Building a Reliable Mutable File System on Peer-to-peer Storage

C. A. Stein, Michael J. Tucker, and Margo I. Seltzer
Harvard University
{stein,mtucker,margo}@eecs.harvard.edu

Abstract

This paper sketches the design of the Eliot File System (Eliot), a mutable filesystem that maintains the pure immutability of its peer-to-peer (P2P) substrate by isolating mutation in an auxiliary metadata service.

The immutability of address-to-content bindings has several advantages in P2P systems. However, mutable filesystems are desirable because they allow clients to update existing files; a necessary property for many applications.

In order to facilitate modifications, the filesystem must provide some atom of mutability. Since this atom of mutability is a fundamental characteristic of the filesystem and not the underlying storage substrate, it is a mistake to violate the integrity of the substrate with special cases for mutability. Instead, Eliot employs a separate, generalized metadata service that isolates all mutation and client state in an auxiliary replicated database. Eliot provides fine-granularity file updates with either AFS open–close or NFS-like consistency semantics. Eliot builds a mutable filesystem on a global resource bed of purely immutable P2P block storage.

1. Introduction

There has been a flurry of recent research into the design of peer-to-peer (P2P) lookup services [14] [10] [9] [15]. These systems aim to provide a stable global addressing structure on top of a dynamic network of constantly failing and arriving nodes. Once addressing is in place, services and distributed applications can be built. Several distributed applications have been designed to use these P2P substrates, including some read-only distributed filesystems [3] [11] and a read-write or mutable filesystem [6].

It turns out that building a mutable filesystem is difficult due to the manner in which the lookup services address data. They derive addresses from the content of the given resource. If the content changes, so will the address. Collision-resistant, one-way hash functions such as SHA-1 or MD5 are popular choices for addressing data and nodes.

There are several reasons for using hashes for addresses. First, by mapping both data and nodes into the same hash space in a pseudo-random fashion, data is likely to be evenly distributed across nodes. Nodes that are proximate in the hash space are unlikely to be proximate in physical space. So a failure due to local factors will not affect segments of the hash space in a systematic way. Second, the collision-resistant property of the hash function guarantees that it is highly unlikely that two different resources will map to the same address. Third, hashes can be used for verification. If the data contents hash to the object's address, then the data was not modified. This is important because it protects against undetected data tampering or communication errors. Fourth, hash-based addressing facilitates sharing of resources. If two blocks written by two different clients have the same contents, they will hash to the same address and only one block will be stored. Fifth, a malicious client cannot arbitrarily construct a data to address binding for a given address. While hashes do not protect the system against denial of service attacks, their collision-resistance property will prevent a malicious user from overwriting existing data. Finally, since hash-based addressing eliminates mutation of address to data bindings, there are no issues of cache consistency. If the bindings were mutable, blocks would have different versions creating problems of coherency and consistency. Being in a WAN environment, P2P networks depend on aggressive caching for performance.

Given these attractive properties, we chose to maintain hash-based addressing. However, since this implies an immutable substrate, it presents us with a challenge for the design of a mutable filesystem.

For the purposes of this discussion, we assume a conventional hierarchical filesystem arrangement with UNIX-style directories and metadata storage (i.e., inode and indirect blocks). If a traditional filesystem is built on an immutable P2P substrate, every update to a block will force updates to every block on the path from the original block to the root of the filesystem. This causes a domino effect from the updated datablocks (i.e. the leaves of the filesystem) to the root. For example, a datablock modification will result in the block being moved to a new address. The inode containing this block's pointer must be modified to include

the new datablock address. This changes the inode's hash and forces the inode to be copied to a new address. Then the directory must be updated with the new inode address. The directory contents change, changing the address of the block containing the directory. This continues all the way up to the root. This problem is similar to the tree update problem in functional language implementations [7].

Eliot implements a mutable filesystem on an immutable P2P substrate. Some atom of mutability is required to support sharing across filesystem clients. How and where this mutability is implemented has a significant impact on the fault-tolerance and sharing semantics of the filesystem. One of the design principles of Eliot is to maintain the immutability of the P2P substrate.

