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 introduces ElmerFS, a geo-replicated, truly concurrent file system built with the aim of addressing these challenges. ElmerFS is based on two key ideas: (1) the use of Conflict-Free Replicated Data Types (CRDTs) for representing file system structures, which ensures that replicas converge to a correct state, and (2) conflict resolution rules, which are determined by the choice of CRDT types and their composition, designed with the principle of being intuitive to the user. We argue that if the state of the file system after resolving a conflict conveys to the user the resolved conflict in an intuitive way, the user can complement or reverse the conflict resolution outcome using traditional file system operations. We present the design of ElmerFS and demonstrate conflict resolution in ElmerFS through an example.

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, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

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 presence of network partitions, and provide support for offline work [4]. To achieve these goals, these services typically 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 directory 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 resolve some conflicts in inconsistent, non-deterministic or unexpected ways [3, 11]. For example, in Google Drive the conflict described above can result in replicas presenting different views of the file system.

This makes it difficult for users to have an intuitive understanding about the behaviors of these services, leading to misconceptions on their expected behavior [10].

A solution to that could involve more flexible conflict resolution mechanisms, for example requiring user input in the process of conflict resolution. However, enabling such functionality 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 introduce 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 to a correct state. The guiding principle for designing conflict resolution in ElmerFS is that it should be intuitive to the user. This is achieved by designing conflict resolution rules that (1) preserve the effects of conflicting operations as much as

possible, and (2) do not introduce changes not explicitly expressed by the conflicting operations.

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 on 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.
- **Cycles**: 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.
- **Divergent renames**: The *rename* operation is semantically a move operation, it move a link from one folder to another. When two concurrent renames move the same link to two different places, if both rename are ultimately accepted, a additional link of the inode will be created. The file system must ensure that the number of link of a inode is always correctly tracked.
- **Deletion of inode**: When operations can be concurrent with the deletion of an inode, the file system must ensure that either the deletion is cancelled and the inode restored or that the deletion is kept honored.
- **Permissions changes**: Updating permission from a replica may take some time to be enforced in 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 Assumptions and objectives

We 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).

The behavior of the file system should remain as close as possible to a local file system. In summary, we want the following properties:

- **Preserve intention**: We minimize changes not explicitly requested by the user. The user should to be able to develop a simple mental model to understand the underlying convergence properties.
- **Truly concurrent operations**: The FS should be always available even under extreme network conditions. 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 or completely discarded.
- **Active-Active**: Several replicas accept operations (structural and updates) concurrently and propagate them from one-another, even after long delays.

Our focus therefore is to leverage 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 Modeling the file system using CRDTs

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 [9].

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.

ElmerFS uses the following CRDT types provided by AntidoteDB [2], a CRDT key-value store:

- **Remove Win Map** (RWMap): A RWMap is a map data type that associate an arbitrary key to a CRDT value, The Remove Win semantic arbitrates conflicting add and remove operations in favor of the remove.
- **Remove Win Set** (RWSet): A set data structure containing LWWRs. It has add and remove operations. As with the RWMAP, it favors remove operations in conflicting situations.
- **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 based on replica identifiers and timestamps, ensures that the same retained across all replicas.

ElmerFS represents the state of the file system using CRDTs. The four main entities are inode objects, symbolic links, blocks and directories. An inode structure stores metadata for an inode in the file system. using a Remove Wins Map

We represent a file as a collection of fixed-size blocks. Each block is represented using a LWWR. Blocks have a fixed size, they can be addressed with the concatenation of an offset and an ino. We do not keep track of the allocated block of a file, we rely on the file size to recover this information.

We represent a symbolic link as a special case of a file, storing exclusively the target path.

We represent a directory using a Remove Win Set (RWSet), a set data type with semantics similar to the RWMap. Directory entries in the set are inode number - name pairs. Directory contains its child directories, a child directory keeps a pointer to its parent through the special ".." named file. Ideally a RWMap whose keys are file names would have been a better choice, however AntidoteDB doesn't support querying specific fields of a map, we had to resort to using a RWSet but we considered two directory entry to be equal if their name are equal as any more practical implementation will want to be able to lookup entries with the parent inode number and the name without loading the whole folder in memory.

The design decision of choosing the Remove Win semantic instead of its Add Win counterpart is discussed in Section 4.2.

3.2 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 a shared-nothing architecture.

An ElmerFS node is a daemon consisting of the following layers.

3.2.1 Interface. The interface layer is responsible for handling interaction between the client applications and the file system.

It is based on the FUSE protocol, a user-space protocol used to implement file systems. The interface layer receive a FUSE request, calls the corresponding operation in the translation layer(§ 3.2.2), and creates the appropriate response.

