SCISPACE formerly Typeset

Open access • Book Chapter • DOI:10.1007/978-3-319-73721-8_17

Co-Design and Verification of an Available File System — Source link

Mahsa Najafzadeh, Marc Shapiro, Patrick Eugster

Institutions: Purdue University, French Institute for Research in Computer Science and Automation

Published on: 07 Jan 2018 - Verification, Model Checking and Abstract Interpretation

Topics: File system, Distributed File System, Concurrency, POSIX and Semantics (computer science)

Related papers:

- CRDTs for truly concurrent file systems
- Syncpal: A Simple and Iterative Reconciliation Algorithm for File Synchronizers
- Operating system level support for coherence in distributed systems
- A FaaS File System for Serverless Computing.
- From Crash Consistency to Transactions

Share this paper:

View more about this paper here: https://typeset.io/papers/co-design-and-verification-of-an-available-file-system-1qjnliuw17

Co-Design and Verification of an Available File System

Mahsa Najafzadeh, Marc Shapiro, Patrick Eugster

To cite this version:

Mahsa Najafzadeh, Marc Shapiro, Patrick Eugster. Co-Design and Verification of an Available File System. VMCAI 2018 - International Conference on Verification, Model Checking, and Abstract Interpretation, Jan 2018, Los Angeles, CA, United States. pp.358-381, 10.1007/978-3-319-73721-8_17 . hal-01696263

HAL Id: hal-01696263
https://hal.inria.fr/hal-01696263

Submitted on 30 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Co-Design and Verification of an Available File System

Mahsa Najafzadeh[1], Marc Shapiro[2], and Patrick Eugster[1,3]

[1] Purdue University, West Lafayette, USA
[2] INRIA-LIP6, Paris, France
[3] Darmstadt University, Darmstadt, Germany

Abstract. Distributed file systems play a vital role in large-scale enterprise services. However, the designer of a distributed file system faces a vexing choice between strong consistency and asynchronous replication. The former supports a standard sequential model by synchronising operations, but is slow and fragile. The latter is highly available and responsive, but exposes users to concurrency anomalies. In this paper, we describe a rigorous and general approach to navigating this trade-off by leveraging static verification tools that allow to verify different file system designs. We show that common file system operations can run concurrently without synchronisation, while still retaining a semantics reasonably similar to Posix hierarchical structure. The one exception is the `move` operation, for which we prove that, unless synchronised, it will have an anomalous behaviour.

1 Introduction

The market for distributed storage is fueled by cloud computing, big data, exascale computing and so on. Classical file system designs continue to represent approximately 30% of distributed storage needs according to IDC [30], especially for entreprise applications. Modern distributed file system design can improve performance and be highly available by replicating data at several servers. A user can access a file as long as at least one replica of it is available [21, 39]. Ideally, we would like the replicated file system to provide the standard Posix semantics [2], as if a single centralised server handled all operations. However, Posix was designed under the assumption of strong consistency, which requires synchronisation in the critical path. For instance, an operation accessing some file might lock all the directories along the path to the file. This *synchronous*, strong-consistency approach is used in systems such as Frangipani, GFS, GPFS, and Lustre [19, 40, 41, 46]. Although safe, it is unlikely to perform well at large scale, and is unavailable in case of network partition [15].

Experience with real-world applications shows that concurrent updates to the same file system objects are rare [7, 26, 33, 47, 49]. Thus, the synchronous approach often causes more synchronization than the application really needs. Therefore, many recent systems, such as HDFS [43], NFS [34, 38], or PVFS

[12] eschew Posix semantics to gain better performance and scalability. A client reads or writes its local replica without synchronisation with other replicas, and immediately returns to the client, while any updates are transmitted to other replicas in the background. We call this approach *asynchronous replication*.

Unfortunately, asynchronous replication faces the challenge of replica divergence, and may violate some application desirable properties, called *integrity invariants* [35, 48]. Consider this simple example: Alice in Anchorage adds file f to directory d to her local replica, while at the same time, Bob in Brussels removes directory d. The outcome may well result in f existing but being unreachable. Such anomalies are undesirable, and this poses a major challenge to the design of a replicated file system [27].

The correctness of applications implemented on top of a given file system is highly dependent on subtle behaviors of the underlying file system. Thus, programmers require to reason about the file system behaviour, taking into account which anomalies are disallowed by a given file system semantics and whether disallowing these anomalies is enough to ensure correctness.

We address this problem by considering the integrity invariants that both sequential and replicated file systems must satisfy. Our goal is to come up with a design that is *as asynchronous as possible while satisfying the invariants*. We are helped in this road by a static analysis tool based on the CISE logic, which was proved sound for replicated data under the causal consistency assumption [20]. CISE is a variant of rely-guarantee reasoning [25]. A successful analysis proves that any execution of operations over replicated data, under a given synchronisation protocol, maintains the given invariants.

We apply this analysis to file system design. For simplicity, we focus only on a single naming tree and ignore hard links, devices, mounts, file attributes, and the like; we are reasonably confident that our analysis extends readily to a more detailed file system model. The targeted invariant is that the directory structure forms a *tree*. Our model covers the Posix commands [2] affecting the tree structure, including creating, removing, moving, and changing directory entries, as well modifying files.

We first formulate a sequential specification of Posix file system, and prove that this semantics maintains the tree invariant. Next, we extend the sequential semantics to support concurrent users; we study two different concurrent semantics, each exposing a different amount of parallelism, and different anomalies.

The first one, called *Fully Asynchronous*, optimistically accepts all concurrent updates, and resolves conflicts by weakening the sequential Posix semantics. This achieves better availability and latency, which are essential for large-scale applications. Applying the CISE analysis verifies that most operations of a replicated file system can execute without synchronisation. The only exception is that concurrent `move` operations may violate the tree invariant, resulting in a disconnected cyclic component. To fix this issue, our design follows the geoFS system [45]: if a cycle would occur, it effectively replaces `move` with a copy-delete, which preserves the tree invariant but might duplicate the directories that would otherwise end up in a cycle.

If replacing move with copy-delete is undesirable, an alternative safe solution is to use synchronisation in order to disallow the concurrent execution of move operations that would violate the tree invariant. However, several concurrency control algorithms are possible; it is not obvious which is best. Synchronising too much hurts performance and availability, but synchronising too little would result in violating the tree invariant. Using the CISE analysis, we identify the *minimal* synchronization sufficient for move operations. Accordingly, we design our *Mostly-Asynchronous file system*, in which the common operations run in asynchronous mode, and only some move-directory operations might be blocked by synchronisation.

In summary, this paper makes the following contributions:

- We provide a rigorous specification for Posix-like file system for both centralized/synchronous and replicated/asynchronous semantics.
- Using the CISE analysis, we describe a rigorous and general approach that helps developers to encode and verify a variety of concurrent file system designs.
- We study and verify two different replicated file system semantics, each exposing a different amount of parallelism, and different anomalies.
- We identify and verify synchronization is necessary and sufficient for a replicated file system to maintain the tree invariant.

The remainder of the paper is organized as follows: Section 2 introduces the file system objects and models, and gives an overview of the CISE logic. Section 3 presents and verifies a sequential specification of the file system. Section 4 does the same with a concurrent specification of the file system. Sections 5 and 6 discuss our proposed Fully-Asynchronous and Mostly-Asynchronous file system semantics respectively. Related work is discussed in Section 8. Finally, Section 9 summarizes our results and concludes the paper.

2 Model

2.1 File System Objects

The abstract state of a file system consists of a naming tree of *directories*. A directory $d \in \mathsf{Dir}$ maps a set of locally-unique names $n \in \mathsf{Name}$, to a set of *inodes* $\in \mathsf{INode}$. An inode represents a file system object, which is either a directory or a file:

$$\mathsf{Dir} : \mathsf{Name} \rightarrow \mathsf{INode}$$
$$\mathsf{INode} : \mathsf{File} | \mathsf{Dir}$$

This hierarchical file system structure has a single fixed *root* directory. Each inode is identified by a *path*. The path is a sequence of directory names, and possibly a final file name, separated by a separator or delimiter. Following the Unix convention, we use the "/" character as the separator.

We use Greek letters to denote paths.

