Position: CRDTs for truly concurrent file systems

Anonymous Author(s)

ABSTRACT

Building scalable and highly available geo-replicated file systems is hard. These systems need to resolve conflicts that emerge in concurrent operations in a way that maintains the file system invariants, is meaningful to the user, and does not depart from the traditional file system interface. Conflict resolution in existing systems often leads to unexpected or inconsistent results. This paper presents the design of ElmerFS, a geo-replicated, truly concurrent file system built with the aim of addressing these challenges. Conflict resolution in ElmerFS is designed with the principle of preserving user intention. File system replicas converge to a common state through the use of Conflict-free replicated data types (CRDTs) for representing file system structures. The resulting state informs the user about the resolved conflict in an intuitive way. The user can then complement or reverse the conflict resolution outcome using traditional file system operations. We demonstrate this mechanism through conflict resolution examples.

CCS CONCEPTS

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

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

SYSTOR'21, June 14-16, 2021, Haifa, Israel
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

File systems services are essential for data sharing and collaboration among users. These services must provide low response time, remain available in the persistence of network partitions, and provide support for offline work [2]. To achieve these goals, these services typically rely on systems that replicate data among geographically distant sites and serve each user request from the replica closer to the user, without coordination with other replicas.

This type of design allows replicas to accept concurrent operations that conflict with one another, for example concurrently creating two files with the same name under the same directly on two different replicas. As a result, these systems face two challenges: resolving conflicts between concurrent operations in a way that is meaningful to the users while maintaining file system invariants, and ensuring support for legacy applications that have not been developed with mechanisms for dealing with concurrency anomalies.

It has been shown that existing file system services that support collaboration and offline work merge some conflicts in inconsistent, non-deterministic or unexpected ways [1, 8]. For example, in Google Drive the conflict described above can result in two replicas presenting two different views of the file system to users.

This makes it difficult for users to have an intuitive understanding about the behaviors of these services, leading to misconceptions on their expected behavior [7]. Moreover, enabling more flexible conflict resolution by exporting some decisions to the users while maintaining support for legacy applications through POSIX compliance is challenging.

In this paper, we present a comprehensive analysis of the challenges in the design of geo-replicated weakly consistent file systems. Guided by this analysis, we propose ElmerFS, a geo-replicated file system that provides intuitive conflict resolution semantics, while maintaining support for legacy applications. The design of ElmerFS leverages the properties of Conflict-free replicated data types (CRDTs) to ensure that concurrent operations on different replicas always converge into a consistent state. The guiding design principle is that conflict resolution should preserve user intention. Finally, we demonstrate through conflict resolution examples that a CRDT-based file system design can provide conflict resolution semantics that are intuitive to the user.

2 DESIGNING A FILE SYSTEM

2.1 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 challenges:

- **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, this is not compatible with the POSIX specification, which requires 8 byte ids.
- **Named links**: Operations that create or move objects (files or directories) may result in conflicts in which concurrent operations n different replicas create objects with the same name in the same directory. Existing systems resolve naming conflicts between files by automatically renaming files, and conflicts between directories either by renaming or by merging them.
- **Preserving the FS tree invariants**: Concurrent move operations without coordination may violate the file system invariant. For example, merging an operation that moves a directory A into a directory B with a concurrent operation that moves B into A can result in a cycle. Merging two concurrent operations that move the same directory to different destinations can result in a directory with two parents.
- **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. Moreover, even if a common conflict resolution is applied to all replicas, the result might not be what the user expects from the system.
- **Permissions changes**: Updating permission from a replica may not be propagated directly to other replicas. Merging an operation that removes a Bob's permission to write to file with a concurrent operation in which Bob writes to that file will result in a different outcome depending on the order in which operations are applied.

2.2 Conflict-Free Replicated Data Types

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

Conflict-Free Replicated Data Types (CRDTs) 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.

Moreover, the rules used for conflict resolution are parts of each CRDT's definition. Therefore, application developers can control their conflict resolution semantics by choosing the types of CRTDs they model their application with.

