Position: CRDTs for truly concurrent file systems

Anonymous Author(s)

ABSTRACT

This paper addresses the problem of building highly available shared and geo-replicated distributed POSIX file systems (FS). We explore the path of building a distributed file system that is geos-distributed, highly resilient and scalable by using Conflict-free Replicated Data Types (CRDTs) i.e. replicated data structures that dont need any coordination. We build upon these data structures and explore solutions to ensure that the application remains correct and in a final state that keeps the original user intention in case of conflicting operations.

CCS CONCEPTS

• **Computer systems organization** → **Redundancy**; *Client-server architectures*; **Availability**; **Reliability**; • **Software and its engineering** → *Software design engineering*; *Software design engineering*.

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Anonymous Author(s). 2021. Position: CRDTs for truly concurrent file systems. In *Proceedings of ACM International Systems and Storage Conference (SYSTOR'21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

System designers have long sought to design geo-replicated file systems allowing concurrent updates on multiple replicas (active-active replication) that are correct, remain available under partitions, and provide high performances (Bell Labs P9 [5], AFS [2]).

SYSTOR'21, June 14-16, 2021, Haifa, Israel

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

One approach is to rely on strong consistency, ensuring that replicas agree on a single, global order of operations. Ordering operations across replicas, however, requires coordination among geographically distributed nodes, and therefore comes at the cost of performance overheads and reduced availability under network partitions.

An alternative approach is to provide weak consistency guarantees. This approach does not require coordination among replicas, and ensures availability under network partitions.

However, resolving conflicts while ensuring that file system invariants are maintained and user operations not discarded is hard.

Existing systems that accept concurrent updates on multiple replicas without coordination fails both to converge to a predictable state and to keep the user's original intention when the operation was issued [1]. Convergence of those systems can leads to directory being merged, files being lost or have their content mixed in unexpected manners.

It is difficult for a user to build a simple intuition of these behaviors, leading to misconceptions on what their tool offers and how it react in these situations [7]

In this paper, we present ElmerFS, a POSIX-compliant geo-replicated file system, that allow users to resolve conflicts using known FS operations.

ElmerFS does not require coordination between replicas, ensures correctness in the face of concurrent conflicting updates and remains available under network partition and server crashes.

The design of ElmerFS is based on Conflict-free Replicated Data Types (CRDTs), replicated data structures that ensure that modifications performed on different replicas can always be merged into a consistent state.

2 DESIGNING A FILE SYSTEM

2.1 File system invariants

POSIX's file systems organize files and directories in a tree structure. To design our file system we focus on the following tree invariants:

(1) Tree nodes are always reachable from the root directory.
(2) Directories can have at most one parent node.
(3) Cycles are possible only through symbolic links.

We categorize two types of operations. Structural operations, which can be used to manipulate the file system

structure (the tree), and content operations, which can be used to store and retrieve information from files.

File systems are commonly represented internally as a collection of inodes. We don't depart from this representation: An inode represents a file system object (file, directory or symbolic link). Each inode is identified by a unique numeric identifier (an ino). A link is a reference to an inode. A directory entry is a named and visible representation of a link.

2.2 File systems under weak consistency

Preserving file system invariants in a replicated file system that allows updates in multiple replicas with coordination among them presents several design issues:

- **Unique identifiers**: Any operation that creates inodes needs to generate a unique identifier. Without coordination among replicas, generated ids might conflict. In practice, this is addressed using 16 byte ids. However, 8 byte ids are required in POSIX.
- **Named links**: Operations that create or move links (rename, link and creation operations) may result in conflicts in which concurrent operations n different replicas create links with the same name in the same directory. Systems may resolve this type of conflict by automatically renaming those links, or expose the conflict resolution to the user.
- **Preserving the FS tree invariants**: Concurrent rename (move) operations without coordination may result in cycles or unexpected additional links. Merging an operation that moves a directory A into a directory B with a concurrent operation that moves B into A violates the FS invariants as it results in a cycle. Furthermore, merging two concurrent operations that move the same directory to different destinations results in a directory with two parents, which violates the FS invariants.
- **Deletion of inodes**: Concurrent link and unlink operations may result in different behavior depending on the order in which operations are applied on each replica. The FS may be left with a partial inode that is not fully deleted but apparent.
- **Permissions changes**: Updating permission from a replica may not be propagated directly to other replicas. Ensuring Operations that have no longer sufficient privileges should be discarded.

