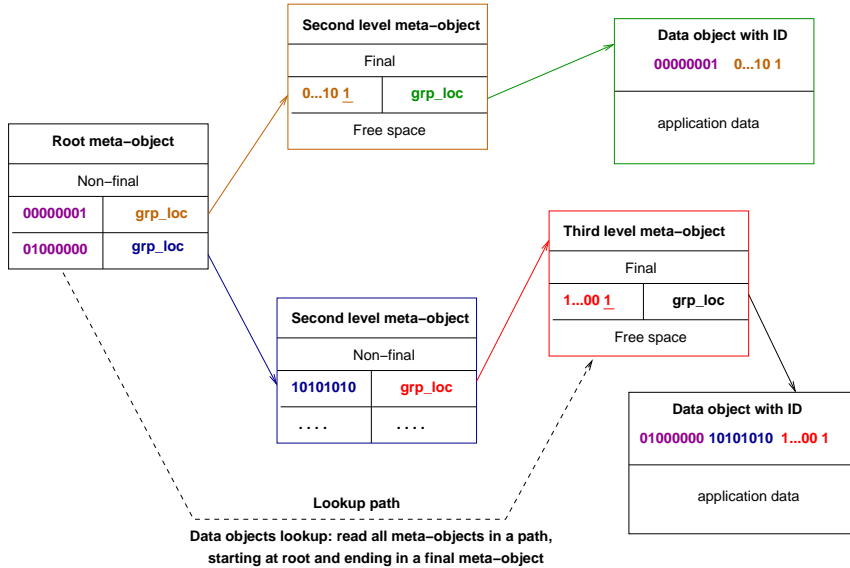


Figure 4. Object look-up mechanism.

resulting hierarchy, starting from the root, to resolve the access permissions for a particular file or directory. Unfortunately, these tree traversals can be costly. Thus, we implement a need-based caching mechanism to allow the lookup mechanism to avoid performing these tree traversals unless the cache of the object has been invalidated.

Figure 4 shows how the metadata objects preserve the hierarchical directory structure utilized by the tree traversal mechanism for determining file access permissions.

3.2 The Direct I/O layer

The primary function of the Direct I/O (DIO) layer is to handle the propagation of data and metadata to stable storage. When a node receives new data to be written to an object that it has stored locally, or if the IIO layer requires the creation of a new object, the DIO layer saves the new data and metadata associated with the object to stable storage by interacting directly with the underlying local filesystem.

However, in order to maintain consistency of this data in the presence of failures, it is necessary for the DIO layer to perform some additional bookkeeping. For example, due to network or node failures there could be a split of a virtual data server such that only some subset of the original members of the virtual server group are still able to communicate with each other. In this case, it is the responsibility of the DIO layer to decide which subset of the members of the original virtual server can continue to make progress and which nodes are prohibited from making progress (blocked)

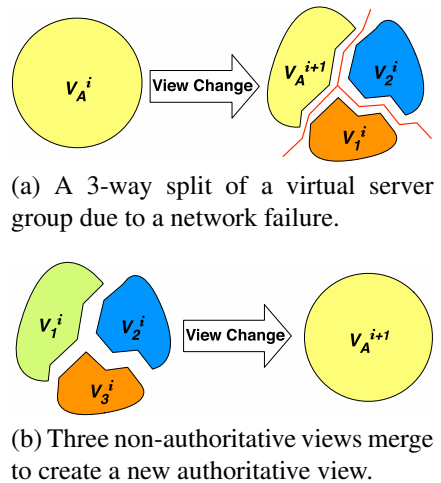


Figure 5. Views split and merge, changing authoritative status.

until the failure is resolved.

To illustrate the need for this bookkeeping, consider the scenario highlighted in Figure 5(a). In this case, a network failure causes a 3-way partition of the nodes that comprise one of the virtual servers in the filesystem. Each of the nodes in the three fragments of the original group are capable of communicating with other nodes within its respective partition, but are unable to communicate with any of the nodes in the other partitions. If the data were not mutable, this might not be a problem. However, it is clear that allowing each of the three fragments to operate normally (i.e. handle both reads and writes) could lead to data inconsistency among the replicas.