Suppose there were special cases built into the P2P substrate to accommodate mutable blocks. Mutable blocks could be stored at a fixed address by taking the hash of something invariant such as the public key of the filesystem. Now any data can be bound to the address to which the public key maps. How would the system verify that a writer has the right to overwrite an address with a new value? Signatures could be used for verification of client writes, authenticating and placing trust in the client. However, to allow for concurrent updates, the private key would have to be shared across all clients; presenting a key distribution and revocation problem. If different keys were used per-client then every P2P node storing a root would have to maintain public-key state for each of the clients authorized to write to the filesystem. We believe such an approach is contradictory to the scalability of P2P networks. P2P network nodes should not maintain per-client state. In addition, this approach has the shortcoming that it provides us with mutability of only the root directory, making fine-grained concurrency impossible.

A filesystem with file-level concurrency allows for two different files to be updated concurrently without their updates interfering. Likewise, a filesystem with block-level concurrency allows for two different blocks within a file to be updated concurrently without interference. Eliot supports file-level concurrency and, depending on the architecture of the clients, can support block-level concurrency. This is discussed further in section 3.4.

Eliot uses a metadata service (MS) external to the P2P substrate to store mutable filesystem metadata. All mutability is isolated to this service, preserving the immutability of the substrate and allowing for concurrency within the filesystem. The P2P substrate is not necessarily under a single administrative control. However, the MS is. The MS is within the physical control of trusted administrators and is engineered for fault-tolerance. For example, it can be implemented on a replicated database in a controlled machine room. The physical proximity of the metadata service to Eliot clients decreases the latency of open and close file operations, as well as other metadata operations. The P2P nodes on the other hand, are contributed by unknown entities. The nodes themselves are much more prone to Byzantine and failstop failure than the locally controlled and administered MS nodes. Failure of P2P nodes is contained by the Charles so that Eliot views a reliable substrate. The merits of the P2P substrate relative to the MS nodes include its accessibility and pervasiveness. The Charles is globally available as a data resource. Multiple metadata servers can share the same Charles substrate.

2. Related Work

The field of distributed filesystems has a lengthy and rich heritage. Although much newer, the field of peer-to-peer systems is beginning to develop its own history. The work we describe here is an integration of the two fields in that we are building a distributed filesystem with a traditional UNIX interface on top of a somewhat conventional peer-to-peer substrate. As a result, in this section, we'll discuss the most relevant systems in both the distributed filesystem area as well as the peer-to-peer area.

Peer-to-peer (P2P) research addresses the design of systems in untrusted, highly dynamic environments. In general, these P2P systems are composed of a low-level lookup service and a higher-level filesystem implementation built on that lookup service.

CFS [3] stores both metadata and data in servers located using the Chord [14] lookup service. Both Eliot and CFS implement a hierarchical filesystem; they are both implemented on potentially untrusted P2P nodes, and both depend on lower layers of software for data persistence and availability. CFS uses DHash, a layer performing replication across Chord, for persistence and availability. Eliot uses the Charles [13]. In CFS, the inodes and datablocks are addressed by their content-hash. Eliot uses content-hashes to address datablocks, but inodes, directories, and other mutable metadata stored in the metadata server, are named by randomly generated unique identifiers.

The biggest difference between CFS and Eliot is in how the two systems support filesystem modification. CFS has a signed, mutable root to allow for mutation. This root cannot be addressed by content hash because that address would change as the contents of the root directory changed. CFS uses the hash of a filesystem's invariant public key to address roots in the underlying Chord substrate. As discussed earlier, once the address is no longer a one-way, collision-resistant function of the contents, illegitimate overwriting of datablocks becomes an issue. CFS solves this with signatures, requiring external coordination for key distribution between clients, In contrast, Eliot avoids this problem by placing mutability outside of the P2P substrate. This allows Eliot to implement filesystem modification in a sim-

pler fashion with finer granularity of updates.

