Garantir la disponibilité et la gestion de la cohérence dans les systèmes géo-distribués

Vinh Tao Thanh

UPMC
SORBONNE UNIVERSITÉS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité : Informatique

École doctorale : Informatique, Télécommunications, et Électronique (Paris)

réalisé :

à Laboratoire d'Informatique de Paris 6 et à Scality

présenté par :

Vinh TAO THANH

pour obtenir le garde de :

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Ensuring Availability and Managing Consistency in Geo-Replicated File Systems

soutenue le

devant le jury composé de :

Mme.	Sara BOUCHENAK	Rapporteur
M.	Pascal MOLLI	Rapporteur
Mme.	Annette BIENIUSA	Examinateur
Mme.	Julia LAWALL	Examinateur
M.	Vivien QUEMA	Examinateur
M.	Vianney RANCUREL	Co-directeur de thèse
M.	Marc SHAPIRO	Co-directeur de thèse

Acknowledgement

My PhD road was on the lonely side (just like any other), and it would not have come to an end without various forms of support and encouragement from people around.

First and foremost, thank you Marc for enduring me the whole 4 years, especially the last several months of being constantly nagged by me for my manuscript. Your desire to do fundamental research, your fairness, and your enthusiasm for nature all inspire my thinking about research in some way.

With the same gratefulness, I would like to thank my co-advisor—Vianney, director of research at Scality. Thank you for being an awesome manager and mentor. Your constant helps and encouragement literally dragged me out of the darkest periods of my PhD. The discussions with you were so inspiring and mind-opening. Your hunger for new ideas and the wildness of your ideas are all radical and exciting. You have proved that degree may not prove anything.

I am thankful to the committee members, including my co-advisors, for spending your time and effort to complete my thesis. Thank you Julia for going over every single detail of the thesis from its early stage. All mistakes left are my still mine though.

Thank you my PhD student fellows at LIP6; you made my PhD life a little less lonely. Thanks Alejandro, Tyler, and João for once in a while sharing a glass of beer or two with me in the early days (Ale and Chris actually shared three bottles of wine with me in Bordeaux, which proudly went into my alcohol history as a record). Thanks Paolo, Sreeja, Ilyas and Dimitrios for accepting my unpredictability the last couple of months; Paolo, you kept your promise to eat noodle during my defense, you are such a man. I am also very much thankful to Françis for translating the résumé of my thesis.

Last but not least, thank you my friends at Scality; you had to suffer my repeated story about the difficulties of being a PhD student, yet you always cheer me up and swallow the (PhD) bitterness with me over a pint of beer. Just to name a few: Thibault, Alex, Mathieu, Antonin, Maxime, David, Lam, Laure, Michael, Juan, Marc, Lauren, Rahul, Electra, Mudit, Nicolas, Jordan, Boris, Kevin. Thanks Jean-Marc for sharing the office many late nights, and going endlessly with me over my many silly stories.

This thesis is dedicated to my family.

Abstract

Geo-distributed systems suffer from high latency and network partitions. Because of this, and to ensure high availability, such systems typically commit updates locally, with no latency, and propagate them in the background. Such optimistic replication faces two major challenges: (i) detecting conflicts between concurrent updates and resolving them in a way meaningful for users, while maintaining system integrity invariants; and (ii) supporting legacy applications that are not prepared to deal with concurrency anomalies.

Our PhD research addresses these challenges for the specific use case of a large-scale geo-distributed file system. This is a good showcase: indeed, a geo-distributed file system has a complex hierarchical structure; maintaining the file-system invariants (e.g., tree structure) against parallel updates is challenging; and applications view the file system through the legacy POSIX API.

Existing optimistic geo-distributed file systems fall short of addressing the challenges. For instance, Dropbox does not support hard links; Andrew File System fails on some concurrent renaming of directories; and all existing systems use automatic conflict resolution that violates the legacy POSIX semantics.

We present our solution to the above problems in the design and implementation of a prototype geo-distributed file system, named Tofu. Its design includes a new session abstraction to support the legacy API, while allowing optimistic updates. Unlike previous approaches, our solution is based on a formal model covering all aspects of a Unix-like file system, including directories, inodes, hard links, etc. It is able to detect all conflicts on those data structures, and resolves them in a way that we believe users will find generally reasonable. Experiments show that Tofu is highly scalable, and incurs linear overhead, improving over existing academic and industrial systems.

keywords: *Replication, Geo-Distribution, File System, Eventual Consistency, CRDT, Conflict Resolution*

Table Of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A geo-distributed file system typically spans multiple replicas that are remote from each other. The inter-replica network of a geo-distributed file system has limited bandwidth and high latency, especially compared to the intra-replica fabric. In order to be available, a geo-distributed file system must address the inherent trade-off between consistency and availability, underscored by the CAP theorem [6, 22].

Many large scale geo-distributed file systems opt for the Eventual Consistency (EC) approach [70,72,73]. In an EC system, an update commits locally on its original replica, before being propagated to the other replicas asynchronously; EC ensures that, when all replicas have received and applied all updates from each other, they will all have the same state. Because a commit is local, without coordination between replicas, users experience low latency. A user can modify a replica, even if that replica remains disconnected for a long time, thus ensuring high availability.

This EC approach has to deal with conflicts between concurrent updates from different replicas. A geo-replicated file system using the EC approach faces two major challenges: (i) detecting (all) conflicts between concurrent updates, and resolving them in a meaningful way for users, while maintaining system integrity invariants; and (ii) supporting legacy applications, which are not prepared to deal with concurrency anomalies.

Detecting and resolving conflicts in a geo-distributed file system is difficult because of the complex hierarchical structure of the file system. A file system typically consists of a tree of directories; a directory can contain other directories or files; however, a file may be included in multiple directories, thanks to hard links. This complex structure is ruled by strict invariants, such as uniqueness of names within a directory, or absence of directory cycles.

Because of the difficulty of detecting and resolving conflicts in such complex sys-

tems, existing approaches often forgo some system invariants, or limit the conflict detection and resolution ability. Some, including Dropbox [13] or Unison [3], simplify the file system model by ignoring hard links, treating multiple links to the same file as if they were different files, thus results in diverged file system replicas and unnecessary network/storage usage. Others, including Ficus [56], Coda [30], or Microsoft OneDrive [37], leave reconciliation of difficult conflict cases to users; they might move the directories and files involved in a conflict into a special directory, expecting users to resolve issues manually. Some others, including AFS [26] and Google Drive [23], opt for a simple Last-Writer-Wins (LWW) approach which chooses an arbitrary update to win over conflicting ones; this approach forgoes update durability.

Furthermore, existing EC geo-distributed file systems do not support legacy applications well. These systems automatically change the file system at unexpected times and in non-intuitive ways. Adapting legacy applications to cope with this behavior would require complex logic. For example, when two users concurrently write to a same file, both Dropbox and Google Drive resolve the conflict by creating new files with somewhat arbitrary names; this can even happen while the file is in active use. Legacy applications are not prepared to cope with such sudden and unpredictable changes.

This thesis addresses these issues: to support both concurrent updates and compatibility with legacy applications. Accordingly, we designed a geo-distributed file system, named Tofu. Our contributions are as follow.

1. The design and implementation of a conflict detection and resolution mechanism that identifies and resolves *all* conflicts, based on: our formal file system model (Chapters 2 and 3) which supports all file system components including hard links, and our conflict resolution semantics (Chapters 7 and 8) that preserves all concurrent updates, while presenting meaningful conflict resolution results. The implementation of Tofu (Chapter 9) is based on the concept of Conflict-free Replicated Data Type (CRDT) [65, 66] to ensure convergence and correctness.

2. The session concept that isolates legacy POSIX applications from unexpected automatic changes in the file system. Our session system (Chapters 4 and 5) divides the usage of a distributed file system into sessions. A session is a kind of a long transaction, within which applications enjoy the strong sequential semantics of a traditional POSIX file system. Multiple sessions may co-exist concurrently, each one being isolated from the others. A session makes a consistent snapshot of the whole file system at the beginning, and atomically merges all its changes into the file system at the end. Tofu automatically resolves any concurrent updates between the committing session and

any other prior committed session. Manual intervention is required only if automatic conflict resolution has made changes that would be incompatible with applications in later sessions. Such a manual intervention is limited to renaming directories and files.

3. Experimental results showing the completeness of our approach compared to those existing with respect to the conflict resolution ability. We show through experiments (Chapter 10) that Tofu detects and resolves all conflicts, including those which have been a difficulty for previous systems. Tofu's session system is able to provide the low latency for the updates of the eventual consistency approach. We also show that the commit of a session has minimal impact on the latency of the other ongoing sessions.

Part I

Sequential Semantics

Chapter 2

Sequential File System Model

In this chapter, we describe our model of sequential file systems; a sequential file system is one that commits updates having the same target sequentially. Our model includes file system data objects, file system layout, and file system invariants. Our model is inspired by POSIX; we will comment on the similarities and differences between POSIX and other file system variants whenever applicable.

2.1 File System Data Objects

A file system is a collection of data objects called inodes.[1] An inode is identified by a unique[2] identifier called its inode number.[3] Each inode has three components, namely, *stat*, *names*, and *data* (Figure 2.1).

2.1.1 Inode Stat

The *stat* of an inode stores an inode's metadata attributes, including both so-called system attributes and extended attributes. The former are predefined, and are created and maintained by the file system, although some can be updated by users. For example, the attribute `st_atime` of an inode records the time when the inode was last accessed; this attribute is automatically updated by the file system. On the other hand,

[1] A record in the Master File Table of Microsoft NTFS is their equivalent of an inode.

[2] The inode numbers contained in a file system are unique relative to the allocation unit (e.g., a partition in POSIX.) However, real-world file system implementations such as EXT for Linux and NTFS for Microsoft Windows employ a strategy called 'inode number reuse' for practical purposes; the inode number of a deleted inode can be reused later. Only NTFS keeps track of how many times an inode number has been reused.

[3] NTFS has two different concepts equivalent to the inode number: *FileReference* number and *ObjectId*. The former more closely resembles an inode number as it is automatically assigned to an inode when it is created and can be reused. The latter is a unique 16-byte string assigned upon the request of application layer; it is used by NTFS in distributed setups.

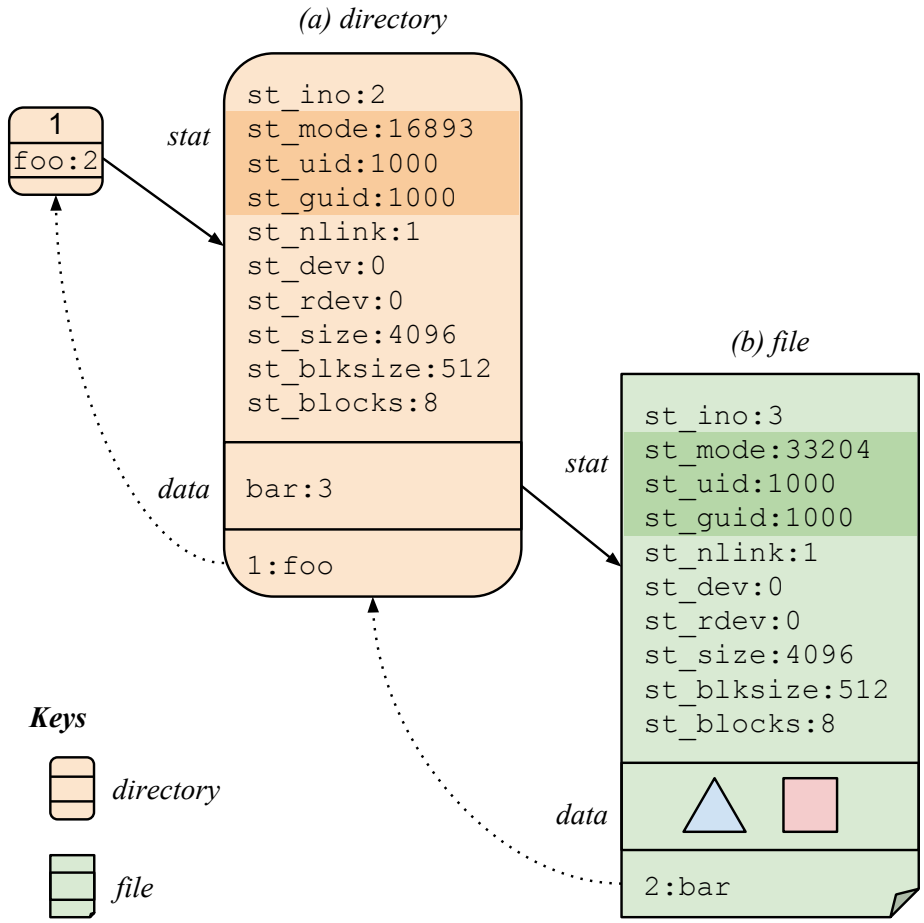


Figure 2.1: Inode model describing the *stat*, *names*, and *data* parts of an inode. With (a) and (b) are a directory and a file, respectively. Manually updatable attributes are highlighted. Triangle and square represent the arbitrary data of a file. Solid arrows represent parent-child mappings, and dotted arrows are the reverse of those mappings. The parent of (a) is represented in a simplified form.

the attribute `st_mode`, which stores the inode's ownership information, is initialized by the file system, but it can be manually updated by users. Extended attributes are manually defined and maintained by users only, and are opaque to the file system semantics. These attributes are interpreted only by users or applications, such as those to keep the geo-location of an image contained in the *data* part of the inode.

Some of the system attributes play a minor role (e.g., `st_atime`); hereafter we focus on the ones that are essential to file system correctness. (1) The type of an inode is indicated by attribute `st_mode`. There are two main inode types, *directory* and *file*.[4]

[4]Throughout this thesis, we will use the terms *directory* and *file* to designate inodes of those specific types, while using *inode* as a generic term for both.

An inode of type *directory* stores the directory contents (a mapping of string names to inode numbers) in its *data* part (next section). An inode of type *file* contains arbitrary (opaque) data in its *data* part, such as a text document or an image.[5] (2) Security information is stored in the `st_mode` (access permissions), `st_uid` (owner), and `st_gid` (group) attributes. This dictates who (user, group) will have which kind of access (read, write, execute) to the inode.[6] (3) The remaining attributes contain accounting information for the inode, such as timestamps (`st_atime`: last-accessed time, `st_mtime`: last-modified time, and `st_atime`: created time), its location (`st_dev` and `st_rdev`: the identifier of the device containing the inode, `st_ino`: inode number), its size on disk (`st_size`, `st_blksize`, `st_blocks`: inode's real- and on-disk- size information), and its number of associated hard links `st_nlink` (more on hard link in the next section).

2.1.2 Inode Data

The *data* part of an inode stores the useful contents of this inode.[7] Depending on inode type, its *data* part may be meaningful or opaque to the file system. For example, a symbolic link's *data* part is a string representing the location of another inode; the file system reads this information and forwards access towards its destination. A directory's *data* part is a map of names (strings) to inode numbers (called target inodes). Each such mapping constitutes a hard link to the target inode. The file system uses this mapping information in its hierarchical structure (Section 2.2). Hereafter, we call the *data* part of a directory its *child map* or just *map*, and a mapping entry (hard link) as a *mapping*. The *data* part of a file is opaque and is of arbitrary type, such as the sequence of characters of a text document, or the set of pixels of an image.

2.1.3 Inode Names

The *names* part of an inode keeps a reverse reference to all of the hard links that have been mapped to the inode. We represent *names* as a map of directory inode numbers to the (string) names of the current inode in these directories. Clearly, this information should be consistent with *st_ nlink* and with the contents of directories (this is formalized by Invariant 5 in Section 2.3). The *names* part is implicit in POSIX and not explicitly materialized in previous implementations. Our model (and implementation) makes it explicit because it plays an important role in file system correctness.

[5] There are other inode types, which we ignore because they are not critical to file system integrity, such as symbolic link, FIFO, or device. For the purposes of this thesis, they can be identified with *file*.

[6] Some modern Unix-like operating systems, such as Linux, use extended attributes for file system security. Abstractly, security extended attributes together constitute an ACL (Access Control List).

[7] The data is often stored as a hierarchy of indirect inodes, each holding a part of the data. We will ignore this detail hereafter.

2.2 File System Structure

A file system can be considered as a graph where an inode constitutes a node, and a mapping constitutes a vertex. In common file systems, the directory sub-graph is actually a tree. An inode linked from a directory is called a child of that directory. Conversely, an inode's *names* part consists of the reverse reference to its parents and its own name within the parents.

A directory may have multiple children, but has only a single parent (the root has no parent). Therefore the *names* part of a directory must always have a single entry, and a directory's `st_nlink` is always 1.[8] The *data* part of a directory may have multiple mapping entries. In contrast, a file may have multiple names and a higher link count.

No two children of a given directory are allowed to have the same name (a name is unique within its directory).

There can be no cycles in a directory tree. A sequential file system ensures this invariant by making it illegal to create a hard link from a directory to one of its ancestors. In particular, the `rename` operation checks this precondition, as discussed later (Section 2.3).

A directory tree has a distinguished 'root directory' (often identified by a specific inode number, such as '`2`' in Linux's Ext file systems). Although the contents of the root directory can change, the root itself cannot be deleted or replaced by a different inode. In POSIX tradition, the root is noted "`/`". The path from the root to some inode is called an *absolute path* of that node. An absolute path is a string represented by the concatenation of the names of all directories on the path, separated by "`/`"[9]; for example the path `/foo/bar` indicates that `foo` is a child of the root `/` and `bar` is a child of `foo`. The inode named by `/foo/bar` can either be a directory or a file; this cannot be deduced from the name alone. At any point in time, no two nodes have the same absolute path. Because a directory has a single parent, a directory thus has a single absolute path.

A file may have multiple names, and thus multiple absolute paths. Therefore, the full graph of both directories and files is not a tree, but a special case of a partially ordered set (poset).

Figure 2.2a presents an example file system composed of three inodes. Inodes `1` and `2` are directories (`1` being the root), and inode `3` is a file. The child map of `1` maps

[8] We exclude the self pointer and back pointer (identified by the names '`.`' and '`..`' respectively) of existing POSIX implementations, which would make the `st_nlink` of a directory be equal to the number of its child directories plus 2 (its name from its parent and its self pointer).

[9] We use throughout this thesis the UNIX notation with `/` for both the root and the hierarchy relation. Microsoft NTFS uses the device name for the root, e.g., `C:`, and the backward slash `\` for the hierarchy relation. In Microsoft NTFS, the path `bar` in this example might be denoted `C:\foo\bar`.

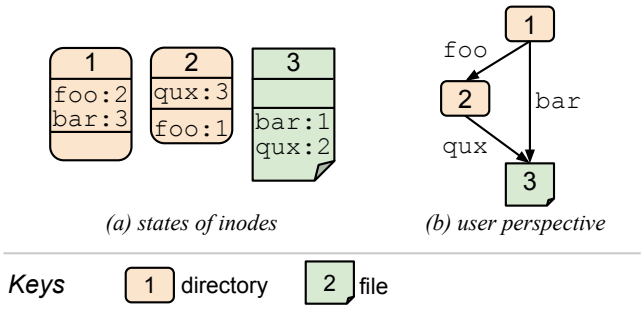


Figure 2.2: Our file system model. With (*a*) describes the data structure of the file system with a collection of inodes and their data structure (inodes are in simplified representation with only important bits described; the parts of an inode represent its *stat*, *data*, and *names*), and (*b*) visually presents the file system.

the name `foo` and `bar` to `2` and `3`, respectively. Inode `2` also maps the name `qux` to `3`. These mappings are reflected in the *names* part of inode `2`'s `1:foo`, and of inode `3`'s `{1:bar, 2:qux}`. Figure 2.2b visualizes how this structure is perceived from the users' point of view. Directory `2` in this structure has a single absolute path `/foo`, whereas file `3` has two absolute paths `/bar` and `/foo/qux`.

2.3 File System Invariants

This section describes the invariants that characterize the correctness of a file system.

2.3.1 Notations

We denote the child map in a directory as a set of (*name*, *ino*) pairs, representing the string name and the inode number of a target inode respectively. We use the dot notation, for example: *m.name* and *m.ino*, where m is an entry in the child map.

We use shorthands `i.ino` for `i.stat.st_ino`, the inode number of an inode i; `i.nlink` for `i.stat.st_nlink`, the number of links to the inode i; `i.type` for its type, which is `DIR` if the inode is a directory, or `FILE` otherwise; and `d.map` to represent the child map of directory d.

We denote the set of all inodes as $\mathfrak{I}$, which is ranged over by $i, i_1, i_2, \ldots$; the set of all inodes minus the root as $\mathfrak{I}^*$; the set of all directories as $\mathfrak{D}$, ranged over by $root, d, d_1, d_2, \ldots$; the set of all directories minus the root as $\mathfrak{D}^*$; and the set of all files as $\mathfrak{F}$. The relations $\mathfrak{I} = \mathfrak{D} \cup \mathfrak{F}$ and $\mathfrak{D} \cap \mathfrak{F} = \varnothing$ hold.

We define the relation *reachable* to indicate whether a directory is a descendant of another, and *parent* to indicate if a directory is the parent of an inode; *parents*(i)

denotes the set of all parent directories of inode i.

$$parent(d,i) = \begin{cases} \texttt{TRUE} & \exists m \in d.map : m.ino = i.ino \\ \texttt{FALSE} & otherwise \end{cases}$$

$$parents(i) = \{d | parent(d,i)\}$$

$$reachable(d,i) = \begin{cases} \texttt{TRUE} & parent(d,i) \vee (\exists d' : parent(d',i) \wedge reachable(d,d')) \\ \texttt{FALSE} & otherwise \end{cases}$$

2.3.2 Invariants

The following invariants characterize the safety properties of a correct file system. Any violation of these invariants would constitute an error. For simplicity, we restrict ourselves to a single file system.

Invariant 1 (fixed root) *A file system has a unique, non-changing root inode.*

Invariant 2 (unique inode number) *The number of an inode is unique in its file system.*

$$\forall i, i' \in \mathfrak{I}, i.ino = i'.ino \implies i' = i$$

Invariant 3 (single parent) *A directory has a single parent directory, except the root, which has no parent.*

$$\begin{cases} \forall d \in \mathfrak{D}^*, |parents(d)| = 1 \\ parents(root) = \varnothing \end{cases}$$

Invariant 4 (unique name) *The children of a directory have a unique name within that directory.*

$$\forall d, \forall m, m' \in d.map, m.name = m'.name \implies m = m'$$

Invariant 5 (mapping coherency) *a parent-child mapping entry in a directory is reflected by a corresponding reverse mapping in the names part of the child inode.*

$$\begin{aligned} \forall d, i, s, &\exists m \in d.map : m.ino = i.ino \wedge m.name = s \\ &\iff \exists n \in i.names : n.ino = d.ino \wedge n.name = s \end{aligned}$$

Invariant 6 (no cycles, reachability) *An inode is not reachable from itself, and an*

inode (not the root) is reachable from the root.

$$\begin{cases} \forall i \in \mathfrak{I}, reachable(i, i) = \texttt{FALSE} \\ \forall i \in \mathfrak{I}^*, reachable(root, i) = \texttt{TRUE} \end{cases}$$

Invariant 7 (metadata coherency) *The* `st_nlink` *of an inode is the number of the names of that inode.*

$$\forall d, f, \begin{cases} d.nlink = |d.names| = 1 \\ f.nlink = |f.names| \end{cases}$$

Invariants We define the conjunction of all invariants as:

$$\text{Invariants} = \bigwedge_{i=1}^{7} \text{Invariant } i.$$

Chapter 3

File System Operations

POSIX is a standard set of specifications for UNIX-compliant operating systems (OSes) to ensure their interoperability. File systems for those OSes need to implement the same set of 'system calls', e.g., `open` and `close`, to perform some common file system operations. In this chapter, we present the sequential semantics of some representative file system operations, such as (in Linux's VFS parlance) `mkdir`, `write`, or `rename`.

3.1 Preliminaries

3.1.1 Helper Functions

To simplify the descriptions of the operations, we use the following helper functions: newInode to create a new inode with a unique inode number, and lookup to find the inode object from inode number. These functions are defined below.

$$\begin{aligned} &\textsc{newInode}() = i : (\forall i' \neq i, i'.ino \neq i.ino) \\ &\textsc{lookup}(ino) = i : (i.ino = ino) \end{aligned} . \tag{3.1}$$

3.1.2 Path Resolution

The interface between a file system and its users/applications is based on absolute paths; users/applications perform operations targeting these paths. A file system has to translate a path used in an operation to the inode object. This process is called "path resolution" and is described by Algorithm 1. We use the function split to get the array of the names (not including the root) in a path using "/" as the delimiter, and we use the bracket notation to access individual names in the path. The inputs are: the path, the index of the next name to resolve, and the current parent directory.

Algorithm 1 path resolution

```
1: procedure PathResolution(path, index = −1, p)   ▷ inputs: path, next name index, current parent directory
2:     let names = split(path, "/")                ▷ split path into an array of names
3:     let name = names[index]                     ▷ get the referring name
4:     if index = |names| then                     ▷ reach the end of the path, return last inode
5:         return p
6:     if index = −1 then                          ▷ begin path resolution process from the root
7:         return PathResolution(path, 0, root)
8:     if p ∉ 𝔇 then ERROR                         ▷ not the end of path but parent is not a directory
9:     if ∄m ∈ p.map : m.name = name then ERROR    ▷ check if name exists in parent
10:    let ino = m.ino : (m ∈ p.map ∧ m.name = name)
11:    return PathResolution(path, index + 1, lookup(ino))   ▷ next name
```

3.2 Read Operations

A read operation returns either the contents or the metadata of an inode. For example `readdir` returns the child map of a directory; `getattr` returns the attributes (*stat*). Read operations are also useful to verify file system integrity by checking if file system invariants hold. In this section, we present the operation `traverse` which helps ensure that the tree structure of a file system has no cycles; we can assert so by checking if the set of all directories `traverse` can visit is equal to the set of all directories in the file system. This operation is described in Algorithm 2.

Algorithm 2 traverse tree

```
1: procedure traverse(d, D = ∅) ▷ inputs: current directory, traversed directories
2:     if d ∈ D then return ERROR
3:     D ← D ∪ {d}
4:     for m ∈ d.map do
5:         i ← lookup(m.ino)
6:         if i.type = DIR then
7:             if traverse(i, D) = ERROR then return ERROR
8: procedure CheckNoCycles
9:     if traverse(root) = ERROR then
10:        return FALSE
11:    return TRUE
```

3.3 Modification Operations

In this section, we describe some representative file system operations in POSIX, including: `create`, `mkdir`, `link`, `setattr`, `write`, `unlink`, `rmdir`, and `rename`; the other

operations, such as `symlink` or `truncate`, are either almost identical to, or an optimization of those representative operations.

We use the generator-effector structure [66] to present the pseudo-code of the following operations. The *generator* of a function ensures all invariants and all preconditions are met, then it computes a state transformation to be applied on the state of the system; this transformation is the *effector*. Applying a transformation is an atomic process, which has no side effect other than those explicitly specified.

3.3.1 `create`

This operation creates an empty file and is used by POSIX's system calls `creat` and `open`. It takes the path to the parent directory, a name, and a *stat* object as its inputs. In the normal case with no errors, it creates a new file with the assigned *stat* in the parent directory. Algorithm 3 describes the details of this operation.

Algorithm 3 CREATE create an empty regular file

1: **procedure** CREATE_GEN($path$, n, $stat$) ▷ inputs: parent directory, file name, stat
2: REQUIRE(Invariants)
3: **let** $p = \text{PATHRESOLUTION}(path)$
4: **if** $\neg(p \in \mathfrak{D} \wedge \nexists m \in p.map : m.name = n)$ **then** `ERROR` ▷ name already exist
5: **procedure** CREATE_EFF(p, n, $stat$)
6: REQUIRE(Invariants)
7: REQUIRE($p \in \mathfrak{D} \wedge \nexists m \in p.map : m.name = n$) ▷ new name does not exist
8: **let** $i = \text{NEWINODE}$ ▷ create an empty inode
9: $i.stat \leftarrow stat$ ▷ assign the desired *stat* from the input
10: $i.type \leftarrow$ `FILE` ▷ set inode type to file
11: $i.names \leftarrow \{(p.ino, n)\}$ ▷ add reverse mapping
12: $i.nlink \leftarrow 1$ ▷ update link count
13: $p.map \leftarrow p.map \cup \{(n, i.ino)\}$ ▷ mapping from parent
14: ENSURE($p \in \mathfrak{D} \wedge (n, i.ino) \in p.map$) ▷ new name exists in parent
15: ENSURE($reachable(root, i)$) ▷ new inode is reachable from the root
16: ENSURE(Invariants)

This operation obviously does not violate any invariant as it creates a file with: a single parent directory (Invariant 3), a unique inode number (Invariant 2), a single name which is unique (as a precondition; Invariant 4 holds) and is reflected in its `st_nlink` (Invariant 7), a reverse mapping (Invariant 5 holds); and this certainly does not change the root (Invariant 1) or any directory structure to create a cycle (Invariant 6 holds).

3.3.2 `mkdir`

The `mkdir` operation is very similar to `create`. It creates an empty directory. Creating a directory requires setting the type of the created inode as *directory* and performing other directory-specific tasks. In UNIX file systems, examples of such tasks usually include creating the default entries '.' and '..'. We omit this, considered to be implementation detail (Section 2.1) and we exclude it from this high level semantic model.

