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

1. **TODO: Talk about POSIX earlier**
2. **TODO: Insist on the GEO distributed part**

File systems exists to abstract and give structure to the general problem of storing information on a storage medium. Storage lifetime requirements can range from storing for a few microseconds to an endless amount of time.

They also have a very wide range of applications. With our personal computers, file systems only have to manage a couple of storage media and handle concurrency locally between applications run by one user. On a broader scale, file hosting services not only need to cope with the usual problems that can be encountered with persistent media but also deal with concurrency between multiple users acting on the same files and directories across multiple data centers, data centers that can suffer high latencies due to their geographic distance.

Being a critical component, high availability for those services is also a strict requirement. Even small failures can lead to hours being wasted waiting for the service to go back to normal and sometimes without the possibility to access to a partially functioning interface to have the possibility to work locally in the meantime.

On the opposite of file systems, we have object stores that are able to persist, often immutable, arbitrary blob of data. Those blobs are uniquely identified under a common namespace and act as the sole unit of abstraction. They have their advantages, notably because they don't suffer from the burden of a strict hierarchy and have a much limited set of available operation: the simplest object store can only provide the put and get operations whereas, a file system, have all operations needed to manipulate the tree hierarchy. As a consequence they are simpler in their structure and are easier to scale, they are now a strong component in Cloud services.

However, hierarchical file systems still is one of most approachable user interface to organize and access data that we have today. The challenges that they bring to be distributed, resilient, scalable and efficient still exist.

Distributed systems are notoriously hard to build and harder to build right. Here, we explore the path of building a plausible distributed file system that is geos-distributed, highly resilient and scalable by using **Conflict Free Replicated Data Types** (CRDTs) as our main framework: data structures that doesn't need any coordination to be replicated. To ease our development we leverage **AntidoteDB**, a distributed database based on CRDTs and we use the rust programming language, a compiled language free of any runtime which enforce memory safety and thread safety with its type system.

Our findings are that is is possible to build a file system spanning multiple data-centers using CRDTs that can work under extreme failure and network conditions while resolving complex concurrent user interaction with an intuitive user interface.

2 Designing a file system

2.1 What a basic file system requires

There are many ways to implements a file system depending on its usage. We decided to design a distributed file system that follows the POSIX standard. More precisely, we implements a user-space file system using the FUSE protocol, which avoid the need of creating a kernel module and all the intricacies that comes along with it.

A basic distributed file system must provide both metadata operations and content operations. Metadata operations are here to manipulate the file system structure, permissions and retrieving some information. The content operations are there to allow the user to persist an arbitrary amount of data to a given file and then read it. It is also common to expect from distributed file system some amount of resiliency and concurrency handling; at least for manipulating the file system structure. For content operations, a user should be able to write files that range from a few bytes to multiple gigabytes.

Drawing from the POSIX standard, we will use in this document few terms that we define here:

- **An inode**: It is the internal representation of a filesystem object: file or directories, symlink all share a same common information stored as an inode.
- **A link**: A reference to some inode. An inode that doesn't have any link, doesn't need to be kept, it can be deleted. Usually, all references are visible by the user.
- **An ino**: A 64bit unique id used to reference an inode.
- **A directory entry**: The named and visible representation of a link. In most cases users references inodes

by using their names which is then resolved into an ino.

- **A block**: This is the smallest unit of storage available. Files always have a effective size that is aligned to the size of a block.

2.2 Assumptions and precise goals

The goals of our implementation is to provide a plausible file system service that is geos-distributed and always available. This means that we aim to implements all basic operations that one might expect from a basic file system. Being geo-distributed we want our file system to be always available and provide good latencies whatever the network conditions are. The file system must be able to work in an active-active situation, two clusters are able to issues read, writes and metadata operations at the same time without waiting on each other.

We aimed for a system that works first and that helps us to see what it takes to use CRDTs for real systems instead of focusing on throughput and raw performances. Of course, waiting for an unbearable amount of time is still considered as a failure. In summary, most of the design focus was done on the metadata operations and their semantics rather than their optimizations.