Definition 1 (Parent Relation). *Directory* $u \in \mathsf{Dir}$ *is the direct parent of inode* $v \in \mathsf{INode}$*, denoted by* $u \downarrow v$*, if and only if* u *contains a mapping to* v*, i.e., there is a name* $n \in \mathsf{Name}$*, such that* $(n, v) \in u$*.*

Definition 2 (Path Prefix). *Path* π *is called a* prefix *of path* γ*, noted* $\pi \sqsubseteq \gamma$*, if and only if* $\gamma = \pi/\alpha$ *for some path* α*.*

The *transitive* closure of the parent relation defines the *ancestor relation*. We say directory $u \in \mathsf{Dir}$ is an ancestor of inode $v \in \mathsf{INode}$, noted $u \downarrow^* v$, if and only if:

$$u \downarrow^* v = \begin{cases} true & if (u \downarrow v) \\ \exists w \in \mathsf{Dir}, (u \downarrow w) \wedge (w \downarrow^* v) & otherwise \end{cases}$$

Definition 3 (Least Common Ancestor). *The Least Common Ancestor of nodes* u *and* v*, noted* $LCA(u, v)$*, is the ancestor of both* u *and* v *that is the farthest from the root. If* u *and* v *are inodes from the same file system then* $LCA(u, v)$ *exists and is unique.*

2.2 Operation Execution Model

We assume a replicated model where each replica stores a full copy of the file system. We use a Read-One-Write-All approach [9], under the operation execution model proposed by Gotsman et al. [20]. A client interacts with the file system through a set of operations Op; it submits an operation to an arbitrary single replica, called the *origin* replica for that operation. The operation is divided into two phases: *generator* and *effector*. The generator executes at the origin replica. This returns a value Val to the client and computes the *effector* of the operation, a function encoding the update to be done by the operation. Every replica eventually applies the effector to its own state.

More precisely, the semantics of operations is defined by a function

$$\mathcal{F} \in \mathsf{Op} \rightarrow (\mathsf{State} \rightarrow (\mathsf{Val} \times (\mathsf{State} \rightarrow \mathsf{State}) \times \mathsf{set}(\mathsf{Token}))).$$

The function is formulated as follows for some operation o:

$$\begin{gathered} \forall \sigma \in \mathsf{State}, o \in \mathsf{Op},\ \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\mathsf{val}}(\sigma), \mathcal{F}_o^{\mathit{eff}}(\sigma), \mathcal{F}_o^{\mathsf{tok}}(\sigma)), \\ \mathcal{F}_o^{\mathsf{val}}(\sigma) \in \mathsf{Val} \\ \mathcal{F}_o^{\mathit{eff}}(\sigma) \in \mathsf{State} \rightarrow \mathsf{State} \\ \mathcal{F}_o^{\mathsf{tok}}(\sigma) \in \mathsf{set}(\mathsf{Token}) \end{gathered}$$

Given state $\sigma \in \mathsf{State}$ at the origin replica in which an operation $o \in \mathsf{Op}$ executes,

- The $\mathcal{F}_o()$ function is the *generator*.
- $\mathcal{F}_o^{\mathsf{val}}(\sigma)$ is the *return value* of operation o. We use $\perp$ for operations that return no value.
- $\mathcal{F}_o^{\mathit{eff}}(\sigma)()$ is the *effector* function of operation o. It will be sent to every replica; when received by a replica, the replica applies this transformation to its own state.
- $\mathcal{F}_o^{\mathsf{tok}}(\sigma)$ is the set of concurrency-control *tokens* acquired by operation o. Tokens are described in Section 2.3.

$$\sigma = \mathsf{map}(\mathsf{Name} \mapsto \mathsf{INode}) \qquad n \in \mathsf{Name} \qquad u \in \mathsf{INode}$$

$$\mathcal{F}_{\mathsf{add}(n)}(\sigma) \ : \ \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{add}(n)}(\sigma) = \bot, \mathcal{F}^{\mathsf{tok}}_{\mathsf{add}(n)}(\sigma) = \emptyset, u = \mathsf{inode}() \\ \mathcal{F}^{\mathit{eff}}_{\mathsf{add}(n,u)}(\sigma) = \lambda(\sigma').\, \sigma'[n \mapsto u] \end{cases}$$

$$\mathcal{F}_{\mathsf{remove}(n)}(\sigma) \ : \ \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{remove}(n)}(\sigma) = \bot, \mathcal{F}^{\mathsf{tok}}_{\mathsf{remove}(n)}(\sigma) = \emptyset \\ \mathcal{F}^{\mathit{eff}}_{\mathsf{remove}(n)}(\sigma) = \lambda(\sigma').\, \sigma'[n \mapsto \emptyset] \end{cases}$$

$$\mathcal{F}_{\mathsf{query}(n)}(\sigma) \ : \ \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{query}(n)}(\sigma) = u \mid \sigma[n \mapsto u] \\ \mathcal{F}^{\mathsf{tok}}_{\mathsf{query}(n)}(\sigma) = \emptyset \\ \mathcal{F}^{\mathit{eff}}_{\mathsf{query}(n)}(\sigma) = \mathsf{skip} \end{cases}$$

Precondition	Operation
$\nexists e \in \mathsf{INode}, (n,e) \in \sigma$	$\mathcal{F}_{\mathsf{add}(n)}(\sigma)$
$\exists e \in \mathsf{INode}, (n,e) \in \sigma$	$\mathcal{F}_{\mathsf{remove}(n)}(\sigma)$
$\exists e \in \mathsf{INode}, (n,e) \in \sigma$	$\mathcal{F}_{\mathsf{query}(n)}(\sigma)$

Fig. 1. A sequential specification of directory

2.3 Concurrency

A replica is a process that executes a sequence of generator and effector events. An operation o is *visible* to operation o' if some replica executes the effector of o before the generator of o'. We assume *causal consistency*, i.e., the visibility relation is transitive. Two operations that are not related by visibility are said *concurrent*.

The tokens mentioned in the previous section are an abstraction of concurrency control mechanisms. Tokens are related by a symmetric *conflict* relation ($\bowtie$). If two operations acquire tokens conflicting according to $\bowtie$, then one must be visible to the other. If their tokens do not conflict by $\bowtie$, the operations are allowed to be concurrent.

2.4 Example

Consider a directory d in the file system. Figure 1 illustrates a simple implementation of d as follows: Let σ denote the state of d at the origin replica. State σ is a map of names to inodes, representing the content of the directory. The directory semantics supports operations to add, remove, and query mappings in the directory.

We start with a sequential specification for the directory, shown in Fig. 1. To add an inode to d, the $\mathsf{add}(n)$ operation's generator in the origin replica, computes return value, tokens, and creates a new inode to prepare its effector if σ satisfies the operation's precondition. The precondition of $\mathsf{add}(n)$ is that no inode under the same name n exists in the directory d. We assume a function $u = \mathsf{inode}()$ to create a new and unique inode identifier u. The effector for $\mathsf{add}(n)$ takes name n and inode u as its arguments, reads state of d at the current replica, denoted σ' (which can be different from σ, due to concurrency), and then adds the inode u under name n to it, denoted by $\sigma'[n \mapsto u]$. Similarly, to remove

an inode named n from directory d, the $\mathsf{remove}(n)$ operation's effector, reads the state of directory at each replica, denoted by σ', and removes the mapping for name n, noted by $\sigma'[n \mapsto \emptyset]$. The $\mathsf{query}(n)$ operation computes the inode u mapped to the name n at the origin replica, and simply returns u, i.e., $\sigma[n \mapsto u]$; its effect is simply skip where $\mathsf{skip} = (\lambda\sigma'.\sigma')$.

2.5 CISE Logic and Analysis

Any replicated system needs to ensure convergence, i.e., executing the same operations produces the same results at different replicas. Furthermore, a given system must maintain a specific integrity *invariant*, i.e., a safety condition over replica states. The invariant of interest for file systems is that its directories form a tree (the *tree invariant*). In this work, we do not consider liveness properties.

To this effect, we leverage CISE [20], a sound logic that allows to verify such safety conditions statically (in polynomial time) for a replicated system with causal consistency. We use the CISE verification tool [31], which takes as input the specification of the system's operations (their preconditions and effects) and of the targeted invariant. The tool checks the following proof rules: *(i)* Individual Correctness: each operation individually maintains the invariant. *(ii)* Commutativity: any two concurrent operations commute (operations that have conflicting tokens need not commute). *(iii)* Stability: every operation's precondition is stable under concurrent updates (but not necessarily against an operation that acquires a conflicting token). A successful analysis proves that *any* execution of the given operations, under the given tokens, maintains the given invariant.