In order to handle such scenarios as might lead to data inconsistency, the DIO layer introduces the notion of an *authoritative* view of a virtual data server. Intuitively, a view, \mathcal{V} , is authoritative when \mathcal{V} corresponds to the initial membership of a newly created virtual server or when \mathcal{V} contains a majority of the membership of the previous authoritative view. To keep track of the order of authoritative views in the system, the DIO layer associates an *epoch* with every view in the system; this value is monotonically increasing and is incremented every time a view experiences a change in membership that results in the view attaining (or maintaining) authoritative status. We denote the value of the epoch, e , of a view, \mathcal{V} , using the superscript notation \mathcal{V}^e (we use the convention that 0 is the epoch of a view corresponding to a newly created virtual server). More formally, the following two conditions govern the authoritative status of a

view.

1. The initial view, \mathcal{V}^0 , associated with a newly created virtual server is authoritative.
2. The view, \mathcal{V}^i , is authoritative if there existed an authoritative view \mathcal{V}^{i-1} such that $\mathcal{V}^i \cap \mathcal{V}^{i-1} \subseteq_m \mathcal{V}^{i-1}$, where $A \subseteq_m B$ denotes that A is a majority subset of B .

Thus, the condition for a view to attain authoritative status ensures that at most one authoritative view of a virtual data server exists in the entire system at any given time. Given this property of authoritative status, it is safe for the DIO layer to allow only authoritative views of virtual data servers to make progress.

Note, however, that the condition for authoritative status does permit scenarios where all the views of a virtual data server become non-authoritative. For example, consider Figure 5(b) and suppose that none of the three fragments contains a majority of the nodes of the initial virtual server group. In such a case, no view of the virtual data server can make progress. This necessitates the adoption of some sort of recovery policy that permits one to “rebuild” an authoritative view out of non-authoritative views. The specific recovery policy we have chosen for our implementation is embodied by the second condition presented above. This condition implies that two or more non-authoritative views could merge to form an authoritative view (provided that the membership of the merged view contains a majority of the nodes that were members of the last authoritative view).

Given the need for these types of recovery policies, the DIO layer must keep track of certain information, both to facilitate the reconstruction of authoritative views and to prevent more than one authoritative view of a particular virtual data server from being created. In particular, our simple recovery policy requires that the DIO layer maintain the membership of the last authoritative view of which a node was a member along with the epoch of that authoritative view. Furthermore, on a split, a resulting non-authoritative view must also keep track of all the messages the view has received for which the members cannot confirm have been delivered to the nodes on the other side of the partition. This is needed to ensure that, if two or more non-authoritative views merge to create an authoritative view, a resulting authoritative view has the most recent set of messages (reads and writes) sent to the view before the split.

It is salient to note that different policies governing authoritative status could be implemented. However, it is important that such a policy enforce the invariant that at most one view is permitted to be authoritative at any time. In general, policies that allow more automatic failure recovery tend to require more bookkeeping. For example, one might

envision a policy that would allow non-authoritative views with different epochs to merge, provided that the data has not changed between these epochs. In this case, the DIO layer would have to keep track of extra information, such as the entire history of authoritative view membership for a node (as opposed to the membership of only the previous authoritative view of which the node was a member). Without this extra information it would not be possible for the DIO layer to determine if the membership of a view resulting from a merge of two or more non-authoritative views contains a majority of the possible nodes that could combine to form an authoritative view (thus ensuring that there could be no other merger involving a disjoint set of nodes that would result in a view being considered authoritative). Though, it may be possible to maintain less information if a metric other than the majority were used.

3.3 The Group Communication Layer

Conceptually, the group communication layer consists of two primary components: a high-level protocol for guaranteeing sequential consistency (described in [16]) and a variant of a low-level total-order group communication protocol implementing atomic multicast.

The requirements for an implementation of our high-level protocol are summarized by the following lemma [16].

Lemma 3.1. An implementation of the protocol provides sequential consistency if the following conditions are met:

- The group communication protocol orders messages sent by the same process to the same group consistently with the order in which they were sent.
- When a process attempts to send a message to a different group than it sent its previous message to, the send operation is *blocked* and is *not performed* until all the messages already sent by this process get assigned to the group total order by the low-level group communication protocol.

By “*assigned to the group total order*” we mean that the group communication protocol commits to putting the message right after some specific message (which is already assigned to the group total order) in the group total order. This is illustrated in Figure 6. Here the node that handled the `touch(Y)` request has to wait for this request to be assigned a sequence number in group Y before it can forward the `touch(X)` request to its peers in group X .

