On the design of AntidoteFS

Paolo Viotti

February 17, 2019

Abstract

This document is a work-in-progress blueprint for an implementation of an available, distributed POSIX-like file system [4] built around the concept of CRDT [12]. Currently, the file system is backed by Antidote [2], which exposes an API to interact with CRDTs. In each section of this document, we elaborate on a different way of emulating a file system data model using Antidote and its CRDTs library.

```
Antidote v.  0.1.0-git-HEAD
Antidote Javascript client v 0.1.7
FUSE v.  2.9.7
```

1 Directories as maps, files as registers `[implemented, abandoned]`

A simple way of emulating a file system data model with Antidote consists in using its library of CRDTs to model directories as *add-wins maps* and files as *last-writer-wins registers*. In this way, nesting directories and files inside directories results in embedding registers and maps inside maps. Thus, the hierarchical structure of the file system is reflected by the actual nesting of CRDT objects. This design has the benefit of being easy to implement and to reason about. Besides, there is no need to check for cycles, as there cannot be any, since the tree structure is enforced in the data model.

Unfortunately, a number of disadvantages and various implementation quirks have to be taken into account:

1. since the hierarchical structure is reflected in the data model, when moving files and directories, data has to be moved around across maps and registers in Antidote;
2. there is currently no support for operations on file system metadata (e.g., hard and soft linking, permission management, etc), due to lack of metadata attached to CRDTs `[TODO [5]]`;

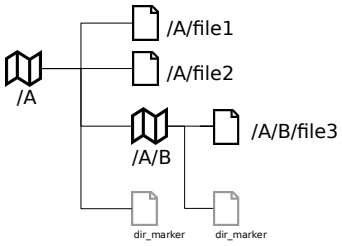

Figure 1: Antidote data structures for design option 1. Antidote's registers and maps are represented respectively with file and map icons.

3. due to the lack of support for nested objects in the Antidote API [3], traversing the file system and writing nested objects is impractical and inefficient;

4. owing to a design choice, in Antidote, object creation and update are not distinguishable. As a result of this semantics, an attempt to read a non-existing object will return an empty object. Therefore, to distinguish empty directory from not existing ones, we use *directory markers*, i.e. empty registers inside maps;

5. Antidote CRDT maps do not support partial reads or key listing. Therefore, in order to list the files embedded in a given map, one has to read the entire content of a map, and thus, all the data of the objects embedded into it. This is clearly not efficient.

Ultimately, due to its overall inflexibility and to the poor support in Antidote for nested objects and partial reads, this design has been abandoned. However, its implementation is still available for future peruse and comparison [7]. Figure 1 illustrates the design discussed in this section.

2 A map of paths, and key indirection `[implemented, abandoned]`

To overcome the drawbacks discussed in Sec. 1, in this section, we describe an alternative design that decouples the file system hierarchical structure from its content. Its corresponding implementation with Antidote CRDTs is represented in Fig. 2.

In this design scheme, a special CRDT map `PATHS` stores a series of CRDT registers referring to different paths in the file system. A register in the `PATHS`

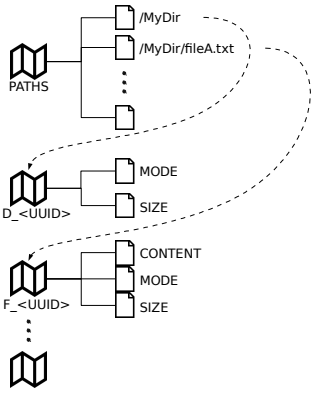

Figure 2: Antidote data structures for design option 2. Antidote's registers and maps are represented respectively with file and map icons.

map stores the key within Antidote of a map containing all data pertaining to that single path, i.e. all the data related to a given *inode* [4]. The keys of maps storing inodes data are composed by a prefix (e.g., "F_" for files and "D_" for directories) and by an identifier which guarantees their uniqueness without requiring coordination (e.g., a UUID [1]). The `PATHS` map is cached locally and refreshed periodically or upon local updates. Rename and move operations on inodes are carried out by means of transactions on the `PATHS` map. Similarly, deleting inodes entails removing their paths, while a client-based background task takes care of removing the maps storing the corresponding data.