Unsuccessful analysis returns a counter-example, which indicates the problem to the developer. The developer can fix the problem, either by weakening the application semantics, or by strengthening the token system, and then run the check again. This process constitutes *co-design* of the application semantics and of its concurrency control.

3 Sequential Specification

In this section, we first formulate a sequential specification of a Posix-like file system, and prove that this semantics preserves the tree invariant.

3.1 Tree Invariant

An directory is a map of names to inodes (files or directories). We model the directory structure as a graph, where a directory is a node, and there is an edge from this *parent* directory to each directory that it names (its *children*). The tree main invariant I of the file system is the conjunction of the following assertions: (1) The file system has a fixed root node. (2) The root is an ancestor of every other node in the tree. (3) Every node has exactly one parent, except the root, which has none. (Since we ignore symbolic links, a file cannot have several parent

$$\sigma = \mathsf{set}(\mathsf{INode}) \quad \sigma_{init} = \{\mathsf{r}oot\} \quad n \in \mathsf{Name} \quad c \in \mathsf{Content}$$

$$d, d_p, d_{p_{new}}, d_{p_{old}} \in Dir \quad f \in \mathsf{File}$$

$$\mathsf{mkdir}(\mathsf{path}) : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), d = \mathsf{inode}() \\ \mathcal{F}^{eff}_{\mathsf{mkdir}(d_p,n,d)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto d] \end{cases}$$

$$\mathsf{rmdir}(\mathsf{path}) : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), \\ \mathcal{F}^{eff}_{\mathsf{rmdir}(d_p,n)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto \emptyset] \end{cases}$$

$$\mathsf{mkfile}(\mathsf{path}) : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), f = \mathsf{inode}() \\ \mathcal{F}^{eff}_{\mathsf{mkfile}(d_p,n,f)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto f] \end{cases}$$

$$\mathsf{rmfile}(\mathsf{path}) : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), \\ \mathcal{F}^{eff}_{\mathsf{rmfile}(d_p,n)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto \emptyset] \end{cases}$$

$$\mathsf{write}(\mathsf{path}, \mathsf{c}) : \begin{cases} f = \mathcal{L}(\mathsf{path}), \\ \mathcal{F}^{eff}_{\mathsf{write}(f,c)}(\sigma) = \lambda(\sigma').\, \sigma'.f.\mathsf{write}(c) \end{cases}$$

$$\mathsf{mvdir}(\mathsf{old}, \mathsf{new}) : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi), d = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma), \\ \mathcal{F}^{eff}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}(\sigma) = \lambda(\sigma').\, (\sigma'.d_{p_{old}}[n \mapsto \emptyset]; \sigma'.d_{p_{new}}[n \mapsto d]) \end{cases}$$

$$\mathsf{mvfile}(\mathsf{old}, \mathsf{new}) : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi), f = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma), \\ \mathcal{F}^{eff}_{\mathsf{mvfile}(d_{p_{old}},n,d_{p_{new}},f)}(\sigma) = \lambda(\sigma').\, (\sigma'.d_{p_{old}}[n \mapsto \emptyset]; \sigma'.d_{p_{new}}[n \mapsto f]) \end{cases}$$

Precondition	Operation
$\nexists e \in \mathsf{INode}, (n,e) \in d_p \wedge root \downarrow^* d_p$	$\mathcal{F}_{\mathsf{mkdir}(d_p,n,d)}(\sigma)$
$\nexists e \in \mathsf{INode}, m \in \mathsf{Name}, (m,e) \in d \wedge root \downarrow^* d$	$\mathcal{F}_{\mathsf{rmdir}(d_p,n,d)}(\sigma)$
$\nexists e \in \mathsf{INode}, (n,e) \in d_p \wedge root \downarrow^* d_p$	$\mathcal{F}_{\mathsf{mkfile}(d_p,n,f)}(\sigma)$
$root \downarrow^* f$	$\mathcal{F}_{\mathsf{rmfile}(d_p,n,f)}(\sigma)$
$root \downarrow^* f$	$\mathcal{F}_{\mathsf{write}(f,c)}(\sigma)$
$\nexists e \in \mathsf{INode}, (n,e) \in d_{p_{new}} \wedge root \downarrow^* d \wedge d_{p_{old}} \downarrow d$ $\wedge root \downarrow^* d_{p_{new}} \wedge d \not\downarrow^* d_{p_{new}}$	$\mathcal{F}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}(\sigma)$
$\nexists e \in \mathsf{INode}, (n,e) \in d_{p_{new}} \wedge root \downarrow^* f$ $\wedge d_{p_{old}} \downarrow f \wedge root \downarrow^* d_{p_{new}}$	$\mathcal{F}_{\mathsf{mvfile}(d_{p_{old}},n,d_{p_{new}},f)}(\sigma)$

Fig. 2. A sequential hierarchy file system design

directories.) (4) The directory graph is acyclic. (5) The names in a directory are unique with respect to that directory. Formally:

$$\begin{aligned} I = {} & \forall e_1, e_2 \in \mathsf{INode}, d_1, d_2 \in \mathsf{Dir}, n_1, n_2 \in \mathsf{Name}, \pi, \pi' \in \mathsf{Path} \\ & (1) \quad root \in \mathsf{INode} \\ & (2) \wedge e_1 \neq root \implies root \downarrow^* e_1 \\ & (3) \wedge (d_1 \downarrow e_1 \wedge d_2 \downarrow e_1 \implies d_1 = d_2 \wedge e_1 \neq root) \\ & (4) \wedge (d_1 \downarrow^* d_2 \wedge d_2 \downarrow^* d_1 \implies d_1 = d_2) \\ & (5) \wedge (\pi/n_1 \mapsto e_1 \wedge \pi/n_2 \mapsto e_2 \wedge e_1 \neq e_2 \implies n_1 \neq n_2) \end{aligned}$$

3.2 File System Operations

The file system semantics that we study in this paper consists of a set of operations, which abstract the Posix commands to manipulate the tree structure and

to update file content. They include *creating*, *deleting*, and *renaming* directories or files, and modifying files. We identify a file or a directory by a path. Given the path argument of an operation, a *resolution* function $\mathcal{L}$ is executed at the operation's origin replica to find the inode located in the path,

$$\mathcal{L} : \mathsf{Path} \rightarrow \mathsf{INode}.$$

Figure 2 shows a sequential specification of the file system. We denote σ the state of the file system, and the dot notation, for instance $\sigma.e$, to refer to a particular inode e in it.

We assume that update operations return no value, i.e., $\mathcal{F}_o^{\mathsf{val}}(\sigma) = \bot$, and an exclusive token is assigned to each operation that forbids the concurrent execution of these operations. For complex operations like move operation, whose effect updates two directories, indicated by old and new, we use semicolon to denote a sequence of changes over the file system state. We assume that inodes have unique identifiers across replicas. The arguments to the effector of some operation includes the inode determined by the generator, rather than its path; this is unnecessary in the sequential specification but will prove useful when we consider concurrent updates.

For instance, consider that Alice wants to create a new directory using the operation $\mathsf{mkdir}(/share/album/paris)$. In Alice's replica of the file system, this resolves to creating name to name *paris*, within the directory whose path is $/share/album$, which evaluates to inode d_p (subscript p stands for "parent"). A minimal precondition is that d_p exists, and it does not contain name *paris*. If satisfied, effector $\mathcal{F}^{eff}_{\mathsf{mkdir}(d_p, "paris", d)}$ is generated and sent to all replicas. On delivering the effector, every replica (including Alice's) applies its effect, which is to create the new directory d, and to update the parent directory d_p by mapping the name "*paris*" to directory d, denoted by $d_p[paris \mapsto d]$. Note, we use the notation $f.\mathsf{write}(c)$ for writing the content c into a file f.