2.3 What are CRDTs and a brief overview of AntidoteDB

Ensuring that all replicas converge to the same state without coordination is not trivial.

Conflict-Free Replicated Data types are data structures that can be replicated across multiple replicas, and these replicas can be updated independently and concurrently without coordination.

By construction, CRDTs guarantee that modifications on different replicas can always be merged into a consistent state without requiring any special conflict resolution code or user intervention.

There are two approaches to CRDTs: Operation-based CRDTs replicate their state by transmitting only the update operation that are commutative. State-based CRDTs send their full local state to other replicas and merge incoming states with a commutative, associative and idempotent function [6].

For example, one of the simplest operation based CRDT is the counter. An unbounded counter is a simple local integer whose operations are increment and decrement. Addition and subtraction commutes, no matter in which order the operations arrived, the replicas update their local integer and will converge to the same value.

AntidoteDB is a key-value store, where each value is a CRDT. AntidoteDB provides us a library of needed CRDT types. This means that the designer must map structures to the appropriate CRDT types and ensure that he can express the applications operations with the set of operations offered by the chosen CRDT types.

AntidoteDB implements for us the replication protocol and the persistence and acts as the only source of truth for our state. From the library of available CRDTs, we mainly use three kind:

- **Remove Win Set** (RWSet): A set data structure with add and remove operations. When a conflicting concurrent add and remove are issued (of entries that are equal), the add operation is discarded.
- **Remove Win Map** (RWMap): A map data structure that have also favor remove operation when conflict updates arise.
- **Last Writer Win Register** (LWWR): A blob of data which keep only the last update issued.

2.4 Assumptions and objectives

We want to leverage CRDTs to develop a file system that is always available and that provides good response times whatever the network conditions. It must support active/active configurations (i.e. two geographically distant

clusters can issue read, write and structural operation at the same time without coordination with each other).

Given the requirements, we want to keep the behavior of the file system as close as possible to a local file system. In summary, we want the following properties:

- **Preserve intention**: We dont want to perform too many actions or changes besides what the ones the user does. We want the user to be able to develop a simple mental model to understand the underlying convergence properties.
- **Truly concurrent operations**: One way to handle concurrency is to serialize operations applied on the relevant objects. CRDTs avoid this and allow true concurrency without the need for a consensus.
- **Follow the POSIX standard**.
- **Atomic operation**: No matter how complex a FS operation is, it should be either performed are completely discarded.
- **Always available**: CRDT types allow us to design systems that are always available even under extreme network conditions.
- **Active-Active**: Several replicas accept operations (structural and updates) concurrently and propagate them from one another, even after long delays.

Focus has been put on building on CRDTs to create a highly resilient and truly concurrent file system that follows the strict POSIX invariants while providing users a simple interface to deal with conflicting updates.

3 SYSTEM OVERVIEW

3.1 The layered architecture of ElmerFS

From a hight level point of view, a ElmerFS deployment consists of a number of data centers.

Each data center holds a full replica of the file system. Clients communicate with the data center nearest to them. All reads and writes are served by the local data center, without the need for coordination across data centers. Updates are asynchronously propagated between data centers. Data centers can continue serving user requests even if connectivity with other data centers is lost due to network partitions.

Within a data center, an ElmerFS cluster consists of an arbitrary number of nodes, and uses a shared-nothing architecture. Unlike other storage systems in the industry, there is no minimal requirement concerning the number of nodes in a cluster. User requests are served as long as their is an available node in the data center.

A node in a data center is an ElmerFS process, which consists of the following layers.

3.2 Interface

The interface layer is responsible for handling the user interaction with the file system.

It is based on the FUSE protocol, a user-space protocol to implements file systems. It receives FUSE requests, matches them with the corresponding operation available in the translation layer, and creates the appropriate response. This layer implements the standard FS operations.

ElmerFS is multi-threaded and asynchronous. Each FUSE request spawns an independent task that runs concurrently with other tasks.

3.3 Translation

The translation layer is responsible for translating FUSE requests to CRDT operations. It translates each high-level FS operation to a collection of operations on CRDTs.

All CRDT operations corresponding to a specific FS operation are bundled into a transaction. This ensures that FS operations are atomic.

3.4 Transactional CRDT

Requests from the translation layers are passed to the transactional CRDT layer that ensures that CRDTs are persisted. This layer is also responsible for replicating CRDTs across data centers. ElmerFS relies on AntidoteDB for implementing this layer.