PAST [11] is an immutable file store built on the Pastry [10] P2P substrate, a Plaxton-based [8] routing protocol. While CFS and Eliot break files into blocks spread across multiple nodes, PAST stores full files on individual nodes and replicates across nodes for persistence. PAST is intended to provide archival storage and content distribution, not a general-purpose filesystem. As a result, its files are immutable and this does not pose a problem as it does for Eliot whose goal is to be a general-purpose filesystem.

Ivy [6] is a mutable filesystem built on DHash. Ivy places mutability in a per-client log-head. Every client writes its updates, both data and metadata, to its log. To satisfy read operations, a client accesses its own as well as other clients' logs to construct the current state of the filesystem.

The distributed filesystem literature introduced the notion of peer-to-peer systems, calling them serverless distributed file systems [1].

xFS uses a distributed data structure known as the manager map to locate file blocks. Designed for a secure, low-latency LAN environment, under a single administrative control, xFS tolerates failstop, but not malicious or Byzantine nodes. Eliot, on the other hand, is built to take advantage of two different environments. The metadata reside in a local, reliable, and low-latency environment. The datablocks reside in a variable latency and dynamic P2P network. For the P2P network, Eliot uses the Charles which assumes an untrusted WAN environment without a central administrator.

Eliot installations have a fair amount of flexibility in the consistency semantics they provide. The semantics are controlled by the clients. This is discussed further in section 3.4.

Eliot borrows one set of filesystem semantics from the Andrew File System (AFS) [4]. AFS provides open-close semantics. An operational definition is as follows; clients fetch a full file on open and cache it on their local disks. They modify the file locally. On close, they write back the full file, committing the modifications to the server. All updates committed by others during the period between the open and close will be lost.

Alternatively, Eliot clients can provide NFS semantics [2].

The histories of distributed systems and name services are closely tied in that multiple nodes can only interact in a coordinated fashion if they can locate one another and discover what services each provides. The metadata service component of Eliot shares some similarities with systems like Grapevine [12] and DNS that provide replicated, fault-tolerant lookup services. Grapevine provided a distributed registry for locating resources and authenticating users. It was built for a messaging system and used the underlying messaging system to maintain state and pass around replication updates. In Grapevine, the propagation of updates to replicas tended to be slow and users often encountered inconsistencies. Eliot's metadata service, on the other hand, must support strong consistency and a high rate of updates, as clients open and close files, and mutate metadata with high frequency under varied filesystem workloads.

3. The Eliot File System

The Eliot filesystem consists of several components. These are:

i An untrusted, immutable, reliable P2P block storage substrate known as the Charles [13].

ii A trusted, replicated database, known as the metadata service (MS), storing mutable inodes, directories, symlinks, and superblocks.

iii A set of filesystem clients.

iv Zero, one, or more cache servers. These are intended to improve performance but are not necessary for correctness.

3.1 The Charles Block Service

The Charles [13] provides a reliable, fault-tolerant, immutable P2P substrate for Eliot. The philosophy behind the design of our substrate is to split a distributed system into a top scalable layer providing global routing correctness and a bottom layer of protocols that provide the abstraction of fault-tolerance to the top layer using less scalable protocols. This philosophy addresses the commonly observed trade-off between the global-knowledge of group membership protocols and performance.

The Charles makes strong safety and liveness guarantees with regard to the blocks that it stores; namely, if a block is written to an address, then a client will be able to request that block, and subsequently receive the block, unmodified, within a reasonable period of time. To maintain these invariants, the Charles must tolerate failstop and Byzantine node failures. Surviving these failures is crucial to making guarantees about both the availability and the correctness of the stored data. In designing a P2P storage substrate, we must assume that some nodes will fail, some may become unreachable, other will send out spurious or incorrect messages. Though the hash values as addresses protects clients from accepting bad data blocks, bad nodes can disrupt the structure of the substrate by providing false routing information to neighbors and new nodes. Therefore, the routing protocols of the Charles Block Service are designed to be fault tolerant for failstop and Byzantine failures.