We apply the CISE effector safety to check if the file system operations preserve the tree invariant in isolation, with the minimal preconditions of the previous paragraph (since no operations are concurrent, the other two rules are void). The CISE tool returns an error for mvdir. The associated counter-example shows that the source directory must not be an ancestor of the destination directory, because otherwise, a cycle occurs. The developer strengthens the pre-condition to avoid this; with this stronger specification, the tool indicates that mvdir is now safe. We proceed similarly for the other operations, thus co-designing the sequential specification. Finally, we reach the specification illustrated in Fig. 2, which is proved safe.

4 Concurrent Specification

In this section, we extend the sequential semantics to support concurrent users. We present a concurrent specification of the file system that *optimistically* accepts all concurrent updates, and resolves conflicts but it trades the sequential

$$\sigma = (d \times \rho) \qquad d = \mathsf{map}(\mathsf{Name} \mapsto \mathsf{INode}) \qquad \rho = \mathsf{map}(\mathsf{INode} \mapsto \mathsf{INode})$$

$$n \in \mathsf{Name} \qquad u, v, w, e_1, \ldots, e_k \in \mathsf{Dir}$$

$$\mathcal{F}_{\mathsf{add}(n)}(\sigma) \;:\; \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{add}(n)}(\sigma) = \bot, \qquad \mathcal{F}^{\mathsf{tok}}_{\mathsf{add}(n)}(\sigma) = \emptyset, u = \mathsf{inode}() \\ \mathcal{F}^{\mathit{eff}^*}_{\mathsf{add}(n,u)}(\sigma) = \lambda(\sigma').\, (\mathsf{IF}(\sigma'.d[n \mapsto v] \wedge w == v \bigoplus u) \\ \qquad\qquad (\sigma'.d[n \mapsto w], \sigma'.\rho[u \mapsto w, v \mapsto w])(\sigma'.d[n \mapsto u], \sigma'.\rho) \end{cases}$$

$$\mathcal{F}_{\mathsf{remove}(n)}(\sigma) \;:\; \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{remove}(n)}(\sigma) = \bot, \qquad \mathcal{F}^{\mathsf{tok}}_{\mathsf{remove}(n)}(\sigma) = \emptyset, \\ \mathcal{F}^{\mathit{eff}^*}_{\mathsf{remove}(n,u)}(\sigma) = \lambda(\sigma').\, (\mathsf{IF}(\sigma'.d[n \mapsto w] \wedge \sigma'.\rho[u \mapsto w]) \\ \qquad\qquad (\sigma'.d, \sigma'.\rho[u \mapsto \emptyset])(\sigma'.d[n \mapsto \emptyset], \sigma'.\rho) \end{cases}$$

$$\mathcal{F}_{\mathsf{query}(n)}(\sigma) \;:\; \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{query}(n)}(\sigma) = w \mid d[n \mapsto w] \wedge (\rho[e_1 \mapsto w, e_2 \mapsto w, \ldots, e_k \mapsto w] \\ \qquad\qquad \implies w = e_1 \bigoplus e_2 \ldots \bigoplus e_k) \\ \mathcal{F}^{\mathsf{tok}}_{\mathsf{query}(n)}(\sigma) = \emptyset, \qquad \mathcal{F}^{\mathit{eff}^*}_{\mathsf{query}(n)}(\sigma) = \mathsf{skip} \end{cases}$$

Fig. 3. A concurrent specification of directory d

Posix semantics for availability. The concurrent design exploits Conflict-Free Replicated Data Types (CRDT) [42] to address conflicts. CRDTs include many useful data types, such as counters, sets, graphs, and maps, which encapsulate conflict resolution policies for automatically merging the effects of operations performed on each object concurrently.

4.1 File System Objects as CRDTs

We use the idea of CRDTs to carefully design conflict-free replicated file system objects, which ensures convergent outcomes reflecting effects of all operations performed to each file or directory at different replicas. To this goal, we first discuss conflict cases that may occur as a result of concurrent execution of file system operations, and then propose the concurrent file system semantics, which converges by design.

4.2 Name Conflict

Users may perform concurrent updates to a directory. Concurrently adding or moving inodes under the same name in the same directory is problematic because it violates the name uniqueness property in the tree invariant (*name conflict*).

To address such conflicts, we define a merge operator $\bigoplus$. The merge operator $\bigoplus$ may have different merge semantics depending on directories or files, or may be different for different files. We choose the following merge semantics: concurrently adding or moving two directories under the same name to the same parent directory merges these directories, i.e., takes their union. For files with the same name, the merge semantics renames files by appending a replica-specific suffix to a locally-unique file name. We assume that type of each file system object is embedded in its name, and hence name conflicts between files and directories cannot occur.

Definition 4 (Union Merge Function). *Let u and v be two different directories with the same name n under parent directory d_p. We define the union merge of u and v as follows:*

$$w = u \bigoplus v \mid d_p[n \mapsto w] \ \wedge (\forall e \in \mathsf{INode}, u[- \mapsto e] \vee v[- \mapsto e] \implies w[- \mapsto e])$$

Where w is a new directory whose content is union of contents of directories u and v. The merge function is recursively applied to sub-directories if there are naming conflicts.

A concurrent effector may still use the old directories u and v. To solve this problem, each replica's state has map ρ, which keeps a record of the equivalence relation between the directories and their merged directory e.g., $\rho[u \mapsto w, v \mapsto w]$ where $w = u \bigoplus v$. Thus, when a replica receives an effector updating a directory, the replica first queries the equivalence relation ρ to check if the directory has been merged. Unused identifiers can be garbage-collected and removed from ρ. For brevity, we do not attempt to formalise this property.

Definition 5 (Rename Merge Function).

Let u and v be two files with the same name n under the same parent directory d_p, which originally are generated in replica r_1 and r_2, respectively. A merge decision to solve the file name conflicts would be to change the names by adding the replica's suffixes to the original name. Thus, we define the rename merge function of files u and v as follows:

$$u \bigoplus v : d_p[n_1 \mapsto u, n_2 \mapsto v]$$

Where $n_1 = n + r_1$ and $n_2 = n + r_2$ are new unique names mapped to u, and v respectively.

The merge operator must satisfy the following properties:

- $u \bigoplus u = u \quad (idempotent)$
- $u \bigoplus v = v \bigoplus u \quad (commutative)$
- $u \bigoplus (v \bigoplus w) = (u \bigoplus v) \bigoplus w \quad (assosiative)$

where u, v, and w are file system objects.

Figure 3 illustrates the concurrent implementation of directory d using the merge operator. $\mathcal{F}_o^{eff^*}$ is the effector of operation o that integrates the merge policy for managing name conflicts. The directory's state σ consists of two maps: d that is a map of names to inodes, representing the directory's content, and ρ that is a map of inodes to inodes, tracking the equivalence relation of merged sub-directories in d. The $\mathsf{add}(n)$ operation creates a new inode u; its effector reads the directory's state at each replica, denoted by σ', and if there is no name conflict, it simply adds the pair (n, u) to the directory's content, denoted by $\sigma'.d[n \mapsto u]$. For simplicity, we assume the name n refers to a directory. In the case where a directory v is concurrently added under the same name n,

$$\sigma = \mathsf{set(INode)} \quad \sigma_{init} = \{root\} \quad \mathsf{Token} = \emptyset \quad \bowtie = \emptyset$$

$$\mathsf{mkdir(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), d = \mathsf{inode()} \\ \mathcal{F}^{eff}_{\mathsf{mkdir}(d_p,n,d)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(\sigma'.d_p); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{rmdir(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), d = \mathcal{L}(\mathsf{path}) \\ \mathcal{F}^{eff}_{\mathsf{rmdir}(d_p,n,d)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{mkfile(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), f = \mathsf{inode()} \\ \mathcal{F}^{eff}_{\mathsf{mkfile}(d_p,n,f)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{add}(n,f)}(\sigma'.d_p); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{rmfile(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), f = \mathcal{L}(\mathsf{path}) \\ \mathcal{F}^{eff}_{\mathsf{rmfile}(d_p,n,f)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{write(path, c)} : \begin{cases} f = \mathcal{L}(\mathsf{path}) \\ \mathcal{F}^{eff}_{\mathsf{write}(f,c)}(\sigma) = \lambda(\sigma').\,\sigma'.f.\mathsf{write}(c); \mathsf{recurAdd}(\sigma'.f) \end{cases}$$

$$\mathsf{mvdir(old, new)} : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi) \wedge d = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma) \\ \mathcal{F}^{eff}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_{p_{old}}); \\ \qquad \mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(\sigma'.d_{p_{new}}); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{mvfile}(old, \mathsf{new}) : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi) \wedge f = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma) \\ \mathcal{F}^{eff}_{\mathsf{mvfile}(d_{p_{old}},n,d_{p_{new}},f)}(\sigma) = \lambda(\sigma').\,((\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_{p_{old}}); \\ \qquad \mathcal{F}^{eff^*}_{\mathsf{add}(n,f)}(\sigma'.d_{p_{new}}); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

