# ElmerFS: Building a file system using CRDTs

Romain Vaillant
romain.vaillant@scality.com

## Abstract

**ACM Reference Format:**
Romain Vaillant. 2021. ElmerFS: Building a file system using CRDTs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

File system is one of the most approachable user interface to organize and access data that we have today.

With the recent evolutions, the need in an storage capacity and good response times to distribute and share information across the world has only increased. Geo-distributed file systems are a key component to respond to this need.

However, building geo-distributed file systems are notoriously hard and harder right, it is an old problem (ref WeTransfer, BellLabs, AFS). Active-Active replication results in conflicts that are hard to resolve. Serializing critical operations with distributed locking can help but it is slow and problematic under failure.

Many existing system are either unavailable or incorrect, or both, under some critical operations, and do not scale well to an enterprise workload.

This paper address the problem of highly available shared and geo-replicated distributed hierarchical 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 don't need any coordination. They are central to build systems that are highly available, truly concurrent and efficient even under extreme failure.

Our designs ensure that our file system remain correct and available under network partition and server crashes. Our experiments shows that ElmerFS can scale up to 3 geo-distant data centers with 5 node each while being issued operation fighting for the same resource.

## 2 Designing a file system

### 2.1 What a basic file system requires

Our requirement is to follow the POSIX standard. We implement a user-space file system using the FUSE protocol (ref).

A file system supports structural operations operations (to manipulate the file system structure, permissions and retrieving some information) and content operations (read/write arbitrary amount of data to a given file).

We use the following terms:

- **A inode**: Is the internal representation of a filesystem structure object (a file or a directory or a symbolic link).
- **A link**: A reference to some inode. An inode that is not linked may be delete
- **An ino**: A unique id used to reference an inode.
- **A directory entry**: The named and visible representation of a link. When a name is used to reference an inode, the system resolve internally the name into an ino.

The key structure of a file system is its tree hierarchy, the most common properties that we require are:

1. Nodes in this tree are links to inodes. A directory is a container of those links.
2. Directories can have at most one existing link.
3. All nodes are always reachable from the root directory.
4. Cycles are possible only through symbolic links.

The file system must also be resilient and concurrent and have a behavior close to what an user might see in a local filesystem supporting multiple users.

We whish to support enterprise workloads with concurrent user interactions. Our expectation are that the application should be able to run up to 5 geo-distanced clusters with a minimum of 5 nodes per cluster.

Each node should be able to provide enough read write performance for writing and reading documents and images.

No focus have been done on I/O performances, but to adhere to this workload we require to stay in the range of 1mBytes/s for writes and more than 10mBytes/s for reads. We don't expect any limit on the file size apart from the underlying storage capacity.

### 2.2 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:

- **Truly concurrent operations**: One way to handle concurrency is to serialize operation applied on the relevant objects. CRDTs avoids this and allow true concurrency without the need for a consensus.

- **Atomic operation**: POSIX impose that every FS operation is atomic.
- **Always available**: CRDT types allows to design system that are always available even under network extreme conditions.
- **Active-Active**: Several replica accept operations (structural and updates) concurrently and propagate them from one another, even after long delays.
- **Least surprise**: We don't want to perform too many actions or changes besides what the ones the users does. We want to avoid rollbacks or drastic structural changes behind the scene to stay close to a local file system behavior.

## 2.3 Concurrency in a File System

Under an environnement without coordination, a number of new issues can arise to preserve the file system invariants. Let's consider some of those problematic operations and their characteristic..

**2.3.1 Inode creation.** All operations that creates inodes (*mkdir* for directories and *mknod* for files and symbolic links) needs to generate an unique id.

This requires to ensure that no replicas can ever generate the same id even if they can't talk to each other. This problems is often solved with the use of UUIDs, a 16 bytes number, which have a large enough domain such that the probability to generate a duplicate is practically null. However, in POSIX, the requirement for this id is to be a 8 bytes number.

**2.3.2 Named links.** Operations that create or move links (*rename*, *link* and creation operations) need to consider the case where two users concurrently want to create a link with an identical name inside a same directory.

In local file systems or distributed with coordination, this problem does not exists as those operation are serialized. Serializing those operation by the use of distributed locking is an option, but that would lessen the advantages of CRDTs as almost all structural operations would need coordination.

Additional problems arise if we decide to accept temporary conflicts. In such cases, we have to inform the user of a conflict in the POSIX interface which wasn't designed for this kind of situations. Care needs also to be taken to not impact the current workload of one user from external structural updates.

**2.3.3 Preserving the FS tree invariants.** By its nature *rename* is an exclusive operation. As soon as a link is moved, subsequent moves on the same link should fail. When we decide to not serialize, this operation can create cycles or unexpected additional links.

For example, without coordination, two users trying to rename two directory into each other might make each directory be the parent of the other. Once this situation arise, the concerned directories aren't reachable from the root directory anymore.

The user can't solve the problem by himself and the application must decide what must be done. Many distributed file system fails to handle this situation, leaving the user with an unrecoverable error.

When *rename* is used concurrently on the same source link to a different destination, if both operations are accepted, it will create one additional link of the inode. Additionally, for directories, the POSIX interface requires that directories can only have one unique parent at all time.