Algorithm 4 MKDIR create an empty directory

1: **procedure** MKDIR_GEN($path$, n, $stat$) ▷ inputs: parent directory, file name, stat
2: REQUIRE(Invariants)
3: **let** $p = \text{PATHRESOLUTION}(path)$
4: **if** $\neg(p \in \mathfrak{D} \wedge \nexists m \in p.map : m.name = n)$ **then** `ERROR` ▷ name already exist
5: **procedure** MKDIR_EFF(p, n, $stat$)
6: REQUIRE(Invariants)
7: REQUIRE($p \in \mathfrak{D} \wedge \nexists m \in p.map : m.name = n$) ▷ new name does not exist
8: **let** $i = \text{NEWINODE}$ ▷ create an empty inode
9: $i.stat \leftarrow stat$ ▷ assign the desired *stat* from the input
10: $i.type \leftarrow$ `DIR` ▷ set inode type to directory
11: $i.names \leftarrow \{(p.ino, n)\}$ ▷ add reverse mapping
12: $i.nlink \leftarrow 1$ ▷ update link count
13: $p.map \leftarrow p.map \cup \{(n, i.ino)\}$ ▷ mapping from parent
14: ENSURE($p \in \mathfrak{D} \wedge (n, i.ino) \in p.map$) ▷ new name exists in parent
15: ENSURE($reachable(root, i)$) ▷ new inode is reachable from the root
16: ENSURE(Invariants)

Similarly to `create`, this operation does not violate any of the invariants; it creates a directory with: a single parent directory, a single unique name (as a precondition), a reverse mapping; and this new directory does not connect to any other directory (have any child) to create a cycle.

3.3.3 `link`

The operation call `link` creates a hard link (an additional name) for an existing file. Algorithm 5 shows such procedure in our file system.

This operation does not violate any of the invariants {1, 2, 3, 4} because it does not change the root directory, nor create a new inode, nor link to a directory, nor use an existing name (checked in precondition); this operation updates the *names* of the target file to keep the coherency of the mapping (Invariants 5 and 7), and it does not change the file system tree structure to create directed cycle (Invariant 6).

Algorithm 5 LINK create a hard link to a file

1: **procedure** LINK_GEN($path1$, n, $path2$) ▷ inputs: parent path, file name, file path
2: REQUIRE(Invariants)
3: **let** $p = \text{PATHRESOLUTION}(path1)$
4: **let** $i = \text{PATHRESOLUTION}(path2)$
5: **if** $\neg(p \in \mathfrak{D} \wedge \nexists m \in p.map \wedge m.name = n)$ **then** `ERROR` ▷ name already exist
6: **if** $\neg(i \in \mathfrak{F})$ **then** `ERROR` ▷ input file does not exist
7: **procedure** LINK_EFF(p, n, i) ▷ inputs: parent directory, file name, file
8: REQUIRE(Invariants)
9: REQUIRE($p \in \mathfrak{D} \wedge \nexists m \in p.map : m.name = n$)
10: $p.map \leftarrow p.map \cup \{(n, i.ino)\}$ ▷ mapping from parent
11: $i.names \leftarrow i.names \cup \{(p.ino, n)\}$ ▷ reverse mapping
12: $i.nlink \leftarrow i.nlink + 1$ ▷ increase file's link count
13: ENSURE($p \in \mathfrak{D} \wedge (n, i.ino) \in p.map$) ▷ mapping exists in parent
14: ENSURE($i \in \mathfrak{F} \wedge (p.ino, n) \in i.names$) ▷ reverse mapping exists in file
15: ENSURE(Invariants)

3.3.4 `setattr`

The operation `setattr` is used by various attribute-related system calls such as `chmod`. It takes an inode and a *stat* object as input and then replaces the *stat* of the inode by the provided *stat* (we use the whole *stat* to represent the general case; actual implementations of each function such as `chmod` are more fine-grained and target security attributes only). Algorithm 6 describes our implementation.

Algorithm 6 SETATTR update the attributes of an inode

1: **procedure** SETATTR_GEN($path$, $stat$) ▷ inputs: inode path, attributes
2: REQUIRE(Invariants)
3: **let** $i = \text{PATHRESOLUTION}(path)$
4: **if** $\neg(i \in \mathfrak{I})$ **then** `ERROR` ▷ input inode is not valid
5: **procedure** SETATTR_EFF(i, $stat$) ▷ inputs: inode, attributes
6: REQUIRE(Invariants)
7: REQUIRE($i \in \mathfrak{I}$)
8: $i.stat \leftarrow stat$
9: ENSURE(Invariants)

The `setattr` operation only changes the *stat* of an inode. It can only change the attributes that that do not violate the invariants, for instance file type and link count cannot be changed manually. Therefore it does not violate any invariant.

3.3.5 `write`

We use the `write` function to abstract over a range of system calls that modify the data content of a file. This includes for instance the `write` to `flush` system calls. This operation takes an inode and new data as its inputs. Similarly to `setattr`, this operation replaces the `data` part of the inode by the argument (Algorithm 7). Real-world implementations optimize this in numerous ways depending on the context of the actual system calls, for example, it might replace only a range of data.

Algorithm 7 WRITE update the data of a file

```
1: procedure WRITE_GEN(path, data)                ▷ inputs: file path, data
2:     REQUIRE(Invariants)
3:     let i = PATHRESOLUTION(path)
4:     if ¬(i ∈ 𝔉) then ERROR                     ▷ input inode is not valid
5: procedure WRITE_EFF(i, data)                   ▷ inputs: inode, data
6:     REQUIRE(Invariants)
7:     REQUIRE(i ∈ 𝔉)
8:     i.data ← data
9:     ENSURE(Invariants)
```

Similarly to the case of `setattr`, this operation does not change the structure of the file system. Therefore it does not violate any invariant.

3.3.6 `unlink`

The `unlink` operation removes a name of a file, thus decreasing the file's `st_nlink` link count. If the `st_nlink` is 0 after this operation, the inode is not reachable and may be deleted (not represented in our model[1]). We describe this operation in Algorithm 8.

The operation `unlink` is the inverse of `link`. It does not violate any file system invariant: it removes a name of a file and reflects this in the *names* and its `st_nlink` of the file, as well as in the map of the file's parent directory.

3.3.7 `rmdir`

The operation `rmdir` is similar to `unlink`. It removes the name of an empty directory (which means a directory that has no children), and thus removes the directory because any directory has a single name only. As with the case of `unlink`, the inode deletion step of `rmdir` is not represented. We describe `rmdir` in Algorithm 9.

[1]The deletion step depends on actual file system implementations; it could be done immediately in the `unlink` process, or it could be delayed until some asynchronous garbage collection. In this thesis, we assume that this step is controlled by file system's garbage collector, thus omitting it from this `unlink` operation.

Algorithm 8 UNLINK delete a (name of) a file

1: **procedure** UNLINK_GEN($path1$, $path2$, n) ▷ parent path, file path, name
2: REQUIRE(Invariants)
3: **let** $p = \text{PATHRESOLUTION}(path1)$ ▷ parent directory
4: **let** $i = \text{PATHRESOLUTION}(path2)$ ▷ target file
5: **if** $\neg(p \in \mathfrak{D} \wedge (n, i.ino) \in p.map)$ **then** ERROR ▷ no such parent or no such name
6: **if** $\neg(i \in \mathfrak{F} \wedge (p.ino, n) \in i.names)$ **then** ERROR ▷ no such file or no such name
7: **procedure** UNLINK_EFF(p, i, n) ▷ parent, file, name
8: **let** $m = (n, i.ino)$ ▷ mapping
9: **let** $m' = (p.ino, n)$ ▷ reverse mapping
10: REQUIRE(Invariants)
11: REQUIRE($p \in \mathfrak{D} \wedge m \in p.map$) ▷ mapping exists in parent
12: REQUIRE($i \in \mathfrak{F} \wedge m' \in i.names$) ▷ reverse mapping exists in file
13: $p.map \leftarrow p.map \setminus \{m\}$ ▷ remove mapping from parent directory
14: $i.names \leftarrow i.names \setminus \{m'\}$ ▷ remove reverse mapping from target file
15: $i.nlink \leftarrow i.nlink - 1$ ▷ update target's link count
16: ENSURE($p \in \mathfrak{D} \wedge m \notin p.map$) ▷ mapping is removed from parent
17: ENSURE($i \in \mathfrak{F} \wedge m' \notin i.names$) ▷ reverse mapping is removed from file
18: ENSURE($\neg reachable(root, i) \iff i.nlink = 0$) ▷ deleted iff link count = 0
19: ENSURE(Invariants)

Algorithm 9 RMDIR delete a directory

1: **procedure** RMDIR_GEN($path1$, $path2$, n) ▷ parent path, directory path, name
2: REQUIRE(Invariants)
3: **let** $p = \text{PATHRESOLUTION}(path1)$ ▷ parent directory
4: **let** $i = \text{PATHRESOLUTION}(path2)$ ▷ target directory
5: **if** $\neg(p \in \mathfrak{D} \wedge (n, i.ino) \in p.map)$ **then** ERROR ▷ no such parent or no such name
6: **if** $\neg(i \in \mathfrak{D}^* \wedge (p.ino, n) \in i.names)$ **then** ERROR ▷ no such dir or no such name
7: **if** $\neg(|i.map| = 0)$ **then** ERROR ▷ target directory is not empty
8: **procedure** RMDIR_EFF(p, i, n) ▷ parent, directory, name
9: **let** $m = (n, i.ino)$ ▷ mapping
10: **let** $m' = (p.ino, n)$ ▷ reverse mapping
11: REQUIRE(Invariants)
12: REQUIRE($p \in \mathfrak{D} \wedge m \in p.map$) ▷ mapping exists in parent
13: REQUIRE($i \in \mathfrak{D}^* \wedge m' \in i.names$) ▷ reverse mapping exists in target
14: REQUIRE($|i.map| = 0$) ▷ target directory is empty
15: $p.map \leftarrow p.map \setminus \{m\}$ ▷ remove mapping from parent directory
16: $i.names \leftarrow i.names \setminus \{m'\}$ ▷ remove reverse mapping from target directory
17: $i.nlink \leftarrow i.nlink - 1$ ▷ update target's link count
18: ENSURE($p \in \mathfrak{D} \wedge m \notin p.map$) ▷ mapping is removed from parent
19: ENSURE($\neg reachable(root, i) \wedge i.nlink = 0$) ▷ target directory is removed
20: ENSURE(Invariants)

The operation `rmdir` may remove empty directories only. Therefore it does not change the file system structure. This operation does not violate any invariant.

3.3.8 `rename`

This is the most complex file system operation, because it is a transaction that involves up to 3 different inodes. The `rename` operation removes an old name of an inode and assigns a new name to it. It consists of elements of `link` and `unlink` or `rmdir`. As the names may reside in different parent directories, this operation modifies up to 3 inodes.

Renaming a directory must not create a directory cycle. This is ensured by a precondition that rejects any `rename` that makes the directory a descendant of itself (lines 9 and 18 of Algorithm 10).

Real-world implementations of this operation may also differ from case to case. For example, some cause the operation to fail if the target name exists, while others overwrite the target. In this thesis, we choose the first approach, as it is simpler and is more in line with operations like `create`. Algorithms 10 and 11 describe the algorithms for renaming a directory and a file, respectively. These algorithms take the paths to the old and new parents and the old and new names as their inputs.

These algorithms ensure Invariant 1 and Invariant 2 as they do not rename the root or create a new inode. They preserve Invariant 6 because the check for reachability from i to $p2$ would rule out the possibility of having a directory cycle. They also include the checks for Invariant 3 and Invariant 4 in the precondition; they ensure Invariant 5 by updating the *names* of the inode and the child maps of the parent directories.

3.3.9 Other Operations

There are many other file system operations that we did not mention to keep the list of operations simple and representative. These include for instance `mknod`, `symlink`, `truncate`, `append`, and `flush`.

The `mknod` operation creates files of special types such as FIFO and devices; these are less-frequently-used types of file; the only difference with normal files is the file type (in *stat*'s `st_mode`). The `mknod` operation is therefore substantially identical to `create`. Therefore, we ignore `mknod`.

Almost the same thing goes for `symlink`. This operation creates an inode that stores a symbolic link (a string) to another inode as its data. Therefore `symlink` is similar to the combination of `create` and `write`, which have already been described.

Other operations, such as `truncate`, `append`, and `flush`, are engineering optimizations of `write`. The operation `truncate` either removes or appends some data at the

Algorithm 10 RENAME_DIR rename a directory

1: **procedure** RENAME_DIR_GEN($path_S$, n_S, $path_D$, n_D) ▷ src/dst parents/names
2: REQUIRE(Invariants)
3: **let** $p_S = \text{PATHRESOLUTION}(path_S)$ ▷ source parent directory
4: **let** $p_D = \text{PATHRESOLUTION}(path_D)$ ▷ destination parent directory
5: **if** $\neg(p_S \in \mathfrak{D} \wedge \exists m \in p_S.map : m.name = n_S)$ **then** `ERROR` ▷ no such name
6: **if** $\neg(p_D \in \mathfrak{D} \wedge \nexists m \in p_D.map : m.name = n_D)$ **then** `ERROR` ▷ new name exists
7: **let** $m = m \in p_S.map : m.name = n_S$
8: **let** $i = \text{LOOKUP}(m.ino)$ ▷ directory to rename
9: **if** $reachable(i, p_D)$ **then** `ERROR` ▷ making a cycle
10: **procedure** RENAME_DIR_EFF(p_S, n_S, p_D, n_D, i)
11: **let** $m_S = (n_S, i.ino)$ ▷ old mapping
12: **let** $m_D = (n_D, i.ino)$ ▷ new mapping
13: **let** $m'_S = (p_S.ino, n_S)$ ▷ old reverse mapping
14: **let** $m'_D = (p_D.ino, n_D)$ ▷ new reverse mapping
15: REQUIRE(Invariants)
16: REQUIRE($p_S \in \mathfrak{D} \wedge m_S \in p_S.map$) ▷ old name exists
17: REQUIRE($p_D \in \mathfrak{D} \wedge \nexists m \in p_D.map : m.name = n_D$) ▷ new name does not exist
18: REQUIRE($\neg reachable(i, p_D)$) ▷ not creating a cycle
19: $p_S.map \leftarrow p_S.map \setminus \{m_S\}$ ▷ remove old mapping from source
20: $p_D.map \leftarrow p_D.map \cup \{m_D\}$ ▷ add new mapping to target
21: $i.names \leftarrow i.names \setminus \{m'_S\}$ ▷ remove old name
22: $i.names \leftarrow i.names \cup \{m'_D\}$ ▷ add new name
23: ENSURE($reachable(root, i)$) ▷ renamed directory is reachable
24: ENSURE($m_S \notin p_S.map \wedge m_D \in p_D.map$) ▷ mappings updated in parents
25: ENSURE($m'_S \notin i.names \wedge m'_D \in i.names$) ▷ names updated in the directory
26: ENSURE(Invariants)

Algorithm 11 RENAME_FILE rename a file

1: **procedure** RENAME_FILE_GEN($path_S$, n_S, $path_D$, n_D) ▷ src/dst parents/names
2: REQUIRE(Invariants)
3: **let** $p_S = \text{PATHRESOLUTION}(path_S)$ ▷ source parent directory
4: **let** $p_D = \text{PATHRESOLUTION}(path_D)$ ▷ destination parent directory
5: **if** $\neg(p_S \in \mathfrak{D} \wedge \exists m \in p_S.map : m.name = n_S)$ **then** `ERROR` ▷ no such name
6: **if** $\neg(p_D \in \mathfrak{D} \wedge \nexists m \in p_D.map : m.name = n_D)$ **then** `ERROR` ▷ new name exists
7: **let** $m = m \in p_S.map : m.name = n_S$
8: **let** $i = \text{LOOKUP}(m.ino)$ ▷ file to rename
9: **procedure** RENAME_FILE_EFF(p_S, n_S, p_D, n_D, i)
10: **let** $m_S = (n_S, i.ino)$ ▷ old mapping
11: **let** $m_D = (n_D, i.ino)$ ▷ new mapping
12: **let** $m'_S = (p_S.ino, n_S)$ ▷ old reverse mapping
13: **let** $m'_D = (p_D.ino, n_D)$ ▷ new reverse mapping
14: REQUIRE(Invariants)
15: REQUIRE($p_S \in \mathfrak{D} \wedge m_S \in p_S.map$) ▷ old name exists
16: REQUIRE($p_D \in \mathfrak{D} \wedge \nexists m \in p_D.map : m.name = n_D$) ▷ new name does not exist
17: $p_S.map \leftarrow p_S.map \setminus \{m_S\}$ ▷ remove old mapping from source
18: $p_D.map \leftarrow p_D.map \cup \{m_D\}$ ▷ add new mapping to destination
19: $i.names \leftarrow i.names \setminus \{m'_S\}$ ▷ remove old name from file
20: $i.names \leftarrow i.names \cup \{m'_D\}$ ▷ add new name to file
21: ENSURE($reachable(root, i)$) ▷ renamed file is reachable
22: ENSURE($m_S \notin p_S.map \wedge m_D \in p_D.map$) ▷ mappings updated in parents
23: ENSURE($m'_S \notin i.names \wedge m'_D \in i.names$) ▷ names udpated in the file
24: ENSURE(Invariants)

end of the *data* of a file; this is a special case of `write`. The operation `append` appends data to the end of a file, instead of replacing the whole *data* part. This means a lot for performance, but is conceptually identical to `write`. The operation `flush` is another optimization of `write` that has no observable side effects unless hardware failure occurs. Therefore, we can also ignore it.

Part II

Concurrency Semantics

Chapter 4

Concurrency Model

In this chapter, we describe the topic of replication in distributed file systems, which provides the context of concurrency, and then we present our concurrency model for achieving high availability and maintaining sequential semantics in such systems.

4.1 Replication in Distributed File Systems

Large-scale distributed file systems are usually replicated at multiple locations to improve availability. A single logical distributed file system is composed of separate local file system instances, called file system replicas. In a so-called "multi-master" setting, each replica can accept local updates. Replicas may or may not synchronize their updates with each other; in either case, an update accepted at some replica is transmitted and replicated at all other replicas. Consistency is the problem of keeping replicas up to date with each other, so that users can see the same file system contents across replicas.

In order to tolerate network latency between replicas and failures, many distributed file systems enforce a weak form of consistency. In such a system, a replica commits an update locally without coordination, and later transmits its locally committed updates asynchronously to the other replicas. Conversely, a replica receiving a remote update applies the update to its own state. In this way, replicas may diverge temporarily, but converge eventually. This eventual consistency (EC) approach ensures low-latency operations, and system availability.

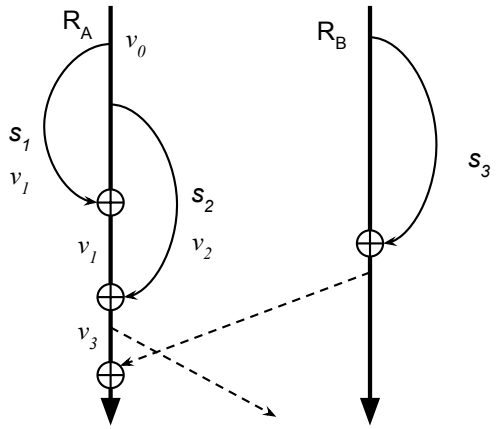


Figure 4.1: Our model for managing concurrency in distributed file systems. A thick line represents the state of the local file system of a replica. A dashed line denotes update propagation between replicas. A curve represents an independent session, in a fork-join model. A crossed circle represents a joining/merging point of a local or a remote session into the local state.

4.2 Concurrency Model

The general eventual consistency model allows update transmission to be scheduled in an arbitrary way. Similarly to the TouchDevelop model [7], we restrict concurrency to a fork-join model per replica; however, in contrast with TouchDevelop, we allow concurrency between replicas, and we do not allow fork off a session to keep the model simple. Our combined fork-join and parallel model is illustrated in Figure 4.1. The parallelism between replicas ensures low-latency updates, inspired by the eventual consistency approach. The fork-join model restriction maintains the traditional sequential semantics despite concurrency.

Replicas execute in parallel; each replica has its own independent state, and performs its updates independently. From time to time, a replica transmits recent updates to the other replicas. Upon receiving an update message from another replica, the receiving replica will atomically deliver the received update into its own state. Updates are delivered in causal order; in particular, they are delivered to remote replicas in the order in which they occurred.

We use the fork-join model to control concurrency inside a replica. An application works in an isolated environment, called a *session*. A session starts by making a consistent snapshot copy (called a fork) of the state of the local replica, containing all the updates of the local and remote sessions previously committed to the current replica. Two sessions of a replica are isolated and independent from each other: the updates of one session will be available only to sessions that fork after the current session

commits (joins). Committing a session atomically merges all the updates of that session into the state of the local replica. Committing a session or receiving remote updates may reveal conflicts between concurrent updates; we will describe conflicts later, in Chapters 6 and 7. At arbitrary points in time (in the implementation: after every join) a replica transmits its locally-committed updates to the other available replicas, in an asynchronous manner. When receiving the updates of a remote session, the receiving replica delivers the received updates atomically. Conflicts are handled in the same way as for local sessions.

Figure 4.1 illustrates the session concept. On replica R_A, session s_1 obtained a snapshot of R_A in state v_0. Applications in s_1 update the snapshot to state v_1. While s_1 was running, s_2 started and made a snapshot of R_A; because there were no updates committed in the interval, the initial content of the s_2's snapshot is identical to v_0. Thus, the updates from s_1 remained isolated. When s_1 finishes and successfully commits its updates, as there were no commits in the interval, the state of R_A becomes v_1. A session started after this point will have contents v_1. When s_2 finishes, it commits its state v_2 into R_A as well. R_A resolves the concurrency between the updates of s_1 and s_2 to produce a state $v_3 = v_1 \oplus v_2$ that merges updates of s_1 and s_2.[1] Similarly, when R_A receives updates from R_B, it merges these concurrent updates with $\oplus$.

The fork-join model provides some compatibility with the traditional sequential semantics by ensuring that updates from concurrent sessions become visible in a controlled manner. Indeed, a session works in isolation (updates from one are not available to another session and thus do not interfere with it). This model ensures that interference from concurrent sessions happens only at the boundaries between sessions. This model also supports parallelism between replicas.

[1] Notation $\oplus$ represents the concurrency resolution algorithm that we will describe in Chapter 7.

Chapter 5

Sessions

A session is similar to a long transaction in traditional DBMSes under the control of user applications. We provide the following interface for managing sessions.

- `start_session(mount_point):session_id`
- `connect_session(mount_point, session_id)`
- `commit_session(session_id)`

5.1 `start_session`

The function `start_session` starts a new session. When receiving this request, the local replica takes a snapshot of its file system state, associates the snapshot with a unique session identifier called `session_id`, and returns the session identifier. Only users and applications associated with this session can *access* that snapshot; updates from a snapshot are not visible except to the processes that belong to the session.

Associating a process to a session in a real-world implementation can be any mechanism that distinguishes between sessions. Our implementation (Figure 5.1) uses the Unix concept of a *mount point*[1] as the mean of access to sessions. A client (a PC for example) of our system must have a mount point on its own facility. A client can start a session by querying the local replica using `start_session`. Upon receiving this request, the replica takes a snapshot of its state, associates that snapshot with a `session_id` and returns the snapshot and the `session_id` to the client. The client then mounts the snapshot into its mount point. Thereafter, accesses to the mount point will access and modify the snapshot.

The *access* to a session is the key for an application to manage its interaction with

[1] A mount point is a directory (typically an empty one) in the currently accessible file system on which an additional file system is mounted (i.e., logically attached) [33].

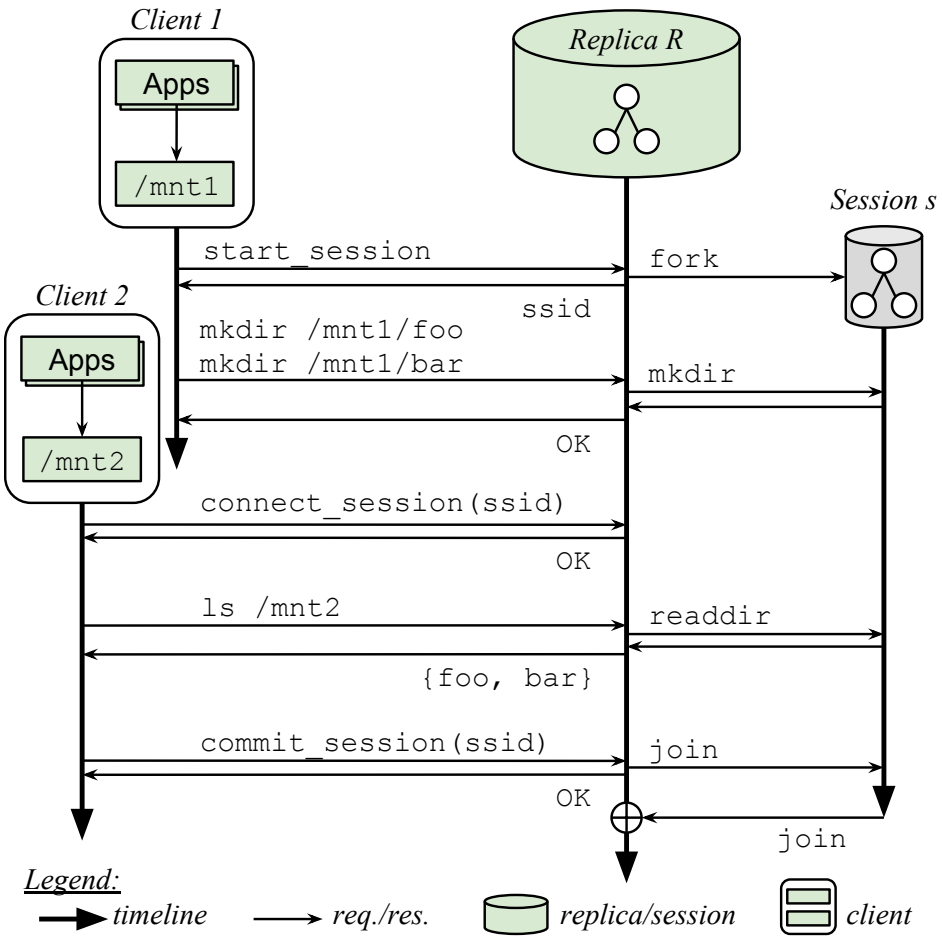


Figure 5.1: An example of our implementation of session. *Client 1* first starts a session with identifier `ssid` on its mount point `/mnt1`, and then creates two directories `/mnt1/foo` and `/mnt1/bar`. *Client 2* later connects to the session through its mount point `/mnt2`; *Client 2* was able to observe both `foo` and `bar` in `/mnt2`.

that session. Generally, by having the *access*, the application can interact with the session, and by giving up the *access*, the application will not be able to work in that session. Depending on specific implementation, how to obtain and give up the *access* to a session will be determined. For example in our implementation, accessing a specific mount point will grant an application the *access* to the associated session; by not going through that mount point, an application gives up the *access* to the associated session.

5.2 `connect_session`

The function `connect_session` requests the local replica to grant the caller an *access* to an existing session specified by a session identifier. If the session exists and the

access is granted, the caller will be able to join existing users/applications to work on the same snapshot of the file system, which means the caller will be able to observe all updates on the session and to experience the sequential semantics and isolation guarantees.

In our implementation, a client can create a mount point on its facility and use `connect_session` to connect to an existing session through that mount point. Because of that, there may be multiple mount points (on the same system or not) connecting to the same session; users/applications can see the same file system contents, as well as experiencing the same sequential semantics, through these different mount points.

5.3 commit_session

The function `commit_session` signals the local replica to merge a session into the replica's state; this corresponds to the 'join' part in the fork-join model. The replica will revoke all access to the snapshot of the session, and atomically[2] commit all updates made to the snapshot into the replica's state. Operations on a finished session are an error. In our implementation, `commit_session` makes the local replica to unmount all existing mount points of a session, then commits the updates to the session's snapshot to the state of the replica.

A replica merges all of its sessions sequentially; it needs to fully commit all updates of one session to its state, before it can commit the other sessions.

Merging a session received from a remote replica is similar. The receiving replica atomically merges all updates of the remote session.

[2]Atomicity: at any point in time, either all of the updates from a session are available in the replica's state or none are.

Chapter 6

Conflicts

Concurrent sessions, either within a replica or between replicas, may update inodes in ways that conflict. In this chapter, we study the conflict cases. Conflict resolution will be described in the next chapter, but we will also make some references to it in this chapter.

6.1 Issues Of Concurrency

In the sequential file system of Chapter 2, a sequential update has a precondition that preserves the file system invariants. This is what ensures the correctness of the file system. However, in a concurrent system, the concurrency between updates may not ensure file system correctness or replica convergence if we apply the same sequential effector algorithms of the updates in Chapter 3. For example, concurrently changing the name of the same directory could violate Invariant 3 (a directory must have a single name). Concurrently updating the *data* of the same file does not directly violate an invariant, but applying these concurrent updates in different orders using the sequential effector algorithms could yield different results, thus causing the replicas to diverge.

To ensure convergence and correctness, conflicts must be resolved. However, a simple conflict resolution approach may not be 'meaningful' to users, making it inappropriate. For example, the Last Writer Wins (LWW) approach picks an arbitrary update out of multiple concurrent ones, and drops the others. Although the LWW approach converges the replicas, lost updates may not be the effect desired by users.

In the next sections of this chapter, we will: (1) define conflicts, and (2) enumerate the conflict cases of our file system design.

6.2 Conflict Definition

The correctness of our distributed file system is described by three safety rules: replica convergence, invariant preservation, and names-data coherency.

The first rule allows replicas to diverge for a while, as long as they ultimately converge. The second rule requires that every replica individually maintains the sequential invariants, described in Chapter 2.

And the last rule is our own definition of conflict that has not been covered by the previous rules: the situation when concurrent updates target the *names* and *data* part of an inode is a conflict. We will describe the rationale of this rule in the next chapter.

Any pair of concurrent updates that violates any of these safety rules constitutes a conflict.