Fig. 4. A concurrent specification of the file system (fully asynchronous)

in such a way that $w = u \bigoplus v$, the effector adds the pair (n, w) to the directory's content, and updates the directory's equivalence relation ρ by adding the corresponding equivalence into, denoted by $\sigma'.\rho[u \mapsto w, v \mapsto w]$. Similarly, the $\mathsf{remove}(n)$ operation's effector simply removes the mapping for name n from the directory's content at each replica, denoted by $\sigma'.d[n \mapsto \emptyset]$. If u is a directory which is merged into new directory w due to name conflicts, i.e., $\sigma'.d[n \mapsto w]$ and $\sigma'.\rho[u \mapsto w]$, the effector removes equivalence mapping for u from the relation ρ at each replica, i.e., $\sigma'.\rho[u \mapsto \emptyset]$. The $\mathsf{query}(n)$ operation computes the directory mapped to the name n by first reading the relation ρ at the origin replica and computing all directories $e_{i_{1 \le i \le k}}$ concurrently added to the directory's content d under the name n, and returns w, where $w = e_1 \bigoplus e_2 \bigoplus \ldots \bigoplus e_k$.

4.3 Remove/Update Conflict

A different kind of conflict happens when a replica updates an inode, while another replica concurrently removes the inode. This kind of conflict is called a *remove/update conflict*. For instance, when a replica receives an operation to add directory u to directory v, if directory v has been removed by a concurrent user, the operation execution results in an unreachable directory u.

The replicated data types support two main approaches, called *add-wins* and *remove-wins*, to address this problem. They differ by the result of concurrent

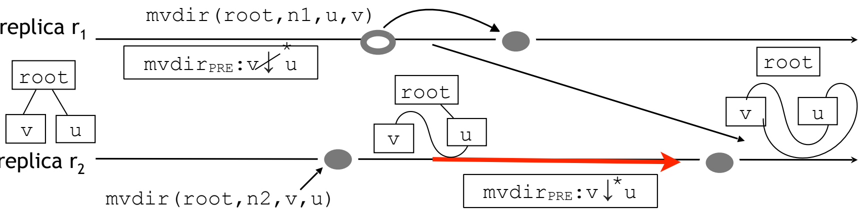

Fig. 5. Counter-example for violation of tree invariant due to of concurrent moves

add and remove of the same elements. In the add-wins semantics, when there are concurrent add or remove of the same element, add wins and the effects of concurrent remove operations are ignored. Remove-wins follows the opposite semantics. When an element is removed, any concurrent updates of the same element are lost.

Given the directory semantics in Fig. 3, concurrent adding inodes into the same directory d commute since each inode is unique. Concurrent removes commute because removing different inodes has independent effects, and removing the same inode is identical. Moreover, concurrent adding and removing inodes to the same directory d commute, i.e., the add wins because the unique inode created by add operation cannot be observed by remove operation.

Figure 4 illustrates a concurrent specification of the file system. The concurrent semantics is token-free, and relies on the effectors $\mathcal{F}^{eff^*}_{\mathsf{add}}$, and $\mathcal{F}^{eff^*}_{\mathsf{remove}}$ presented in Fig. 3 to handle name conflicts occurring within a directory. Following the add-wins semantics, function $\mathsf{recurAdd}(d)$ recursively re-creates the removed directories, which have been concurrently updated. The function takes an inode, e.g., directory d, as input, and then checks whether the directory is reachable by the root, if not, the *full* directory's path from the root is re-created as follows:

$$\mathsf{recurAdd}(d) = \mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(d.\mathsf{Parent}) + \mathsf{recurAdd}(d.\mathsf{Parent})$$

where $d.\mathsf{Parent}$ is parent of d. The function $\mathsf{recurAdd}(d)$ uses the effector $\mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(d.\mathsf{Parent})$ to re-add a removed directory d into its parent directory.

For instance, consider the concurrent semantics of $\mathsf{mkdir}(\mathsf{path})$ operation for creating a new empty directory identified by the path argument. In the origin replica, this operation resolves to creating name to name n, within the directory whose path is π, which evaluates to inode d_p. Its sequential semantics is to create the new directory d, and to update the parent directory d_p by mapping the name n to directory d. However, concurrent conflicts may arise: 1) other directories concurrently added or moved into parent directory d_p under the same name n, 2) the directory d_p has been removed concurrently. To address name conflicts when applying mkdir effector on directory d_p in any replica state σ', the concurrent semantics uses the name conflict resolutions given by $\mathcal{F}^{eff^*}_{\mathsf{add}(n,f)}(\sigma'.d_p)$.

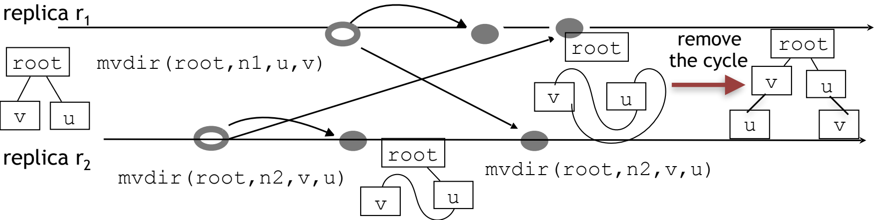

Fig. 6. Asynchronous solution design for conflicting move operations

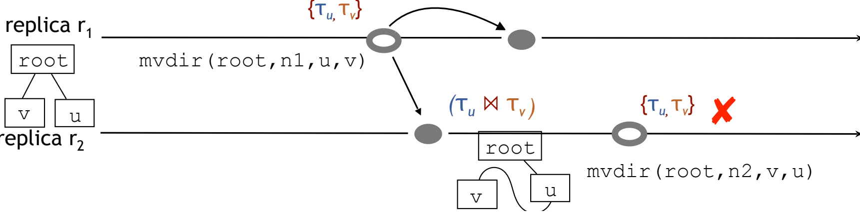

Fig. 7. Avoiding concurrent execution of conflicting moves

The concurrent semantics also uses function $\mathsf{recurAdd}(\sigma'.d_p)$, which re-creates directory d_p if it has been removed concurrently from the replica state σ'.

We remark that removing a directory does not actually delete the directory, as it only removes the mapping for the directory from its parent, as the directory exists but is unreachable. Re-creating a directory adds it into its parent directory again. Therefore, if multiple replicas concurrently re-create directory d, they all will end up with the same state.

We use the CISE-enabled analysis to verify the concurrent design. The commutativity analysis verifies that the concurrent operations results in a convergent state because all possible pairs of concurrent operations commute. The application of the CISE stability analysis for the concurrent file system design verifies that most operations of the file system can execute without synchronisation, and only concurrent move operations may violate the tree invariant. The precondition of move directory operation is not *stable* when there is another concurrent move operation. Figure 5 illustrates a counter-example: consider a file system with three directories, $root$, u and v, replicated at two replicas. Initially, the $root$ is parent of u and v. One replica asks to move directory u named n under directory v using the move operation $\mathsf{mvdir}(root, n, u, v)$. The precondition of this move operation is true, i.e., the directory u is not an ancestor of directory v. However, concurrently, another replica moves directory v under directory u, and hence, the precondition of move is not true any more. If indeed we were to continue and apply the effect of the first move operation, we come to the state, with a cycle of u and v, disconnected from the $root$. Obviously, it is not a tree.