The presence of the explicit read messages, the group total order, and the constraining condition (above) on the way nodes issue read and write messages to different groups guarantees global sequential consistency across the entire filesystem service. We have proved the correctness of our

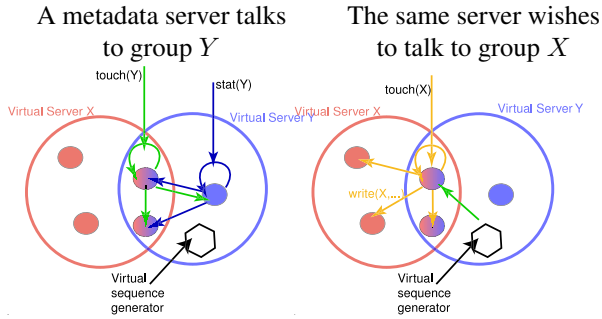


Figure 6. Imposing an order on messages sent to different metadata groups.

protocol using a mathematical model and have also verified safety and liveness properties of our system using several model checkers.

Note that although the system as a whole may consist of hundreds or thousands of data servers, in our implementation we assume that only a fraction of the total number of data servers will maintain membership in any single virtual server group, so it is feasible to use a total-order group communication protocol in each individual group.

3.4 Optimization

One important issue in designing a distributed filesystem is to understand the difference between the data and the metadata. Metadata differs from file data in several ways. On the positive side, metadata is usually smaller than file data. Loss or corruption of metadata can be catastrophic, leading to a disproportionately large loss of data. The frequency of metadata operations often exceeds the frequency of operations on file data. In many applications, 50-80% of all filesystem accesses are to metadata [5]. Metadata services are prone to hotspots where, as a computation shifts to a new dataset, a burst of operations are focused on a single file or on the files in a single directory or subdirectory. In general, metadata operations must be serializable—that is, they must be sequentially consistent and atomic.

We propose a separation of metadata servers from data servers, where the former would implement a strict consistency model, while the latter will allow various relaxed consistency models, as specified by users. Figure 7 illustrates this optimization. We define metadata broadly to include any data in which sequential consistency and/or reliability are principal concerns. This includes, for example, filesystem metadata such as directory and security information where loss or corruption of the metadata can be severe; as well as synchronization operations like locking, where sequential consistency is critical.

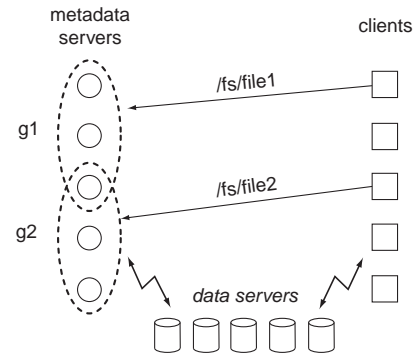


Figure 7. Each filename maps to a virtual metadata server group. While not required, the metadata and data servers that are shown here are in separate server farms.

4 Related work

One of the first distributed file systems used on a large scale was the Andrew File System (AFS) [9]. While AFS provides scalability and security, it offers a relaxed consistency model, rather than sequential consistency (as we provide in MojaveFS), for concurrent accesses to shared data. This makes real-time concurrent file sharing very difficult. Another drawback of AFS is data availability. By requiring entire files to be transferred to the initiator of an open call AFS limits the concurrent accesses to shared data. MojaveFS is more selective in sending data, providing higher data availability and lower overhead on calls to “open”.

The CODA [2] filesystem was designed for mobility and does not enforce strict data consistency. Thus, it requires user input to solve conflicts that occur due to unguarded concurrent accesses to data. To provide remote access to data CODA uses a “hoarding” mechanism to copy files on the local machine to make them available while the machine is off-line. While MojaveFS does not address the issue of data availability for disconnected users, it is better fit for distributed environments that have a more stable behavior and, due to its file splitting mechanism, it does accommodate low-end machines with scarce resources.

The latest file system in the series initiated by AFS is InterMezzo [4]. Similarly to MojaveFS, InterMezzo acts like a filter between the VFS and the local file systems. However, InterMezzo has a client-server architecture and it follows the direction taken by CODA in providing access to data while the computer is off-line. MojaveFS is more decentralized and it does provide better access to data.