2.3 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 minimize changes not initiated by the user. The user should 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 use consensus 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 permit the design of 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.

A Focus has been placed 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

An 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. Every operation is served by the local data center, without the need for coordination across data centers, and updates are asynchronously propagated between data centers. A data center continues 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 node, and uses a shared-nothing architecture. Typically, each node is deployed on a separate physical server. Unlike other storage systems in the industry, there is no minimal requirement concerning the number of servers in a cluster.

An ElmerFS node is daemon consisting of the following layers.

3.1.1 Interface. The interface layer is responsible for handling interaction between the user and the file system.

It is based on the FUSE protocol, a user-space protocol used to implement file systems. The interface layer receives FUSE requests, matches them with the corresponding operation in the translation layer(§ 3.1.2), and creates the appropriate response.

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

3.1.2 Translation. The translation layer is responsible for translating FUSE requests to CRDT operations. Each high-level FS operation is translated to a collection of operations on CRDTs.

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

3.1.3 CRDT. The CRDT layer is responsible for replicating CRDTs across data centers and providing persistence.

ElmerFS relies on AntidoteDB for implementing this layer. AntidoteDB is a key-value store that models data as CRDTs.

ElmerFS represents file system structures as CRDTs. It uses the following CRDT types provided by AntidoteDB:

- **Remove Win Set** (RWSet): A set data structure with add and remove operations. In an RWSet, when add operation conflicts with a concurrent remove operation, the add operation is discarded.
- **Remove Win Map** (RWMap): A map data structure that favors remove operations when conflicts arise.
- **Last Writer Win Register** (LWWR): A blob of data that retains only the last applied update.

3.2 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 Remove Wins Map (RWMap). A RWMap is a map data that merges conflicting add and remove operations in favor of the remove operations

We represent files as a collections of fixed size blocks. Each block is represented using a Last Writer Win Register (LWWR). A LWWR can be viewed as a blob of data that retains only the last applied update. For concurrent updates, a mechanism, typically based on replica identifiers, ensures that the same retained across all replicas.

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

We represent each directory using a Remove Win Set (RWSet) (a set data structure that merges conflicting operations in favor or remove operations). Directory entries in the set are inode number - 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 ".." named file.

We chose the Remove Win semantics instead of its Add Win counterpart because it becomes easier to prevent partial state. We discuss this further in section 4.2.

4 ENSURING CORRECTNESS

CRDTs ensure Strong Eventual Consistency (SEC) [6] but do not ensure that the application invariants remain correct nor that convergence leads 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 conflict resolution.

In this section, we present how we address the challenges discussed in section 2.1 in ElmerFS through the choice of CRDT types for representing file system structures, the metadata that ElmerFS maintains, and the transaction of file system operations to operations on CRDTs.

4.1 Generating the inode number

We address this using a global counter. Access to this counter is serialized through a distributed lock.

To reduce the overhead of the contention on this lock, each access to the global counter reserves a range of inode number which can then be consumed locally.

Furthermore, ElmerFS, does not recycle the inode number of deleted inodes. This is because ensuring that all replicas will converge to a state in which an inode number is not used anymore is not compatible with supporting offline operation (a replica in which the inode number is still in use can reconnect to the system after an arbitrarily long period of time). It can be noted that, 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 maps and sets that honor concurrent removes.

Choosing a remove win semantic for our CRDT ensure that we don't get a partial state after a conflict resolution. If some operation that updates some keys of the inode structure is concurrent with an unlink that deletes all the keys, the CRDT won't be left with only the keys that were updated. All keys will be effectively removed.

4.3 Keeping the user intention

In ElmerFS, we allow names conflict to happen and we expect users to solve those conflicts using standard FS operations that they are 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.