Table 1. Required tokens for the mostly-asynchronous file system

Effector	Tokens
$\mathsf{mvdir}(d_{p_{old}}, n, d_{p_{new}}, d)$	$\{\ \tau_d, \tau_{d_{p_{new}}}\ \} \cup$ $\{\tau_e \mid e \in \mathsf{Dir} \wedge e \downarrow^* d_{p_{new}} \wedge$ $LCA(d, d_{p_{new}}) \downarrow^* e\}$
$\mathsf{mvfile}(d_{p_{old}}, n, d_{p_{new}}, f)$	τ_f

5 Fully-Asynchronous File System

If high performance and availability of update operations are important to the file system application, a simple approach to fix move conflicts is to allow move operations to execute without restriction, and to repair the tree invariant violations after the fact. Thus, we design a fully-asynchronous file system that accepts all concurrent operations, and if cycles occur due to concurrently moving two directories u, and v, it has a merge function, which will duplicate all directories in the cycle. For instance, in Fig. 6, the cycle between directories u and v is removed by making copies of the directories u and v. This merge semantics is used in real file systems, such as geoFS [45].

We use the CISE analysis to verify the merge semantics for move operations. The analyser proves that the merge function is commutative, and also guarantees the tree invariants.

6 Mostly-Asynchronous File System

One alternative approach for handling move conflicts is to add synchronisation in order to avoid concurrent execution of move operations that would violate the tree invariant. Thus, we co-design the file system semantics, in which the common operations run in asynchronous mode, and only some move directory operations need to be synchronised.

A developer may define a mutually exclusive token for each inode $e \in \mathsf{INode}$: τ_e, such that $\tau_e \bowtie \tau_e$. To ensure that cycles do not happen, we assign a set of tokens to each move operation. For any pair of move operations, if their tokens are conflicting, only one of them can take effect, because token semantics requires that the operations exchange messages, which ensures that one of the operations is aware of the other. However, other move operations are causally independent, and hence can proceed in parallel.

Using the CISE analysis, we identify and verify the necessary and sufficient token assignments for move directory operations ensuring the tree invariant in any possible executions.

Lemma 1. *Assume a* move *directory operation moving directory d into its destination directory $d_{p_{new}}$. Let set A be the set of ancestors of the destination directory $d_{p_{new}}$ up to $LCA(d, d_{p_{new}})$. Token set $T = \{\tau_d, \tau_{d_{p_{new}}}\} \cup \{\tau_e \mid e \in A\}$ represents the necessary and sufficient tokens required by the* move *directory operation.*

Proof. First, we follow the CISE analysis to prove that the set T is indeed sufficient for maintaining the tree invariant. To this goal, we prove that precondition of move operation is stable against all concurrent move operations allowed by the tokens. Assume operation $\mathcal{F}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}$ moving directory d located in the parent directory $d_{p_{old}}$ to its new parent directory $d_{p_{new}}$. The operation is associated with tokens τ_d, and $\tau_{d_{p_{new}}}$ over the directory d and $d_{p_{new}}$, and a set of tokens τ_e, for all directories e, which are ancestors of the new parent directory $d_{p_{new}}$ up to $LCA(d, d_{p_{new}})$.

The precondition of move operation requires that (1) directory $d_{p_{old}}$ is the parent of directory d, and (2) directory d is not reachable from $d_{p_{new}}$. The first assertion becomes false when another move operation concurrently moves directory d. Token τ_d in set T disallows this concurrent execution. The second assertion will be violated when another move operation concurrently move directory $d_{p_{new}}$ under directory d. Acquiring token $\tau_{d_{p_{new}}}$ and the tokens over $d_{p_{new}}$'s ancestors in set T forbids such concurrent situation.

However, we only need to acquire tokens over ancestors of $\tau_{d_{p_{new}}}$ up to the least common ancestor of d and $\tau_{d_{p_{new}}}$. We use contradiction to support this claim. For brevity, we use $\mathsf{mvdir}(d_{p_{new}}, d)$ to indicate moving directory d in to directory $d_{p_{new}}$. Let assume that the ancestors' tokens up to the $LCA(d, d_{p_{new}})$ is not sufficient, and a cycle is created as follows:

$$a \downarrow c \ldots \downarrow d_{p_{new}} \downarrow d \ldots b \downarrow a$$

Where c, b, a are directories. This happens when there are move operations concurrently moving the $d_{p_{new}}$'s ancestors, which are located above the $LCA(d, d_{p_{new}})$. The left side of this cycle ($a \downarrow c \ldots \downarrow d_{p_{new}} \downarrow d$) indicates that there is an operation $\mathsf{mvdir}(a, c)$ concurrently moving one of common ancestors of $d_{p_{new}}$ and d, say c, in to directory a. This operation succeeds iff directory a is not a descendant of c (it's the move's precondition).

Now, consider the right side ($b \downarrow a$), where another concurrent operation $\mathsf{mvdir}(b, a)$ moves directory a in to one of d's descendants, say b. This operation requires tokens over directory b up to $LCA(b, a)$. Depending on the location of $LCA(b, a)$, we consider two cases: 1) directory d is located between $LCA(b, a)$ and the destination directory b, i.e., directory d is in set A of $\mathsf{mvdir}(b, a)$ operation ($LCA(b, a) \downarrow^+ d \downarrow^+ b$). Thus, moving a to b requires token over d, which conflicts with the token set of $\mathsf{mvdir}(d_{p_{new}}, d)$ operation. 2) $LCA(b, a)$ is located under d. This means that directory a is d's descendant. Knowing c is d's ancestor, a is also c's descendant. This violates the precondition of operation $\mathsf{mvdir}(a, c)$, which requires directory c not to be a's ancestor, and hence, the execution of operation $\mathsf{mvdir}(a, c)$ cannot happen. Unless, there was another operation $\mathsf{mvdir}(d, a)$ moving directory a in to directory d concurrently with operation $\mathsf{mvdir}(a, c)$. However, this move operation also requires token d conflicting with the tokens of $\mathsf{mvdir}(d_{p_{new}}, d)$ operation.

Thus, independent of LCA's location, the right and left hand side of cycle cannot be true at the same time, i.e., directory a cannot move in to one of d's descendants while moving d's ancestors in to a. This contradicts the original

assumption that the cycle is created, and the above is impossible. Therefore, we conclude that acquiring tokens up to $LCA(d, d_{p_{new}})$ is sufficient.

Now, we show that set T is necessary, i.e., it contains the minimal set of tokens, by contradiction: We assume that T is not minimal, meaning that it includes unnecessary tokens. We remove a token $\tau \in T$, and then check whether concurrent executions of move operations still maintain the tree invariant. If so, set T is not minimal.

1. τ_d is the token over the source directory d. Removing token τ_d from set T allows concurrent operations to move the same directory d to another destination directory d_1. If $d_1 \neq d_{p_{new}}$, then d will have two parents; violating the tree invariant.
$$d_{p_{new}} \downarrow d \wedge d_1 \downarrow d$$
2. $\tau_{d_{p_{new}}}$ is the token over destination directory $d_{p_{new}}$. Removing token $\tau_{d_{p_{new}}}$ from set T allows another move operation to concurrently move destination directory $d_{p_{new}}$ to directory d_1. If $d_1 = d$, or if directory d_1 is a descendent of directory d, i.e., $d \downarrow^+ d_1$, then cycles occur.
$$d_1 \downarrow d_{p_{new}} \wedge d_{p_{new}} \downarrow d \wedge d \downarrow^+ d_1$$
3. τ_{d_1} is the token over one of $d_{p_{new}}$'s ancestors, say d_1. . Removing token τ_{d_1} from set T allows another move operation to concurrently move directory d_1 to directory d_2. If $d_2 = d$, or if directory d_2 is a descendent of source directory d, i.e., $d \downarrow^+ d_2$, then cycles occur.
$$d_2 \downarrow d_1 \wedge d_1 \downarrow^+ d_{p_{new}} \wedge d_{p_{new}} \downarrow d \wedge d \downarrow^+ d_2$$

□

Thus, for moving directory d, we only require to acquire tokens over d, its destination directory $\tau_{d_{p_{new}}}$, and all ancestors of $\tau_{d_{p_{new}}}$ up to $LCA(d, d_{p_{new}})$. Concurrent move operations are allowed as long as their token sets are compatible. The intuition behind acquiring tokens over ancestors up to the least common ancestor is: if a directory is a common ancestor of directory d and its destination directory, the directory cannot be involved in concurrent move operations that result into the tree invariant's violation because it is disallowed by the move operation's precondition.