3.2 The Metadata Service

The metadata service (MS) stores Eliot's metadata. This includes inodes, symlinks, directories, and the filesystem superblock. Indirect blocks and datablocks are stored in the Charles. The guiding principle behind the placement of data objects is that those objects that are mutated by operations other than a file write are stored in the MS, while objects that are only mutated during the course of a write are stored in the Charles.

The MS is implemented as a replicated database for high-performance and fault-tolerance.

The MS and Eliot clients authenticate one another. Every Eliot client has a public key registered with MS. Every RPC from an Eliot client to the MS is authenticated by verifying the signature. Alongside the client's public key, MS stores a sequence number. The RPC is tagged with a sequence number. To avoid replay attacks, the RPC is discarded if the sequence number is not strictly increasing.

Each Eliot filesystem installation has one MS. The MS has a public key well-known to its Eliot clients. The MS signs all messages sent to clients. The clients attempt to verify messages and discard those they cannot verify.

3.3 The Cache Service

The latency of Charles' datablock accesses is alleviated by cooperative caching [1]. A cache server stores soft state mapping Charles addresses to clients that hold the corresponding blocks in their cache. Clients update the cache server periodically with changes in their cache state. Multiple cache servers can operate concurrently within a single Eliot installation, should one become a bottleneck. The cache server is not required for correctness. It exists solely to improve performance.

3.4 Implementation and Consistency

Eliot presents a traditional POSIX filesystem interface to applications. Clients are implemented either as NFS loopback servers, user-level filesystem libraries, or kernel filesystems beneath the VFS/Vnode layer. Eliot filesystems are mounted in the client's filesystem namespace and present a hierarchical directory structure. Unlike most filesystems, Eliot clients access the contents of inodes directly, reading the addresses to request datablocks directly from the Charles. The MS does not lie on the datapath.

Depending on the architecture of the client, different file system semantics can be supported. For example, many of the recent file systems built using the SFS toolkit [5] provide NFS semantics, because SFS implements filesystems as NFS loopback servers.

If the Eliot client is structured as an NFS loopback server then Eliot will provide NFS-like semantics. Traffic to and from the MS will be higher, because inodes will not be cacheable at the client. The client will contact the MS for operations that could otherwise have been satisfied locally from a cached inode. An NFS-like Eliot client will support block-level concurrency.

Providing AFS semantics requires a meaningful implementation of file open and close. This is not possible with an NFS loopback client because NFS lacks a file open and close. NFS clients use opaque, server-generated descriptors to specify files within NFS RPCs. These descriptors are obtained with a lookup RPC, which includes a parent directory descriptor and a component name. An open will usually cause a lookup RPC, but the NFS server has no way of differentiating a lookup caused by an open from a lookup caused by another filesystem operation, such as a name component lookup within a directory. NFS servers were designed to be stateless to ease crash recovery. Since the NFS server maintains no client state, a server has nothing to do on a client's file close, and consequently there was no reason for the designers of the NFS protocol to include a close RPC.

Under AFS-like semantics, updates associated with the last close win. Other updates associated with a file close following the winning close's open and preceding the winning close will be lost entirely. AFS provides these semantics by fetching the full file from the fileserver to the local disk cache on open and writing the full file back on close. For large files, open and close will have high latency. Interestingly, the immutability of the datablock substrate allows an AFS-like Eliot client to accomplish open–close semantics without copying any data other than the data and metadata blocks updated. The beauty of open–close semantics is that, unlike NFS, that has probabilistic and undefinable semantics, updates will not be interleaved and partial. AFS semantics, however, prohibit block-level concurrency.

To support AFS semantics within Eliot, the client must be implemented either underneath the VFS/Vnode interface or within a user-level library. In this case, the client can cache inodes locally until file close, avoiding communication with the MS to both improve the latency of filesystem operations and reduce the aggregate load on the MS.

Operating AFS-like and NFS-like clients within the same Eliot filesystem will result in unexpected behavior for both the NFS-like and AFS-like clients. The NFS-like clients' writes will lose persistence when interleaved with the writes of AFS-like clients and the AFS-like clients' reads will see data that are in a transient (but fully acceptable for NFS) state between the NFS client application's open and close. This is no worse than write-sharing with a misbehaving client that writes bad data.