Rule 1 (Replica Convergence) *Replicas that have delivered the same updates have the same state. A replica is required to eventually deliver all updates that others have delivered. Updates may be applied in any order consistent with their partial order.*

Rule 2 (System Integrity) *At every replica, and at every point in time, the sequential invariants must be true.*

Rule 3 (Names-Data Coherency) *Concurrent updates to the names and data parts of an inode constitute a conflict.*

6.3 Types Of Conflict

With the exception of Invariant 6, by iterating through the above safety rules and by inspection of the invariants, we can see that to violate any of them, concurrent operations must target the same inodes. For example to create multiple names for a directory (violating Invariant 3), concurrent operations would have to update the *names* part of that directory; to create different children with the same name in a directory (violating Invariant 4), concurrent operations would have to update the map (*data* part) of that directory. We therefore call this type of conflict *direct conflict*; this is the case when concurrent updates target the same inodes and violate the defined safety rules. The other type, if exists, is called *indirect conflict*; this is the cases when concurrent updates violate Invariant 6 even though they are not targeting the same inode. The next sections of this chapter will describe these types of conflict.

To simplify the analysis, we treat the *stat* and *data* parts of an inode as if they were a single piece of data; this means that updating the *stat* of an inode is equivalent to updating the *data* of that inode.

Table 6.1: Common direct concurrency cases for all inode types.

any inode	delete	update
delete	*delete-delete*	*delete-update*
update	*delete-update*	*update-update cases* (see Tables 6.2 and 6.3)

6.4 Direct Conflict Cases

A direct conflict is one caused by concurrent operations that target the same inode. In this section, we enumerate such cases according to the type of the target inode.

6.4.1 Common Concurrency Cases of All Inode Types

These are the concurrency cases in which we consider that the same resolution can apply whether the target inode is a directory or a file (Table 6.1).

delete-delete. This is the situation when concurrent sessions both delete the same inode. Because both replicas agree on the same value of the inode, this concurrency case trivially converges without violating any of the safety rules.

delete-update. This case happens when a session deletes some inode and concurrently another session updates the same inode. Because applying the concurrent updates in different orders using the respective sequential effector algorithms (Chapter 3) would not generate the same results (thus violating the rule of convergence—Rule 1), this case is a conflict and requires conflict resolution; we call this a *state conflict*.

update-update. When neither concurrent update is a deletion in Table 6.1), our analysis depends on the type of the targeted inode. In the next sections, we will describe the details of these cases for each type of inode.

6.4.2 Update-Update Concurrency On Files

Table 6.2 describes the concurrency cases where two sessions concurrently update the same file. We go through the detail of each case below.

File data-data conflict. When updates concurrently change the *data* part of the same file, this does not violate any of the invariants (Rule 2), however, applying them in different orders (using the effector algorithms in Chapter 3) could yield different results (thus violating the rule of convergence—Rule 1). If the file contains a data type whose updates are commutative, such as a CRDT [44, 50, 58, 78], concurrent updates could be merged according to the rules of that data types and converge; in this case,

Table 6.2: Direct update-update concurrency cases when the target is a file.

file	data	names
data	*file data-data conflict*	*file names-data conflict*
names	*file names-data conflict*	*file names-names conflict*

the concurrent updates are not in conflict. However, in general, the type of file contents is opaque to the file system, and a file might contain anything. Thus, for generality, we ignore such commutative data types. Our conflict resolution approach will be described in Section 7.2.2.

File names-data conflict. When an update changes the *names* and concurrently another update changes the *data* parts of the same file, these updates break the relationship between the name and the corresponding file contents as expected by the users (violating our rule of *names-data* coherency—Rule 3). The conflict resolution for this case will also be described in Section 7.2.2.

File names-names conflict. When concurrent updates change the *names* part of a file, they may create the same name and thus violate Invariant 4 (a name is unique in the map of a directory). Our solution is presented in Section 7.2.3.

6.4.3 Update-Update Concurrency On Directories

We describe in Table 6.3 the direct concurrency cases when the target inode is a directory. We go through the detail of each case below.

Directory data-data conflict. Whereas the *data* part of a file is opaque to the file system, for a directory, it consists of the child map. If we implement it using a commutative data type, then concurrent updates can be merged.[1]

Merging the concurrent updates in this case triggers a recursive merge of the directory's children (mapping entries).

Conflicts happen when these updates target the same mapping entries, or there are different mapping entries with the same name; this is a conflict because it violates Invariant 4 (a name is unique in a map). We present our resolution for this conflict case in Section 7.2.4.

Directory names-data conflict. Concurrent updates in this case break the relationship between different names and different directory contents, violating Rule 3. We describe our resolution for this case in Section 7.2.5.

[1]Recall that in Section 6.3 we consider the *stat* of an inode a part of its *data*; merging the *data* parts of directories therefore involves merging their *stat*s. The algorithm for merging directory *stat*s is described in Appendix A.

Table 6.3: Direct update-update concurrency cases when the target is a directory.

directory	data	names
data	*dir. data-data conflict*	*dir. names-data conflict*
names	*dir. names-data conflict*	*dir. names-names conflict*

Directory names-names conflict. Concurrent updates to the *names* part of a directory might result in multiple names for the same directory, violating Invariant 3 (a directory has a single name). As we explain and justify in Section 7.2.5, our conflict resolution is to copy the directory in order to maintain all of the updated mappings.

6.5 Indirect Conflict Case

In this section, we present our definition of indirect conflict and our method for detecting this case.

6.5.1 Indirect Conflict Definition

By iterating through the invariants, we can see that concurrent operations that target the same inodes are not able to violate Invariant 6 (no cycles). In order to do that (violating the invariant), updates would have to transitively impact each other in a way that creates a directory cycle. We describe the conditions for concurrent updates to violate Invariant 6, and then our definition of indirect conflict as below.

Formally, consider I_1 and I_2 as the sets of inodes updated by operations op_1 and op_2, respectively. The condition for them to violate Invariant 6 is as below.

$$\begin{cases} I_1 \cap I_2 = \varnothing \\ \exists i_1 \in I_1, i_2 \in I_2 : \mathit{reachable}(i_1, i_2) \wedge \mathit{reachable}(i_2, i_1) \end{cases} . \tag{6.1}$$

The above requirements specify that concurrent updates, even though they do not necessarily target the same inodes, do in fact create a directory cycle.

This is easy to achieve; Figure 6.1 shows an example of creating a directory cycle. Suppose that there initially exist directories 1, 2, 3, 4, 5, and 6 such that:

$$\begin{cases} \mathit{reachable}(1,3) \wedge \mathit{reachable}(3,5) \\ \mathit{reachable}(2,4) \wedge \mathit{reachable}(4,6) \\ \neg(\mathit{reachable}(1,2) \vee \mathit{reachable}(2,1)) \end{cases} . \tag{6.2}$$

Then, the following concurrent operations are valid on separate replicas whose states are as described in Condition 6.2 above: $op_1 = \text{RENAME}(3, 6)$, corresponding to the command `mv A/foo B/bar/quz` in the figure, and $op_2 = \text{RENAME}(4, 5)$, corresponding to the command `mv B/bar A/foo/qux` in the figure (inputs of op_1 and op_2: inode to rename, target directory; we omit the other arguments). The sets of updated inodes of respective operations then become: $I_1 = \{1, 3, 6\}$, $I_2 = \{2, 4, 5\}$. We have the following:

$$\begin{cases} I_1 \cap I_2 = \varnothing \\ \mathit{reachable}(4, 3) \quad \text{result of: } \text{RENAME}(3, 6) \\ \mathit{reachable}(3, 4) \quad \text{result of: } \text{RENAME}(4, 5) \end{cases}. \tag{6.3}$$

This clearly shows that there is a directed cycle between 3 and 4 in the merging result; this violates Invariant 6 and may result in data lost. However it cannot be easily detected by direct inspection, as the concurrent operations op_1 and op_2 target different sets of inodes.

Therefore we define an indirect conflict as following: an indirect conflict happens when concurrent updates, even those not targeting the same inodes, modify the tree structure of the file system, thus causing directory cycles and violating Invariant 6.

Our conjecture is that concurrent `rename`s is the only situation that might create directory cycles. According to Condition 6.1, concurrent updates have to transitively impact each other; this can only be achieved with `rename` because it is the only operation that transitively updates the sub-tree of a directory when renaming that directory.

6.5.2 Indirect Conflict Detection

In the case of an indirect conflict, the concurrent updates that cause the conflict do not target the same inodes, making it not possible to detect the conflict based on the concurrency on individual inodes. As these concurrent updates transitively modify the path of each other's target inodes (Condition 6.1), our intuition is that an indirect conflict can be identified by checking if concurrent updates modify the path of each other's target inodes.

In order to do this, we mark all ancestors of an updated inode as updated. We call this 'back-propagation' of updates. For example, when the file `/foo/bar/qux` is updated, all its ancestor directories `/`, `foo`, and `bar` are marked as updated, even though there were no direct modifications to these inodes.

An update back-propagation from a child to a parent directory has the same effect as if the parent's *names* part and the mapping entry to the child were updated. This

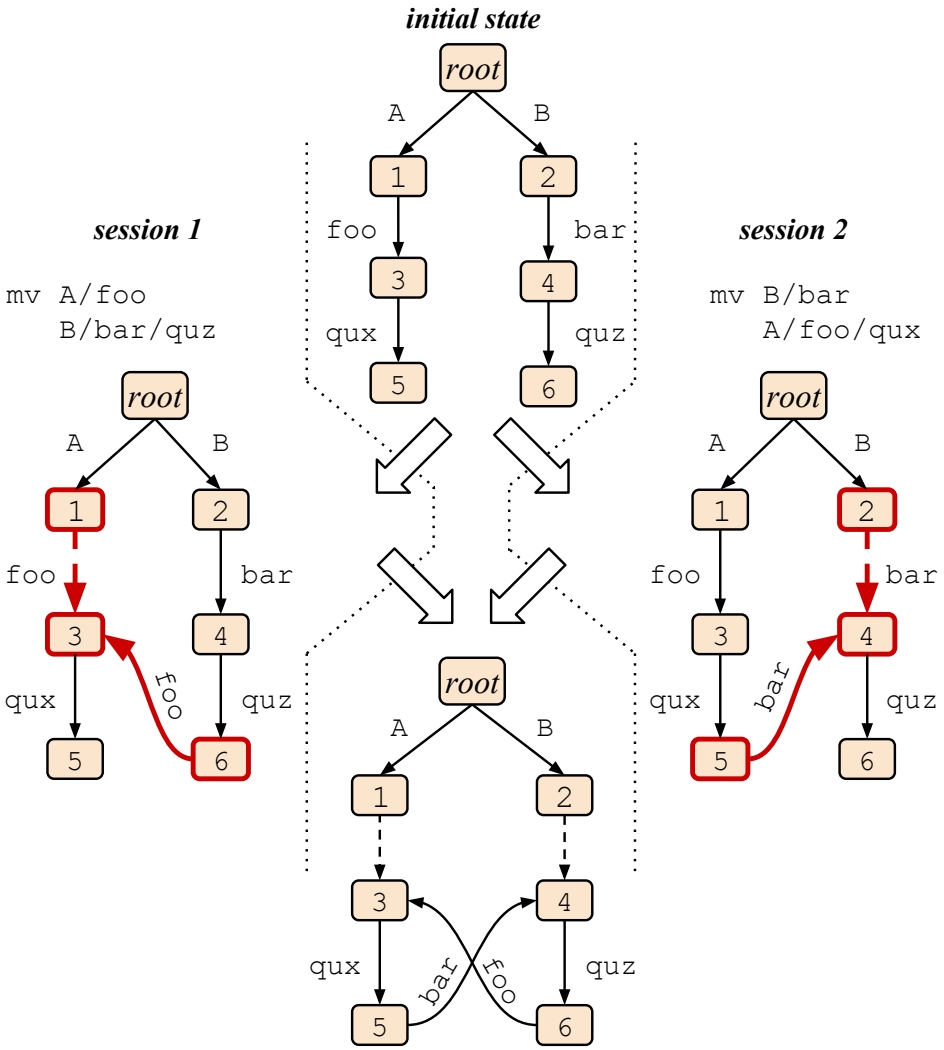


Figure 6.1: An example of indirect conflict created by concurrent `rename` operations. Dashed arrows are deleted mappings; bold-red shapes are those updated.

means it may cause a conflict with another concurrent update on the parent directory; however, concurrent back-propagations on the same directory do not cause any conflict because they do not actually change that directory.

With this approach, we are able to detect indirect conflicts systematically. Indeed, the Condition 6.1 indicates that to make a cycle between directories i_1 and i_2, each of these directories must be updated in a concurrent operation, such that one becomes an ancestor of the other. By using update back-propagation, these directories become updated in both sessions, thus we can easily detect the direct conflict on them.

Applying update back-propagation to the example in Figure 6.1, we can see that 2 and 4 were updated when updating 6 (by RENAME(3, 6)), and 1 and 3 were up-

dated when updating 5 (by RENAME(4, 5)). Combining the updated inodes of these concurrent `rename`s, we can clearly see the direct conflicts on {1, 3, 2, 4} representing the indirect conflict; this would otherwise have not been identified without update back-propagation.

Chapter 7

Conflict Resolution

In this chapter, we describe our conflict resolution for the conflict cases from the previous chapter. This high level description shows how we resolve conflicts with respect to system integrity (Rule 2) and inode names-data coherency (Rule 3). Recall that a conflict is a violation of the safety rules that include the aforementioned rules and the rule of replica convergence (Rule 1). We defer discussion of conflicts with respect to the latter to the next chapter.

7.1 Conflict Resolution Principles

Resolving a conflict means presenting the effects of the concurrent updates in a way that does not violate the safety rules. Technically, any approach that preserves system integrity and correctness would be safe, including losing all updates. Of course, we prefer solutions that enhance system liveness and user experience. Accordingly, we propose the following principles for conflict resolution.

Principle 1 (No Lost Updates) *Conflict resolution should preserve the effects of all updates.*

Clearly, arbitrarily dropping updates is undesirable, but it does occur in approaches such as Last-Writer-Wins (LWW), which resolves a conflict by choosing one of the updates and dropping the others. We decompose the No-Lost-Updates (NLU) principle into: (1) preserving updated file data contents, and (2) preserving the paths of updated directories and parent directories of updated files. The latter rule implies, for instance, that if an updated directory had a path p, there must be a corresponding directory at the path p after conflict resolution. This enables users to see their updated contents at

the same path and to not surprise them. This is the rationale for the inode names-data coherency rule (Rule 3) of the previous chapter.

Principle 2 (No Ghost Updates) *Conflict resolution should not make up updates that are not requested explicitly by users.*

This principle restricts conflict resolution from creating new updates out of thin air. For example, resolving a conflict on a file `/foo` should not create an unrelated directory `/bar` as part of the result.

Both of these principles are reasonable, but it is impossible to follow them rigidly. In fact, we show in this chapter that there are situations where this either would result in violating the safety rules of Chapter 6, or (where applying both simultaneously) would violate the file system semantics of Chapters 2 and 3. For example, preserving two concurrent updates to a same file requires either keeping both versions inside that file, which is not supported by POSIX file system APIs, or to store them in two different files (with different names), which violates the No-Ghost-Updates (NGU) principle.

Our design decision is, if there are no other options, to favor the NLU principle over NGU principle. For instance, in the previous example, we create new files to store the concurrent updates. Although for directories, concurrent updates can be merged without violating either principle.

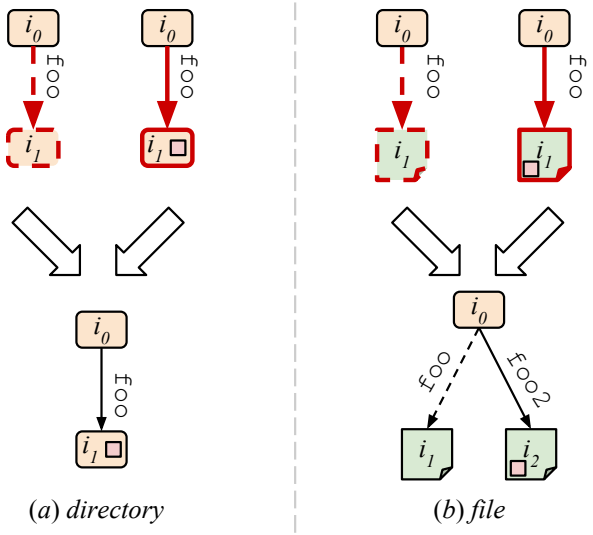
We also relax the NLU principle in some cases where the concurrent updates are contradicting with each other. For example, when an inode is concurrently both deleted and updated, it is impossible to keep both, and we preserve the update and ignore the deletion.

In the next sections, we will detail conflict resolution for the direct and indirect conflict cases of Chapter 6. Two main details will be presented: conflict resolution is a recursive process starting from the root directory and going down the tree; and resolving concurrent updates on an inode may require generating new inodes to store these updates.

7.2 Conflict Resolution For Direct Conflicts

This section describes the conflict resolution for the direct conflict cases discussed in Chapter 6. These include the *delete-update conflict*, *data-data conflict*, *names-data conflict*, and *names-names conflict* for inodes of any type.

The next sections will discuss the conflict resolution algorithms for *delete-update conflict*, *data-data conflict*, *names-data conflict*, and *names-names conflict*. The algorithm for a *file names-data conflict* is similar to that for a *file data-data conflict*. This



(*a*) *directory* (*b*) *file*

Figure 7.1: Resolving inode delete-update conflict for each type of the target inode.

is because these cases both map different names of an inode to different data contents (violating Rule 3), thus these cases can be resolved in the same way. Similarly, we apply the same conflict resolution algorithm for *directory data-names conflict* and for *directory names-names conflict*.

7.2.1 Delete-Update Conflict

Recall that a *delete-update* conflict happens when concurrently deleting and updating the same inode. Our resolution algorithm depends on the type of the inode, but generally, we preserve the update and ignore the deletion, thus violating NLU. Intuitively, this is because users can manually roll back an update if this is not the desired conflict resolution outcome, but if a deletion takes place, there is no direct way to roll it back. The conflict resolution for each type of inode is as below.

When the target inode is a directory, we preserve the updated directory and use it as the merging result, in order to maintain the updated path, following NLU (Fig. 7.1a). This helps maintaining a familiar structure of the file system through conflict resolution, following the names-data coherency rule (Rule 3).

When the target inode is a file, we preserve the updated contents in a new file, and delete the original one (Fig. 7.1b) favoring NLU over NGU. Chapter 8 will describe how we generate the new file's name and new inode number. We do not keep the update in the original file, in order to unify the conflict resolution mechanism with the *file data-data conflict*.

7.2.2 File Data-Data Conflict

A file data conflict happens when concurrent sessions update the *data* part of the same file. Our conflict resolution algorithm keeps both updates following the NLU principle.

Some existing approaches that preserve both updates in the context of a distributed file system are: (1) merging the contents, (2) keeping the contents as different versions, and (3) keeping the contents in new files.

The first approach (Fig. 7.2a) merges the updated contents together. This is applicable when the data type contained in the file is is known to be mergeable; in fact, this is our approach for directories. A shared document represented by a sequence of characters is an example that has been extensively studied for collaborative text editors, such as Google GSuite Colaborate [24], Microsoft Office Online [36], Dropbox Paper [14], or similar academic systems [44, 50, 58, 78]. This approach is not general, however, since not all file types are mergeable.

The second approach (Fig. 7.2b) is to store the updates in concurrent versions (or branches) of the file. Then users can access these versions or merge them manually. Version control systems, such as Git [42] or SVN [2], extensively use this approach to manage collaborative source code (text-based files) development. This solution is also the approach of versioning file systems [12, 34, 38] that keep multiple versions of a file. This approach however relies mostly on manual intervention from users to resolve conflicts, which is not scalable.

The last approach (Fig. 7.2c) is to create new files with different names, in order to keep the different updated contents, while preserving Invariant 4 (a name is unique in the map of a directory). This breaks the NGU principle, but this is necessary to maintain the NLU principle with a file system that does not support versioning. The names of these new files are chosen to make users aware of the conflict and to resolve that manually if necessary. A simple approach is to append a unique identifier to the original names. For example, we use `foo.A` and `foo.B` to represent the concurrent updates to `foo` from sessions A and B, respectively. Chapter 8 describes the details of our format for new file names, motivated by the convergence requirement.

The last approach however violates the NGU principle since it creates new files that do not correspond directly to a user request. We will discuss the issues with this approach in Section 8.3.

We chose this approach because it works with any file type and of its compatibility with the traditional POSIX semantics.

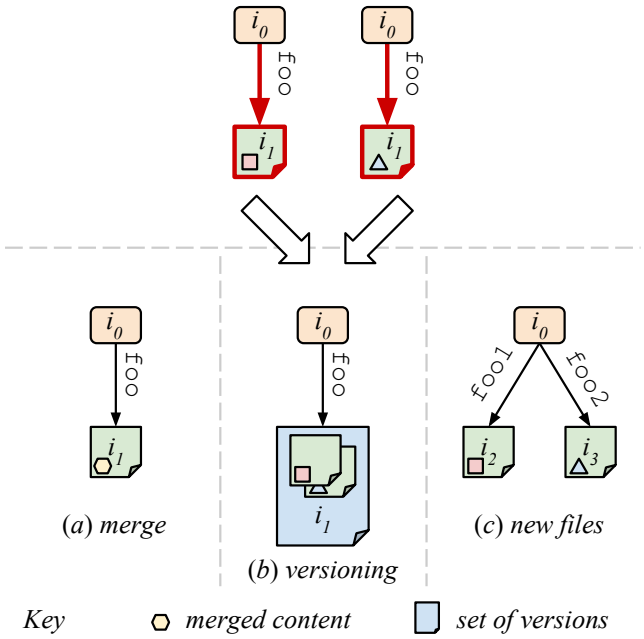


Figure 7.2: Resolving file data-data conflict with different approaches.

7.2.3 File Names-Names Conflict

This is the situation where concurrent sessions update only the *names* part of the same file. Because the data type of the *names* part of an inode is a map, just like the *data* part (child map) of a directory, we therefore can merge the concurrent updates on this part of a file together.

When a name of a file is concurrently updated (Figure 7.3), we store the concurrent updates to the name in new names to preserve these updates. For example, when concurrent sessions A and B create the same name `/bar` for the same inode, resolving this conflict results in new names `/bar.A` and `/bar.B` to preserve the updates of A and B, respectively. Creating new names in this case follows the same mechanism of creating new names for file data-data conflicts (Section 7.2.2).

7.2.4 Directory Data-Data Conflict

Concurrent updates to the child map (the *data* part) of a directory are merged together because its data type (a map) is known to be mergeable. Recall that the map of a directory maps local names to inodes; merging takes the union of the updated mapping entries of the concurrent updates.

Merging the concurrent updates in this case is a recursive process because conflicts might happen on the mapping entries of that directory. Conflicts happen when concur-

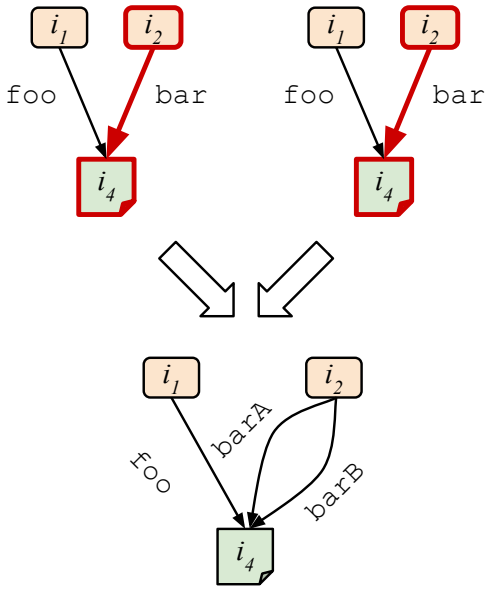


Figure 7.3: Merging semantics for file naming conflicts.

rent updates modify the same mapping entries, thus violating the Invariant 4 (a name is unique in the map of a directory). A conflict on a mapping entry requires conflict resolution on that entry, as well as recursive conflict resolution on the target inodes of that entry. For example, concurrently creating the same directory `/foo/bar` cause a directory data-data conflict on `/foo`; merging these updates on `/foo` recursively merges the directories with the same name `bar`.

There are multiple situations that lead to this type of conflict, depending on: the state of the mapping entries (*deleted* or *updated*), target inode number (the same inode or different inodes), and target inode type (directory or file). Table 7.1 shows the possible combinations of these factors. Our conflict resolution algorithm for each of these cases is as follows.

Case 1: This is the situation when concurrent updates delete the same mapping entry. Similar to the case of concurrent deletes of the same inode, there is no conflict in this case because the updates commute and do not violate any invariant.

Case 2: In this case, a mapping entry is concurrently both deleted and updated, causing a state conflict on the mapping. Similarly to the case of concurrent update-delete on an inode, we preserve the update. If the updated mapping entry maps to a file, we change the name of the mapping; changing the name follows the same naming algorithm as with the conflict cases described earlier.

Case 3: Concurrent updates in this case target the same name which maps to a directory; this recursively creates another directory data-data conflict on the target

Table 7.1: Concurrency cases when mappings with the same name has been updated concurrently. A row shows the state of the mappings (*updated* or *deleted*), their target inode (same or different), and the types of the target inodes if different.

concurrent updates	**target**	**target type**	**resolution**
delete + delete	-	-	1: no conflict
delete + update	-	-	2: preserve update
update + update	same	directory	3: recursive merge
	same	file	4: rename file
	different	directory + directory	5: recursive merge
	different	directory + file	6: rename file
	different	file + file	7: rename files

directory. The conflict resolution algorithm recursively merges the concurrent updates on the target directory of the mapping.

Case 4: The concurrent updates in this case target the same file, may cause on the target file either *file data-data conflict* or *file names-names conflict*. We recursively resolve the conflict on the file according to the type of conflict. This creates new file names to preserve these concurrent updates.

Case 5: Concurrent updates in this case map the same name to different directories; this violates the uniqueness of the names (Invariant 4). This is similar to Case 3. The conflict resolution algorithm recursively merges the contents of the directories (Figure 7.4a).

Case 6: Concurrent updates map the same name to a directory and a file; this also violates the name's uniqueness invariant as with the previous case. We recursively resolve this conflict by keeping the mapping entry to the directory as is, and changing the name of the mapping entry to the file (Figure 7.4b); this makes the names of the mapping entries unique again.

Case 7: The same name are mapped to different files; this also violates Invariant 4 (name uniqueness). This case is similar to the file data-data conflict, except that here different contents are in different inodes. Similarly, the conflict resolution algorithm renames both mapping entries (Figure 7.4c), but without creating new inodes.

7.2.5 Directory Names-Names Conflict

When concurrent sessions rename the same directory to different names, this makes the directory to have different names, thus violating the Invariant 3 (a directory must have a single name). We preserve both names in different copies of the directory (Figure 7.5); making new copies of the directory implies recursively making new copies

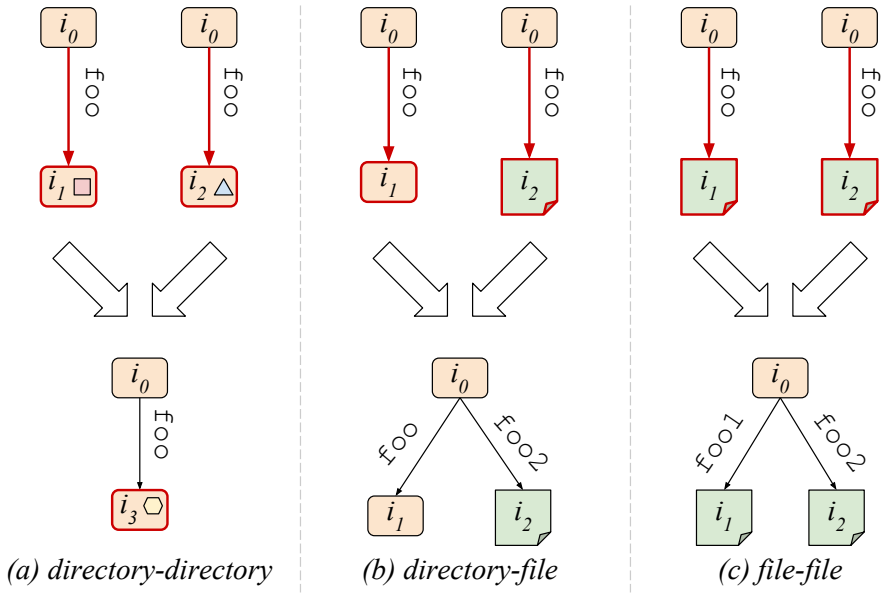


Figure 7.4: Conflict resolution for directory data-data conflicts.

of its children. This approach violates Principle 2 (No-Ghost-Update) but preserves Principle 1 (No-Lost-Update). We choose this approach because it would otherwise lose updates if we opted for preserving a single name (LWW).

7.3 Conflict Resolution For Indirect Conflicts

As has been described earlier in Section 6.5.2, with update back-propagation, an indirect conflict becomes a set of direct conflicts on ancestors of updated inodes of concurrent updates. This reduces resolving an indirect conflict to resolving direct conflicts, for which conflict resolution has been discussed in the previous section. In this section, we will go through the example of the creation of a directory cycle by concurrent `rename`s.

Figure 7.6 presents an example. In this example, making inode 3 a descendant of inode 6 (`mv /A/foo /B/bar/quz`) indirectly updates inodes 2 and 4 by update back-propagation. Similarly, making inode 4 descendant of inode 5 indirectly updates inodes 1 and 3. The sets of updated inodes of the sessions are $\{1, 2, 3, 4, 6\}$ and $\{1, 2, 3, 4, 5\}$, respectively. This translates to the *directory data-data conflicts* on inodes 1 and 2, and the *directory names-names conflicts* on inodes 3 and 4.

By resolving the *directory data-data conflicts* on inodes 1 and 2, we retain the mappings `foo` and `bar`, respectively; by resolving the *directory naming conflict* on inodes 3 and 4, we preserve the sub-trees rooted by these inodes.