For instance, Fig. 7 illustrates how the token assignment avoids previous counter example. Operation mvdir$(root, n_1, v, u)$ acquires tokens $\{\tau_u, \tau_v\}$, and operation mvdir$(root, n_2, u, v)$ acquires the tokens $\{\tau_u, \tau_v\}$. Their token sets are not compatible, token τ_u is not compatible with token τ_v, i.e., $\bowtie = \{(\tau_u, \tau_v)\}$. When another move operation executes at a different replica r_2. This will force it to synchronise with other replicas to find out if there are other move operations. So, it will get the information about the first move operation in replica r_1, and cannot succeed.

We add the corresponding tokens to the move semantics, and perform the CISE stability analysis again. This time, the tool generates a counter-example

Table 2. A summary of file semantics verified by tool

Semantics	#Op	#Tokens	#Invariant	Anomaly	Verification Time(ms)
Sequential Design	7	7	1	NO	278
Concurrent Design	7	0	1	invariant violation	1297
Fully-Asynchronous Design	7	0	1	duplication	2350
Mostly-Asynchronous Design	7	2	1	NO	1570

that indicates that two concurrent users might move the same file to different locations. Thus, the file would end up with two parent directories; violating the tree invariant. To avoid this issue, we assign an exclusive move token τ_f over file f to each move file operation. Table 1 presents the move tokens required in the mostly- asynchronous file system design. The semantics successfully passes all three CISE analyses. The analyser proves that the consistency choices for different move operations are sufficient to preserve the tree invariant.

7 Evaluation

We have developed a verification tool that leverages the CISE analysis to co-design and verify a replicated file system. Our tool is currently implemented as a few hundred lines of Java code that reduces the CISE obligations to Satisfiability Modulo Theories (SMT) queries. We built the tool on the Z3 SMT solver [1], developed by Microsoft Research for the verification and analysis of software applications. Using the tool, we are able to encode a variety of file system semantics. The challenge of file system verification using the SMT solver was to translate reachability property because the SMT solver does not support any built-in transitive closure operator. We employed the tactics and strategies proposed in [28] and [16] to incorporate the reachability property in the context of the SMT solver. Table 2 summaries the results of verification of four file system semantics and the time taken by the tool. The tool was run on a Mac Mini, 3 GHz Intel Core i7. The number of operations is given without taking into account operation arguments. The number of tokens specifies the number of operations that require synchronization. The analyzer shows that the concurrent execution of move operations is anomalous, i.e., it may violate the tree invariant. It follows that no file system can support an unsynchronised move without anomalies, such as loss or duplication.

8 Related work

First-Order Logic Reasoning. A number of formalisations of file systems have been proposed using first-order logic [6, 17, 23]. Most of them focus only on primitive file I/O operations, such as reading and writing file content [6, 29]. Arkoudas et al. [6] have proved the correctness of read and write operations for a basic file system implementation using Athena, an interactive theorem prover. Given a simple file system implementation, Athena constructs 283 lemmas and

theorems in order to verify the isolation of reading and writing files in a directory. Hughes [24] has specified a visual file system using the Z notations [44]. He focuses on modelling of a hierarchical file system, so that its model covers basic operations affecting the tree structure, including move and remove directories. However, its specification does not consider the no-loop property, it only takes transitive closure (i.e., reachability) as the main property of a tree structure. Inspired by Hughes's specification, Damchoom et al. [14] have formalised and proved a tree-structured file system by using Event-B and Rodin platform [5]. Like our specification, their model is based on acyclic directory structure. A set of permissions are attached to an object, so that accesses to the object depend on the permissions allowed. The Rodin toolset generates 162 proof obligations to verify the specification model. Hesselink and Lali [23] have introduced an alternative approach to formulate the file structure using partial functions from paths to data.

However, the first-order logic reasoning does not scale well when reasoning about operation executions of a Posix file system [32]. The Posix English specification defines a set of preconditions for each operation, which must be satisfied before its execution. For instance, moving a source directory into a destination directory takes effect, if the source directory is not an ancestor of the destination directory. Encoding such conditions using first-order logic entails many proof obligations and constraints that increase non-linearly with respect to the size of programs [32].

Separation Logic Reasoning. Recent work on file system verification relies on separation logic [37]. Haogang et al. [22] have introduced Crash Hoare Logic (CHL) for developing and verifying sequential and fault-tolerant file systems. The CHL logic checks whether a storage system implementation will recover to a state consistent with its specification after a failure. Using the analysis, the authors specified and verified FSCQ, a crash-safe user-space file system implemented in Haskell. The FSCQ's interface consists of a series of Hoare triples over high-level operations. The specification model of FSCQ relies on the separation logic to reason about operations at different level of abstractions including disk, files, directories, and logical disk. FSCQ uses a write-ahead log for failure recovery. The CHL analysis proved that the write-ahead log guarantees atomicity of updates by adding fault-conditions into the Hoare triples.

Biri and Galmiche [10] have proposed a separation logic rule for trees and local reasoning over global paths. However, their simple tree model forbids structural modifications, as neither new nodes can be created nor nodes can be moved, i.e., the tree structure is static.

Gardner et al. [18] have proposed a formal model of Posix file system based on separation logic. The semantics of Posix operations are captured with preconditions and postconditions in a Hoare-logic style. Some permissions are associated into each operation to control access to shared paths. Before applying an update, the necessary permissions must be obtained in order to ensure that the effect of the update is propagated to entries whose path may overlap. However, the specification model does not support concurrent Posix users.

Conflict Resolution in File systems. Clements et al. [13] have proposed a cache conflict-free implementation of Posix file system on a shared-memory multiprocessor system. They explore the commutativity of Posix operations to design a scalable file system implementation. They have presented an analyser, called COMMUTER, which checks the commutativity of Posix operations. COMMUTER relies on symbolic executions for program testing. A symbolic model tests all permutations of operations, and computes necessary conditions under which those operations commute. Using the commutativity conditions, they modify the Posix semantics. COMMUTER generates different test cases to verify the semantics in a real implementation. However, they focus only on scalability, not on the safety of executing commutative operations; they do not verify that the commutative operations maintain the tree invariant.

Balasubramaniam and Pierce [8] have proposed an optimistic files system replication model from a semantics perspective. Causally-dependent operations are ordered according to a happen-before relation, while concurrent operations may be executed in any orders. Concurrent updates on the same directory are allowed if they do not conflict. For instance, concurrent users can add different files with different names to the same directory, but if one user modifies a file, and another deletes its parent directory, a conflict happens. The model requires users to manually resolve conflicts. This specification model was later formalised and proved by Ramsey and Csirmaz [36]. However, the operation-based model is limited i.e., the algebra model contains 51 different rules for few operations, including create, remove, and edit. It is not clear how one can extend the model to support more complex operations, such as move operations involving different directories. In addition, the model does not check the tree invariant; it is difficult to describe the acyclic property by using their model.

Bjϕrner [11] has proposed a replicated file system reconciler (DFS-R) that automatically resolves conflicts when they arise. The author uses model checkers to verify the conflict resolution strategies. Similarly to the CISE-enabled tool, the analysis gives a counter-example for concurrent moves, meaning that concurrent move operations do not maintain directory hierarchies as tree-like structure. However, this reconciler does not address how to add synchronisation when the tree invariant is violated. In this vein, Microsoft One Drive [4] discards the directories involved, thus restoring a tree. Similarly, Google Drive [3] moves the involved directories directly under the root. The geoFS [45] system effectively executes a problematic move as copy-delete, duplicating the directories involved.

9 Conclusion and Future Work

We have applied the CISE analysis to verify and co-design an available file system. Initially, the file system specification models the POSIX file system. The main invariant is that the file system structure must be shaped as a tree. We verified that our co-design approach is able to remove synchronisation for the common file system operations, while ensuring the tree invariant.

There are several avenues for future work from both verification and performance perspective. First, the CISE analysis only verifies the correctness of the file system against concurrent executions. In the future,we plan to propose proof rules that allow developers to reason about the operation executions in the presence of replica and network failures. Thus, programmers would be able to prove that a file system specifications model handles properly any possible faults. This entails formalisation of failure models, as the specification of the file-system API captures its semantics under crashes. We are going to implement the three file system semantics to compare their actual performance under real workloads. The plan is to integrate our co-design findings into a highly-scalable geo-replicated file system. The challenge is to translate the tokens into an efficient concurrency control protocol, which is also dead-lock free. We are looking for dynamic and heuristic analysis that allow to measure and improve the token implementations.