3.5 Modeling the file system using CRDTs

ElmerFS represents the state of the file system using CRDTs. In particular, it models four main entities: inode objects, symbolic links, blocks, and directories. An inode structure stores metadata for an inode in the file system. We represent the inode structure using a map data type (RWMap).

In ElmerFS, files are sparse (gaps are allowed). We use a register data type (LWReg) of fixed size file blocks. We only allocate blocks for ranges that have been written.

We represent symbolic links as a special case of files that store only the target path.

We represent each directory using a RWSet. Directory entries in the set are ino - name pairs.

The file system hierarchy is implicit. A parent directory contains its child directories and a child directory keeps a pointer to its parent through the special ".." file.

4 REMAINING CORRECT

CRDTs ensure Strong Eventual Consistency (SEC) [6] but do not ensure that the application invariants remains correct nor it ensures that convergence lead to a state that is meaningful to the user.

The challenge is to keep those invariant always correct under any sequence of operation while ensuring that no data or user intention is lost through unintended conflict resolution.

When translating our high level operations, we had to consider problems stated in section 2 and chose the proper CRDT types, what metadata needs to be stored and what sequence of CRDT operations needs to be issued to remain correct.

4.1 Generating the inode number

To address this, we use a global counter. Access to this counter is serialized through a distributed lock.

To reduce the overhead of the contention on this lock, each time that we access the global counter we reserve a range of ino that we will then consume locally. Note that we also dont recycle ino of deleted inodes, we cant assert that all replicas have converged to a state where the ino is not used anymore. Generating 100 000 files per second, it would take around 8 million years to exhaust this counter.

4.2 Ensuring deletion

To ensure that delete operations are honored, we chose to use map and set that honors concurrent removes.

Favoring concurrent remove operations in our file systems means that when an unlink is concurrent with a link or a rename, the unlink will prevail when replicas will converge.

We can only update some key of the map and issue a remove operation on all possible existing keys (which is a finite amount when we are mapping structures) to ensure that the map will never converge to a partial state.

4.3 Keeping the user intention

In ElmerFS, we allow names conflict to happen and we expect the user solves those conflicts using standard FS operations that he is familiar with.

To be able to distinguish between two inodes sharing the same name under the same parent directory, we use an additional internal unique identifier, the ViewID.

Apart from being unique, there is no particular requirement for this identifier. We chose to use the file system user id (uid) and we expect that a user wont issue operations from two different processes.

Each time a user creates a link, we not only store the name and the ino of the link but also the ViewID of the user that created it. Because we use RWSet for directories with an equality put on names, entries that would have been previously considered the same (sharing the same name and not necessarily the same ino) are now distinct.

To interface with the user, we use the concept of partial and Fully Qualified Names (FQNs). Partial names are how the user named the link, FQN are partial names concatenated with the ViewID. Since the ViewID is unique, we know that all visible links are uniquely identifiable. Meaning that at anytime the user can chose to refer to its file from the partial name of the FQN.

In a situation where no conflict exists, we always show partial names. When there is a conflict, we only show FQN of entries that have a ViewID that doesnt match the one used by the user.

Showing only FQN when the ViewID mismatch allows the user to continue to work by name on the file while other conflicting operations might be merged. We always favor the ViewID of the user that issued the request.

For example, in a situation with two users, Bob and Alice where both have created a file name "us". If Bob tries to list the directory, he will see two files: "us" and "us:alice". To him "us" is still its own file and we do not lose any content when the CRDTs are merged.

4.4 Divergent renames

Without coordination, *mkdir* and *mknod* are not the only operations that can create a link. (See section 2.2 for problematic renames).

Because of this situation, using a counter to track the number of links is not enough anymore because at the time we issue the *rename* operation we cannot know if the operation will be concurrent.

To keep track of the number of links, we use another RWSet that is always updated alongside the directory entries updates.

Each link contains the parent inode number and the FQN which contains the ViewID. We use the ViewID again to have the exact same semantics as the set storing the directory entries. Thus the link set is always valid with respect to links currently visible in the FS.

However, this solution by itself is not enough to ensure that directories have a unique parent at all times.

This is solved with an additional Last Writer Win register that serves us as an arbitrator to decide which link is valid.

Each time we load directories entries, we check the parent register of each entry that points to a directory.