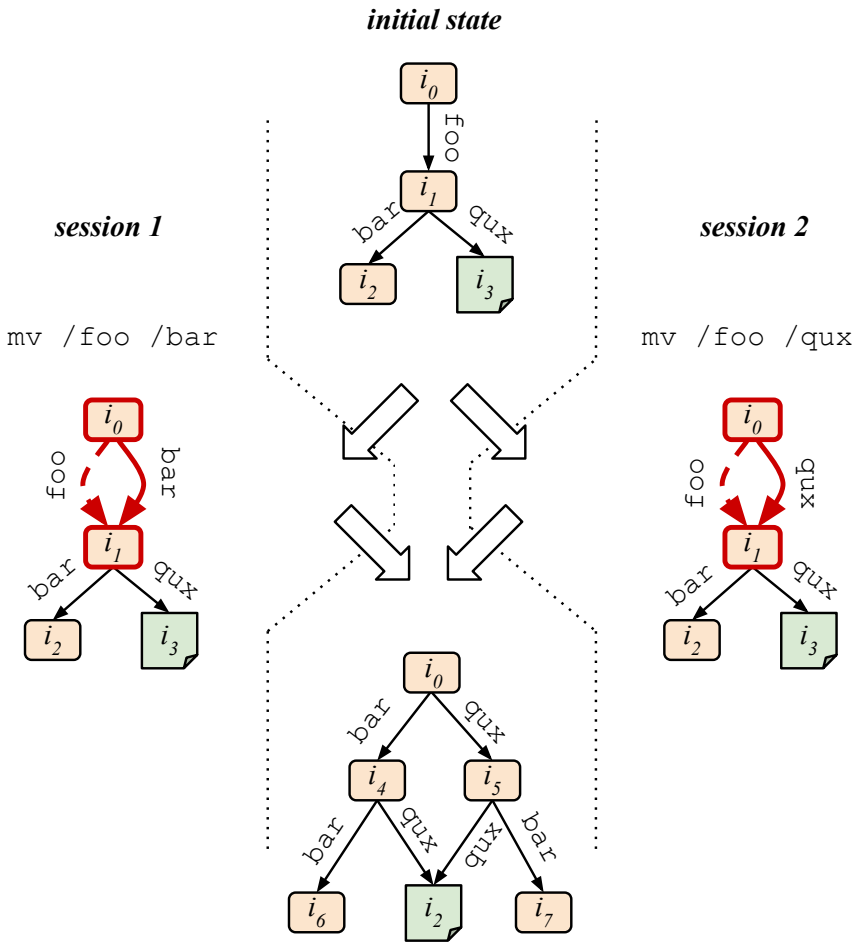


Figure 7.5: Conflict resolution for directory names-names conflicts.

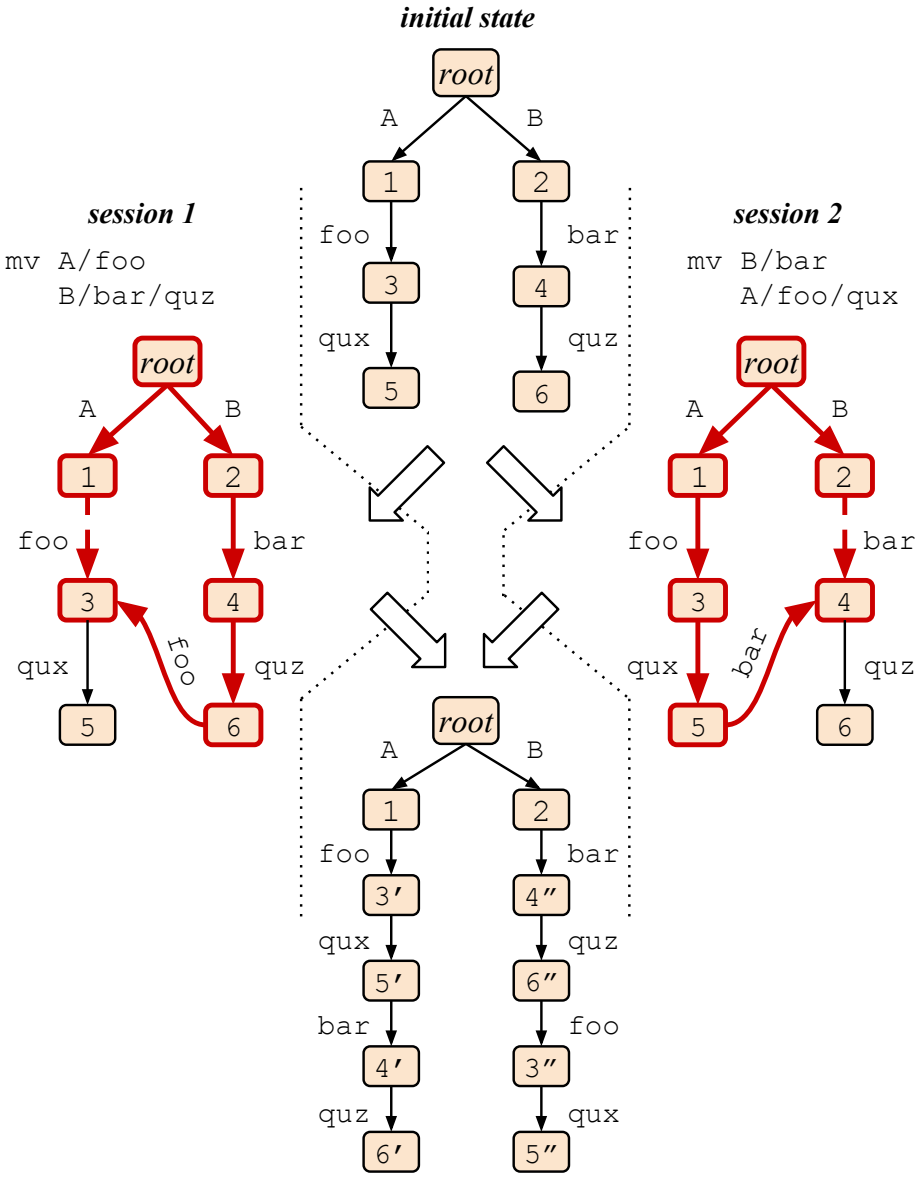


Figure 7.6: Merging semantics for indirect conflicts.

Chapter 8

Replica Convergence

In the previous chapter, we showed how we semantically resolve the conflicts between any pair of concurrent sessions, with respect to system integrity (Rule 2) and to inode names-data coherency (Rule 3). This semantic resolution however does not guarantee replica convergence. In this chapter, we study the violations of replica convergence (Rule 1), and we propose our conflict resolution. Our approach is based on CRDTs [65, 66], a set of priciples for eventually consistent data types.

8.1 CRDTs

CRDT, which stands for Conflict-Free Replicated Data Type, is a set of principles for replicated data types to ensure the eventual consistency of their replicas. We summarize these principles and how to apply them to converge our file system replicas as below.

8.1.1 CRDT Principles for State-based Replication

Replica State The state of each replica advances after a modification, with respect to a predefined partial order between the states.

Merging Replicas Merging the states of concurrent replicas computes their Least Upper Bound (LUB); the LUB of two states is the least one among those states equal or more advanced than them, with respect to the defined partial order. By definition, computing the LUB (denoted by $\oplus$) is *idempotent*, *commutative*, and *associative*; these properties are formalized as below, where s, s_i, s_j, s_k are states.

$$\begin{aligned} \textit{idempotent}: &\quad s \oplus s = s \\ \textit{commutative}: &\quad s_i \oplus s_j = s_j \oplus s_i \\ \textit{associative}: &\quad (s_i \oplus s_j) \oplus s_k = s_i \oplus (s_j \oplus s_k) \end{aligned} \quad . \tag{8.1}$$

8.1.2 Replica Convergence Using CRDT

A file system is defined as a collection of individual inodes $\mathfrak{I} = i_0, i_1, \ldots, i_n$, where an inode i_j might have multiple versions $i_j^0, i_j^1, \ldots, i_j^m$, each of which is created by an update to the inode. Note that a deletion creates a version called a *delete marker* (as will be described in Section 8.3.1); a delete marker appears to the users as if the inode was deleted. In this chapter, we consider a file system as a set of inode versions, and a *correct* file system state as one that satisfies the invariants of Chapter 2.

We define the partial order between two correct file system states S_A and S_B as following (we use $\|$ to represent the concurrency between two states, and $\cap$, $\cup$, and $\setminus$ for set intersection, union, and difference operations, respectively):

$$\begin{cases} S_A = S_B & \iff S_A \cap S_B = S_A \wedge S_B \cap S_A = S_B \\ S_A < S_B & \iff S_A \cap S_B = S_A \wedge S_B \setminus S_A \neq \varnothing \\ S_B < S_A & \iff S_B \cap S_A = S_B \wedge S_A \setminus S_B \neq \varnothing \\ S_A \parallel S_B & \quad otherwise \end{cases} . \tag{8.2}$$

Because an update on an inode creates a new version of that inode, the state of the file system therefore advances upward after each update, with respect to our definition of partial order above. Indeed, suppose S^t and S^{t+1} are correct file system states before and after an update, which targets i_j and creates a new version i_j^k, then we have $S^t \cap S^{t+1} = S^t$ and $S^{t+1} \setminus S^t = \{i_j^k\}$, therefore $S^t < S^{t+1}$.

Merging two correct states also produces another correct state (upper bound) which is equal or more advanced then both of them. Consider a pair of concurrent updates that change a correct state S to new correct states S_A and S_B, respectively; these updates target inode i_j of S, and create different versions i_j^A and i_j^B, respectively. Merging these states involes computing the union of the diverged states, and resolving the conflict between i_j^A and i_j^B, then storing the conflict resolution result in a new version i_j^C (cases *delete-delete*, *delete-update*, *file names-names conflict*, and *directory data-data conflict*) or new inodes i'_j, i''_j (cases *file data-data conflict* and *directory names-names conflict*). In any case, the merged state S_C ($S_C = S_A \cup S_B \cup \{i_j^C\}$ or $S_C = S_A \cup S_B \cup \{i'_j, i''_j\}$) is always more advanced than each of the diverged states S_A and S_B.

We also make merging correct file system states to have the LUB properties, with respect to the correctness of the file system and our merging rules; the LUB properties are idempotency, commutativity, and associativity. Though we do not formally prove it, we conjecture that the upper bound computed by merging is the LUB of the states.

In the next sections, we describe how we ensure the LUB properties of merging

file system states, which is translated into ensuring the LUB properties of our conflict resolutions. We present the commutativity and associativity properties of our conflict resolutions; the idempotency property is automatically achieved as there is no concurrency in the case of $s \oplus s$.

8.2 Ensuring Commutativity

In the previous chapter, we resolved conflicts using the following main actions: (1) generating new file names, (2) generating new inode numbers, and (3) merging directories. Ensuring the commutativity of our conflict resolution algorithms implies ensuring the commutativity of these actions, which in turn implies generating a new name or inode number, or merging inodes deterministically (while of course ensuring other file system invariants, such as name and inode number uniqueness). We describe how we achieve determinism for these actions below.

8.2.1 Deterministic File Name Generation

The problem of deterministically generating a new file name can be formally represented by a deterministic function $n' = f(n)$ where n and n' are the old and the new file names.

Technically, there are many ways f can achieve its determinism. For our system, we define f to generate new names by appending information to the original file name, according to the following format: *n.ino.ssid.hash*. In this format, *ino* is the original inode number of the file; *ssid* is the identifier of the session that has updated the file; and *hash* is the result of hashing all the previous fields to make the name appear more unique. For example, the generated names `foo.12345.A.abcde` and `foo.12345.B.edcba` indicate that they were generated from resolving a file data-data conflict on file `foo` between concurrent updates from sessions A and B, respectively.

We have decided to use this format because, first, the process is deterministic (assuming the hash function is deterministic); second, the informative name helps users to reason about the conflict; and last, the generated name is unique with high probability. However, absolute uniqueness cannot be guaranteed since a user may manually create any arbitrary name. To reliably ensure uniqueness, naming convention enforcement or coordination between replicas would be necessary; the latter is possible at the expense of availability in the sense of latency. Our current implementation simply assumes such naming collisions do not occur.

To simplify the presentation, in this thesis, we use the simplified format *n.ssid* to describe automatically generated names; the implementation uses the full format.

8.2.2 Deterministic Inode Number Generation

The commutativity problem of generating a new inode is to deterministically generate a new inode number for the new inode, so that replicas have the same inode number for the same file content.

To ensure determinism, we compute a new inode number i' as the result of hashing the identifier *ssid* of the updating session, and the original inode number i, as in: $i' = hash(i, ssid)$.

The uniqueness of the inputs ensures the uniqueness of the output with high probability, assuming a good hash function. Inode number collisions however might still happen depending on the hash function being used. As above, global coordination would be necessary to ensure uniqueness. In our implementation, we assume no hashing collisions as well.

8.2.3 Deterministic Directory Merge

When the map of a directory is concurrently changed, or when different directories with the same name concurrently exist, we merge the updated maps into a single directory inode. Commutativity of directory merging reduces to ensuring that the directory inode to store the merging result is deterministically chosen, and thus ensuring commutativity of conflict resolution.

Different approaches are possible, for example, generating a new inode based on the combination of the input inodes, or arbitrarily picking an inode among those conflicting (as in LWW). However for its known simplicity and support for both commutativity and associativity, we choose the LWW approach to store the merge result in the inode whose inode number is larger.

Consider, for instance, merging three directories whose inodes are 1, 2, and 3, respectively. When merging in the order $(1 \oplus 2) \oplus 3$, we will pick inode 2 to store the result of $1 \oplus 2$, then to pick inode 3 to store the result of merging then 2 with 3. The same outcome goes for another grouping $1 \oplus (2 \oplus 3)$; resolving this results in 3 as the final inode to store the merging result of the inodes.

8.3 Ensuring Associativity

In this section, we present the problem of achieving associativity. For this, we introduce a new concept, the *delete marker*.

8.3.1 Delete-Update Dectection Issue and Delete Marker

In a non-versioning POSIX file system implementation, a deletion physically removes the target mapping entry and inode (if its `nlink` reaches 0) from the file system. This however makes it difficult to tell if there is a delete-update concurrency, or there is only an update on an inode.

In order to detect the delete-update concurrency cases, we use the concept of *delete marker*. In our file system, a deletion does not remove its target (mapping and inode), but instead, it changes its target into a delete marker. A delete marker remains internal to our implementation and is not visible to any file system operation. However, detecting update-update conflicts takes delete markers into account.

In the next sections, we will go through all concurrency cases between three updates and will show that our conflict resolution algorithms are associative with the help of delete marker.

8.3.2 Concurrency Cases Between Three Updates

Between any three updates A, B, and C, there are three ordering possibilities as follows (we use $\rightarrow$ to represent the happen-before relationship, following Lamport's notation [32], and $\parallel$ for concurrency): $A \rightarrow B \rightarrow C$, $A \rightarrow (B \parallel C)$, $(A \rightarrow B) \parallel C$, and $A \parallel B \parallel C$.

In the first case, the updates are causally ordered. Because of that, these updates follow the sequential semantics described in Chapter 2; there are no conflicts between the updates. In the second case, the concurrent updates (B and C) both happen-after A; resolving this case is equal to merging B and C, therefore converging the replicas. We are interested in the last two concurrency cases and will present our approach for ensuring associativity for these cases.

8.3.3 Associativity For Causally Related Updates

This section discusses associativity for the concurrency situation with causally related updates; this is the case of $(A \rightarrow B) \parallel C$ as described before. Let S_A, S_B, and S_C be the states of the file system as the results of the updates, respectively, the goal of associativity for this case is to have $(S_A \oplus S_B) \oplus S_C = (S_C \oplus S_A) \oplus S_B$. However because $A \rightarrow B$, thus $S_A < S_B$, therefore $S_A \oplus S_B = S_B$, the requirement to ensure associativity becomes: $S_B \oplus S_C = (S_C \oplus S_A) \oplus S_B$.

The requirement is however not satisfied with our conflict resolution algorithms so far. When they automatically generate new files; the names of these new files do not follow the causal relationship between subsequent sessions. For example when A, B,

and C all update the *data* part of a file `foo`, even though $S_A < S_B$, the result of $S_A \oplus S_C$ is not subsumed by $S_B \oplus S_C$, thus $(S_C \oplus S_A) \oplus S_B \neq S_B \oplus S_C$ as following:

$$\underbrace{\underbrace{S_B \oplus S_C}_{\texttt{foo.B, foo.C, foo}(marker)}}_{\neq} \quad \underbrace{\underbrace{(S_C \oplus S_A)}_{\texttt{foo.A, foo.C, foo}(marker)} \oplus S_B}_{\texttt{foo.A, foo.B, foo.C, foo}(marker)}.$$

We can see that on the right-hand side, even though S_A is subsumed by S_B, the existing conflict resolution still generates `foo.A` to store S_A's `foo`.

To solve this issue, when resolving the conflict of a session B with a concurrent one C, we remove the result of resolving A (such that $A \rightarrow B$ and $\nexists A' : A \rightarrow A' \wedge A' \rightarrow B$) and C if $A \parallel C$. In the example above, when merging $S_B \oplus S_C$, we remove `foo.A` (the result of $S_A \oplus S_C$) because $A \rightarrow B$ and $A \parallel C$; the final outcome is the desired {`foo.B`, `foo.C`, `foo` $(marker)$}.

8.3.4 Associativity For Full Concurrency On Inodes

This section describes the concurrency cases between three concurrent sessions $A \parallel B \parallel C$ that update the same inode i. Table 8.1 shows all possible concurrency cases. In the following, we show that our conflict resolution for each of these cases is associative.

We use *deleted* and *updated* to refer to different states of an inode i; i_A, i_B, and i_C are inodes generated when resolving the conflicts of the respective sessions.

Case 1: This is the case when all the sessions delete the inode. Because resolving concurrent deletions of an inode results in a deleted inode, resolving any number of concurrent deletions in any grouping always result in a deleted inode.

$$\underbrace{\underbrace{(deleted \oplus deleted)}_{deleted} \oplus deleted}_{deleted} = \underbrace{deleted \oplus \underbrace{(deleted \oplus deleted)}_{deleted}}_{deleted}$$

Case 2: This is the case when there are two deletions and one update on an inode. For the case of a directory:

$$\underbrace{\underbrace{(deleted \oplus deleted)}_{deleted} \oplus updated}_{updated} = \underbrace{deleted \oplus \underbrace{(deleted \oplus updated)}_{updated}}_{updated}.$$

Table 8.1: Concurrency cases on an inode between three updates.

concurrency case	**merging result**	
	directory	*file*
1: deleted ‖ deleted ‖ deleted	deleted	deleted
2: deleted ‖ deleted ‖ updated	updated	updated
3: deleted ‖ updated ‖ updated	data/naming conflict	data/naming conflict
updated ‖ updated ‖ updated		
4: data ‖ data ‖ data	merged	split
5: data ‖ data ‖ names	merged + copy	split
6: names ‖ names ‖ data	copies	split + merged
7: names ‖ names ‖ names	copies	merged

For the case of a file:

$$\underbrace{\underbrace{(deleted \oplus deleted)}_{deleted} \oplus updated}_{deleted + i_C} = \underbrace{deleted \oplus \underbrace{(deleted \oplus updated)}_{deleted + i_C}}_{deleted + i_C}.$$

We can see that in all cases, committing the updates in any grouping has the same result. When i is a directory, the final outcome is the updated directory. When i is a file, it is i being a marker, and i_C storing C's update (as described in Chapter 7).

Case 3: This is the case when when one of the concurrent sessions deletes the inode and the other two update it. For any type of the inode, the conflict between these sessions on the inode becomes the conflict between the two updates. Resolving the conflict between a pair of updates on an inode is commutative as described earlier, therefore we have the same result when committing these sessions in any grouping. For the case i is a directory:

$$\underbrace{\underbrace{(deleted \oplus updated)}_{updated} \oplus updated}_{dir.\ data/naming\ conflict} = \underbrace{deleted \oplus \underbrace{(updated \oplus updated)}_{dir.\ data/naming\ conflict}}_{dir.\ data/naming\ conflict}.$$

For the case i a file:

$$\underbrace{\underbrace{(deleted \oplus updated)}_{deleted + i_B} \oplus updated}_{deleted + i_B + i_C} = \underbrace{deleted \oplus \underbrace{(updated \oplus updated)}_{deleted + i_B + i_C}}_{deleted + i_B + i_C}.$$

Case 4: This is the case when the sessions update only the *data* part of the inode.

For i is a directory, because we can merge the updates together, the final outcome of conflict resolution is always i storing the merged of the concurrent updates. This case is associative.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(dir.\ data\ conflict)\\ merged(A+B)}} \oplus data}_{\substack{(dir.\ data\ conflict)\\ merged(A+B+C)}} = data \oplus \underbrace{\underbrace{(data \oplus data)}_{\substack{(dir.\ data\ conflict)\\ merged(B+C)}}}_{\substack{(dir.\ data\ conflict)\\ merged(A+B+C)}}.$$

For i is a file, the concurrent updates in this case cause pairwise *file data conflict*s on the inode. Resolving the conflicts in any order generates i_A, i_B, and i_C storing the updates from A, B, and C, respectively.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B}} \oplus data}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B + i_C}} = data \oplus \underbrace{\underbrace{(data \oplus data)}_{\substack{(file\ data\ conflict)\\ deleted + i_B + i_C}}}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B + i_C}}.$$

Case 5: In this case, two sessions update the *data* of the inode and the other session updates the *names* of it. For i is a directory, resolving this conflict merges the updates to the *data* of the directory while generating a new directory for the update to the *names* part; the merging process is as below.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(dir.\ data\ conflict)\\ merged(A+B)}} \oplus names}_{\substack{(dir.\ naming\ conflict)\\ merged(A+B)+i_C}} = data \oplus \underbrace{\underbrace{(data \oplus names)}_{\substack{(dir.\ naming\ conflict)\\ i+i_C}}}_{\substack{(dir.\ data\ conflict)\\ merged(A+B)+i_C}}.$$

For i is a file, resolving this conflict generates new files to store the updates to the *data* part, while keeping the last update in the original file.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B}} \oplus names}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B + i_C}} = data \oplus \underbrace{\underbrace{(data \oplus names)}_{\substack{(file\ data\ conflict)\\ deleted + i_B + i_C}}}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B + i_C}}.$$

Case 6: The conflict in this case is between two concurrent updates to the *names* part

of the inode, and an update to its *data* part. For i is a directory, our conflict resolution generates new directories to store the updates to the *names* part of the directory, while keeping the original inode to store the update to the *data* part.

$$\underbrace{\underbrace{(names \oplus names)}_{(dir.\ naming\ conflict)} \oplus data}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_A + i_B}} = names \oplus \underbrace{\underbrace{(names \oplus data)}_{(dir.\ naming\ conflict)}}_{\substack{(dir.\ naming\ conflict)\\ i_B + i}}.$$
$$\underbrace{\qquad\qquad}_{\substack{(dir.\ naming\ conflict)\\ i_A + i_B + i}} \qquad\qquad \underbrace{\qquad\qquad}_{\substack{(dir.\ naming\ conflict)\\ i_A + i_B + i}}$$

For i is a file, our conflict resolution generates a new file to store the updates to the *data* part of the file, while keeping the merged of the other updates in another file.

$$\underbrace{\underbrace{(names \oplus names)}_{(file\ naming\ conflict)} \oplus data}_{\substack{(file\ naming\ conflict)\\ i}} = names \oplus \underbrace{\underbrace{(names \oplus data)}_{(file\ data\ conflict)}}_{\substack{(file\ data\ conflict)\\ i + i_C}}.$$
$$\underbrace{\qquad\qquad}_{\substack{(file\ data\ conflict)\\ i + i_C}} \qquad\qquad \underbrace{\qquad\qquad}_{\substack{(file\ naming\ conflict)\\ i + i_C}}$$

Case 7: In this case, the concurrent sessions all update the *names* part of the inode. For i is a directory, this causes two consecutive *directory naming conflict*s. Resolving this case is similar to the previous case, except in this case, our conflict resolution generates new directories for all of the updates.

$$\underbrace{\underbrace{(names \oplus names)}_{(dir.\ naming\ conflict)} \oplus names}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_A + i_B}} = names \oplus \underbrace{\underbrace{(names \oplus names)}_{(dir.\ naming\ conflict)}}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_B + i_C}}.$$
$$\underbrace{\qquad\qquad}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_A + i_B + i_C}} \qquad\qquad \underbrace{\qquad\qquad}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_A + i_B + i_C}}$$

For i is a file, the updates in this case cause pairwise *file naming conflict*s. Resolving this case merges all the names of each update in the original file.

$$\underbrace{\underbrace{(names \oplus names)}_{(file\ naming\ conflict)} \oplus names}_{\substack{(file\ naming\ conflict)\\ i}} = names \oplus \underbrace{\underbrace{(names \oplus names)}_{(file\ naming\ conflict)}}_{\substack{(file\ naming\ conflict)\\ i}}.$$
$$\underbrace{\qquad\qquad}_{\substack{(file\ naming\ conflict)\\ i}} \qquad\qquad \underbrace{\qquad\qquad}_{\substack{(file\ naming\ conflict)\\ i}}$$

8.3.5 Associativity For Concurrency On Mapping Entries

The concurrency cases between three concurrent updates on a mapping is similar to those for an inode; we also briefly describe these cases in Table 8.2.

The conflict resolution for the cases including at least one deletion of a mapping is also similar to the resolution for an inode. For the case of three concurrent deletions, the merge deletes the mapping at the end; for the case of a single update, this update is preserved by the end, meaning the mapping is the updated mapping; for the case with two updates, the conflict resolution for a pair of updates on a mapping has been described in the resolution for *directory data conflict* (Section 7.2.4).

For the cases with three concurrent updates on a mapping, the conflict resolution is simple. Basically, a directory has a higher priority of maintaining its name in a conflict resolution with a file, and our conflict resolution always renames a file in any file conflict. Based on these characteristics of conflict resolution, in this case of conflict, if the mapping in any of the update is a directory, the final conflict resolution result for that mapping is a directory; if the mapping of an update is a file, it is renamed deterministically, therefore the mapping in the end may be a marker if the input mappings are all file. The conflict resolution for each pair of conflicts using the deterministic inode and/or name generation as has been described before; this makes resolving a conflict on a mapping associative.

For example, consider concurrent updates create two directory mappings and a file mapping with the same name `/foo`. Regardless of the target inodes, we can easily see that our conflict resolution will create `/foo.C` (with '`.C`' as the deterministic suffix described before) to store the file mapping, and it will merge the content of the directories together if they are different directories. In the end, there will be a mapping `/foo` which maps to a directory, and a mapping `foo.C` that maps to a file.

Table 8.2: Concurrency cases on an inode between three updates. A row represents a combination of them.

concurrency case	**merging result**	
	directory	*file*
1: deleted ‖ deleted ‖ deleted	deleted	deleted
2: deleted ‖ deleted ‖ updated	updated	updated
3: deleted ‖ updated ‖ updated	data/naming conflict	data/naming conflict
updated ‖ updated ‖ updated		
4: data ‖ data ‖ data	merged	split
5: data ‖ data ‖ names	merged + copy	split
6: names ‖ names ‖ data	copies	split + merged
7: names ‖ names ‖ names	copies	merged

Part III

Implementation And Evaluation

Chapter 9

Implementation

In this chapter, we present the design and implementation of the prototype of our file system. It targets, first, achieving the desired functionalities (concurrent sessions and conflict resolution), and second, to some extent, optimizing the performance of the system. We focus on the following particular points:

- MetaData-Data decoupling for system performance
- Directory layout for system scalability
- Session implementation

9.1 Metadata-Data Decoupling

Data (the *data* part of a file) and metadata (everything else including directory child maps) of a file system have different access patterns; data is immutable and throughput oriented, whereas metadata is mutable and latency sensitive. Therefore existing large scale distributed file systems, such as GFS [20, 21] and HDFS [67], usually separate them to improve system performance.

For the same reason, we decouple metadata from data in the implementation of our file system. We store them in separate distributed key-value stores, named the *datastore* for data and the *metastore* for metadata.

The datastore is an object store that provides a simple key-value interface and is specialized for storing large objects. We use the Ring from Scality [63] for this purpose. When it stores an object, the datastore generates a random key identifier. The application (our file system in this context) can later use this key to access the object content. The datastore divides a large object (whose size exceeds a certain threshold) into small chunks, called data stripes, each one is stored under its own random key that is returned to the application. We call these identifiers *data keys*.

The datastore provides other functions such as byte range locking to facilitate data access, and erasure coding for replication. In what follow, we ignore the Ring internals and treat it as a black box.

The metastore is a separate key-value store, optimized for small and mutable objects. We implemented our metastore by combining multiple instances of LevelDB [19] as the storage building block, with Raft [41] as the replication protocol. The metastore supports a simple key-value store API: `PUT` to store or update an object under a string key, which we will call a *meta key*, `GET` to retrieve it, `DELETE` to delete it, `BATCH` to atomically perform a batch of `PUT` and/or `DELETE` operations, and `LIST` to list all keys in a given range.

We use the metastore to store different types of information, such as inodes and directory mapping entries. To better separate these types (to make it easy to list all keys objects of a certain type, for example), we separate the metastore into logical namespaces, each representing a different information type. To implement that, we append a prefix to every meta key, for example we use "`I::`" and "`N::`" as the prefixes for meta keys of inodes and directory mapping entries, respectively.

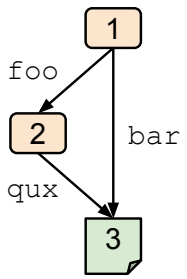
The meta key of an inode is of the format: "`I::inode_number`", where "`I::`" is the namespace of the inodes, and "`inode_number`" is the decimal string representation of the inode number of the inode. For example, the inode identified by inode number `4` is stored under the key "`I::4`" in the metastore.

A directory is stored entirely in the metastore. In contrast, the *data* part of a file in the metastore contains the data keys of the data stripes of that file. To write a file, our file system first stores the file's contents in the datastore, which returns the list of data keys for the data stripes. The file system then stores this list in the *data* part of the file inode in the metastore.[1] Accessing a file's contents is the reverse: first look up desired file in metastore to get its data keys, then read the corresponding data stripes from the datastore.