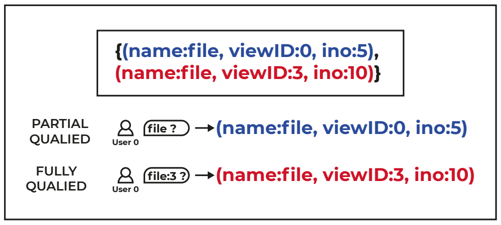
Each time a user creates a link, as illustrated in Figure 1, we not only store the name and the ino of the link but also the ViewID of the user that created it. Because we use RWSet for directories with an equality put on names, entries that would have been previously considered the same (sharing the 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. This means 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 does not match the one used by the user.

Showing only FQN when the ViewIDs 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.

Figure 1: An example of the name resolution in ElmerFS. The set above is what the folder contains.


{(name:file, viewID:0, ino:5),
(name:file, viewID:3, ino:10)}
PARTIAL QUALIED
User 0
file ?
(name:file, viewID:0, ino:5)
FULLY QUALIED
User 0
file:3 ?
(name:file, viewID:3, ino:10)

4.4 Divergent renames

Without coordination, *mkdir* and *mknod* are not the only operations that can create a link. (See section 2.1 for problematic renames).

Because of this situation, using a counter to track the number of links is no longer sufficient because at the time we issue the *rename* operation we cannot know if the operation will end up being 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 sufficient 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 garbage collected 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 conflict.

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

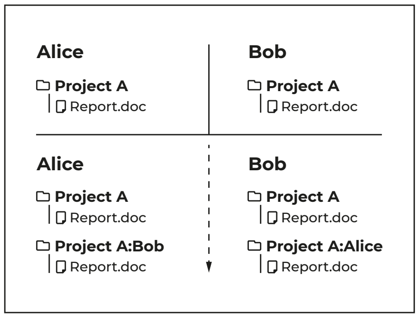
Once the connection reestablished, Bob see that there is now two folders: his own folder *projectA* and another one *projectA:Alice*.

They can both continue to work on their project without worrying about conflict or implicit merges. After some time Bob and Alice agree to merge their work. Alice remove its old no longer relevant folder.

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.

Figure 2: Alice and Bob conflict scenario.



5 DIRECTION FOR FUTURE RESEARCH

5.1 Concurrent ino number generation and recycling

Our solution to generate an inode number described in section 4.1 is not truly concurrent. To address this, and to be able to recycle inode numbers, a specialized CRDT type might be needed. A possible solution for specifying such a CRDT type would be to draw inspiration from sequence CRDT used for text processing application where each inserted character is assigned an unique position [3, 5].

5.2 Cycle with concurrent rename

Our implicit hierarchy using map CRDTs does not prevent the creation of cycles. Some CRDT tree design exists [4] 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 key idea would be to merge operations only if they satisfy a specified condition.

For cycles, this would simply be that the resulting tree does not have a cycle. Transactions that do not satisfy this condition should be discarded in a deterministic manner to ensure convergence.

5.3 Dealing with Orphan CRDTs

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

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

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

We are not 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 explore the challenges in the design of a truly concurrent shared geo-replicated file systems under weak consistency, and present the design of ElmerFS, a CRDT-based file system.

Conflict resolution in ElrmerFS is based on the goal of preserving user intention. ElmerFS ensures all file system replicas eventually converge to the same state, while also allowing users to complement or reverse the results of conflict resolution through a simple interface that preserves the standard file system POSIX semantics.

We have implemented a prototype of ElmerFS and are in the process of performing experimental evaluation.

While there remain open problems to be addressed, we believe that leveraging the properties of CRDTs is a promising path towards highly available and truly concurrent file systems and believe that future work should go in 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] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. 2012. Abstract unordered and ordered trees CRDT. *arXiv preprint arXiv:1201.1784* (2012).

[5] 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.

[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] Vinh Tao Thanh. 2017. *Ensuring availability and managing consistency in geo-replicated file systems*. Theses. Université Pierre et Marie Curie - Paris VI. https://tel.archives-ouvertes.fr/tel-01673030