ElmerFS is multi-threaded and asynchronous. Each FUSE request spawns an independent task that runs concurrently with other tasks. The kernel will ensure that on the same inode are serialized.

3.2.2 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.

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

ElmerFS uses AntidoteDB for implementing this layer.

4 ENSURING CORRECTNESS

CRDTs ensure Strong Eventual Consistency (SEC) [9] but do not ensure that the FS invariants remain correct nor that convergence leads to a state that is meaningful to the user.

The challenge is to maintain those invariants correctness under any sequence of operations while ensuring that no data or user intention is lost through conflict resolution.

In this section, we present how we address the correctness 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

As introduced in Section 2.1, file systems cannot leverage universally unique id generation algorithm due to the low number of bits an inode number is (8 bytes).

We are left with two possible choices. Adding synchronisation around a unique counter to prevent two replica allocating the same number or to shard the number generation with a fixed, known number of shard.

ElmerFS use the first solution, a 8 bytes counter. To reduce the overhead of the contention on this lock, each access to the global counter reserves fixed range of inode number which can then be consumed locally.

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, this would require strong consistency, 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 8 byte counter.

4.2 Ensuring deletion

Following the two possible choice exposed in Section 2.1, ElmerFS always honors an inode deletion.

Choosing a remove win semantic for our CRDT ensure that we don't get a partial state (a state where some fields of the inode's metadata remains) after a conflict resolution.

For example, if an operation updates the inode *ctime* and is concurrent with the deletion of the inode, using Add Wins Map would leave the inode with no field but the *ctime* one. With a Remove Wins semantics, because we issue the deletion of all the map's keys, the system always converge to an empty map.

The drawback of this approach is that we must know in advance all the keys that might exists map/set to issue a deletion for all the possible keys of this map/set.

4.3 Resolving name conflicts

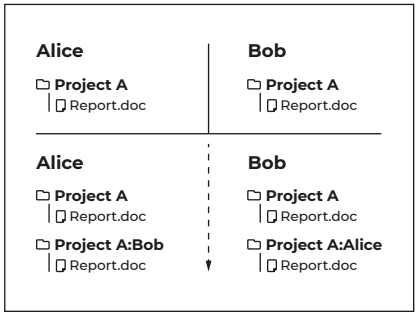
For availability under partition, ElmerFS allows name conflicts to happen. We expect users to solve those conflicts using standard, familiar file system operations.

4.3.1 A simple conflict scenario. To illustrate this, let's consider a scenario in which Alice and Bob collaborate on a common project. Alice is in a flight without internet access, therefore their replicas are partitioned. Both Bob and Alice create a directory named *projectA*. Because the project will likely need a report, they both create a report file *report.doc* and start working on their report.

Once connection is re-established, changes are propagated among the two replicas. As a result, Bob sees that there are now two directories: his own directory *projectA* and a new directory named *projectA:Alice*. Alice in turn see her own directory *projectA* and a new directory *projectA:Bob*. 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 by using one of the two folders and removing the other one. Finally, they both see a unique file tree *projectA/report.doc*.

Note that this sequence is very close to what a user might expect when working with a local file system. While this example was about folder name conflicts file behaves in the same manner, reducing risks of data loss.

Figure 1: Alice and Bob conflict scenario.



4.3.2 Name conflict resolution in ElmerFS. 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 user id (uid) and we expect that a user wont issue operations from two different processes. Note that it is a simplistic choice, many application might log in under the same user on a system, we chose this to be able to map the ViewID to sensible name. A more robust system could use an unique id associated with the ElmerFS process and then add metadata to map it back to a username for example.

Each time a user creates a link, as illustrated in Figure 2, the system stores the name and the ino of the link

as well as the ViewID of the user that created it. Entries that would have been previously considered the same (sharing the same name) 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.

At any time, the user can chose to refer to its file from the partial name or the FQN.

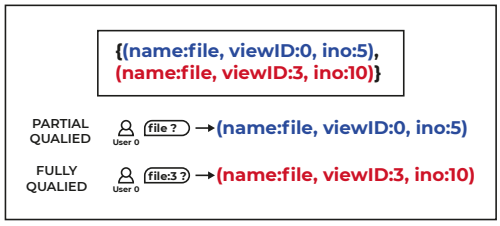
Since the ViewID is unique, we know that all visible links are uniquely identifiable. Because we cannot show duplicate names under a folder, when a conflict has occurred we do not display the partial names but the FQNs of the conflicting files. Otherwise, the system behaves as a local file system, only showing the original file names.

To prevent applications that do not expect files to be renamed. When there is a conflict, if a request only use a partial name, we always favor the ViewID of the requester. If the application's file is subject to a conflict, the application will still be able to refer to its file directly without interruption.