9.2 Directory Data Layout in the Metastore

The scalability of a file system is directly impacted by the scalability of a directory, i.e., how many mapping entries a directory can hold without increasing latency. The problem of scaling out directories for large scale distributed file systems has always been an inherent issue in the area of high performance computing [47, 57, 76, 77]. In this section, we describe and justify our directory design to support system scalability.

[1] A possible optimization would be to store a small file in the metastore entirely, though we did not implement it in our prototype.



key	value
I::1	{stat,names}
I::2	{stat,names}
I::3	{stat,data,names}
N::1::foo	{ino:2}
N::1::bar	{ino:3}
N::2::qux	{ino:3}

Figure 9.1: Example of our metadata system implementation using key-value store abstraction. The figure on the left shows the file system structure visually; the table on the right presents the data structure of that file system in metastore.

9.2.1 Directory Inode-Data Decoupling

To address the directory scalability issue while being simple, we rely on the scalability of the metastore to scale out the directories. We store each mapping entry of a directory as a separate object in the metastore, and store the directory itself (without its child map) as a separate object. We use the following format for the meta key of a directory mapping entry:

`N::inode_number::mapping_name`

where "`N::`" is the namespace for meta keys of directory mapping entries, `inode_number` is the decimal string representation of the inode number of the directory containing the mapping, `mapping_name` is the name of the child in this directory. For example in Figure 9.1, we use "`I::2`" as the meta key of inode `2` (which is directory `foo`), and "`N::2::qux`" as the meta key for the mapping of inode `2`'s child "`qux`".

With this approach, we can implement most operations similarly to traditional inode-based implementations, except for listing a directory. For example, to access inode `4`, our file system queries the metastore for the value associated with key "`I::4`".

Different from classical file system designs, to list a directory (primitive `readdir`), we rely on the range query capability of the metastore. Listing directory inode `4`, for example, is done by searching for all entries with prefix "`N::4::`", whereas it is a simple retrieval of "`I::4`" in the classical file system implementations. In the example in Figure 9.1, listing directory `foo`, whose inode is `2`, involes searching for all objects whose key starts with "`N::2::`".

9.2.2 Advantages and Disadvantages

Implementing our metastore as a distributed key-value store has many advantages. The simple key-value API abstracts away many of the complexities of file system implementation. For example, having the datastore to handle data avoids having to deal with traditional indirect block pointers of large inodes. The scalability of the key-value store removes the need to worry about the scalability in the file system implementation as we rely on the scalability of the metadata store for scaling out directories.

The implementation of the metadata system using the key-value store abstraction also has some disadvantages. One is about the size of the metadata of a file. Because our implementation stores all data keys of a file in its metadata, the metadata object thus may be large and may thus incur larger overhead than traditional indirect inode pointers (which stores some data keys at the first level). However, we believe that this case is rare and could be solved by standard techniques, such as adjusting the data stripe size or by caching. We could also store the *data* part of a file in a separate key, similarly to directories, but this will increase access latency as a trade-off. Because this depends on workload, we leave this issue of scalability to future work.

Using range queries to list directories (using `LIST`) may be slower compared to classical direct inode retrieval (using `GET`), and may make caching difficult. This is a trade-off between using a simple design and having to deal with the complexity of low-level indirect inode pointers. Furthermore, because of its size, listing a very large directory in a distributed file system is usually moderated, for example by limiting the number of entries in a listing result (to 1000 as with Amazon's S3 service [1]). This approach may reduce the potential performance gap between our approach and traditional approach in listing a large directory. Nevertheless, distributed key-value stores are a dynamic research topic; we believe that this active research area will help improving the performance of listing in the future.

9.3 Session Data Layout in the Metastore

To support sessions in our file system, the metastore must be able to store and retrieve different versions of an object. In this section, we describe how we support object versioning in the metastore.

9.3.1 Object Versioning in the Metastore

We store each version of an inode or a mapping entry as a separate object in the metastore. Furthermore, for each inode or mapping entry, we also keep a *master*

version, which has the contents of the last committed version, and the lists of committed and non-committed versions. The meta key of the master version is the meta key of the inode or the mapping entry as described earlier. The key of a version other than the master is the concatenation of the master key with the identifier of the session that created it. Figure 9.2 shows an example of versioning in metastore. In this example, the master version of inode `4` is stored under the key `I::4`, and its versions, created by session S_A, S_B, and S_C, are stored under the keys `I::4::`S_A, `I::4::`S_B, and `I::4::`S_C, respectively. In this example, S_B's version is the latest committed version, and S_C is a non-committed version. The master version keeps the lists of committed versions and non-committed versions in its V_C and V_A, respectively; the master version contains the contents of S_B's version, as S_B is the latest committed version.

Writing an object will create or update the version specific to the current session; reading an object will either read the session's specific version if it exists, or the last version committed before the current session started. In the above example, consider a session S_D that starts after S_A's commit (but before S_B's and S_C's commits), S_D's writes will create or update `I::4::`S_D, and it reads will read from `I::4::`S_D if that version exists, or `I::4::`S_A, otherwise.

9.3.2 Object Versioning With Version Vectors

We use version vectors [18, 32, 35] to encode the partial order between versions. A version vector is an array of monotonically increasing integers, each representing the state (number of committed sessions in our case) of a corresponding replica. A version vector can be less than, greater than, equal to, or concurrent with another, indicating the partial order between the version vectors.

The different parts of an inode are each equipped with their own independent version vector. A file has a version vector for each of its *data* and *names* parts, as well as a version vector for the whole file, called its general version vector. If the *names* part of the file is updated, both its general version vector and the *names* version vector are increased; if its *data* part is updated, all of its version vectors are increased because of update back-propagation. A directory also has a version vector for each of its mapping entries; updating a mapping entry is updating the directory's *data* part, thus increasing the version vectors of the updated mapping entry, the *data* part, the *names* part, and the directory itself.

In our implementation, we have an entry for each session in the version vectors to distinguish concurrent sessions from the same replicas; updates inside a session increase the value of the entry for that session; we merge this value back to the entry of the

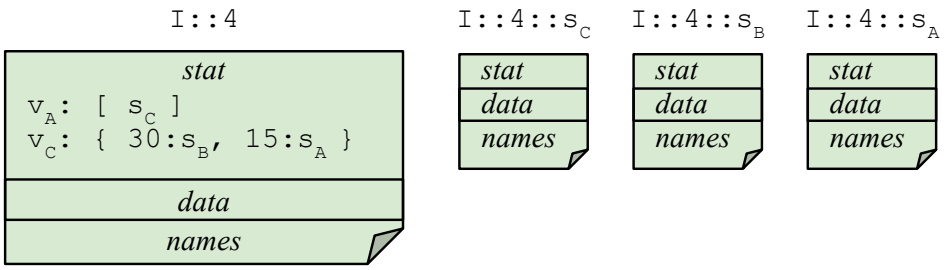


Figure 9.2: Example of our metadata system implementation with versioning. Where V_A and V_C in the master version are respectively the lists of uncommitted versions and of committed versions.

replica of the session when the session is committed. For example, in a system with replicas A, B, and C, if the version vector of an inode updated by a session S on replica B is $<A{:}1, B{:}2, C{:}3, S{:}1>$, then that version vector when S is committed is $<A{:}1, B{:}3, C{:}3>$.

9.4 Committing a Session

Our session system is basically a transaction processing system, specialized in processing long transactions. This section describes how we ensure the transactional properties[2] of committing the sessions in a replica.

In a replica, sessions are committed according to their partial order relationship: causally related sessions are committed in their causal order; non-overlapping concurrent sessions (those modified disjoint sets of inodes) can be committed concurrently, however, those overlapping are committed sequentially.

Committing a session is to apply (or install) all of its updates into the state of the current replica. Installing the updates is done in the Depth-First-Search manner from the root directory. Installing all updates of a session is an atomic process which means all updates or none are visible in the state of the current replica at any moment.

There are some available approaches to committing the sessions, such as the 2 Phase Commit (2PC) protocol or using centralized sequencer; we chose the latter approach to commit the sessions in a replica. This simplifies the implementation of our system. In the next sections, we will describe how we use the centralized sequencer to commit sessions atomically, and how we improve this approach to fit the session system.

[2]These are Atomicity, Consistency, Isolation, and Durability in the definition of transaction in the database community.

9.4.1 Committing With a Sequencer

The central sequencer in a replica defines a total order of committing the sessions in that replica. Every starting or committing session is thus aware of those that are being committed before it by contacting the sequencer. The sequencer has the following API:

- `commit_session(ssid):map` – commit a session whose identifier is `ssid`; return a map of the identifiers of the committing sessions (including the current session) and commit sequence numbers defining a total order of the committing sessions.
- `start_session():{ssid, map, n}` – start a session, return the identifier `ssid` associated with the session, the map of committing sessions as above, and the commit sequence number `n` of the latest committing or committed session.

In a common usage scenario (Figure 9.3), a session S_1 starts committing by registering with the sequencer using the function `commit_session`. The sequencer then keeps S_1 in its list of committing sessions, and then returns to S_1 a commit sequence number and the list of committing sessions. At this point, S_1 can safely notify users that it has been committed, and asynchronously start to install its updates.

When another session S_2 starts to commit before S_1 has finished installing its updates, S_2, just like S_1 earlier, registers itself to the sequencer, the sequencer returns to S_1 a commit sequence number and the list of the sessions that are being committed (which contains S_1), then S_2 notifies users as committed, and starts installing its updates asynchronously. While installing its updates on an inode, if S_2 encounters a update of S_1 that has not been installed, S_2 stops its installing process on that inode and will wait until the update of S_1 has been successfully installed. This ensures that all updates of S_1 are installed before S_2 can install its updates, thus ensuring the atomicity of S_1.

Similarly, when a session S_3 starts, it also needs to contact the sequencer to get the commit sequence number of the latest committing or committed sessions. In the situation when S_3 starts before both S_1 and S_2 have finished installing their updates, the list returned to S_3 will contain both of these sessions. When accessing an inode, if S_3 finds any non-installed update of the these sessions on that inode, S_3 will need to wait until these updates are installed. This ensures the atomicity of all sessions committed before S_3 starts.

9.4.2 Committing with Fine-Grain Locking

Fine-grain locking is an optimization to avoid waiting for non-installed updates to be committed when accessing an inode. In our system, a session when seeing a non-installed update on an inode will wait only if this pending update does not have any

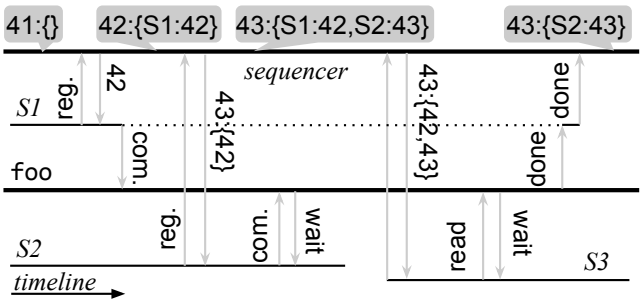


Figure 9.3: Example showing how sequencer is used to ensure session atomicity. Thick lines are the states of the sequencer and a file `foo`, respectively; thin black lines are the sessions; gray arrows are messages and their direction between agents; callouts showing the states of the sequencer with the sequence number and the list of committing sessions; timeline is from left to right.

conflict with the replica's state of the inode. These are the cases when there is no concurrency on an inode, or concurrent updates do not cause conflicts. In this case, it is totally safe to proceed to access the inode. For example, S_1 updates a file `/foo/bar` and then commits; after S_1 has done installing its update, S_2 starts and updates `/foo/qux` then commits; when S_3 starts and updates `/foo/quz` before S_2 has installed its update, it is safe for S_3 to proceed its access on `/foo` because the pending update in `/foo` of S_2 does not cause any conflict with the installed update of S_1 as they are both updated by back-propagation (which does not change anything of `/foo`).

The advantage of our fine-grain locking optimization over the traditional 2PC locking system is that concurrent sessions are only serialized at the parts that they conflict with when committing. This optimization is in fact valuable because the ratio of conflicts between sessions might be low[3] and the committing time for a session is long (several seconds for thousands of updated files in our experiments); pessimistically locking in these cases would be neither necessary nor efficient. Our session system is comparable to 2PC in the worst case when all sessions update the same set of directories and files. In such case, starting and stopping sessions will not be blocked, but accessing the updated directories and files will be blocked until all conflicts are resolved.

We can also proactively commit the pending updates of a committing session by other sessions. For example, even if the updates of S_1 and S_2 are conflicting, a session S_3 can proactively resolve the conflict and commit S_1 and S_2's updates, thus S_3 does not need to wait. This optimization is however is not implemented in our prototype; our future work will target this.

[3]The ratio of conflicts in the home and office environments is about 0.03% in a study of Ficus [56].

Chapter 10

Evaluation

In this chapter, we present our experiments to evaluate our file system design and implementation. The main objective of the evaluation is to compare our conflict resolution with existing approaches, and to analyse its performance.

10.1 Basic Conflict Resolutions

We compare the behaviour of our approach to those of commercial systems, including: Dropbox [13], Google Drive [23], and Microsoft OneDrive [37]. The criteria for the comparison is the conflict resolution principles (Chapters 6 and 7); these include the convergence of the replicas and the meaningfulness (Rules 1 and 2) of the merging results. The result is reported in Table 10.1

We implemented the prototype of our geo-distributed file system Tofu in NodeJS and FUSE. Because this experiment aims to test the conflict resolution mechanisms, we simply deployed Tofu on a single physical server and had two concurrent sessions as replicas A and B; these sessions are mounted on different mount points. The host server ran Ubuntu Desktop 14.04 LTS.

The deployment of Dropbox composes of two replicas, named A and B. Each is a virtual machine running Dropbox client for Linux v.3.0.3 on Ubuntu Server 14.04 LTS. These virtual machines were hosted on the same physical machine, with Network Address Translation networking. The setup for Google Drive and Microsoft OneDrive was a Mac running Mac OS X v.10.10 as replica A, and a PC with Windows 8.1 Enterprise as replica B. These replicas were in the same local network. The versions of Google Drive for Mac and Windows were the same, v.1.18.7821.2489, while those of Microsoft OneDrive on these replicas were v.17.3.4501 and v.6.3.9600.17334, respectively.

In the following experiments, we determined how well these systems could resolve

Table 10.1: Evaluation of our merging semantics with commercial systems.

Feature/Support	Dropbox	Google Drive	OneDrive	Tofu
Preserve Updates	✓	×	✓	✓
Preserve Structure	×	×	×	✓
Hard link	×	×	×	✓
Same name dir./files	✓	diverged[a]	✓	✓
Write ‖ Write	✓	last writer wins[b]	✓	✓
Direct Delete ‖ Edit	✓	delete wins[c]	✓	✓
Indirect Delete ‖ Edit	✓	delete wins	✓	✓
Cycles	✓	arbitrary[d]	×	✓

[a]Diverged: elements are preserved, but replicas' structures diverged.
[b]Last-Writer-Wins: the write with the last timestamp wins over the others.
[c]Delete-Wins: the element, if deleted on any site, is deleted after merging.
[d]Arbitrary: the directories in the cycles are placed at *root* after merge.

the conflict cases that we described in Chapter 6. In all of the cases, our prototype was able to resolve the conflicts and to produce the desired outcomes, with respect to the target of our merging semantics. For such result, we only describe the behaviours of the commercial systems with the experiments in the followings.

Experiment 1. *Hard link support* This experiment is to determine if a system supports hard links when resolving conflict. In this experiment, we created a hard link `bar` to an existing file `foo` in a session s on A, then we checked if the same hard link is created on B or not. After merging session s, the replica B of all setups of Dropbox, Google Drive, and Microsoft OneDrive had the same files named `foo` and `bar`, however, these file names on B pointed to different inodes, which means there is a divergence in the structures of the replicas.

There were also some anomalous results we observed with Dropbox in this experiment. Continuing the previous experiment, we updated `bar` on A, after merging the update to B, we deleted both `foo` and `bar` on B. However, `foo` appeared again on both A and B after the merge. From this observation, we speculated that Dropbox passively detected the update of `foo` on A caused by update to `bar`, and its concurrency with the deletion of `foo` from B was then resolved by resurrecting `foo`. This can be explained that Dropbox uses *inotify*, which is a Linux tool for detecting updates in a directory, but not at the inode level. In the case updating `bar` on A, inotify was able to report that `bar` was updated but not `foo`; when receiving `foo`'s deletion from B, Dropbox checked if it can merge this operation by inspecting the *stat* of `foo` (which is stored in `foo`'s inode). The found information indicated that `foo` was also updated and thus this merging resulted in `foo` reappeared on all sites when done resolving conflict.

Experiment 2. *Directory Data-Data Conflict* This experiment compares how the systems resolve a conflict where files with the same name exist. We concurrently created a file `foo` on each replica A and B to make a situation that multiple mappings with the same name exist. After resolving the conflict, the replicas of the setups (Dropbox, MS OneDrive, and Google Drive) created different files to store the data of these conflicting updates. However, the setup using Google Drive did not converge the replica to the same structure, i.e., `foo.A`'s content on A was equal to `foo.B`'s on B, and vice versa.

The results were repeated when we created a directory with the same name on each of the replicas. The setups using Dropbox and OneDrive were able to make their replicas converge by merging the contents of these directories, while Google Drive did not merge them, but changed the name of one of the directories, even though directories with the same name on both replicas do not have the same contents; Google Drive thus failed to make the replicas converge.

Experiment 3. *File Data-Data Conflict* This experiment compares how the systems resolve the conflict of concurrent writes to the same file. We concurrently updated a file `foo` on both replicas A and B. The setups using Dropbox and OneDrive resolved the conflict by creating different files to store these different updates of the file as expected. The setup using Google Drive however only retained the update from one replica and dropped the other. Because of this, we believe that Google Drive uses the LWW approach to resolve the concurrent updates on a file; this approach, while being simple, does not preserve all concurrent updates. We consider Google Drive failed in this test with respect to our measures of conflict resolution properties (Principle 1: no lost updates); the other setups, including ours, even though violated Principle 2 (no ghost updates), did not lose updates, which we objectively valuate more.

Experiment 4. *State Conflict* This experiment determines how a system resolves the concurrent delete and update to the same inode. We concurrently deleted a file `foo` on A, while writing to it on B. Google Drive deleted `foo` on both replicas after the merge. The other setups preserved the updated file. Our conjecture is that Google Drive resolves this conflict using the Delete-Wins approach, where the deletion of an element wins over the other concurrent operations on that element in conflict resolution. The others use a Write-Wins approach like ours that prefers the update over the delete.

Experiment 5. *Indirect Conflict* The purpose of this experiment is to find out if a conflict resolution could handle the concurrent `rename`s that create a directory cycle. Initially, we started both A and B with the same file system structure of directories `/test/foo` and `/test/bar`. On A, we moved `foo` into `bar`, while on B, we moved `bar` into `foo` to make the cycle of these directories. We expected the merged namespace

would be `/test/foo/bar` and `/test/bar/foo`, which preserves both updates of A and B while not violating the No-Directory-Cycle invariant (Invariant 6). The Dropbox system was able to make the expected outcome, while the Google Drive system put all the directories in the cycle in *root*, and the OneDrive system stayed diverged with only `/test/bar/foo` on A and only `/test/foo/bar` on B. The result of Google Drive in this situation though technically converges the replica (by moving both `foo` and `bar` to *root*), it does not preserve the updated directories (`/test/foo` and `/test/bar`), thus violating Principle 1 (no lost updates).

10.2 Conflict Resolution Performance

We compared the conflict resolution performance of our prototype of Tofu against Dropbox—the most popular public cloud storage system. The performance is expressed in terms of the time to converge the replicas and the network usage for update propagation, with varying numbers of conflicting files. These measures stand for how efficient the conflict resolution is and how much overhead it has.

We knew that it would be unfair to compare an industrial level system, which has to handle many real-world cases, to a prototype, which has a lot of assumptions. Therefore, the interpretation of the benchmark results should be moderated, for example we do not directly compare the times it takes different systems to complete the same task; we focus more on whether a system could keep its efficiency and what is the increase in the overhead when the size of the test increases. In this experiment, we kept increasing the number of conflicts for this purpose.

For both systems, we deployed the same setup of four replicas. Each of the replicas is a virtual machines running Ubuntu Server 14.04 LTS, and each has the configurations of one CPU core and 1GB memory of the host; these replicas were in the bridge mode of networking. For the setup of Dropbox, the clients was with the *lansync* mode on; this mode is supposed to enable replicas in the same network to communicate directly.

On each of the replicas, we concurrently created n files, being named *1*, *2*, to n, thus creating n conflicts between each pair of replicas; we use a single session on each replica in our system. The content of each file was the one-byte identifier of the replicas (A, B, C, and D respectively) on which the file was created. We repeated the experiment with different values of n: 100, 400, and 900. We kept the number of replicas unchanged.

We measured the time from when we started to create the files on all replicas, until these replicas finished their synchronization. For our system *tofu*, it is easy to measure this time span because we can manipulate the code; however it is difficult to measure this information of the Dropbox system because its proprietary software

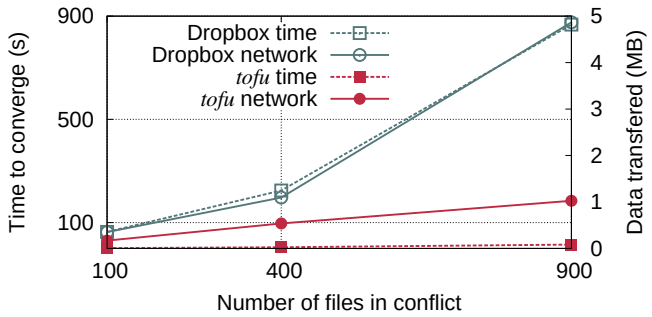


Figure 10.1: Evaluation of Dropbox and Tofu. Values are the average for each replica.

cannot be modified. Therefore we decided to use a manual method, in which we manually watched the status of all Dropbox's replicas and identified when they finished synchronizing. We used a daemon[1] provided by Dropbox to check the synchronization status of each replica, we identified the termination of synchronization only when the status of all replicas is "Up to date"; we used Linux tools `time` and `watch` with an interval of 2 seconds to monitor the replicas' status. In addition, we used another tool `iftop` to track the network usage of both systems.

The results of this experiment are shown shown in Figure 10.1; Tofu used much less time and network bandwidth to synchronize its replicas in all cases compared to Dropbox. More importantly, we see the exponential increase in both the time and the bandwidth that Dropbox used in synchronization as the number of conflicts increased. On the other hand, the increase of these measures for Tofu was linear. This can be explained by the synchronization mechanism of each system. As we inspected the network usage of Dropbox, we saw a lot of traffic between each Dropbox replica and `dropbox.com`. Furthermore, Dropbox's synchronization status showed that the status of each replica repeatedly changed between being stable ("Up to date") then being active ("synchronizing", "uploading", and "downloading") for multiple times before all of these replicas finally converged. From these observations, we believe that Dropbox uses a form of centralized synchronization where `dropbox.com` acts as the central point. As opposed to Dropbox, our prototype Tofu is totally decentralized in every sense (network traffic and conflict resolution); this led to the linear increase of time and network usage for synchronization when the number of conflicts increased.

[1] https://www.dropbox.com/download?dl=packages/dropbox.py

10.3 Micro-Benchmarks

In this experiment, we evaluate the impact of concurrent sessions on the normal usage of users in a session, and we analyze the impact of committing a session on a concurrent session.

10.3.1 Experimental Setup

Our experimental setup comprises two groups of storage servers inside a single data center. Each group comprises 6 physical servers, each of which has a 4-core Intel Xeon CPU, 64GB of memory, and 2TB of HDD storage. The network interface between all servers has a bandwidth of 1 Gbps. The average round-trip latency between these servers is 0.24 ms. We mimicked geo-distribution by adding latency to the network between these server groups; this latency is a normal distribution with the mean of 100 ms and the standard deviation of 10%. The bandwidth for the network between these server groups however remained unchanged.

We labeled the groups as replicas A and B; these groups act as the replicas of a geo-distributed file system. On each of the physical storage servers, we created 8 processes to serve as logical storage nodes for the metastore of each replica. There were thus 48 storage nodes per metastore deployment. On each of the replicas, we also deployed an instance of the commercial Scality Ring as the datastore of that replica. In each group, we included an additional server where clients could create mount points to connect to the local replica.

10.3.2 Normalization

In order to understand the performance limits that we could expect with our geo-distributed file system, we measured the best performance we can have for a single replica in our experimental setup.

Our specific strategy is to tune our system to find the best combination of concurrent mount points and number of concurrent processes for each mount point. We started by varying the number of concurrent processes on a single mount point of the file system. Each process sequentially created 10,000 directories under the root. As shown in the left sub-figure of Figure 10.2, the system achieved the best throughput-latency tradeoff at 420 ops and 5ms when using 2 concurrent processes. We therefore used 2 as the default number of processes for each mount point in our experiments.

We then varied the number of the mount points per replica with two concurrent processes each. We achieved the best throughput-latency tradeoff with the configu-

rations of 8 and 16 mount points per replica. The difference of throughput between these level is 5% with the setup of 16 mount points having higher throughput while the difference in latency is 83% with the setup of 8 mount points having lower latency (as shown by *local updates* in the right sub-figure of the Figure 10.2). We therefore used 8 as the number of mount points in our experiments because this setup enables our experiments to have a wider range of tradeoff when the concurrency changes.

We also included in the right sub-figure of the Figure 10.2 our conjecture of the performance characteristics that a geo-distributed file system may have using our experimental setup. There are two extremes in this sub-figure, one (solid line) describes the tradeoff between throughput and latency of the system under local updates while the other (dashed line) describes that of the global updates; global updates are those that require coordination between replicas to commit. The former workload represents that of the eventual consistency systems and the later workloads represents that of the strong consistency systems. In the local case, the latency was maintained at a good level around 5ms when we increased the number of mount points to 8. As we further increased the number of the mount points, the latency increased significantly. It reached the latency level of strong consistency approach at 640 mount points. We were not able to test with more mount points due to unknown technical issues. Meanwhile, the latency for global updates was at around 429ms when the number of mount points varied between 1 and 512, and started to deteriorate when more mount points were used.

Because of the way our conflict resolutions work, for each operation, we need to write it twice, one when writing it in the session and another one is when committing it. Therefore, we can expect the throughput of our system to theoretically be half of the optimal throughput of the setup.

10.3.3 Workloads

Due to the lack of standard workloads for eventual consistency geo-distributed file systems, we designed our own synthetic workload for our experiments. Our workload design is to satisfy the following testing criteria: (1) the workload should have enough contention level to put the file system to the limit, and (2) the workload should be wide enough to cover a large part of the file system; this second criterion is to test the impact of committing a long session whose modified elements span all over the system.

In our synthetic workload, we use 8 mount points per replica, each of which receives workloads from two concurrent processes. Each process creates a large number of directories (10,000) in the root directory; processes in the same session create different

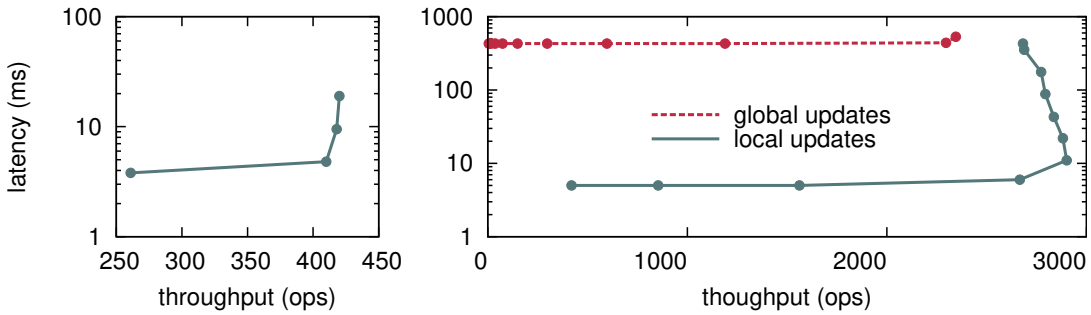


Figure 10.2: The baseline of the system for evaluation. Points of the left sub-figure show the throughput-latency information of a single mount point when the number of concurrent clients was changed between 1, 2, 4, and 8. Points of the right sub-figure show the throughput-latency information of a replica when the number of mount points ranging in $\{2^0, \ldots, 2^9, 640\}$. Solid line connects the points of local updates; dashed line connects those of global updates.

directories; all sessions create the same set of directories. We chose to have all sessions create the same sets of directories to create a large number of conflicts, which helps testing our conflict resolution system. The large number of directories to be created is also to fulfill the second criterion of the workload design which is to have a large coverage; this helps distinguishing our optimized commit protocol with the locking-based 2PC. We chose to only create directories under the root to have a clearer distribution of latency and throughput. This is because, in a file system, a single file system operation issued by users would then be translated into a set of operations on the elements on the absolute path of the target; this significantly increases the latency of the operations and makes it difficult to compare systems using random workloads. For example, to create a directory `/foo/bar/qux`, the file system has to check for the availability and the permission of each of the directories `/`, `/foo`, and `/foo/bar` before creating the desired directory. This will be translated into a large set of sequential accesses to the file system, which will hamper the latency in the case of global updates and will make comparing different approaches less fair. By using our workload, the latency of each operation is minimized due to the short paths of the targets.