While this design is evidently more flexible and better suited to the Antidote data model than the one described in Sec. 1, it presents some drawbacks. Namely, as described in Sec. 1 (item 5), read operations on the `PATHS` map are total and key listing is not supported. As a result, refreshing `PATHS` map might become onerous as the number of paths increases, and listing the files in a directory entails scanning the whole keyset of the `PATHS` map ($O(n)$).

3 An inode-based data model `[80% implemented]`

The designs described in the previous sections adopt entities like folders, files and paths which are familiar to file systems' users. As an alternative to this, we propose in this section a data model which reflects the data model implemented in Unix-like file systems and later formalized in the POSIX specification [4]. The POSIX standard defines an *inode* data structure representing file system objects

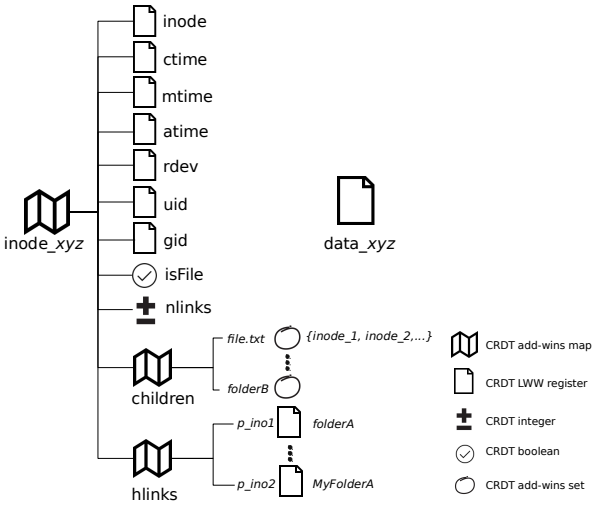

Figure 3: Antidote data structures for the data model design inspired by the inode data structure.

such as files or folders. The inode data structure, which according to POSIX is defined in <sys/stat.h>, includes a number of fields that specify ownership, permissions and other metadata of given file system object.

We emulate this basic data structure by using CRDTs as illustrated in Fig. 3. Each inode is a CRDT map named as `inode_X`, where `X` is the inode number. The inode map contains a number of LWW registers accounting for permission, ownerships and timestamp metadata (e.g., `ctime`, `atime`, `mode`, etc.), and several other CRDTs that describe the relationships of this inode with other inodes. In particular, if it is a folder inode, it includes a `children` map that contains all references to files and folders names contained in the folder in question. To each of this name is associated a CRDT add-wins set that contains the inode numbers of the inode referring to a same name that might be concurrently created at different sites of the distributed file system. Conversely, the `hlinks` CRDT map includes the inode numbers of inode folders that contain the file or folder being considered. This map and the CRDT integer counter `nlinks` are updated when performing operations like `unlink`, or when creating hard links of the inode. In this design, we store the data of a certain inode `X` in the corresponding LWW register called `data_X`.

Clearly, this design enjoys the benefits of decoupling the data model from the path hierarchy. However, this translates to an increased difficulty in checking for file system structural anomalies such as path cycles.

Conflict type	Example	AntidoteFS policy	Notes
Naming	Concurrent creation of files and folder with the same name.	Rename files, merge folders.	-
Data	Concurrent updates the same file	LWW or optional lock on file content.	[TODO]
State	Concurrent update and delete of a same inode.	Use add-win policy.	-
Mapping	Divergent concurrent move of folders.	Synchronize move operations.	[TODO]
Indirect	Path cycles	Synchronize move operations.	[TODO]

Table 1: Conflicts types and related resolution policies in AntidoteFS.

