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, 3 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 tody. This paper address the problem of highly available shared and geo-replicated distributed hierarchical file systems (FS).

This is an old problem, the Andrew File System, a pioneer in distributed file systems, dates back from 1982 and still receive updates today (refs). It is based on a client-server architecture enabling file shared and replicated read-only content distribution. Firsts versions were able to scale up to hundreds of concurrent users (ref).

Bell Labs released a distributed operating system (OS) in 1992 (ref). Based on the 9P protocol, this OS represents almost every resources and service interfaces as files. Remote access and control is done with the standard FS operations: mount, reads and writes.

Recently, cloud based file systems have appeared, focusing on ease of use. Google Drive is a file hosting service that allows user to store, share and synchronize files on multiple devices. It supports concurrent access across the world and integrate with other services such as Google Doc or Google Slide for collaborative work. Dropbox (ref) offer the same kind of services.

WeTransfer (ref), a file sharing service, can rely on a simpler abstractions, object stores. They allow to persist often immutable blob data identified by a numeric key. They don't provide a strict hierarchy and support a limited set of operations: put, get and delete. They are easier to scale and have become prevalent in cloud services.

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

Distributed systems are notoriously hard to build and harder to build right. Here, 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. We leverage **AntidoteDB**, a distributed database based on CRDTs. Our implementation uses the rust programming language, an efficient language which enforces memory and thread safety statically.

Our experience shows that is is possible to build a file system, spanning multiple data-centers, using CRDTs, that remains correct under extreme failure and network conditions, while supports a heavy concurrent user workloads with an intuitive user interface.

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 with up to 100 concurrent users.

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

Our main objective is to implements a file system based on CRDTs. 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.

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.

3 Concurrency in a File System

From a high level point of view, AntidoteDB is a key-value store, where each value is a CRDT. AntidoteDB provides us a library 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.

First, let's consider FS operations and their characteristics under an environnement without coordination.

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.

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.

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.

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.

4 Building blocks

4.1 Atomic Operations

Each operation in ElmerFS is implemented as an AntidoteDB transaction. They are interactive and are across multiple objects with causally consistent reads an writes. Atomicity of our FS operations comes "for free" by using this functionality and we designed ElmerFS as a layer between the POSIX interface and the AntidoteDB.

4.2 Composing CRDTs

We split ElmerFS state into four mains entities: inodes, blocks, directories entries.

Inodes stores all POSIX metadata information. We represents this entity using a Remove Win Map (RWMap). RWMaps favor remove operations in face of concurrent adds.

This convergence property is crucial. Most operations only update a subset of the inode attributes. In case of a concurrent *unlink*, if no link remains, we explicitly remove all fields. Concurrent partial updates coming from the other FS operations will be dropped.

Blocks are fixed size Last Writer Win register which can store arbitrary content. They serve as storing file content and symlink target paths. the application addresses the blocks implicatively when it receives a write or read requests.

The advantage of this representation is that we do not need to store an index of all existing blocks of a file. The drawbacks are is that if we delete an inode, we cannot explicitly delete an infinite amount of blocks, potentially leaving blocks that are never freed.

Directory Entries are represented as Remove Win set (RWSet). Simplest directory entries are pairs of an ino and a name. For the same reasons as stated above, the remove win convergence property is necessary. A set is better suited for directory entries than a map. When we handles name conflicts we don't want to ever have two entries with the same name and ino.

5 Experimentation and future explorations

5.1 Convergence and overhead.

5.2 About conditional transactions ?

6 Conclusion