One can expect the following properties from our file system:

- **No locking**: One way to handle concurrency is to use distributed locking to serialized operation applied on the relevant objects. CRDTs provides a way for us to escape this and to allow true concurrency without the need for any consensus protocol. We aimed make the locking optional and applying it at the will of the user. All metadata and updates operations do not takes any distributed locks by default.
- **Atomic operation**: We want to keep the behavior of POSIX file systems in which every operations are atomic.
- **Active-Active**: As stated above, we want updates to be able to come from any source at anytime no matter how old the update is.
- **Least surprise**: We don't want to perform to many actions or change besides what the users does. Meaning we want to avoid rollbacks or drastic structural changes behind the scene.

2.3 A high level interface for CRDTs

As an applications can rely on an external database service for persisting its data preserving ACID constraints, here we use AntidoteDB to build to provide us the existing tooling and infrastructure to handle CRDTs and their replication. AntidoteDB by default uses the Cure protocol, allowing us to performs interactive transactions that concerns multiple objects with consistent read and writes.

Each operation in ElmerFS are wrapped inside an AntidoteDB transaction. Any failure is able to rollback all operations that we meant to commit. No other specific work have been done is this regards, all is handled by the protocol as when using transactions in common databases.

Using this infrastructure and abstraction, ElmerFS is only a thin layer between the application interface (here POSIX) and the AntidoteDB database. We only have to be concerned by the application main logic: chose the right CRDTs for our structural representation and issue the correct sequence of operation inside a transaction.

2.4 Predictability in latencies

As ElmerFS is a thin layer which rely on multiple roundtrip with AntidoteDB, we always deploy ElmerFS and AntidoteDB side by side on the same machine. AntidoteDB receives our requests, applies the CRDT transformations locally and give us a response as soon as it is complete.

This allows to reduce the latency between the user's operation and our response. The local AntidoteDB is expected to join permanent clusters with multiple running AntidoteDB instances. The latency between the local AntidoteDB and the cluster might be high, but the perceived latency for the user stays relatively low. Since we do not require mandatory locking, we never have to wait for a majority of the quorum, everything is as local as possible.

3 Building blocks for building a file system

3.1 Composing CRDTs

AntidoteDB provide us a few basic CRDTs that we can use and compose together to represent the state of the application. From a high level point of view, AntidoteDB is a key value store where each value is a CRDT. When dealing with a database, we have to map the in memory application state to CRDTs. In a statically typed programming language such as Rust, it means that we must map our structures and enumeration to some CRDTs.

For structures, we were naturally leaned toward map CRDTs. In AntidoteDB they are multiple kind of maps to chose from each with specific convergence properties. It is up to the application to chose what would be the best fit for its business logic.

Grow Only Maps (GOMaps) are maps where the removal of elements is not possible. At first, we thought that this would be a good fit to represent our structures. Rust is statically typed, we cannot really remove one specific field: either all fields are initialized or none of them. However, AntidoteDB doesn't have the notion of existence and we cannot ask it to simply delete a GOMap when the entity has been deleted (**TODO: Maybe explain why**). Remove Win Maps (RWMap) are used instead, with the nice property that we can only issue updates of specific fields and not worrying of having structure partially existing as that would be the case with Add Win Maps (AWMaps).

Most of the fields can be encoded as simple registers. The CURE (**TODO: ref!**) protocol which is used by AntidoteDB allows us to perform read-write transactions spanning multiple CRDTs and ensure that all registers stay consistent in respect to each other.

We will see in what follows that by only composing CRDTs we can represents most of the file system state with complex convergence properties. In a way, composing CRDTs is the tool for the application to create one final CRDT, which is the application state. **Having the right abstracted and formalized CRDTs is key to allow an application to model it state without having to formalize one specialized CRDT for each domain and without having to bend its domain logic in a diminishing way**.

3.2 A common base

We won't describe in details how ElmerFS identify its entities, it simply computes a composite key from a kind (inode, block, directory, ...) and an ino.

The ino generation is also very simple. Each instance reserve a predetermined amount of ino beforehand and keep them in memory. To ensure that multiple instances doesn't produce the same ino, we have to use a lock. A counter CRDT doesn't work for this (we have to read the value, not just incrementing it). In fact this is the only place where a mandatory lock is used. A sequence CRDT (**TODO: which one**) might be a solution but aren't available in AntidoteDB.