If the entry comes from a parent that doesnt match the value of the register, it is lazily removed and never shown to the user. Because the register is always updated inside the same transaction, the value of the register always matches the update that add the entry to the parent.

4.5 A simple conflict scenario

Using the design described above we can imagine how ElmerFS behaves in case of a conflicting scenario.

To illustrate this, let's take again Bob and Alice. Alice and Bob work on the same project but can't communicate. They don't want to wait and start to create a folder named *projectA*. Because the project will likely needs a report, they both start to create a report file *report.doc* to enter their early work.

Once the connection reestablished, Bob see that there is now two folders: its own folder *projectA* and another one *projectA:Alice*. He knowns that Alice must have put useful information in her folder and he decide to merge the two and simply move all file from *projectA:Alice*.

Alice inspect its folder and sees that Bob wasn't really attentive and merged their folder even though they had two identical *report.doc* files. She sees both *report.doc* and *report.doc:Bob*, adds the modification done by Bob in its file.

After an agreement between Alice and Bob, Bob decide to remove its report file. Now they both see a unique file tree *projectA/report.doc*.

This sequence is very close to what a user might expect when working with a local file system.

5 FUTURE WORK AND EXPLORATION IDEAS

5.1 Concurrent ino number generation and recycling

Our solution to generate an ino number depicted in 4.1 is not truly concurrent. To solve this issue and to be able to reuse previously used id, a specialized CRDT might be needed. One possible path to construct such CRDT would be to take inspiration from sequence CRDT used for text processing application where each inserted character is assigned an unique position (LSEQ [4], Tree-Doc [3]].

5.2 Cycle with concurrent rename

Our implicit hierarchy using map CRDTs does not prevent cycles to be created. Some CRDT tree design exists [Tree CRDT] but relies on multiple correction layers that perform additional operations to recover from the broken invariants. We believe that both this issue could be solved by the use of a conditional transaction framework. The idea would be to merge operations only if they fulfill a user provided condition.

For cycles, this would be that the resulting tree does not have a cycle. Transactions that don't respect the condition should be discarded in a deterministic manner to ensure convergence.

5.3 Dealing with Orphan CRDTs

Our deletion strategy solely relies on issuing a delete operation for all known CRDT of an entity.

For file content, where we store an implicit and unbounded number of CRDT, concurrent add operations merges can lead to some content lingering around without an entity to reference them.

Tombstones are sometimes used in CRDT design, but here we need a mechanism to link and propagate deletion across multiple CRDT.

We arent aware of any protocol that allows this. A possible framework could rely on a unique tombstone and use conditional transactions described in the previous section to discard concurrent add operations.

6 CONCLUSION

In the paper, we explored the design of a truly concurrent shared geo-replicated file system based on CRDTs.

The solutions that we propose for our problems allow ElmerFS to expose conflicts in a simple interface while preserving the POSIX FS invariants for some sequence of operations.

We hope that the solutions described in this paper to ensure that the application remains correct with only set and maps CRDTs can provide ideas and can help solving similar convergence issues that might arise in other applications.

In section 5 we see that they are still many paths to explore, conditional transaction, if they are applicable, might be a solution to some of these issue. Permissions and CRDTs merge semantics are a challenging topic [8].

We believe that CRDTs for file system are a path toward highly available and truly concurrent file systems and hope that more work goes into this direction.

REFERENCES

[1] Weiwei Cai, Agustina Ng, and Chengzheng Sun. 2018. Some Discoveries from a Concurrency Benchmark Study of Major Cloud Storage Systems. In *International Conference on Cooperative Design, Visualization and Engineering*. Springer, 44–48.

[2] 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 Transactions on Computer Systems (TOCS)* 6, 1 (1988), 51–81.

[3] Mihai Letia, Nuno Preguiça, and Marc Shapiro. 2009. CRDTs: Consistency without concurrency control. *arXiv preprint arXiv:0907.0929* (2009).

[4] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM symposium on Document engineering*. 37–46.

[5] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1995. Plan 9 from bell labs. *Computing systems* 8, 3 (1995), 221–254.

[6] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[7] John C Tang, Jed R Brubaker, and Catherine C Marshall. 2013. What do you see in the cloud? Understanding the cloud-based user experience through practices. In *IFIP Conference on Human-Computer Interaction*. Springer, 678–695.

[8] Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. 2021. Access Control Conflict Resolution in Distributed File Systems using CRDTs. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. 1–3.