10.3.4 Experiments

Overall

Our first experiment was to get an overview of the performance characteristics of our file system under a stress workload. In this experiment, we created 8 concurrent sessions on each of the replicas A and B; the sessions on each replica were initialized sequentially

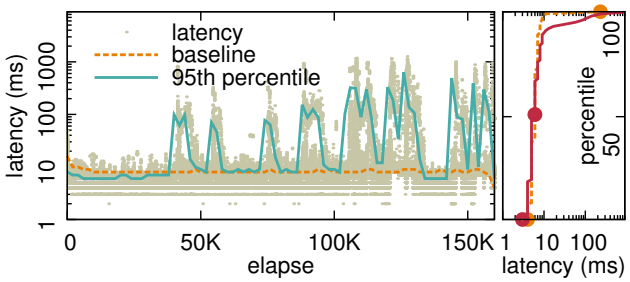


Figure 10.3: The overall experiment results showing 90% of updates was done in under 10ms with the overall average latency was 15.6ms. The left figure shows the latency distribution overtime; dots are measured values; the solid line is the 95th percentile level; the orange dashed line is the 95th percentile level of the baseline system. The right figure shows the CDF of the latency in the experiment.

with one started 5s after another. Each session had a dedicated mount point, to which each of 2 concurrent processes generated 10,000 directories. All sessions received the same set of directories from their concurrent processes. A session commits when it has finished creating all of these directories.

The results of the experiment (Fig. 10.3) show our expected outcome despite a large fluctuation in the latency during the experiment. Each replica in our system was able to fully commit all 16 sessions with the total of 320,000 directories in 271.594s from the start of the first session to the end of the last session. This gives our system in this test an average throughput of 1178 ops with an average latency of 15.6ms (not considering the delay of sequentially starting the session). This result is in line with our analysis that committing a session requires revisiting all modified entries of that session and thus may halve the throughput.

In a long period at the beginning, the latency remained stable around the baseline level, which is the latency of the single replica setup with 8 mount points and 2 processes each. The latency then fluctuated as each replica received more local and remote session commits. This fluctuation continued until all 16 sessions were fully committed. However, as shown in the latency distribution graph on the right of Figure 10.3, the large majority of the operations resulted in low latency with 90% under 10ms. The overall average was 15.6ms.

Session Commit

We further investigated decomposing the impact of the commit of a session by instrumenting our system to measure the exact moments when a session started and finished; we used a single session per replica, and a light workload with only 10K updates per

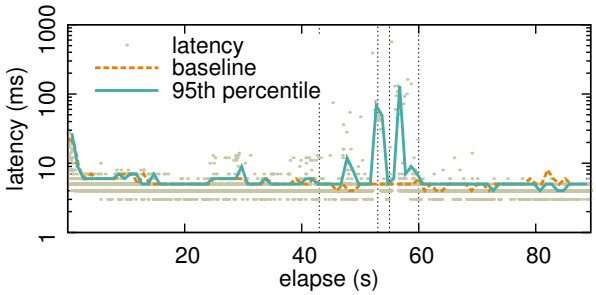


Figure 10.4: The latency during a session with the commit of another session. Vertical dotted lines are the events of committing another session.

session. The results in Figure 10.4 shows the throughput of operations on replica A when receiving the commit of a session from replica B. In this experiment, A started receiving the remote updates at the 43rd second from when the session on A started, and finished receiving the remote session from B after 10s until the 53rd second. A then received the instruction to start committing B's session at the 55th second and A finished committing in 5s. The first and second spikes of latency in between the period of 43rd and 55th seconds were likely due to the fact that A needed to apply all remote updates of the session on B into itself. The third spike happened when A started to commit the remote session which requires it to revisit all of the 10K directories to merge their session version into their main version.

Figure 10.5 shows the timeline of a session from when its first process started until its commit was finished on the local replica. This figure describes the (1) relationship between the time spans of writing and of committing, and (2) the time span for registering a session—the only moment when we actually lock a replica using the sequencer to decide the order of committing the session. We can see that the committing time, started when the session was registered, is longer than the amount of time spent for making all the directories. This is because the process of commit has to both resolve conflicts and merge the session version of each of the directories into their main version. The back-propagation process (which back propagates all updates up to the root in order to detect indirect conflicts) also took a large chunk of time; this is because back-propagations may have to go through multiple storage nodes on different servers of the metastore. The time that a session actually requires a global lock to decide its order (using the sequencer) is about 4ms; this is significantly less than the length for the whole commit process, which would have been the length of locking using locking-based 2PC to commit a session.

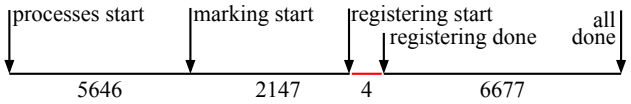


Figure 10.5: The timeline of the events of a session. Numbers show the timespans between events in milliseconds. Timespan scale is not supposed to reflect the real scale.

Chapter 11

Previous Work

In this section, we summarize the state-of-the-art geo-distributed storage systems, both academic and industrial. Our study subjects include legacy distributed file systems, as well as modern cloud storage systems, and related systems such as version control.

11.1 Distributed File Systems By Consistency

Distributed file systems can be classified based on their consistency model, into strongly- and eventually-consistent distributed file systems, and hybrids.

11.1.1 Strongly-Consistent Distributed File Systems

Traditional strongly-consistent distributed file systems provide the legacy POSIX API or similar. However, this inherently requires some form of coordination between replicas to commit updates, which increases update latency. Some, such as CalvinFS [71], Warp [17], and HopsFS [40], straightforwardly use distributed transactions. Distributed transactions in geo-setups suffer high latency; this is reported in the evaluation of these systems, where global operations commit in 300 ms to 1000 ms. Other systems, such as GlobalFS [45], SOFS [64], CFS [9, 10], and XtreemFS [27, 51] use a primary-backup approach; they partition data and assign to each partition a primary replica; all writes to a partition need to go through its primary. As an optimization, the primary of a partition can then be placed on the replica where most of the accesses originate. This approach works well with highly local workloads, but not with those spanning multiple partitions where committing updates will require inter-replica coordination.

Generally, strong consistency approaches fit well where manual reconciliation is difficult, and for highly parallel and disjoint short workloads (so that conflicts are less likely to occur). Otherwise, these approaches result in high update latency. In contrast,

our system provides low update latency; it is a good fit for long working sessions with complex structure of the file systems.

11.1.2 Eventually Consistent Distributed File Systems

Eventually consistent distributed file systems range from basic versioning to complex conflict resolution. These systems either do not provide automatic conflict resolution or do not provide the traditional file system API.

Distributed file systems usually use a versioning file system, such as [8,38,59–61], as the local file system of their replicas as a basic building block to support disconnected updates. With versioning-enabled replicas, distributed file systems can take the simple approach of exchanging and storing all concurrent versions of their data on all replicas without ever resolving the conflicts of these versions; Tofu without all conflict resolution mechanism is a versioning distributed file system; other examples of these systems include Ori [34] and TierStore [12]. Though they can provide the traditional POSIX API, these simple versioning distributed file systems always require manual intervention to resolve conflicts. Tofu differs from them by providing automatic conflict solution.

To another extent, eventually consistent file systems using fully automatic conflict resolution (such as GeoFS [68, 69], and the cloud storage systems [13, 23, 37]) do not provide the traditional POSIX semantics. Their automatic resolution causes trouble for legacy applications, which are not prepared for concurrency anomalies. The synchronization systems [3, 5, 29, 52, 53, 70] do not have the support for all file system components such as hard links. The conflict resolution of these systems is usually ad-hoc, for example, some move those conflicting into a special directory and notify users after that. These systems also sometimes use simple resolution approaches such as Last-Writer-Wins which does not retain all updates. Tofu offers a superior approach with support for both eventual consistency and legacy software.

The class of pioneer optimistic replication file systems such as Coda [30, 62], Ficus [56], Locus [48, 74, 75], Rumor [25], Roam [54, 55], Ivy [39], and TierStore [12] exhibit various degrees of limitations, similarly to the aforementioned cloud storage systems. They usually do not fully model file systems (by dropping hard links) and they usually opt for LWW or ad-hoc approaches (such as moving all conflicting directories and files into a special directory, which requires manual intervention to resolve conflicting updates).

The closest work to our system so far has been BatchFS [79]. BatchFS briefly studied the idea of using a single session per replica to improve the latency of the geo-distributed file system while maintaining the traditional POSIX semantics during a

session. This system however does not provide automatic conflict resolution or multiple sessions per replica.

Our system Tofu is classified as an eventually-consistent system. It offers a full solution compared to the others in this group. Tofu provides traditional POSIX semantics during each session; it automatically detects and resolves all conflicts between concurrent sessions; and it has a concurrency model that enables low-latency local commit of updates.

11.1.3 Consistency-Tunable Distributed File Systems

The Andrew File System (AFS) [26, 28] and OceanStore [31] are hybrids that can be either strongly consistent or eventually consistent, depending on usage. This is because, though enabling client caching, these systems require applications to manage the desired consistency themselves. In the case of AFS, a cache flush to an AFS server when a file is closed would overwrite anything before that without warning; applications must build their own logic and control to achieve their desired consistency level. AFS in a setup which has the coordination of the distributed applications would be a strongly consistent distributed file system; in another setup without any application coordination, AFS is an eventually consistent distributed file system which uses LWW exclusively to resolve conflicting updates. Whereas in the case of OceanStore, applications need to build and manage their proper set of tuples *predicates*, *update*, and *merge*, following Bayou's conflict resolution model [70]; these are the instructions for the primary replicas of OceanStore to resolve the conflict if any and commit an update. Compared to our approach, AFS and OceanStore do not have enough solutions to handle all conflict cases and they are not coordination-free for ensuring strong POSIX semantics.

11.2 State-Based And Operation-Based Approaches

Orthogonally to the consistency model, geo-distributed file systems can also be classified into two groups, based on their approaches to conflict resolution: *operation-based* and *state-based*.

Operation-based approaches keep a log (journal) of file system operations on each replica. A replica from time to time propagates its log to the other replicas. A receiving replica replays the log to keep its state consistent with the sender. Examples of this approach include IceCube [29], Bayou [70], OceanStore [31], and Ramsey's algebraic approach [52, 53].

The operation-based approach is however computing intensive. It usually requires calculating a new order of updates from all replicas, then applying this new sequence of updates on the replicas to converge them. This is not practical in real-world geo-distributed file systems. Moreover, an experience with a large scale file system [64] for a telecommunications service provider in France has shown that the number of operations in that real-world system is three orders of magnitude larger than the number of changed files and directories, which results in much larger log sizes as compared to the changes in the state of file system; reordering updates in such system is computationally expensive as compared to the state-based approach, which does less computation.

The state-based approach keeps track of the state of each file and directory in a replica, then propagates either the final states or deltas of the changed files and directories to the other replicas. Examples of this approach include Coda, Ficus, Unison [3], Andrew File System, and Microsoft's DFS-R [5]. Our system Tofu is also an instance of this approach.

11.3 Namespace-Based and Inode-Based Approaches

Another dimension is whether conflict resolution of a geo-distributed file system is *namespace-based* or *inode-based*.

11.3.1 The Namespace Approach

The namespace approach models a file system as a collection of paths, each of which represents a different directory or file. Merging the replicas of a file system computes the union of the path collections. Conflicts happen when the same path represents different contents. Examples of this approach are Unison [3], Dropbox [13], and version control systems such as Git [42] and SVN [2].

The namespace approach is free from the indirect conflict because the path of an updated file or directory acts as our update back-propagation; an updated directory at a specific path already ensures that a directory must exist at the same path after conflict resolution, this has the same effect as update back-propagation which is used to detect indirect conflicts in our system.

The namespace approach however does not fully model file systems. It assumes a *one-to-one* mapping between paths and inodes, and thus this approach does not take into account hard links. This incorrect model results in the waste of storage and bandwidth to store duplicate data contents, the divergence the replicas, and other anomalous behaviors as shown in our experiment with Dropbox (Section 10.1).

11.3.2 The Inode Approach

The inode approach models a file system as a collection of inode objects (or database records as in the case of DFS-R [5]). The namespace is stored in the directory inodes, and data is stored in the file inodes. Merging diverged replicas involves computing the union of the corresponding inode collections. Examples of this approach are Locus [46, 48, 74] and its descendants, such as Ficus [56] Rumor [25] and Roam [54, 55].

The inode approach however is prone to indirect conflicts where updates target elements that are on the same path but are stored in different inodes. Detecting and resolving indirect conflicts are usually ad-hoc. For example in Ficus and DFS-R, the directories that form a directory cycle are moved into some arbitrary directory for manual resolution later.

11.4 Other Conflict Resolution Systems

11.4.1 Merging framework

The problem of merging diverged replicas of a class inheritance graph has been discussed by Pottinger and Bernstein [49]; the problems in this work are similar to that of a file system: a class may have parents and children, a class may be concurrent updated and cycles between classes can be made by concurrent updates.

This work proposes some merging semantics including *element preservation* and *relationship preservation*. It presents some basic resolution algorithms for conflicting updates. For example, for a conflict in the *type-of* relationship between parent-child classes, which is known to be *one-to-one* similarly to the parent directory - child in a file system, this work solves the conflict in the same way as our system: it creates new types (classes) to store concurrent updates. A cycle between classes is either collapsed into a single class or requires manual intervention. Apart from the different domains, our work in file systems is different from this work in the objectives of merging, i.e., we try to preserve the structure of the replicas by using the No-Lost-Update Principle, while this work does not target that.

11.4.2 Version control systems

Git [42] and SVN [2] are representative examples of distributed and centralized version control systems, which could also be viewed as simplified file systems. Their main objective is to keep replicas of some project consistent, by keeping their namespaces synchronized. At the same time, there could be different versions of the project in different branches; concurrent updates to the same file in the same branch are automatically

merged together using three-way merging, but users are expected to manually resolve the conflicts semantically. Version control systems can be classified as namespace-based approaches as well.

11.4.3 Database Systems

The problem of resolving conflicting updates has also been studied in the field of database systems. A database models its data as a collection of tables in the case of traditional relational databases or as a key-value store in the case of modern NoSQL databases. A database supports operations to *insert*, *update*, or *delete* at the table row or key level. A conflict is a situation where a row or a key is concurrently created or updated. Databases support a very limited number of conflict cases, as compared to those in file systems. Because of the space constraints, we present only some conflict cases and conflict resolutions in Oracle [43], which is a representative example of relational database, and Dynamo [11] and Riak [4], which are examples of key-value stores.

In Oracle supports *update conflict*, *uniqueness conflict*, and *delete conflict*. An update conflict happens when a row is concurrently updated. Oracle resolves this conflict by using either the LWW approach or some specific conflict resolution algorithms for known data types, such as using an *additive* algorithm to aggregate the updated values or a row of numeric data. A uniqueness conflict happens when different rows with the same primary key are concurrently created; this conflict is resolved by adding a sequence, such as site identifier, to the value of the primary keys to make them unique. A delete conflict, which happens when a row is concurrently deleted and updated, requires manual intervention.

Similar cases arise in key-value stores, such as concurrently updating the value of the same key. For all of these cases, Dynamo and Riak solve this conflict by using either the LWW approach or by keeping these values as different versions of the key.

11.4.4 Collaborative Text Editing Systems

A collaborative text editor is a system that enables multiple users to concurrently edit a text-based document. Each user can edit a separate (and maybe disconnected) replica of the document.

Collaborative text editing systems share some common targets with geo-distributed file systems: they support asynchronous updates, and resolving conflicting updates to converge replicas. The approaches to collaborative text editing can be classified either as state-based or operation-based.

State-based approaches usually target synchronizing the state of the replicas to converge them; the state of a replica is usually represented by a list of characters—the building block—with their unique identifiers. Systems such as Woot [44] and Logoot [78] use the combination of identifiers such as replica name and document position to represent the index (unique identifier) of document characters; TreeDoc [50] uses a binary tree representation and RGA [58] uses a linked list respectively for their character positioning mechanism; for whatever they use, these systems usually resolve the concurrent edits at the same position by having these characters sharing the same or adjacent positions (a kind of LWW). In addition, by considering only *insert* and *delete* as the available operations, these systems can use the commutativity property of these operations as a way to resolve conflicts. The state-based approaches presented so far can work well with basic text-based documents and simple operations, but may face challenges in a real-world collaborative implementation where more advanced data types and operation types are used.

The operation-based approach may support better the advanced features and operations, but may be more computationally expensive. In a representative operation-based system, Operational Transformation (OT) [15,16], concurrent operations from different replicas are transformed into a new set of operations, such that applying this transformation on both replicas results in the same outcome of the document. By ensuring the preconditions of the operations and preserving their effect, OT can work better with advanced elements and operations than the state-based approach, though it is more computationally intensive.

We also conducted some experiments with the existing popular collaborative text editing systems Google Docs and Dropbox Paper. With the experience with the file system synchronizers, we set some expectations to the behaviors of these collaborative systems, such as commutative operations can be merged together, and merging concurrent edits preserves the users' intentions through preserving the updated states. In the first experiment, we tested a simple scenario in which we concurrently changed the type of a block of text to be **bold** and *italic*; both Google Docs and Dropbox Paper handled this situation well and merged these styles ***together***. In the next experiment, we moved two blocks of text into each other; after being converged, both text blocks were deleted in Google Docs, while in Dropbox Paper, their positions were swapped without retaining the result of the update on each replica. Though the replicas converged, we believe that the merging results could be improved, with respect to our presented principle of conflict resolution (Section 7.1).

Chapter 12

Conclusions and Future Work

In this thesis, we have presented our solutions to the problems of supporting asynchronous replication and of supporting the traditional API of geo-distributed file systems. We present the design and implementation of our prototype geo-distributed file system, named Tofu. Its design includes a new session abstraction to support the legacy API, while allowing optimistic updates. Unlike previous approaches, our solution is based on a formal model covering all aspects of a Unix-like file system, including directories, inodes, hard links, etc. It is able to detect all conflicts on those data structures, and resolves them in a way that we believe users will find generally reasonable. Experiments show that Tofu is highly scalable, and incurs linear overhead, and improving over existing academic and industrial systems.

In the future work, we will target some complementary features to Tofu, including garbage collection of old object versions in Tofu, support for sessions between sites, and micro sessions. We will also expand our study to related areas such as indexing and searching in such distributed storage, and conflict resolutions for collaborative text editing.

Bibliography

[1] Amazon. GET Bucket (List Objects) Version 1. http://docs.aws.amazon.com/AmazonS3/latest/API/RESTBucketGET.html. Accessed: 2017-01-01. (page 72)

[2] Apache. Subversion. https://subversion.apache.org/. Version 1.7. Accessed: 2014-12-31. (pages 48, 92, and 93)

[3] Sundar Balasubramaniam and Benjamin C Pierce. What is a file synchronizer? In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 98–108. ACM, 1998. (pages 2, 90, 92, and 113)

[4] Basho. Conflict Resolution. http://docs.basho.com/riak/latest/dev/using/conflict-resolution/. Accessed: 2015-04-27. (page 94)

[5] Nikolaj Bjørner. Models and software model checking of a distributed file replication system. *Formal methods and hybrid real-time systems*, pages 1–23, 2007. (pages 90, 92, and 93)

[6] Eric A Brewer. Towards Robust Distributed Systems. In *PODC*, page 7, 2000. (page 1)

[7] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *Programming Languages and Systems*, pages 116–135. Springer, 2011. (page 30)

[8] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 27–27, Berkeley, CA, USA, 2004. USENIX Association. (page 90)

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM. (page 89)

[10] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev.*, 35(5):202–215, October 2001. (page 89)

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007. (page 94)

[12] Michael Demmer, Bowei Du, and Eric Brewer. TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 3:1–3:14, Berkeley, CA, USA, 2008. USENIX Association. (pages 48 and 90)

[13] Dropbox. Dropbox. https://www.dropbox.com/. Accessed: 2014-12-31. (pages 2, 77, 90, 92, 113, and 136)

[14] Dropbox. Paper. https://www.dropbox.com/paper. Accessed: 2017-01-01. (page 48)

[15] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. *SIGMOD Rec.*, 18(2):399–407, June 1989. (page 95)

[16] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM. (page 95)

[17] Robert Escriva and Emin Gun Sirer. The Design and Implementation of the Warp Transactional Filesystem. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 469–483, Santa Clara, CA, 2016. USENIX Association. (page 89)

[18] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988. (page 73)

[19] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb. Accessed: 2017-01-01. (page 70)

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. (page 69)

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. (page 69)

[22] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002. (page 1)

[23] Google. Drive. https://www.google.com/drive/. Accessed: 2014-12-31. (pages 2, 77, 90, 113, and 136)

[24] Google. GSuite. https://gsuite.google.com/. Accessed: 2017-01-01. (page 48)

[25] Richard Guy, Peter Reiher, D Rather, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. *Lecture notes in computer science*, pages 254–265, 1999. (pages 90 and 93)

[26] John H Howard et al. *An Overview of the Andrew File System*. Carnegie Mellon University, Information Technology Center, 1988. (pages 2, 91, and 113)

[27] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS Architecture - a Case for Object-based File Systems in Grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, December 2008. (page 89)

[28] Michael Leon Kazar et al. *Synchronization and caching issues in the Andrew file system*. Carnegie Mellon University, Information Technology Center, 1988. (page 91)

[29] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218. ACM, 2001. (pages 90 and 91)

[30] James J Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992. (pages 2, 90, and 113)

[31] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000. (page 91)

[32] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. (pages 59, 73, and 131)

[33] Linux Information Project. Mount Point Definition. http://www.linfo.org/mount_point.html. Accessed: 2014-12-31. (page 33)

[34] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, History, and Grafting in the Ori File System. In *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles*, pages 151–166, New York, NY, USA, 2013. ACM. (pages 48 and 90)

[35] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989. (page 73)

[36] Microsoft. Office Online. https://www.office.com/. Accessed: 2017-01-01. (page 48)

[37] Microsoft. OneDrive. https://onedrive.live.com/. Accessed: 2014-12-31. (pages 2, 77, 90, 113, and 136)

[38] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 115–128, Berkeley, CA, USA, 2004. USENIX Association. (pages 48 and 90)

[39] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, December 2002. (page 90)

[40] Salman Niazi, Mahmoud Ismail, Steffen Grohsschmiedt, Mikael Ronström, Seif Haridi, and Jim Dowling. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. *CoRR*, abs/1606.01588, 2016. (page 89)

[41] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014. USENIX Association. (page 70)

[42] Open Source. Git. http://git-scm.com/. Accessed: 2014-12-31. (pages 48, 92, and 93)

[43] Oracle. Oracle Database Advanced Replication, 12c Release 1 (12.1). http://docs.oracle.com/database/121/REPLN/E53117-02.pdf. Accessed: 2015-04-27. (page 94)

[44] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 259–268, New York, NY, USA, 2006. ACM. (pages 39, 48, 95, and 121)

[45] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Rivière, and Pascal Felber. GlobalFS: A Strongly Consistent Multi-Site File System. Technical Report 2016/01, University of Lugano, April 2016. (page 89)

[46] Douglas Stott Parker Jr, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, (3):240–247, 1983. (page 93)

[47] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association. (page 70)

[48] Gerald J. Popek and Bruce J. Walker, editors. *The LOCUS Distributed System Architecture*. MIT press, 1985. (pages 90 and 93)

[49] Rachel A Pottinger and Philip A Bernstein. Merging models based on given correspondences. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 862–873. VLDB Endowment, 2003. (page 93)

[50] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A Commutative Replicated Data Type for Cooperative Editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, June 2009. (pages 39, 48, 95, and 121)

[51] Quobyte Inc. XtreemFS replication. http://www.xtreemfs.org/how_replication_works.php. Accessed: 2017-01-01. (page 89)

[52] Norman Ramsey and Elöd Csirmaz. An Algebraic Approach to File Synchronization. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 175–185, New York, NY, USA, 2001. ACM. (pages 90 and 91)

[53] Norman Ramsey and Elöd Csirmaz. An Algebraic Approach to File Synchronization. *SIGSOFT Softw. Eng. Notes*, 26(5):175–185, September 2001. (pages 90 and 91)

[54] David Ratner, Peter Reiher, and Gerald J Popek. Roam: A scalable replication system for mobile computing. In *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on*, pages 96–104. IEEE, 1999. (pages 90 and 93)

[55] David Ratner, Peter Reiher, and Gerald J Popek. Roam: a scalable replication system for mobility. *Mobile Networks and Applications*, 9(5):537–544, 2004. (pages 90 and 93)

[56] Peter L Reiher, John S Heidemann, David Ratner, Gregory Skinner, and Gerald J Popek. *Resolving file conflicts in the Ficus file system*. UCLA Computer Science Department, 1994. (pages 2, 76, 90, and 93)

[57] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 237–248, Piscataway, NJ, USA, 2014. IEEE Press. (page 70)

[58] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011. (pages 39, 48, 95, and 121)

[59] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: the file system that never forgets. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 2–7, 1999. (page 90)

[60] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to Forget in the Elephant File

System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 110–123, New York, NY, USA, 1999. ACM. (page 90)

[61] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to Forget in the Elephant File System. *SIGOPS Oper. Syst. Rev.*, 33(5):110–123, December 1999. (page 90)

[62] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on*, 39(4):447–459, 1990. (page 90)

[63] Scality. Ring. http://www.scality.com/ring/object-storage-overview/. Accessed: 2017-01-01. (page 69)

[64] Marc Segura, Vianney Rancurel, Vinh Tao, and Marc Shapiro. Scality's experience with a geo-distributed file system. *Proceedings of the Posters & Demos Session*, pages 31–32, 2014. (pages 89 and 92)

[65] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. (pages 2, 55, 113, and 127)

[66] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. (pages 2, 17, 55, 113, and 127)

[67] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. (page 69)

[68] Vinh Tao, Vianney Rancurel, and João Neto. A Name Is Not A Name: The Implementation Of A Cloud Storage System. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM. (page 90)

[69] Vinh Tao, Marc Shapiro, and Vianney Rancurel. Merging Semantics for Conflict Updates in Geo-Distributed File Systems. In *Proceedings of the 8th ACM Inter-*

national Systems and Storage Conference, Systor '15, New York, NY, USA, 2015. ACM. (page 90)

[70] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM. (pages 1, 90, 91, and 112)

[71] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, Santa Clara, CA, February 2015. USENIX Association. (page 89)

[72] Werner Vogels. Eventually Consistent. *Queue*, 6(6):14–19, October 2008. (pages 1 and 112)

[73] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009. (pages 1 and 112)

[74] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 49–70, New York, NY, USA, 1983. ACM. (pages 90 and 93)

[75] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. *SIGOPS Oper. Syst. Rev.*, 17(5):49–70, October 1983. (page 90)

[76] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. (page 70)

[77] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA, 2004. IEEE Computer Society. (page 70)

[78] S. Weiss, P. Urso, and P. Molli. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *2009 29th IEEE International*

Conference on Distributed Computing Systems, pages 404–412, June 2009. (pages 39, 48, 95, and 121)

[79] Qing Zheng, Kai Ren, and Garth Gibson. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, pages 1–6, Piscataway, NJ, USA, 2014. IEEE Press. (page 90)

Appendix A

Merging Inode *stat*

In the cases when we merge the contents of inodes, such as merging the *data* parts of directories, we also need to merge their *stat* parts.

Because different *stat* attributes store different properties of an inode, such as `st_atime` for last access time and `st_mode` for file permission, we need to merge these attributes differently. The merging algorithm for each attribute is described below.

`st_atime`, `st_ctime`, `st_mtime`: these are the attributes to store the timestamps of certain events of the inode. We use the LWW approach to merge concurrent values of `st_atime` (last access time) and of `st_mtime` (last modify time); the final value of these attributes is the latest among those concurrent. The `st_ctime` (creation time) of an inode, is never changed, therefore there is no concurrency on this entry.

`st_mode`, `st_gid`, `st_uid`: these attributes stores the inode's type, such as file or symlink, and security information (such as owner, group, and read/write/execute permission). The type of an inode never change; however the security information can be manually changed by users. We can systematically merge the concurrent updates to these attributes, but we cannot retain the security intention of users when changing these values; users need to manually check and modify it after conflict resolution if necessary. Nevertheless, we show how we merge concurrent updates on this attributes in Algorithms 12 and Algorithm 13. In these algorithms, we create a new user and/or a new group to represent the users and groups of the concurrent updates; we merge the permission settings (a sequence of bits) by using the bitwise `OR` operation.

The extended attributes of an inode are defined by developers; these attributes have their own meaning and their own usage. We therefore, again, can merge them technically, but we cannot preserve the intention of users with our own merging policy. Developers or users thus should have their own merging semantics for these properties in order to have their desired conflict resolution result; the implementation of these

merging semantics has to follow the CRDT rules of *idempotent*, *commutative*, and *associative*. We ignore extended attributes in this work.

The other system attributes, such as `st_nlink` and `st_size`, are changed when the *data* of an inode is changed. They are updated by the conflict resolution algorithms to reflect the results of merging the *data* part and of merging the *names* part of an inode.

Algorithm 12 Merge inode's `st_mode`

procedure MERGESTATMODE($mode_A$, $mode_B$, $mode_O$) ▷ $O \to (A \parallel B)$
  $mask_A \leftarrow \text{XOR}(mode_A, mode_O)$ ▷ A's modified bits since O
  $mask_B \leftarrow \text{XOR}(mode_B, mode_O)$ ▷ B's modified bits since O
  $mask \leftarrow \text{OR}(mask_A, mask_B)$ ▷ modified bits of A and B (bitmask)
  $mask' \leftarrow \text{NOT}(mask)$ ▷ inverse of bitmask
  $mode'_O \leftarrow \text{AND}(mask', mode_O)$ ▷ make all O's masked bits zero
  $masked \leftarrow \text{AND}(mask, mode_O)$ ▷ value of O's masked bits
  $masked' \leftarrow \text{NOT}(masked)$ ▷ toggled O's masked value
  $masked'' \leftarrow \text{AND}(mask, masked')$ ▷ clear toggled O's all but masked bits
  $mode \leftarrow \text{OR}(masked'', mode'_O)$ ▷ apply toggled bits to get result
  return *mode*

Algorithm 13 Merge inode's `st_gid` and `st_uid`

procedure MERGESTATOWNER($stat_A$, $stat_B$) ▷ inputs: stats
  $stat \leftarrow stat_A$ ▷ stat to return
  if $stat_A.st\_gid \neq stat_B.st\_gid$ **then**
    $gid \leftarrow$ MAKENEWGROUP($groupname$) ▷ new group, random name
    $users_A \leftarrow$ GETUSERSOFGROUP($stat_A.st\_gid$) ▷ get all users from group A
    $users_B \leftarrow$ GETUSERSOFGROUP($stat_B.st\_gid$) ▷ get all users from group B
    ADDUSERSTOGROUP($users_A, gid$) ▷ add A's users to common group
    ADDUSERSTOGROUP($users_B, gid$) ▷ add B's users to common group
    $stat.st\_gid \leftarrow gid$
  if $stat_A.st\_uid \neq stat_B.st\_uid$ **then**
    $uid \leftarrow$ MAKENEWUSER($stat.st\_gid, username$) ▷ new user, random name
    $i.stat.st\_uid \leftarrow uid$
  return *stat*

Appendix B

Résumé de la thèse

B.1 Abstract

Les systèmes géo-distribués souffrent de latences élevées et de partitions réseau. À cause de cela, et pour assurer une haute disponibilité, de tels systèmes effectuent généralement des mises à jour localement, sans latence, et les propagent ensuite en arrière-plan.