A consequence of this approach is that ino are never recycled, but 64bit is a large enough domain for this to not be problematic.

File, directories and symbolic links use a same common base, an inode. An inode represents any file system entity by storing metadata information: permissions, owner, creation, modification and access time. As described above, an inode is represented as a RWMap. We made the decision to not store data information of the file and the symbolic link path alongside the metadata.

As only inode represents the existence of entities, this separation leads to potentials leaks if some the inode is deleted in one site and written in another. However, it is more a limitation of the API of AntidoteDB than CRDTs which doesn't support partial read or update.

3.3 The heart of a file system interface

One of a key component of a POSIX file system is its tree hierarchy. This tree as many complex properties, the ones that are the most common 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.

It easy to see that without care and just applying CRDTs merging semantics can easily breaks those constraints. There are active research (ref?) to provide a Tree CRDT with add, remove and move operations. In ElmerFS we didn't implement one of such CRDT that would have ease respecting some of those constraints. We explored the path of only using RWMaps. Even though we still have found any simple solution to prevent cycle apart from keeping track of all rename operations and cancelling the one in conflict in a deterministic way, we still manage to handle most of the concurrent tree operations gracefully without distributed locks.

4 Embracing the CRDT merging semantics

To handle most conflicting operations only with common CRDTs ElmerFS introduce the concept of an owner of each operation.

4.1 Some problematic cases

First, let's consider some problematic cases of the rename operation. On an POSIX file system, a rename operation is expected to be exclusive with respect to the the renamed inode.

In a concurrent scenario, the first problematic situation arise when two users tries to rename distinct inodes to the same target location with the same name. In that case, CRDT maps merging semantics to associate names to ino do not correspond to what we would want as one of the inode reference will be lost.

One should note that name conflict can also happens with all operations that can create links: link, mkdir, mknod…

The second situation happens when a unique reference is renamed by two users at the same time to two different location. The standard behavior is that one of the operation will fail. This pose two new problems: In a completely asynchronous situation, how do converge to a state that is meaningful to the users and how do we preserve the reference count with an operation that semantically doesn't affect it ?

4.2 Keeping conflicts where they make sense

In ElmerFS we solve those problems by using an unique value that we call the ViewID. This unique ID is used to identify the origin of the operations and to use the merge semantic of map and set CRDTs to our advantage. Apart from being unique, there is no particular requirement for this ID. In ElmerFS we used the file system user id (uid) assuming that a user might not be issuing operations from two different processes. Another solution would have been to tie an unique id to each process but we find that the mapping between the user and its actions is more straightforward in the former solution.

In our application we use a RWSet CRDT to represent directories. We depart from standard file system by not only storing a pair of ino and name to represent a link but also by storing the ViewID alongside this original pair. In a basic scenario where previously we could end up with two entries with the same name and with potentially different ino numbers. Here, we ensure that we either don't have a conflicts (no tuple in the set share a same name entry) or that we keep all succeeded operations in our application state.

By adding uniqueness to our directories entries we allow precise conflicts to bubble up to the application logic when they make sense. When two user tries to create a new file inside a same directory, we don't really want one operation to fail afterward. As soon as the user created the file and started writing it, it is crucial for the file system to keep this file. Conflicts here allow us to choose how to present and how to manage conflict without designing a specific CRDTs.

4.3 Linking and reference counting

By using the same mechanism we can also keep track of the number of reference of each inode where a counter CRDT would obviously fall short. Imagine two users renaming the same file to a different location. In ElmerFS, we accept this kind of operation but it means that this operation is similar to creating an additional link. We use a set CRDT to store all links of an inode. Inside a transaction, we issue a remove operation of the previous link and an add operation of the new link. The remove operation will contains the exact tuple matched (with the ViewID that created the link), When the CRDT converges, it receives to identical remove operations, and two distinct operation add operations leading to the right reference count.

Directories have the additional requirements that they can only have one existing link in the file system. To preserve this invariant, we also use a register with last writer win semantics to decide which reference is valid. Note that we could also allow multiple link of a directories, this is only a limitation of the POSIX interface.

4.4 Interfacing with the user

5 Experimentation and future explorations

5.1 Convergence and overhead.

5.2 About conditional transactions ?

6 Conclusion