Comparing to have a system that renames files, we always preserve the original name, when conflict happens, the application can still function without worrying of external updates. Intuitively, the user can always see its own file as untouched by the underlying system.

As a drawback, inodes can be queried in two ways at all time, which departs from the POSIX standard. Additionally, it adds the risk that two applications thinks to work on the same inode where in fact they are not. This can occurs for applications that unconditionally create a file expecting that one of the creation request will fail.

Figure 2: An example of the name resolution in ElmerFS. The set above is what the folder contains.



4.4 Divergent renames

As explained in Section 2.1, without coordination, *rename* can not only create cycles but also additional links.

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. We risk wrongly count the number of links and deleting an inode prematurely.

We use another RWSet that is always updated in the same transactions (§ 3.2.2) that create or remove a directory entry.

Each link contains the parent inode number and the FQN that 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, POSIX forbids directory to have multiple links. While we have the correct set of link for our inode, we also need to ensure that even after a divergent rename, only one link of the directory will stay visible.

An additional LWWR is used as an arbitrator to decide which link is valid. The LWWR is updated inside the *rename*'s transaction (§ 3.2.2) and stores the parent inode number.

When ElmerFS loads a directory entry, it first check that the LWWR stored parent correspond to the directory being looked up. If they do not correspond, the entry is removed and the file system correctly inform the user that the entry does not exists. The drawback is that we may never reclaim the entry with the wrong parent if the parent is never looked up.

5 CONCLUSION

In this paper, we explore the challenges in the design of a truly concurrent shared geo-replicated file systems under weak consistency.

We propose ElmerFS, a CRDT-based file system. ElmerFS ensures file system replicas eventually converge to a common, correct state in the present of conflicting operations. Conflict resolution in ElmerFS is designed with the aim of not resulting in unexpected results. We argue that this enables users to complement or reverse the results of conflict resolution through traditional file system operations.

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.

6 DISCUSSION

In this section, we introduce directions for further research and open questions that we would welcome feedback on:

- **Concurrent ino number generation and recycling.** Our solution to generating a inode numbers, described in section 4.1, is not truly concurrent. A possible solution for addressing this could be a specialized CRDT type. The specification of such a CRDT could draw inspiration from the sequence CRDTs used for text processing application where each inserted character is assigned a unique position [5, 8].
- **Cycles with concurrent renames** Our implicit hierarchy using map CRDTs does not prevent the creation of cycles. CRDT tree designs have been proposed [6][1], but rely on multiple correction layers that perform additional operations to recover from broken invariants. We believe that both these issues could be solved by the use of post-conditions on merging concurrent updates. The key idea would be to merge operations only if the resulting state satisfies a given condition. For cycles, this would condition would be be that the resulting tree does not have a cycle [7]. Transactions that do not satisfy this condition would be discarded in a deterministic manner to ensure convergence.
- **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 CRDTs, concurrent add operations conflict resolution 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.
- **On performance and scaling**: We are conducting performance and scaling evaluation of ElmerFS. Our initial results show that ElmerFS lacks optimizations that more mature file system implement to achieve high throughput. We currently only implement write gathering and we would like to explore the behavior of our distributed file system in larger scale to support enterprise level workloads.
- **On conflict resolution for other operations**: We have not explored all possible conflict in ElmerFS yet. Permissions in weakly consistent systems are challenging [12]. Cycles through rename operation are still possible due to our implicit tree representation. We would like to test specialized CRDT that prevent such occurrences while still allowing our current behavior. Recent work on tree CRDT might be a good fit [7].
- **On content aware file systems**: The way ElmerFS currently stores file content puts user data at risk in case of conflicting updates. A possible path to explore would be to select specific CRDTs depending on the file extension to allow concurrent file updates. Another path would be to design snapshot-able file to keep multiple versions in case of conflicts.

REFERENCES

[1] Mehdi Ahmed-Nacer, Stéphane Martin, and Pascal Urso. 2012. File system on CRDT. *arXiv preprint arXiv:1207.5990* (2012).

[2] Deepthi Devaki Akkoorath and Annette Bieniusa. 2016. Antidote: the highly-available geo-replicated database with strongest guarantees. *SyncFree Technology White Paper* (2016).

[3] 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.

[4] 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.

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

[6] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. 2012. Abstract unordered and ordered trees CRDT. *arXiv preprint arXiv:1201.1784* (2012).

[7] Sreeja S Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, and Marc Shapiro. 2021. *A coordination-free, convergent, and safe replicated tree*. Research Report RR-9395. LIP6, Sorbonne Université, Inria de Paris ; Universidade nova de Lisboa. 36 pages. https://hal.archives-ouvertes.fr/hal-03150817

[8] 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.

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

[10] 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.

[11] 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

[12] 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.