Cette réplication optimiste est confrontée à deux défis majeurs : (i) détecter les conflits entre les mises à jour simultanées et les résoudre d'une manière significative pour les utilisateurs, tout en maintenant les invariants d'intégrité du système; et (ii) la prise en charge d'applications qui n'ont pas été conçues pour gérer les anomalies de concurrence.

La recherche menée dans ce doctorat répond à ces défis pour le cas d'utilisation spécifique d'un système de fichiers géo-distribué à grande échelle. Ce cas s'y prête parfaitement : en effet, un système de fichiers géo-distribué a une structure hiérarchique complexe. Maintenir les invariants du système tout en le mettant à jour est une tâche compliquée. De plus les applications visualisent le système de fichiers via l'API POSIX qui n'a pas été pensée pour faire face à ce problème.

Les systèmes de fichiers géo-distribués optimistes existants ne permettent pas de relever ces défis. Par exemple, Dropbox ne supporte pas les liens matériels. Le système de fichiers AndrewFS échoue sur certains changements de noms de répertoires; et tous les systèmes existants utilisent la résolution automatique des conflits qui viole la sémantique POSIX.

Nous présentons notre solution aux problèmes posés ci-dessus dans la conception et la mise en œuvre d'un prototype de système de fichiers géo-distribué, nommé Tofu. Sa conception inclut une nouvelle abstraction de session pour prendre en charge l'API, tout en permettant des mises à jour optimistes. Contrairement aux approches précédentes,

notre solution est basée sur un modèle formel couvrant tous les aspects d'un système de fichiers Unix, y compris les répertoires, les nœuds d'index, les liens matériels, etc. Il est capable de détecter tous les conflits sur ces structures de données et de les résoudre d'une façon que nous pensons que les utilisateurs trouveront généralement raisonnable. Les expériences montrent que Tofu est hautement évolutif et qu'il entraîne des surcoûts linéaires, améliorant ainsi les systèmes académiques et industriels existants.

B.2 Introduction

Un système de fichiers géo-distribué s'étend généralement sur plusieurs réplicas distants les uns des autres. Le réseau inter-réplique d'un système de fichiers géo-distribué a une bande passante limitée et une latence élevée, en particulier comparé à la structure intra-réplica. Pour être disponible, un système de fichiers géo-distribué doit aborder le compromis inhérent entre cohérence et disponibilité, souligné par le théorème CAP [70, 72, 73].

Beaucoup de systèmes de fichiers géo-distribués à grande échelle optent pour l'approche de cohérence consécutive (Eventual Consistency - EC) [70,72,73]. Dans un système EC, une mise à jour est validée localement sur son réplica d'origine avant d'être propagée de manière asynchrone aux autres réplicas; EC s'assure que, lorsque toutes les répliques ont reçu et appliqué toutes les mises à jour des autres répliques, elles auront toutes le même état. Parce qu'un changement est local, sans coordination entre les réplicas, les utilisateurs ont une faible latence. Un utilisateur peut modifier une réplique, même si cette réplique reste déconnectée pendant une longue période, assurant ainsi une haute disponibilité.

Cette approche EC doit gérer les conflits entre les mises à jour simultanées provenant de différentes répliques. Un système de fichiers géorépliqué utilisant l'approche EC est confronté à deux défis majeurs: (i) détecter (tous) les conflits entre les mises à jour simultanées, et les résoudre de manière significative pour les utilisateurs, tout en maintenant les invariants d'intégrité du système; et (ii) la prise en charge des applications qui ne sont pas préparées pour gérer les anomalies de concurrence.

La détection et la résolution de conflits dans un système de fichiers à répartition géographique sont difficiles en raison de la structure hiérarchique complexe du système de fichiers. Un système de fichiers consiste généralement en une arborescence de répertoires; un répertoire peut contenir d'autres répertoires ou des fichiers; cependant, un fichier peut être inclus dans plusieurs répertoires, grâce à des liens matériels. Cette structure complexe est régie par des invariants stricts, tels que l'unicité des noms dans un répertoire, ou l'absence de cycles de répertoires.

En raison de la difficulté de détecter et de résoudre les conflits dans des systèmes aussi complexes, les approches existantes renoncent souvent à certains invariants du système ou limitent la capacité de détection et de résolution des conflits. Certains, notamment Dropbox [13] ou Unison [3], simplifient le modèle de système de fichiers en ignorant les liens matériels et en traitant plusieurs liens vers le même fichier comme des fichiers différents, ce qui entraîne des divergences entre les fichiers sur les différentes répliques des système de fichiers. D'autres, y compris Ficus [3], Coda [30], ou Microsoft OneDrive [37], laissent la résolution des cas de conflit difficiles aux utilisateurs; ils peuvent déplacer les répertoires et les fichiers impliqués dans un conflit dans un répertoire spécial, en attendant que les utilisateurs résolvent les problèmes manuellement. D'autres, notamment AFS [26] et Google Drive [23], optent pour une approche simple de Last-Writer-Wins (LWW) qui choisit une mise à jour arbitraire pour gagner les conflits; cette approche perd la durabilité de la mise à jour.

De plus, les systèmes de fichiers géo-distribués EC existants ne prennent pas bien en charge les applications anciennes. Ces systèmes changent automatiquement le système de fichiers à des moments inattendus et de manière non intuitive. Adapter les applications anciennes pour faire face à ce comportement nécessiterait une logique complexe. Par exemple, lorsque deux utilisateurs écrivent simultanément dans un même fichier, Dropbox et Google Drive résolvent le conflit en créant de nouveaux fichiers avec des noms quelque peu arbitraires. Cela peut même arriver lorsque le fichier est en cours d'utilisation. Les applications héritées ne sont pas préparées pour faire face à de tels changements soudains et imprévisibles.

Cette thèse aborde ces problèmes : pour prendre en charge les mises à jour simultanées et la compatibilité avec les applications anciennes. En conséquence, nous avons conçu un système de fichiers géo-distribué, nommé Tofu. Nos contributions sont les suivantes :

1. Conception et implémentation d'un mécanisme de détection et de résolution des conflits qui identifie et résout tous les conflits, basé sur : notre modèle de système de fichiers formel (Chapitres 2 et 3) qui supporte tous les composants du système de fichiers incluant les liens matériels (Chapitres 7 et 8) qui préserve toutes les mises à jour simultanées, tout en présentant des résultats de résolution de conflit significatifs. La mise en œuvre de Tofu (chapitre 9) est basée sur le concept de type de données répliquées sans conflit (CRDT) [65, 66] pour assurer la convergence et l'exactitude.

2. Le concept de session qui isole les applications POSIX des changements automatiques inattendus dans le système de fichiers. Notre système de session (Chapitres 4 et 5) divise l'utilisation d'un système de fichiers distribué en sessions. Une session est

une sorte de longue transaction, au sein de laquelle les applications bénéficient de la forte sémantique séquentielle d'un système de fichiers POSIX traditionnel. Plusieurs sessions peuvent coexister simultanément, chacune étant isolée des autres. Une session crée un instantané cohérent de l'ensemble du système de fichiers au début et fusionne atomiquement toutes ses modifications dans le système de fichiers à la fin. Tofu résout automatiquement toutes les mises à jour simultanées entre la session de validation et toute autre session validée auparavant. L'intervention manuelle n'est requise que si la résolution automatique des conflits a apporté des modifications incompatibles avec les applications des sessions ultérieures. Une telle intervention manuelle se limite à renommer les répertoires et les fichiers.

3. Les résultats expérimentaux montrant l'exhaustivité de notre approche par rapport à celles existant en ce qui concerne la capacité de résolution des conflits. Nous montrons à travers des expériences (Chapitre 10) que Tofu détecte et résout tous les conflits, y compris ceux qui représentaient une difficulté pour les systèmes précédents. Le système de session de Tofu est capable de fournir la faible latence pour les mises à jour de l'approche de cohérence éventuelle. Nous montrons également que la validation d'une session a un impact minimal sur la latence des autres sessions en cours.

B.3 Modèle de système de fichiers

Un système de fichiers est une collection de structures de données appelées nœud d'index. Un nœud d'index est identifié par un identifiant unique appelé numéro de nœud d'index. Chaque nœud d'index a trois composants, à savoir, son *métadonnées* (stat), son *nom* et ses *données*.

B.3.1 Métadonnées d'un nœud d'index

Cette partie stocke les attributs de métadonnées d'un nœud d'index, y compris les attributs dits système et les attributs étendus. Les premiers sont prédéfinis et sont créés et maintenus par le système de fichiers, bien que certains puissent être mis à jour par les utilisateurs. Par exemple, l'attribut `st_atime` d'un nœud d'index enregistre l'heure à laquelle le nœud d'index a été accédé pour la dernière fois; cet attribut est automatiquement mis à jour par le système de fichiers. D'autre part, l'attribut `st_mode`, qui stocke les informations de propriété du nœud d'index, est initialisé par le système de fichiers, mais il peut être mis à jour manuellement par les utilisateurs.

Les attributs étendus sont définis et gérés manuellement par les utilisateurs uniquement et sont opaques pour la sémantique du système de fichiers. Ces attributs sont

interprétés uniquement par des utilisateurs ou des applications, tels que ceux qui conservent la géolocalisation d'une image contenue dans la partie concernant les données du nœud d'index. Certains des attributs du système jouent un rôle mineur (par exemple, `st_atime`); ci-après, nous nous concentrons sur ceux qui sont essentiels pour le bon fonctionnement du système de fichiers. (1) Le type d'un nœud d'index est indiqué par l'attribut `st_mode`. Il existe deux types de nœud d'index principaux : répertoire et fichier. Un nœud d'index de type répertoire stocke le contenu du répertoire (une association des chemin d'accès aux numéros de nœud d'index) dans sa partie de données (section suivante). Un nœud d'index de type fichier contient des données arbitraires (opaques) dans sa partie données, comme un document texte ou une image. (2) Les informations de sécurité sont stockées dans les attributs `st_mode` (autorisations d'accès), `st_uid` (propriétaire) et `st_gid` (groupe). Ces informations dictent qui (utilisateur, groupe) aura quel type d'accès (lecture, écriture, exécution) au nœud d'index. (3) Les attributs restants contiennent des informations de comptabilité pour le nœud d'index, telles que les horodatages (`st_atime` : heure de dernier accès, `st_mtime` : heure de dernière modification et `st_atime` : heure de création), son emplacement (`st_dev` et `st_rdev` : l'identifiant de l'appareil contenant le nœud d'index, `st_ino` : numéro du nœud d'index), sa taille sur le disque (`st_size`, `st_blksize`, `st_blocks` : informations sur la taille réelle et sur la taille sur le disque du nœud d'index), et son nombre de liens physiques associés `st_nlink` (nous reviendrons sur les liens matériels dans la section suivante).

B.3.2 Données d'un nœud d'index

La partie de données d'un nœud d'index stocke le contenu utile de ce nœud d'index. Selon le type du nœud d'index, sa partie de données peut être significative ou opaque pour le système de fichiers. Par exemple, la partie de données d'un lien symbolique est une chaîne représentant l'emplacement d'un autre nœud d'index; le système de fichiers lit cette information et transmet l'accès vers sa destination.

La partie de données d'un répertoire est une table de noms (chaînes) associés à des numéros de nœud d'index (appelés nœuds d'index cibles). Chaque association de ce type constitue un lien direct avec le nœud d'index cible. Le système de fichiers utilise cette information d'association dans sa structure hiérarchique (section 2.2). Ci-après, nous appelons la partie données d'un répertoire sa carte enfant ou simplement mappage, et une entrée de mappage (lien physique) en tant que mappage.

La partie de données d'un fichier est opaque et est de type arbitraire, comme la séquence de caractères d'un document de texte, ou l'ensemble de pixels d'une image.

B.3.3 Noms d'un nœud d'index

La partie des noms d'un nœud d'index conserve une référence inverse à tous les liens matériels qui ont été mappés au nœud d'index. Nous représentons les noms sous la forme d'une carte de numéros de nœud d'index répertoire aux noms (de chaîne) du nœud d'index courant dans ces répertoires. Cette information devrait être cohérente avec `st_nlink` et avec le contenu des répertoires (ceci est formalisé par l'Invariant 5 dans la section 2.3). La partie noms est implicite dans POSIX et n'est pas explicitement matérialisée dans les implémentations précédentes. Notre modèle (et implémentation) la rend explicite car elle joue un rôle important dans l'exactitude du système de fichiers.

B.4 Structure du système de fichiers

Un système de fichiers peut être considéré comme un graphe où un nœud d'index constitue un nœud, et une association constitue un sommet. Dans les systèmes de fichiers courants, le sous-graphe d'un répertoire est en fait un arbre. Un nœud d'index lié à partir d'un répertoire s'appelle un enfant de ce répertoire. Inversement, la partie des noms d'un nœud d'index est constituée de la référence inverse de ses parents et de son propre nom au sein des parents.

Un répertoire peut avoir plusieurs enfants, mais n'a qu'un seul parent (la racine n'a pas de parent). Par conséquent, la partie noms d'un répertoire doit toujours avoir une seule entrée, et `st_nlink` d'un répertoire est toujours 1. La partie de données d'un répertoire peut avoir plusieurs entrées de mappage. En revanche, un fichier peut avoir plusieurs noms et un nombre de liens plus élevé.

Aucun enfant d'un répertoire donné ne peut avoir le même nom (un nom est unique dans son répertoire). Il ne peut y avoir de cycles dans une arborescence de répertoires. Un système de fichiers séquentiel assure cet invariant en rendant illégal la création d'un lien physique d'un répertoire vers l'un de ses ancêtres. En particulier, l'opération de changement de nom (`rename`) vérifie cette condition préalable, comme nous le verrons plus loin (section 2.3).

Une arborescence de répertoires possède un «répertoire racine» (souvent identifié par un numéro de nœud d'index spécifique, tel que «2» dans les systèmes de fichiers Ext pour Linux). Bien que le contenu du répertoire racine puisse changer, la racine elle-même ne peut pas être supprimée ou remplacée par un autre nœud d'index. Dans la tradition POSIX, la racine est notée `/`. Le chemin de la racine à un nœud d'index est appelé un chemin absolu de ce nœud. Un chemin absolu est une chaîne représentée par la concaténation des noms de tous les répertoires sur le chemin, séparés par des `/` ; par

exemple, le chemin `/foo/bar` indique que `foo` est un enfant de la racine `/` et que `bar` est un enfant de `foo`. Le nœud d'index nommé par `/foo/bar` peut être un répertoire ou un fichier; cela ne peut pas être déduit du nom seul. À aucun moment, deux nœuds n'ont le même chemin absolu. Parce qu'un répertoire a un seul parent, un répertoire a donc un seul chemin absolu.

Un fichier peut avoir plusieurs noms, et donc plusieurs chemins absolus. Par conséquent, le graphique complet des répertoires et des fichiers n'est pas un arbre, mais un cas particulier d'un ensemble d'ordres partiel (partial ordered set).

B.5 Invariants du système de fichiers

Cette section décrit les invariants qui caractérisent l'exactitude d'un système de fichiers.

B.5.1 Notations

Nous désignons la carte enfant dans un répertoire sous la forme d'un ensemble de paires (nom, ino), représentant respectivement le nom de la chaîne et le numéro de nœud d'index d'un nœud d'index cible. Nous utilisons la notation par points, par exemple: `m.name` et `m.ino`, où `m` est une entrée dans la carte enfant.

Nous utilisons les raccourcis `i.ino` pour `i.stat.st_ino`, le numéro du nœud d'index d'un nœud d'index i; `i.nlink` pour `i.stat.st_nlink`, le nombre de liens vers le nœud d'index i; `i.type` pour son type, qui est `DIR` si le nœud d'index est un répertoire, ou `FILE` sinon; et `d.map` à représente la carte enfant du répertoire d.

Nous désignons l'ensemble de tous les nœuds d'index par $\mathfrak{I}$, qui est rangé par $i, i_1, i_2, \ldots$; l'ensemble de tous les nœuds d'index privé de la racine comme $\mathfrak{I}^*$; l'ensemble de tous les répertoires comme $\mathfrak{D}$, rangés par $root, d, d_1, d_2, \ldots$; l'ensemble de tous les répertoires en excluant la racine comme $\mathfrak{D}^*$; et l'ensemble de tous les fichiers comme $\mathfrak{F}$. Les relations $\mathfrak{I} = \mathfrak{D} \cup \mathfrak{F}$ et $\mathfrak{D} \cap \mathfrak{F} = \varnothing$ sont vraies.

Nous définissons la relation d'accessibilité (*reachable*) pour indiquer si un répertoire est un descendant d'un autre, et la relation de parenté (*parent*) pour indiquer si un répertoire est le parent d'un nœud d'index; $parents(i)$ désigne l'ensemble de tous les

répertoires parents du nœud d'index i.

$$parent(d,i) = \begin{cases} \texttt{TRUE} & \exists m \in d.map : m.ino = i.ino \\ \texttt{FALSE} & otherwise \end{cases}$$

$$parents(i) = \{d | parent(d,i)\} \quad .$$

$$reachable(d,i) = \begin{cases} \texttt{TRUE} & parent(d,i) \vee (\exists d' : parent(d',i) \wedge reachable(d,d')) \\ \texttt{FALSE} & otherwise \end{cases}$$

B.5.2 Invariants

Les invariants suivants caractérisent les propriétés de sécurité d'un système de fichiers correct. Toute violation de ces invariants constituerait une erreur. Pour simplifier, nous nous limitons à un système de fichiers local.

Invariant 1 (racine fixe) *Un système de fichiers possède un nœud d'index racine unique et non modifiable.*

Invariant 2 (numéro de nœud d'index unique) *Le numéro d'un nœud d'index est unique dans son système de fichiers.*

$$\forall i, i' \in \mathfrak{I}, i.ino = i'.ino \implies i' = i$$

Invariant 3 (parent unique) *Un répertoire a un seul répertoire parent, à l'exception de la racine qui n'a pas de parent.*

$$\begin{cases} \forall d \in \mathfrak{D}^*, |parents(d)| = 1 \\ parents(root) = \varnothing \end{cases}$$

Invariant 4 (nom unique) *Les enfants d'un répertoire ont un nom unique dans ce répertoire.*

$$\forall d, \forall m, m' \in d.map, m.name = m'.name \implies m = m'$$

Invariant 5 (cohérence de mappage) *une entrée de mappage parent-enfant dans un répertoire est reflétée par un mappage inverse correspondant dans la partie des noms de l'nœud d'index enfant.*

$$\begin{aligned} \forall d, i, s, &\exists m \in d.map : m.ino = i.ino \wedge m.name = s \\ &\iff \exists n \in i.names : n.ino = d.ino \wedge n.name = s \end{aligned}$$

Invariant 6 (pas de cycles, accessibilité) *Un nœud d'index n'est pas accessible depuis lui-même, et un nœud d'index (sauf la racine) est accessible depuis la racine.*

$$\begin{cases} \forall i \in \mathfrak{I}, reachable(i, i) = \texttt{FALSE} \\ \forall i \in \mathfrak{I}^*, reachable(root, i) = \texttt{TRUE} \end{cases}$$

Invariant 7 (cohérence des métadonnées) *Le lien* `st_nlink` *d'un nœud d'index est le numéro du noms de cet nœud d'index.*

$$\forall d, f, \begin{cases} d.nlink = |d.names| = 1 \\ f.nlink = |f.names| \end{cases}$$

Invariants Nous définissons la conjonction de tous les invariants comme:

$$\text{Invariants} = \bigwedge_{i=1}^{7} \text{Invariant } i.$$

B.6 Cas de Conflit

Des sessions simultanées, dans une réplique ou entre des réplicas, peuvent mettre à jour des nœuds d'index de manière conflictuelle. Dans ce chapitre, nous étudions les cas de conflit.

B.6.1 Définition d'un conflit

L'exactitude de notre système de fichiers distribué est décrite par trois règles de sécurité: la convergence des répliques, la préservation des invariants et la cohérence des noms et des données.

La première règle permet aux répliques de diverger pendant un certain temps, tant qu'elles finissent par converger. La deuxième règle exige que chaque réplique conserve individuellement les invariants séquentiels décrits au chapitre 2.

Et la dernière règle est notre propre définition d'un conflit qui n'a pas été couverte par les règles précédentes : la situation où les mises à jour simultanées ciblent les noms et la partie de données d'un nœud d'index est un conflit. Nous décrirons la raison d'être de cette règle dans le chapitre suivant.

Toute paire de mises à jour simultanées qui enfreint l'une de ces règles de sécurité constitue un conflit.

Rule 4 (Convergence de réplica) *Les réplicas qui ont fourni les mêmes mises à jour ont le même état. Une réplique est requise pour éventuellement fournir toutes les mises à jour fournies par d'autres. Les mises à jour peuvent être appliquées dans n'importe quel ordre compatible avec leur commande partielle.*

Rule 5 (Intégrité du système) *À chaque réplique, et à chaque instant, ses invariants séquentiels doivent être vrais.*

Rule 6 (Cohérence des noms et des données) *Les mises à jour simultanées des noms et des parties de données d'un nœud d'index constituent un conflit.*

B.6.2 Types de conflict

En parcourant les règles de sécurité ci-dessus et en examinant les invariants (à l'exception de l'Invariant 6), nous pouvons voir que pour violer l'un d'entre eux il faut que les opérations simultanées ciblent les mêmes nœuds d'index.

Par exemple pour créer plusieurs noms pour un répertoire (violation de l'Invariant 3), les opérations concurrentes devraient mettre à jour la partie des noms de ce répertoire. Pour créer différents enfants avec le même nom dans un répertoire (violation de l'Invariant 4), les opérations concurrentes devraient mettre à jour la carte (partie de données) de ce répertoire.

Nous appelons donc ce type de conflit conflit direct; c'est le cas lorsque des mises à jour simultanées ciblent les mêmes nœuds d'index et enfreignent les règles de sécurité définies. L'autre type, s'il existe, est appelé conflit indirect; C'est le cas lorsque les mises à jour simultanées violent l'Invariant 6 même si elles ne ciblent pas le même nœud d'index. Les prochaines sections décriront ces types de conflits.

Pour simplifier l'analyse, nous traitons les parties statistiques et données d'un nœud d'index comme s'il s'agissait d'une seule donnée; cela signifie que la mise à jour des métadonnées d'un nœud d'index équivaut à mettre à jour les données de cet nœud d'index.

suppression-suppression. C'est le cas lorsque des sessions simultanées suppriment le même nœud d'index. Parce que les deux réplicas visent le même nœud d'index, ce cas de concurrence converge trivialement sans violer aucune des règles de sécurité.

suppression-mise à jour. Ce cas se produit lorsqu'une session supprime des nœuds d'index et qu'une autre session met à jour l'un des nœuds d'index supprimé. Parce que l'application des mises à jour simultanées dans différents ordres en utilisant les algorithmes effecteurs séquentiels respectifs (Chapitre 3) ne générerait pas le même

résultat (violant ainsi la règle de convergence - Règle 1), ce cas est un conflit et nécessite une résolution de conflit; nous appelons cela un conflit d'état.

mise à jour-mise à jour. Lorsque aucune des mises à jour simultanées n'est une suppression (Tableau 6.1), notre analyse dépend du type du nœud d'index ciblé. Dans les sections suivantes, nous décrirons les détails de ces cas pour chaque type d'nœud d'index.

1. **Conflit de données-données sur un fichier**. Lorsque les mises à jour modifient simultanément la partie données du même fichier, cela ne viole aucun des invariants (Règle 2), cependant, les appliquer dans différents ordres (en utilisant les algorithmes effecteurs du chapitre 3) peut donner des résultats différents (violant ainsi la règle de convergence - Règle 1). Si le fichier contient un type de données dont les mises à jour sont commutatives, comme un CRDT [44,50,58,78], les mises à jour simultanées peuvent être fusionnées selon les règles de ces types de données et converger; dans ce cas, les mises à jour simultanées ne sont pas conflictuelles. Cependant, en général, le type de contenu du fichier est opaque au système de fichiers et un fichier peut contenir n'importe quoi. Pour la généralité, nous ignorons ces types de données commutatives.
2. **Conflit de noms-données sur un fichier**. Lorsqu'une mise à jour modifie les noms et qu'une autre mise à jour modifie simultanément les parties de données du même fichier, ces mises à jour rompent la relation entre le nom et le contenu du fichier comme prévu par les utilisateurs (violation de la règle 3).
3. **Conflit de noms-noms sur un fichier**. Lorsque les mises à jour simultanées changent la partie des noms d'un fichier, elles peuvent créer le même nom et donc violer Invariant 4 (un nom est unique dans la carte d'un répertoire).
4. **Conflit de données-données de répertoire**. Alors que la partie données d'un fichier est opaque au système de fichiers, pour un répertoire, elle est constituée de la mappe enfant. Si nous l'implémentons en utilisant un type de données commutatif, alors les mises à jour simultanées peuvent être fusionnées.
   La fusion des mises à jour simultanées dans ce cas déclenche une fusion récursive des enfants du répertoire (entrées de mappage).
   Des conflits se produisent lorsque ces mises à jour ciblent les mêmes entrées de mappage ou qu'il existe différentes entrées de mappage portant le même nom. c'est un conflit car il viole l'Invariant 4 (un nom est unique sur une carte).
5. **Conflit de noms-données de répertoire**. Dans ce cas, les mises à jour simultanées rompent la relation entre différents noms et différents contenus d'annuaire, en violation de la règle 3.

6. **Conflit de noms-noms de répertoire**. Les mises à jour simultanées de la partie des noms d'un répertoire peuvent aboutir à plusieurs noms pour le même répertoire, violant l'Invariant 3 (un répertoire a un nom unique). Comme nous l'expliquons et le justifions dans la Section 7.2.5, notre résolution de conflit consiste à copier le répertoire afin de maintenir tous les mappages mis à jour.

Conflit indirect. En itérant à travers les invariants, nous pouvons voir que les opérations simultanées qui ciblent les mêmes nœuds d'index ne peuvent pas violer l'Invariant 6 (pas de cycles). Pour ce faire, les mises à jour doivent avoir un impact transitif d'une manière qui crée un cycle de répertoire. Nous décrivons les conditions pour que les mises à jour simultanées violent l'Invariant 6, puis notre définition de conflit indirect comme ci-dessous.

Formellement, considérons I_1 et I_2 comme les ensembles nœuds d'index mis à jour par les opérations op_1 et op_2, respectivement. La condition pour qu'ils violent l'invariant 6 est la suivante.

$$\begin{cases} I_1 \cap I_2 = \varnothing \\ \exists i_1 \in I_1, i_2 \in I_2 : \mathit{reachable}(i_1, i_2) \wedge \mathit{reachable}(i_2, i_1) \end{cases}.$$

Les conditions ci-dessus spécifient que la mise à jour simultanée même si ne cible pas nécessairement les mêmes nœud d'indexs, en fait créer un cycle de répertoire.

B.7 Résolution des conflits

Dans cette section, nous décrivons les principes généraux et les règles spécifiques pour la résolution des conflits.

B.7.1 Principes de résolution de conflit

La résolution d'un conflit consiste à présenter les effets des mises à jour simultanées d'une manière qui ne viole pas les règles de sécurité. Techniquement, toute approche qui préserve l'intégrité et l'exactitude du système serait sûre, y compris la perte de toutes les mises à jour. Bien sûr, nous préférons les solutions qui améliorent la vivacité du système et l'expérience utilisateur. En conséquence, nous proposons les principes suivants pour la résolution des conflits.

Principle 3 (Pas de mise à jour perdue (No-Lost-Updates - NLU)) *La résolution des conflits devrait préserver les effets de toutes les mises à jour.*

Il est clair que l'abandon arbitraire des mises à jour n'est pas souhaitable, mais cela se produit dans des approches telles que Last-Writer-Wins (LWW), qui résout un conflit en choisissant l'une des mises à jour et en supprimant les autres. Nous découplons le principe NLU en: (1) préservant le contenu des données de fichier mises à jour, et (2) préservant les chemins des répertoires mis à jour et des répertoires parents des fichiers mis à jour. Cette dernière règle implique, par exemple, que si un répertoire mis à jour a un chemin p, il doit y avoir un répertoire correspondant sur le chemin p après la résolution du conflit. Cela permet aux utilisateurs de voir leurs contenu mis à jour sur le même chemin et de ne pas les surprendre. C'est la raison d'être des noms de données d'un nœud d'index règle de cohérence (règle 3) du chapitre précédent.

Principle 4 (Pas de mise à jour fantôme (No-Ghost-Updates - NGU)) *La résolution des conflits ne devrait pas faire de mises à jour qui ne sont pas demandées explicitement par les utilisateurs.*

Ce principe empêche la résolution de conflits de créer de nouvelles mises à jour à partir de rien. Par exemple, la résolution d'un conflit sur un fichier `/foo` ne doit pas créer de répertoire `/bar` sans rapport avec le résultat.