Bibliography

[1] Https://github.com/Z3Prover/z3

[2] POSIX.1-2008. The Open Group Base Specifications Issue 7

[3] Google Drive (2017), https://www.google.com/drive/

[4] Microsoft OneDrive (2017), https://onedrive.live.com/

[5] Abrial, J.R.: A system development process with event-b and the rodin platform. In: Proceedings of the Formal Engineering Methods 9th International Conference on Formal Methods and Software Engineering (ICFEM). pp. 1–3. Berlin, Heidelberg (2007)

[6] Arkoudas, K., Zee, K., Kuncak, V., Rinar, M.: Verifying a file system implementation. In: Proceedings of the Formal Methods and Software Engineering. pp. 373–390. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[7] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K.: Measurements of a distributed file system. In: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles. pp. 198–212. SOSP '91, ACM, New York, NY, USA (1991)

[8] Balasubramaniam, S., Pierce, B.C.: What is a file synchronizer? In: Int. Conf. on Mobile Comp. and Netw. (MobiCom '98). ACM/IEEE (Oct 1998)

[9] Bernstein, P., Radzilacos, V., Hadzilacos, V.: Concurrency Control and Recovery in Database Systems. Addison Wesley Publishing Company (1987)

[10] Biri, N., Galmiche, D.: Models and separation logics for resource trees. Journal of Logic and Computation 17(4), 687–726 (2007)

[11] Bjørner, N.: Models and software model checking of a distributed file replication system. In: Formal Methods and Hybrid Real-Time Systems. pp. 1–23 (2007)

[12] Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference. pp. 28–28. ALS'00, Berkeley, CA, USA (2000)

[13] Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., Kohler, E.: The scalable commutativity rule: Designing scalable software for multicore processors. In: Symp. on Op. Sys. Principles (SOSP). pp. 1–17. ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery, Farmington, PA, USA (2013)

[14] Damchoom, K., Butler, M., Abrial, J.R.: Modelling and proof of a tree-structured file system in event-b and rodin. In: Proceedings of the 10th International Conference on Formal Methods and Software Engineering. pp. 25–44. ICFEM '08, Springer-Verlag, Berlin, Heidelberg (2008)

[15] Davidson, S.B., Garcia-Molina, H., Skeen, D.: Consistency in a partitioned network: a survey. ACM Comput. Surv. 17(3), 341–370 (Sep 1985), http://doi.acm.org/10.1145/5505.5508

[16] El Ghazi, A.A., Taghdiri, M.: Analyzing alloy constraints using an smt solver: A case study. In: 5th International Workshop on Automated Formal Methods (AFM). Edinburgh, United Kingdom (2010)

[17] Freitas, L., Woodcock, J., Butterfield, A.: Posix and the verification grand challenge: A roadmap. In: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 153–162 (March 2008)

[18] Gardner, P., Ntzik, G., Wright, A.: Local reasoning for the posix file system. In: Shao, Z. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 8410, pp. 169–188. Springer Berlin Heidelberg (2014)

[19] Ghemawat, S., Gobioff, H., Leung, S.T.: The Google File System. In: Symp. on Op. Sys. Principles (SOSP). pp. 29–43. Assoc. for Computing Machinery, Bolton Landing, NY, USA (Oct 2003)

[20] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In: Symp. on Principles of Prog. Lang. (POPL). St. Petersburg, FL, USA (2016)

[21] Guy, R., Heidemann, J.S., Mak, W., Popek, G.J., Rothmeier, D.: Implementation of the ficus replicated file system. In: USENIX Conference Proceedings. pp. 63–71 (1990)

[22] Haogang, C., Daniel, Z., Tej, C., Adam, C., Frans, K.M., Nickolai, Z.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. SOSP '15, ACM, New York, NY, USA (2015)

[23] Hesselink, W.H., Lali, M.: Formalizing a hierarchical file system. Formal Aspects of Computing 24(1), 27–44 (2010)

[24] Hughes, J.: Specifying a visual file system in z. In: IEEE Colloquium on Formal Methods in HCI: II. pp. 3/1–3/3 (Dec 1989)

[25] Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. North-Holland (1983)

[26] Kistler, J.J., Satyanarayanan, M.: Disconnected operation in the Coda file system. In: Symp. on Principles of Dist. Comp. (PODC). vol. 10, pp. 3–25 (Feb 1992)

[27] Kumar, P., Satyanarayanan, M.: Flexible and safe resolution of file conflicts. In: Usenix Tech. Conf. New Orleans, LA, USA (Jan 1995)

[28] Leino, K.R.M.: Automating induction with an smt solver. In: Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 315–331. VMCAI'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-27940-9_21`

[29] Morgan, C., Sufrin, B.: Specification of the unix filing system. vol. SE-10, pp. 128–142 (1984)

[30] Nadkarni, A.: Scale-out file systems on object-based storage platforms. IDC Technology Assessment 258393, International Data Corporation (IDC), Framingham, MA, USA (2015)

[31] Najafzadeh, M., Gotsman, A., Yang, H., Ferreira, C., Shapiro, M.: The CISE tool: Proving weakly-consistent applications correct. In: W. on Principles and Practice of Consistency for Distributed Data (PaPoC). EuroSys 2016 workshops, ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery, London, UK (Apr 2016)

[32] Ntzik, G., Gardner, P.: Reasoning about the posix file system: Local update and global pathnames. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 201–220. OOPSLA 2015, ACM, New York, NY, USA (2015)

[33] Ousterhout, J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M., Thompson, J.G.: A trace-driven analysis of the unix 4.2 bsd file system. In: SIGOPS Oper. Syst. Rev. vol. 19, pp. 15–24. ACM, New York, NY, USA (Dec 1985)

[34] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz, D.: NFS version 3 design and implementation. In: Proceedings of the Summer 1994 USENIX Technical Conference. pp. 137–152 (1994)

[35] Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Symp. on Op. Sys. Principles (SOSP). pp. 288–301. ACM SIGOPS, Saint Malo (Oct 1997)

[36] Ramsey, N., Csirmaz, E.: An algebraic approach to file synchronization. Tech. Rep. TR-05-01, Harvard University Dept. of Computer Science, Cambridge MA, USA (May 2001)

[37] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02, IEEE Computer Society, Washington, DC, USA (2002)

[38] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.: Design and implementation of the Sun Network Filesystem. In: Summer 1985 USENIX Conf. pp. 119–130. USENIX, Portland OR, USA (Jun 1985)

[39] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. In: IEEE Trans. on Computers. vol. 39, pp. 447–459 (1990)

[40] Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies. FAST '02, USENIX Association, Berkeley, CA, USA (2002)

[41] Schwan, P.: Lustre: Building a file system for 1,000-node clusters. In: Proceedings of the 2003 Linux Symposium (2003)

[42] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS). Lecture Notes in Comp. Sc., vol. 6976, pp. 386–400. Springer-Verlag, Grenoble, France (Oct 2011)

[43] Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–10. MSST '10, IEEE Computer Society, Washington, DC, USA (2010)

[44] Spivey, J.M.: The z notation: A reference manual. In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (1998)
[45] Tao, V., Shapiro, M., Rancurel, V.: Merging semantics for conflict updates in geo-distributed file systems. In: ACM Int. Systems and Storage Conf. (Systor). pp. 10.1–10.12. Haifa, Israel (May 2015)
[46] Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani: A scalable distributed file system. In: SIGOPS Oper. Syst. Rev. vol. 31, pp. 224–237. ACM, New York, NY, USA (Oct 1997)
[47] Vogels, W.: File system usage in windows nt 4.0. In: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP). pp. 93–109. ACM, New York, NY, USA (1999)
[48] Vogels, W.: Eventually consistent. In: ACM Queue. vol. 6, pp. 14–19 (Oct 2008)
[49] Wang, A.I., Reiher, P., Bagrodia, R., Kuenning, G.: Understanding the behavior of the conflict-rate metric in optimistic peer replication. In: Proceedings of the 13th International Workshop on Database and Expert Systems Applications. pp. 757–761 (Sept 2002)