**2.3.4 Deletion of inodes.** The *unlink* operation, which removes links, poses problems in the presence of concurrent operations that can create links.

Consider the simple case where one user *unlink* a file while another one *link*s it. The inode's state stays correct when the application will process the link operation from another replicas.

Recycling the inode (its unique id for example) also becomes a problems. We cannot know when all replicas that once saw this inode will converge to a state where this inode is not reachable anymore.

Similar issues concerning permissions can be observed. Usually coordination is needed to ensure that permission change are honored by every replicas at the time at which they are applied.

## 3 System Overview

ElmerFS is a process that represent a file system replica. Replicas sharing the same data-center share their storage as one unit and forms a cluster to answer local user operations. For example, a user issuing a write at a replica will effectively write to one node of the cluster. Following reads, will lookup the correct node that contain the relevant content.

When multiple data-centers are involved, a full asynchronous replication takes place, with each data center holding the most recent view of file system that it is aware of. With our design, it is possible for this data-center to become isolated for an arbitrary amount of time. Also, no central authority is needed. The is not a minimal requirement of node per cluster.

An ElmerFS process is composed of the following layers:

### 3.1 The FUSE layer

The FUSE layer: This is the layer that allow us to interface with the user. We provide a mount point and answer user requests coming from the kernel. This layer implements the standard FS operations.

It takes incoming FUSE requests, match them with the correct operation available in the translation layer and then create the appropriate response.

ElmerFS is multi-threaded and asynchronous, each FUSE request spawns a single independent task that will run concurrently with the rest of the system.

### 3.2 The persistent CRDT layer

The persistent layer is responsible to persist CRDT. In our case, persistence and replication is achieved by AntidoteDB, a distributed CRDT database.

It 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 application's operations with the set of operation offered by the chosen CRDT types.

AntidoteDB implements for us the replication protocol and the persistence and act as the only source of truth for our state.

From the library of available CRDTs, we mainly use three kind: register which are blobs of data with Last Writer Win (LWReg) semantics and Remove Win Maps/Sets (RWMap and RWSet) that favor remove operations in case of conflicting adds.

### 3.3 The model Layer

The model layer is where we map CRDT to the in memory state of the application. We split ElmerFS state into four mains entities: inodes, symlink paths blocks, directories entries.

Inodes which store POSIX metadata information of any inode in the file system are stored as RWMap mapping each attribute.

Files in ElmerFS are sparse, gaps are allowed. We use LWReg of fixed size to store arbitrary user content. We only allocate blocks for ranges that have been written.

Symbolic links are just a special case of arbitrary file content. They store the actual target path.

Folders are stored as RWSets of ino and names. The file system hierarchy is implicit, parent folder are contains their child directories and child directory keeps a pointer to their parent through the special ".." file.

### 3.4 The Translation Layer

This translation layer is where we maps the application operations to the correct sequence of CRDT operations. This is where the main logic of the file system happens.

This layer also ensure that FS operation are atomics. This achieved "for free" by the use of the interactive transactions provided by AntidoteDB. Each high level operation wrapped inside a single transaction. In these transaction we are free to perform any number of read or updates on multiple objects. Updates will all be visible after the final commit.

## 4 The translation of high level operations

CRDT ensure Strong Eventual Consistency (SEC) (ref) but do not ensure that the application invariants remains correct. The challenge is to keep those invariant always correct under any sequence of operation.

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

As stated in section 2.3.1, POSIX requirements on the inode number are hard to satisfy in a system without coordination.

When *mkdir* and *mknod* are called, we need to find an unique id for the created inode. To solve this problem, we use a global counter whose access is serialized through a distributed lock. To limit this contention point, each time that we access the global counter we reserve a range of ino that we will then consume locally.

We still haven't found a proper CRDT that provides an operation to generate unique identifiers across replicas.

Note that we also don't recycle ino of deleted inode, we can't assert that all replicas have converged to a state where the ino is not used anymore.

### 4.2 Ensuring deletion

To ensure that delete operations are honored, we had to chose RWMap and sets. Favoring concurrent remove operations in our file systems means that when a *unlink* is concurrent with a *link* or a *rename*, the *unlink* will prevail when replicas will converge.

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

This address the some of the issues mentioned in section 2.3.4. However, we believe a better and more general solution would be to allow transaction to be paired with deterministic conditions to decide if it should be committed or aborted when they are concurrent with other transaction concerning the same objects.

### 4.3 Interfacing with the user

In ElmerFS, we allow names conflict to happens and we expect the user solves those conflict using FS operations that are available to him.

To be able to distinguish between two inodes sharing a same name under a same parent directory, we use an additional 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 won't issue operations from two different processes.

Each time an user creates 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, entries that would have been previously considered the same 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 link are uniquely identifiable.

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 doesn't match the one used by the user. Showing only FQN when the ViewID mismatch allow the user to continue to work by name on the file while other conflicting operation might be merged. We always favor the ViewID of the user that issued the request

For example, in a situation with two user, 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.

This solve the issue on named links described in section 2.3.2.

## 5 Experimentation and future explorations

### 5.1 Convergence and overhead.

### 5.2 About conditional transactions ?

## 6 Conclusion