SPRITE [12], the filesystem component of a distributed operating system with the same name, was designed to transparently provide a distributed file system and to sup-

port process migration [7]. Unlike MojaveFS, SPRITE relies on a clear distinction between clients and servers and it is less adaptable to dynamic changes on the server side.

Lately, there have been other attempts at developing distributed filesystems. For example, there is the zFS filesystem [14] (the successor to the DSF project [8]) that makes use of transactions and a lease-based scheme to provide consistency in the presence of concurrent file access. The primary difference between zFS and our system is that our approach uses a group communication protocol to guarantee consistency. As a result, MojaveFS is also more decentralized as it does not rely on a transaction server (such as the one that zFS uses).

Also, following a recent trend towards the development of specialized metadata management services, several projects, like LH [5], NFS [13], Lustre [3], Vesta [6], InterMezzo [1], and Coda [15] have incorporated specialized metadata handlers into their filesystems. Still, most of these existing projects have taken a relatively static approach to the dissemination of their respective metadata services. All of the approaches listed above make use of a fixed set of dedicated metadata servers with little or no replication of the metadata. One of the deficiencies of these current mechanisms is that they do not have the flexibility to dynamically shift resources around in an efficient manner to match the specific access patterns of their clients.

Kerrighed [17] is a complete single-system image solution that addresses issues beyond the scope of the filesystem, extending to processes and other shared resources. It requires a system-wide deployment of a specific operating system kernel. MojaveFS is a lightweight solution tailored exclusively to address the issues arising in the filesystem domain, both in a desktop and mobile environment.

5 Conclusion and Future Work

We presented the design and implementation overview of a scalable distributed filesystem with support for sequential consistency. The main contributions of this work are the following: coupling existing hashing schemes with replication mechanisms and a sequential consistency protocol to provide reliability of data and scalability; using a file chunking mechanism to enhance the availability of data; and using a modular design that allows for easy replacement of various services provided by the system.

Future directions of research include optimizations, like separating the data services from the metadata services, automatic load balancing based on access patterns to files, using speculative execution to improve performance of internal protocols and supporting speculative execution from the user's perspective.

References

- [1] P. Braam, M. Callahan, and P. Schwan. The InterMezzo filesystem, 1999.
- [2] P. J. Braam. The coda distributed file system. *Linux Journal*, June 1998.
- [3] P. J. Braam. Lustre: A scalable, high-performance file system, 2002.
- [4] P. J. Braam, M. Callahan, and P. Schwan. The intermezzo filesystem, 1999.
- [5] S. Brandt, L. Xue, E. Miller, and D. Long. Efficient metadata management in large distributed file systems, 2003.
- [6] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. In H. Jin, T. Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 285–308. IEEE Computer Society Press and Wiley, New York, NY, 2001.
- [7] F. Douglass and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software-Practice and Experience*, 21(8), August 1991.
- [8] Z. Dubitzky, I. Gold, E. Henis, J. Satran, and D. Scheinwald. Dsf - data sharing facility. technical report. Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, <http://www.haifa.il.ibm.com/projects/systems/dsf.html>, 2000.
- [9] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [10] A. A. Kamil, J. Z. Su, and K. A. Yelick. Making sequential consistency practical in titanium. In *The International Conference for High Performance Computing and Communications, SuperComputing 2005*, 2005.
- [11] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. Comput. C*, 28(9):690–691, 1979.
- [12] J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson, and B. Welch. The sprite network operating system. *IEEE Computer*, 21(2):23–26, February 1988.
- [13] B. Pawłowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.
- [14] O. Rodeh and A. Teperman. zfs - a scalable distributed file system using object disks. In *IEEE Symposium on Mass Storage Systems*, pages 207–218, Asilomar Conference Center, Pacific Grove, U.S., 2003. ACM Press.
- [15] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.
- [16] C. Țăpuș, A. Nogin, J. Hickey, and J. White. A simple serializability mechanism for a distributed objects system. In D. A. Bader and A. A. Khokhar, editors, *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004)*. International Society for Computers and Their Applications (ISCA), 2004.
- [17] G. Valle, R. Lottiaux, L. Rilling, J.-Y. Berthou, I. Dutka-Malhen, and C. Morin. A case for single system image cluster operating systems: Kerrighed approach. 2003.