3.1 Conflict management

A crucial part of designing a distributed file system is defining how the POSIX semantics are rendered in a distributed and fault-prone environment. Indeed, a distributed file system will incur a number of anomalies due to concurrency and faults that are not codified by the POSIX standard. Therefore some conflicts may arise between the file system data and metadata seen by different distributed sites. Tao et al. [13] classify these conflicts as *direct* and *indirect* ones. Direct conflicts may pertain data, state, mapping or naming, while indirect conflicts are about the overall structure of the file system and generally can be seen as a composition of direct conflicts.

Table 1 summarizes the approach of AntidoteFS to these conflicts. We note that for most concurrent file system operations the Antidote's causal+ consistency semantics [8] is sufficient to preserve the file system invariants and thus avoiding anomalies. Najafzadeh and Shapiro [11] prove that concurrent move operations require additional coordination. A coordination mechanism is also needed to avoid editing concurrently a same file. The implementation of this design scheme [6] currently lacks such synchronization mechanism. A possible way of implementing this would entail exploiting bounded counters CRDTs [9]

or implementing from scratch a locking primitive in Antidote based on a total order broadcast implementation.

4 A tree CRDT [TODO]

A fourth design option consists in implementing in Antidote a CRDT that would enclose the tree-like file system data model and expose an API to read and update elements of the file systems as they were nodes and edges.

In essence, the data abstraction of a POSIX file system is that of a tree with some additional invariants [11]. Therefore, to implement it as a CRDT one may constrain an existing graph CRDT [12] to comply with both tree and file system invariants [10].

A formal specification of a tree CRDT is currently subject of ongoing work.

References

[1] IETF RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. `https://tools.ietf.org/html/rfc4122`, 2005.

[2] AntidoteDB. `http://syncfree.github.io/antidote/`, 2013.

[3] Embed nesting information in Keys - GitHub Issue. `https://github.com/SyncFree/antidote-java-client/issues/3`, 2015.

[4] POSIX.1-2008, IEEE 1003.1-2008, 2016 Edition, The Open Group Base Specifications Issue 7. `http://pubs.opengroup.org/onlinepubs/9699919799.2016edition/`, 2016.

[5] Mechanism for associating metadata attributes to objects - GitHub Issue. `https://github.com/SyncFree/antidote/issues/314`, 2017.

[6] AntidoteFS prototype. `https://github.com/SyncFree/antidote-fs`, 2017.

[7] AntidoteFS prototype - nesting version. `https://github.com/SyncFree/antidote-fs/tree/v0.9-nesting`, 2017.

[8] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 405–414, Nara,

Japan, June 2016. doi: 10.1109/ICDCS.2016.98. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.2016.98.

[9] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 31–36, Montréal, Canada, September 2015. IEEE Comp. Society, IEEE Comp. Society. doi: 10.1109/SRDS.2015.32. URL http://dx.doi.org/10.1109/SRDS.2015.32.

[10] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. Abstract unordered and ordered trees CRDT. Research Report RR-7825, INRIA, December 2011. URL https://hal.inria.fr/hal-00648106.

[11] Mahsa Najafzadeh and Marc Shapiro. Co-design and verification of an available file system. In *under submission*, 2017.

[12] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL http://www.springerlink.com/content/3rg3912287330370/.

[13] Vinh Tao, Marc Shapiro, and Vianney Rancurel. Merging semantics for conflict updates in geo-distributed file systems. In *ACM International Systems and Storage Conference, SYSTOR*, 2015. URL http://doi.acm.org/10.1145/2757667.2757683.

A Related work

In this section we collect a series of related work to study:

- Git tree objects: `https://git-scm.com/book/en/v2/Git-Internals-Git-Objects`

B File system, FUSE and Antidote API calls

[TODO]

CLI commands	FUSE API calls	AntidoteFS logic	Notes
`touch <file>`	-	-	-
`mkdir <dir>`	-	-	-
`echo "hello" > <file>`	-	-	-
`echo "hello" >> <file>`	-	-	-
`mv <src> <dst>`	-	-	-
`rm <file>`	-	-	-