3.5 Updates

To illustrate the operation of Eliot, we will trace through an open-modify-close sequence of the file `/etc/rc.conf`.

1 An Eliot client sends a request to MS for the inode associated with name `rc.conf` within the directory `/etc/`. This will have been preceded by a lookup of `etc` within the root directory. The root directory has a fixed and well-known inode number.

2 The MS authenticates the user, checks access permissions and returns the inode of `rc.conf` if appropriate, otherwise returning an error.

3 The client uses the list of Charles addresses in the inode to translate the block's logical address into its Charles address. This translation is analogous to a local filesystem's translation from logical block number to physical disk address. Indirect blocks may be retrieved from the Charles during address translation.

4 Given the block's Charles address, the Eliot client contacts a cache server to see if another client is caching the block. If one is specified, the client will request the block and attempt to verify it. If the cache server does not specify a client, or if the specified client either does not respond or returns a datablock that fails the content-hash check, then the client will retrieve the datablock from the Charles. If the datablock returned from the Charles fails the content-hash verification, the client will return an error code to the application.

5 Once the target datablock is in the local cache, the client updates it. The datablock now has a new Charles address because the hash of its contents has changed.

6 The client writes the datablock (and possibly modified indirect blocks) to the Charles. The new address is written into the locally cached inode.

7 The client contacts MS to update the inode. After the commit of this update, the old datablock is no longer reachable from the file. If the client provides open–close semantics, then it will hold off on committing the inode to the MS until an application's file close.

3.6 Fault Tolerance

Eliot depends on the Charles for liveness; the Charles will eventually return the correct block. Within the Charles, this is achieved through replication and the assumption that the number of Byzantine or failstop nodes will be no more than a fraction of the nodes. If a Byzantine Charles node tampers with data, Eliot will detect this because the address will not match the hash, and will ask another node for the block.

If the MS becomes unavailable, Eliot clients will be unable to read or write metadata. Currently, we protect against failstop failure of MS through database replication.

4 Current and Future Work

We are currently implementing Eliot. Future work will include investigation into how traditionally difficult, but certainly important, operations such as backup, snapshots, and filesystem replication can be supported by Eliot and the Charles. We expect that the database implementation of the mutable metadata will facilitate consistent snapshots, backups, and rollback. In addition, the global accessibility of the Charles will allow for portable filesystems and fast filesystem copy. Only the metadata need be copied and sent. The new Eliot installation can plug into the same Charles substrate to access the data. The new and old filesystems will operate in coexistence without interference.

5 Conclusion

This paper has sketched the design of Eliot, a mutable filesystem built on an immutable and reliable P2P storage substrate. Eliot uses an auxiliary database for storing mutable metadata. Eliot offers concurrency within the filesystem and either AFS open–close or NFS-like file consistency semantics, while taking advantage of a global P2P network for its data store.

6 Acknowledgments

We thank Christian Lindig, Norman Ramsey, Jim Waldo, and Chris Okasaki for comments and feedback.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] B. Callaghan, B. Pawlowski, and P. Staubach. *NFS version 3 protocol specification. RFC 1813*. Network Working Group, June 1995.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[4] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, volume 6, February 1988.

[5] D. Mazieres. A toolkit for user-level file systems. In *Proceedings of the 2001 Usenix Annual Technical Conference*, June 2001.

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Submitted to OSDI 2002*, May 2002.

[7] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[8] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, Aug. 2001.

[10] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[11] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, pages 188–201, Oct. 2001.

[12] M. D. Schroeder, A. Birrell, and R. M. Needham. Experience with grapevine: The growth of a distributed system. In *ACM Transactions on Computer Systems*, volume 2, pages 3–23, 1984.

[13] C. A. Stein, M. J. Tucker, and M. I. Seltzer. Reliable and fault-tolerant peer-to-peer block storage. In *Harvard CS Technical Report HU-TR-04-02*, May 2002.

[14] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, 2001.

[15] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.