Ces deux principes sont raisonnables, mais il est impossible de les suivre de manière rigide. En fait, nous montrons dans ce chapitre qu'il y a des situations où cela irait à l'encontre des règles de sécurité du chapitre 6, ou (si l'application simultanée des deux) viole la sémantique du système de fichiers des chapitres 2 et 3. Par exemple, préserver deux mises à jour simultanées d'un même fichier requièrent soit de conserver les deux versions dans ce fichier, ce qui n'est pas supporté par les API POSIX, soit de les stocker dans deux fichiers différents (avec des noms différents), ce qui viole le principe d'absence de mise à jour fantôme.

Notre décision de conception est, s'il n'y a pas d'autre option, de favoriser le principe NLU sur le principe NGU. Par exemple, dans l'exemple précédent, nous créons de nouveaux fichiers pour stocker les mises à jour simultanées. Bien que pour les répertoires, les mises à jour simultanées peuvent être fusionnées sans violer l'un ou l'autre principe.

Nous détendons également le principe NLU dans certains cas où les mises à jour simultanées sont en contradiction les unes avec les autres. Par exemple, lorsqu'un nœud d'index est simultanément supprimé et mis à jour, il est impossible de conserver les deux, et nous conserverons la mise à jour et ignorerons la suppression.

Dans les sections suivantes, nous détaillerons la résolution des conflits pour les cas de conflits directs et indirects du chapitre 6. Deux détails principaux seront présentés: la résolution des conflits est un processus récursif qui part du répertoire racine et descend

dans l'arborescence; et la résolution des mises à jour simultanées sur un nœud d'index peut nécessiter de générer de nouveaux nœuds d'index pour stocker ces mises à jour.

B.7.2 Détails de résolution de conflit

Le conflit supprimer-mettre à jour. Rappelons qu'un conflit de suppression-mise à jour se produit lors de la suppression et de la mise à jour simultanées du même nœud d'index. Notre algorithme de résolution dépend du type du nœud d'index, mais généralement, nous conservons la mise à jour et ignorons la suppression, violant ainsi NLU. Intuitivement, cela est dû au fait que les utilisateurs peuvent annuler manuellement une mise à jour si ce n'est pas le résultat de résolution de conflit souhaité, mais si une suppression a eu lieu, il n'existe aucun moyen direct de le restaurer. La résolution de ce conflit pour chaque type de nœud d'index est expliquée ci-dessous.

Lorsque le nœud d'index cible est un répertoire, nous conservons le répertoire mis à jour et l'utilisons comme résultat de la fusion, afin de maintenir le chemin mis à jour, suivant NLU. Cela permet de maintenir une structure familière du système de fichiers grâce à la résolution des conflits, en suivant la règle de cohérence noms-données (règle 3).

Lorsque le nœud d'index cible est un fichier, nous conservons le contenu mis à jour dans un nouveau fichier et supprimons celui d'origine (figure 7.1b) en privilégiant NLU sur NGU. Le chapitre 8 décrira comment nous générons le nom du nouveau fichier et le nouveau numéro du nœud d'index. Nous ne conservons pas la mise à jour dans le fichier original, afin d'unifier le mécanisme de résolution de conflit avec le conflit de données-données du fichier.

Le conflict données-données de fichier. Un conflit de données de fichier se produit lorsque des sessions simultanées mettent à jour la partie de données du même fichier. Notre algorithme de résolution de conflit consiste à conserver les deux mises à jour en suivant le principe NLU.

Notre approche consiste à créer de nouveaux fichiers avec des noms différents, afin de conserver les différents contenus mis à jour, tout en préservant l'Invariant 4 (un nom est unique dans la carte d'un répertoire). Cela casse le principe NGU, mais cela est nécessaire pour maintenir le principe NLU avec un système de fichiers qui ne supporte pas le versionnage. Les noms de ces nouveaux fichiers sont choisis pour sensibiliser les utilisateurs au conflit et pour le résoudre manuellement si nécessaire. Une approche simple consiste à ajouter un identifiant unique aux noms d'origine. Par exemple, nous utilisons `foo.A` et `foo.B` pour représenter les mises à jour simultanées de foo à partir des sessions A et B, respectivement. Le chapitre 8 décrit les détails de notre format

pour les nouveaux noms de fichiers, motivés par l'exigence de convergence.

Cette approche viole cependant le principe NGU puisqu'elle crée de nouveaux fichiers qui ne correspondent pas directement à une requête d'utilisateur. Nous discuterons des problèmes avec cette approche à la section 8.3.

Nous avons choisi cette approche car elle fonctionne avec n'importe quel type de fichier et car elle est compatibile avec la sémantique POSIX traditionnelle.

Le conflict noms-noms de fichier. C'est la situation où les sessions simultanées mettent à jour uniquement la partie des noms du même fichier. Comme le type de données de la partie noms d'un nœud d'index est une carte, tout comme la partie données (carte enfant) d'un répertoire, nous pouvons donc fusionner les mises à jour simultanées sur cette partie d'un fichier.

Dans le cas où un nom du fichier est mis à jour simultanément, nous stockons les mises à jour simultanées du nom dans de nouveaux noms afin de préserver ces mises à jour. Par exemple, lorsque les sessions simultanées A et B créent le même nom `/bar` pour un même nœud d'index, la résolution de ce conflit entraîne de nouveaux noms `/bar.A` et `/bar. B` pour préserver les mises à jour de A et B, respectivement. La création de nouveaux noms dans ce cas suit le même mécanisme de création de nouveaux noms pour les conflits de données de données de fichier (Section 7.2.2).

Le conflict données-données de répertoire. Les mises à jour simultanées de la carte enfant (la partie de données) d'un répertoire sont fusionnées car son type de données (une carte) est connu pour être fusionné. Rappelons que la carte d'un répertoire mappe les noms locaux aux nœuds d'index; la fusion prend l'union des entrées de mappage mises à jour des mises à jour simultanées.

La fusion des mises à jour simultanées dans ce cas est un processus récursif car des conflits peuvent se produire sur les entrées de mappage de ce répertoire. Des conflits surviennent lorsque des mises à jour simultanées modifient les mêmes entrées de mappage, violant ainsi l'Invariant 4 (un nom est unique dans la carte d'un répertoire). Un conflit sur une entrée de mappage nécessite une résolution de conflit sur cette entrée, ainsi qu'une résolution de conflit récursive sur les nœuds cibles de cette entrée. Par exemple, créer simultanément le même répertoire `/foo/bar` provoque un conflit de données de données de répertoire sur `/foo`; la fusion de ces mises à jour sur `/foo` fusionne récursivement les répertoires du même nom `/bar`.

Il existe plusieurs situations qui conduisent à ce type de conflit, selon : l'état des entrées de mappage (supprimées ou mises à jour), le numéro du nœud d'index cible (le même nœud d'index ou des nœuds d'index différents) et le type du nœud d'index cible (répertoire ou fichier). Le tableau B.1 montre les combinaisons possibles de ces

Table B.1: Cas de concurrence lorsque des mappages portant le même nom ont été mis à jour simultanément. Une ligne montre l'état des mappages (supprimées ou mises à jour), leur inode cible (identique ou différent) et les types des inodes cibles s'ils sont différents.

concurrent updates	**target**	**target type**	**resolution**
delete ‖ delete	-	-	1: no conflict
delete ‖ update	-	-	2: preserve update
update ‖ update	same	directory	3: recursive merge
	same	file	4: rename file
	different	directory + directory	5: recursive merge
	different	directory + file	6: rename file
	different	file + file	7: rename files

facteurs et notre algorithme de résolution de conflit pour chacun de ces cas.

Le conflit noms-noms de répertoire. Lorsque des sessions simultanées renomment le même répertoire en noms différents, le répertoire a des noms différents, violant ainsi l'Invariant 3 (un répertoire doit avoir un seul nom). Nous conservons les deux noms dans différentes copies du répertoire; faire de nouvelles copies du répertoire doit faire récursivement de nouvelles copies de ses enfants. Cette approche viole le principe 2 (mise à jour sans fantôme) mais préserve le principe 1 (mise à jour sans perte). Nous avons choisi cette approche car nous perdrions des mises à jour si nous choisissions de conserver un seul nom (LWW).

Conflit indirect. Comme cela a été décrit précédemment dans la Section 6.5.2, avec la rétropropagation de mise à jour, un conflit indirect devient un ensemble de conflits directs sur les ancêtres des nœuds d'index mis à jour des mises à jour simultanées. Cela permet de résoudre un conflit indirect et de résoudre les conflits directs, pour lesquels la résolution des conflits a été discutée dans la section précédente. Dans cette section, nous allons passer en revue l'exemple du cycle d'annuaire par des renommages (`rename`) simultanés.

La Figure 7.6 présente un exemple. Dans cet exemple, nous souhaitons que le nœud d'index 3 devienne un descendant du nœud d'index 6 (`mv /A/foo /B/bar/quz`), ceci met indirectement à jour les nœud d'indexs 2 et 4 en effectuant une rétropropagation de mise à jour. De même, faire du nœud d'index 4 un descendant du nœud d'index 5 met à jour indirectement les nœuds d'index 1 et 3. Les ensembles de nœuds d'index mis à jour des sessions sont respectivement $\{1, 2, 3, 4, 5\}$ et $\{1, 2, 3, 4, 6\}$. Cela se traduit par les conflits données-données de répertoire sur les nœuds 1 et 2, et les conflits noms-noms de répertoires sur les nœuds 3 et 4.

En résolvant les conflits données-données de répertoire sur les nœuds d'index 1 et 2, nous conservons les mappages `foo` et `bar`, respectivement; en résolvant les conflits noms-noms de répertoire sur les nœuds d'index 3 et 4, nous conservons les sous-arbres enracinés par ces nœuds d'index.

B.8 Convergence de réplique

Dans cette section, nous étudions les violations de la convergence des répliques (Règle 1), et nous proposons notre résolution de conflit. Notre approche est basée sur les CRDT [65,66], un ensemble de priciples pour des types de données finalement cohérents.

B.8.1 CRDTs

CRDT, qui signifie Conflict-Free Replicated Data Type, est un ensemble de principes pour les types de données répliqués afin d'assurer la cohérence éventuelle de leurs réplicas. Nous résumons ces principes et comment les appliquer pour faire converger nos répliques de système de fichiers comme ci-dessous.

État de réplique L'état de chaque réplique avance après une modification, par rapport à un ordre partiel prédéfini entre les états.

Fusion des répliques La fusion des états des réplicas simultanés calcule leur LUB (Least Upper Bound); le LUB de deux états est le moins parmi ces états égaux ou plus avancés qu'eux, par rapport à l'ordre partiel défini. Par définition, calculer LUB (noté $\oplus$) est idempotent, commutatif et associatif; ces propriétés sont formalisées comme ci-dessous, où s, s_i, s_j, s_k sont des états.

$$\begin{aligned} \textit{idempotent} : &\quad s \oplus s = s \\ \textit{commutative} : &\quad s_i \oplus s_j = s_j \oplus s_i \\ \textit{associative} : &\quad (s_i \oplus s_j) \oplus s_k = s_i \oplus (s_j \oplus s_k) \end{aligned} \quad . \tag{B.1}$$

B.8.2 Convergence de répliques à l'aide du CRDT

Un système de fichiers est défini comme une collection de nœuds d'index individuels $\mathfrak{I} = i_0, i_1, \ldots, i_n$, où un inode i_j peut avoir plusieurs versions $i_j^0, i_j^1, \ldots, i_j^m$, dont chacun est créé par une mise à jour du nœud d'index. Notez qu'une suppression crée une version appelée un marqueur de suppression (comme cela sera décrit dans la section 8.3.1); un marqueur de suppression apparaît aux utilisateurs comme si le nœud d'index avait été supprimé. Dans ce chapitre, nous considérons qu'un système de fichiers est un ensemble

de versions de nœuds d'index, et un état de système de fichiers correct est celui qui satisfait aux invariants du chapitre 2.

Nous définissons l'ordre partiel entre deux états de système de fichiers corrects S_A et S_B comme suit (nous utilisons $\|$ pour représenter la concurrence entre deux états, et $\cap$, $\cup$ et $\setminus$ pour définir des opérations d'intersection, d'union et de différence, respectivement):

$$\begin{cases} S_A = S_B & \iff S_A \cap S_B = S_A \wedge S_B \cap S_A = S_B \\ S_A < S_B & \iff S_A \cap S_B = S_A \wedge S_B \setminus S_A \neq \varnothing \\ S_A \parallel S_B & \quad otherwise \end{cases} . \tag{B.2}$$

Parce qu'une mise à jour sur un nœud d'index crée une nouvelle version de cet nœud d'index, l'état du système de fichiers avance donc vers le haut après chaque mise à jour, par rapport à notre définition de l'ordre partiel ci-dessus. En effet, considérons S^t et S^{t+1} sont des états de système de fichiers corrects avant et après une mise à jour, qui cible i_j et crée une nouvelle version i_j^k, alors nous avons $S^t \cap S^{t+1} = S^t$ et $S^{t+1} \setminus S^t = \{i_j^k\}$, donc $S^t < S^{t+1}$.

La fusion de deux états corrects produit également un autre état correct (limite supérieure) qui est égal ou plus avancé que les deux. Considérons une paire de mises à jour simultanées qui changent un état correct S aux nouveaux états corrects S_A et S_B, respectivement; ces mises à jour ciblent le nœud d'index i_j de S, et créent des versions différentes i_j^A et i_j^B, respectivement. La fusion de ces états permet de calculer l'union des états divergents, et de résoudre le conflit entre i_j^A et i_j^B, puis de stocker le résultat de la résolution des conflits dans une nouvelle version i_j^C (cas suppression-suppression, suppression-mise à jour, conflit données-données de répertoire) ou des nouveaux nœuds d'index i'_j, i''_j (cas conflit données-données de fichier et conflit noms-noms de répertoire). Dans tous les cas, l'état fusionné S_C ($S_C = S_A \cup S_B \cup \{i_j^C\}$ ou $S_C = S_A \cup S_B \cup \{i'_j, i''_j\}$) est toujours plus avancé que chacun des états divergents S_A et S_B.

Nous faisons également fusionner les états de système de fichiers corrects pour avoir les propriétés LUB, en ce qui concerne l'exactitude du système de fichiers et nos règles de fusion; les propriétés LUB sont l'idempotence, la commutativité et l'associativité. Bien que cela ne soit pas formellement prouvé, notre conjecture est que la borne supérieure calculée en fusionnant est le LUB des états.

Dans les sections suivantes, nous décrivons comment nous assurons les propriétés LUB de la fusion des états du système de fichiers, ce qui se traduit par la garantie des propriétés LUB de nos résolutions de conflits. Nous présentons les propriétés de commutativité et d'associativité de nos résolutions de conflit; la propriété d'idempotence

est automatiquement obtenue car il n'y a pas de concurrence dans le cas de $s \oplus s$.

B.8.3 Assurer la commutativité

Nous avons résolu les conflits en utilisant les actions principales suivantes: (1) générer de nouveaux noms de fichiers, (2) générer de nouveaux numéros d'nœud d'indexs, et (3) fusionner des répertoires. Assurer la commutativité de nos algorithmes de résolution de conflits implique d'assurer la commutativité de ces actions, ce qui implique la création d'un nouveau nom ou numéro d'nœud d'index ou la fusion déterministe des nœud d'indexs (tout en assurant bien sûr l'invariance des autres systèmes de fichiers). Nous décrivons comment nous réalisons le déterminisme pour ces actions ci-dessous.

Génération de nom de fichier déterministe

Le problème de la génération déterministe d'un nouveau nom de fichier peut être formellement représenté par une fonction déterministe $n' = f(n)$ où n et n' sont les anciens et les nouveaux noms de fichiers.

Techniquement, il y a plusieurs façons d'atteindre ce déterminisme. Pour notre système, nous choisissons f pour générer de nouveaux noms en ajoutant des informations au nom de fichier d'origine, selon le format suivant: *n.info.ino.ssid.hash*. Dans ce format, *info* décrit le type de conflit, tel que «conflit de données de fichier» ou «conflit de noms de fichiers»; *ino* est le numéro du nœud d'index d'origine du fichier; *ssid* est l'identifiant de la session qui a mis à jour le fichier; et *hash* est le résultat du hachage de tous les champs précédents. Par exemple, les noms générés `foo.data_conflict.12345.A.abcde` et `foo.data_conflict.12345.B.edcba` indiquent qu'ils ont été générés à partir de la résolution d'un conflit de données de données de fichier sur le fichier `foo` entre des mises à jour simultanées des sessions A et B, respectivement.

Nous avons décidé d'utiliser ce format car, premièrement, le processus est déterministe (en supposant que la fonction de hachage soit déterministe); deuxièmement, le nom informatif aide les utilisateurs à raisonner sur le conflit; et enfin, le nom généré est unique avec une probabilité élevée. Cependant, l'unicité absolue ne peut pas être garantie car un utilisateur peut créer manuellement n'importe quel nom arbitraire. Pour assurer l'unicité de façon fiable, il serait nécessaire de nommer les conventions ou de coordonner les répliques; ce dernier est possible au détriment de la disponibilité. Notre implémentation actuelle suppose simplement que de telles collisions de noms ne se produisent pas.

Pour simplifier la présentation, dans cette thèse, nous utilisons le format simpli-

fié *n.ssid* pour décrire les noms générés automatiquement; l'implémentation utilise le format complet.

Génération de nombre de nœud d'index déterministe

Le problème de commutativité de la génération d'un nouveau nœud d'index est de générer de façon déterministe un nouveau numéro de nœud d'index pour le nouveau nœud d'index, de sorte que les réplicas aient le même numéro d'nœud d'index pour le même contenu de fichier.

Pour garantir le déterminisme, nous calculons un nouveau numéro de nœud d'index i' comme résultat du hachage de l'identifiant *ssid* de la session de mise à jour et du numéro de nœud d'index d'origine i, comme dans: $i' = hash(i, ssid)$.

L'unicité des entrées assure l'unicité de la sortie avec une probabilité élevée, en supposant une bonne fonction de hachage. Cependant, des collisions de numéros de nœuds d'index peuvent toujours se produire en fonction de la fonction de hachage utilisée. Comme ci-dessus, une coordination mondiale serait nécessaire pour assurer l'unicité. Dans notre implémentation, nous n'assumons pas non plus de collisions de hachage.

Fusion déterministe des répertoires

Lorsque la carte d'un répertoire est modifiée simultanément, ou lorsqu'il existe plusieurs répertoires portant le même nom, nous fusionnons les cartes mises à jour dans un seul nœud d'index de type répertoire. La commutativité de la fusion des répertoires se réduit pour s'assurer que le nœud d'index répertoire pour stocker le résultat de la fusion est choisi de façon déterministe, assurant ainsi la commutativité de la résolution des conflits.

Différentes approches sont possibles, par exemple, générer un nouveau nœud d'index basé sur la combinaison des nœuds d'index d'entrés, ou choisir arbitrairement un nœud d'index parmi ceux qui sont en conflit (comme dans LWW). Cependant, pour sa simplicité connue et son support pour la commutativité et l'associativité, nous choisissons l'approche LWW pour stocker le résultat de la fusion dans le nœud d'index dont le numéro du nœud d'index est plus grand.

Considérons, par exemple, la fusion de trois répertoires dont les inodes sont 1, 2, et 3, respectivement. Lors de la fusion dans l'ordre $(1 \oplus 2) \oplus 3$, nous allons choisir le nœud d'index 2 pour stocker le résultat de $1 \oplus 2$, puis choisir le nœud d'index 3 pour stocker le résultat de la fusion de 2 avec 3. Le même résultat vaut pour un autre groupe $1 \oplus (2 \oplus 3)$; la résolution résultat en 3 comme le nœud d'index final pour stocker le

résultat de fusion des inodes.

B.8.4 Assurer l'associativité

Dans cette section, nous présentons le problème de la réalisation de l'associativité. Pour cela, nous introduisons un nouveau concept, le marqueur de suppression.

Le problème de détection conflit supprimer-mettre à jour

Dans une implémentation de système de fichiers POSIX sans versionnage, une suppression supprime physiquement l'entrée de mappage cible et le nœud d'index (si son `nlink` atteint 0) du système de fichiers. Cela rend cependant difficile de dire s'il existe une concurrence de suppression de mise à jour, ou s'il y a seulement une mise à jour sur un nœud d'index.

Afin de détecter les cas de simultanéité delete-update, nous utilisons le concept de marqueur de suppression. Dans notre système de fichiers, une suppression ne supprime pas sa cible (mapping et nœud d'index), mais change sa cible en marqueur de suppression. Un marqueur de suppression reste interne à notre implémentation et n'est visible à aucune opération du système de fichiers. Toutefois, la détection des conflits de mise à jour-mise à jour prend en compte les marqueurs de suppression.

Dans les sections suivantes, nous passerons en revue tous les cas de concurrence entre trois mises à jour et montrerons que nos algorithmes de résolution de conflit sont associatifs à l'aide du marqueur de suppression.

Cas de concurrence entre trois mises à jour

Entre trois mises à jour A, B et C, il y a trois possibilités de classement comme suit (nous utilisons $\rightarrow$ pour représenter la relation indiquant qu'un événement est arrivé avant un autre événement, suivant la notation de Lamport [32], et $\parallel$ pour la simultanéité): $A \rightarrow B \rightarrow C$, $A \rightarrow (B \parallel C)$, $(A \rightarrow B) \parallel C$ et $A \parallel B \parallel C$.

Dans le premier cas, les mises à jour sont commandées de manière causale. Pour cette raison, ces mises à jour suivent la sémantique séquentielle décrite au chapitre 2; il n'y a pas de conflits entre les mises à jour. Dans le second cas, les mises à jour simultanées (B et C) se produisent toutes deux après A; résoudre ce cas équivaut à fusionner B et C, donc à faire converger les répliques. Nous nous intéressons aux deux derniers cas de concurrence et présenterons notre approche pour assurer l'associativité pour ces cas.

Associativité pour les mises à jour liées de manière causale

Cette section traite de l'associativité pour la situation de concurrence avec mises à jour causales; c'est le cas de $(A \rightarrow B) \parallel C$ comme décrit précédemment. Considérons S_A, S_B, et S_C sont les états du système de fichiers que les résultats des mises à jour, respectivement, l'objectif de l'associativité pour ce cas est d'avoir $(S_A \oplus S_B) \oplus S_C = (S_C \oplus S_A) \oplus S_B$. Cependant parce que $A \rightarrow B$, donc $S_A < S_B$, donc $S_A \oplus S_B = S_B$, l'exigence d'assurer l'associativité devient: $S_B \oplus S_C = (S_C \oplus S_A) \oplus S_B$.

L'exigence n'est cependant pas satisfaite de notre résolution de conflit algorithmes jusqu'à présent. Quand ils génèrent automatiquement de nouveaux fichiers; les noms de ces nouveaux fichiers ne suivent pas la relation causale entre les session. Par exemple, lorsque A, B et C mettent tous à jour la partie *data* d'un fichier `foo`, même si $S_A < S_B$, le résultat de $S_A \oplus S_C$ n'est pas subsumé par $S_B \oplus S_C$, donc $(S_C \oplus S_A) \oplus S_B \neq S_B \oplus S_C$ comme suit:

$$\underbrace{\texttt{foo.B, foo.C, foo}(\textit{marker})}_{S_B \oplus S_C} \neq \underbrace{\underbrace{\texttt{foo.A, foo.C, foo}(\textit{marker})}_{(S_C \oplus S_A)} \oplus S_B}_{\texttt{foo.A, foo.B, foo.C, foo}(\textit{marker})}.$$

Nous pouvons voir cela sur le côté droit, même si S_A est subsumé par S_B, la résolution de conflit existante génère toujours `foo.A` pour stocker S_A `foo`.

Pour résoudre ce problème, lors de la résolution du conflit d'une session B avec un un C simultané, nous supprimons le résultat de la résolution de A (tel que $A \rightarrow B$ et $\nexists A' : A \rightarrow A' \wedge A' \rightarrow B$) et C si $A \parallel C$. Dans l'exemple ci-dessus, lors de la fusion $S_B \oplus S_C$, nous supprime `foo.A` (le résultat de $S_A \oplus S_C$) parce que $A \rightarrow B$ et $A \parallel C$; le résultat final est le {`foo.B` désiré, `foo.C`, `foo` (*marqueur*)}.

Associativité pour la simultanéité complète sur les nœuds d'index

Cette section décrit les cas de concurrence entre trois sessions simultanées $A \parallel B \parallel C$ qui met à jour le même nœud d'index i. La table B.2 montre tous les cas de concurrence possibles. dans le suivant, nous montrons que notre résolution de conflit pour chacun de ces cas est associatif.

Nous utilisons *deleted* et *updated* pour faire référence à différents états d'un nœud d'index i; i_A, i_B et i_C sont des nœuds d'index générés lors de la résolution les conflits des sessions respectivement.

Cas 1: C'est le cas lorsque toutes les sessions suppriment le nœuds d'index. Parce que la résolution des suppressions simultanées d'un nœuds d'index aboutit à un nœuds

Table B.2: Cas de concurrence sur un nœuds d'index entre trois mises à jour.

concurrency case	**merging result**	
	directory	*file*
1: deleted ‖ deleted ‖ deleted	deleted	deleted
2: deleted ‖ deleted ‖ updated	updated	updated
3: deleted ‖ updated ‖ updated	data/naming conflict	data/naming conflict
updated ‖ updated ‖ updated		
4: data ‖ data ‖ data	merged	split
5: data ‖ data ‖ names	merged + copy	split
6: names ‖ names ‖ data	copies	split + merged
7: names ‖ names ‖ names	copies	merged

d'index supprimé, résoudre un nombre quelconque de suppressions simultanées dans un regroupement entraîne toujours un nœuds d'index supprimé.

$$\underbrace{\underbrace{(deleted \oplus deleted)}_{deleted} \oplus deleted}_{deleted} = \underbrace{deleted \oplus \underbrace{(deleted \oplus deleted)}_{deleted}}_{deleted}$$

Cas 2: C'est le cas lorsqu'il y a deux suppressions et une mise à jour sur un nœuds d'index. Pour le cas d'un répertoire:

$$\underbrace{\underbrace{(deleted \oplus deleted)}_{deleted} \oplus updated}_{updated} = \underbrace{deleted \oplus \underbrace{(deleted \oplus updated)}_{updated}}_{updated}.$$

Pour le cas d'un fichier:

$$\underbrace{\underbrace{(deleted \oplus deleted)}_{deleted} \oplus updated}_{deleted + i_C} = \underbrace{deleted \oplus \underbrace{(deleted \oplus updated)}_{deleted + i_C}}_{deleted + i_C}.$$

Nous pouvons voir que dans tous les cas, commettre les mises à jour dans un groupe a le même résultat. Lorsque i est un répertoire, le résultat final est la mise à jour annuaire. Quand i est un fichier, i est un marqueur et i_C Mise à jour de C (comme décrit dans le chapitre 7).

Cas 3: C'est le cas lorsque l'une des sessions simultanées supprime le nœuds d'index et les deux autres le mettent à jour. Pour tout type de nœuds d'index, le conflit entre ces sessions sur le nœuds d'index devient le conflit entre le deux mises à jour. La résolution

du conflit entre une paire de mises à jour sur un nœuds d'index est commutatif comme décrit précédemment, nous avons donc le même résultat lorsque engager ces sessions dans n'importe quel groupe. Pour le cas i est un répertoire:

$$\underbrace{\underbrace{(deleted \oplus updated)}_{updated} \oplus updated}_{dir.\ data/naming\ conflict} = \underbrace{deleted \oplus \underbrace{(updated \oplus updated)}_{dir.\ data/naming\ conflict}}_{dir.\ data/naming\ conflict}.$$

Pour le cas i est un fichier:

$$\underbrace{\underbrace{(deleted \oplus updated)}_{deleted + i_B} \oplus updated}_{deleted + i_B + i_C} = \underbrace{deleted \oplus \underbrace{(updated \oplus updated)}_{deleted + i_B + i_C}}_{deleted + i_B + i_C}.$$

Cas 4: C'est le cas lorsque les sessions ne mettent à jour que les *data* partie du nœuds d'index. Pour i est un répertoire, parce que nous pouvons fusionner les mises à jour ensemble, le résultat final de la résolution des conflits est toujours i stocker la fusion des mises à jour simultanées. Ce cas est associatif.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(dir.\ data\ conflict)\\ merged(A+B)}} \oplus data}_{\substack{(dir.\ data\ conflict)\\ merged(A+B+C)}} = \underbrace{data \oplus \underbrace{(data \oplus data)}_{\substack{(dir.\ data\ conflict)\\ merged(B+C)}}}_{\substack{(dir.\ data\ conflict)\\ merged(A+B+C)}}.$$

Pour i est un fichier, les mises à jour simultanées dans ce cas provoquent deux à deux conflits données-données de fichier sur le nœuds d'index. Résoudre les conflits dans n'importe quel ordre génère i_A, i_B et i_C stockant les mises à jour de A, B et C, respectivement.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B}} \oplus data}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B + i_C}} = \underbrace{data \oplus \underbrace{(data \oplus data)}_{\substack{(file\ data\ conflict)\\ deleted + i_B + i_C}}}_{\substack{(file\ data\ conflict)\\ deleted + i_A + i_B + i_C}}.$$

Cas 5: Dans ce cas, deux sessions mettent à jour la donnée de le nœuds d'index et l'autre session met à jour le noms de celui-ci. Pour i est un répertoire, résolvant ce conflit fusionne les mises à jour des données du répertoire pourquoi générer un nouveau

répertoire pour la mise à jour de la partie noms; la processus de fusion est comme ci-dessous.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(dir.\ data\ conflict)\\ merged(A+B)}} \oplus names}_{\substack{(dir.\ naming\ conflict)\\ merged(A+B)+i_C}} = data \oplus \underbrace{\underbrace{(data \oplus names)}_{\substack{(dir.\ naming\ conflict)\\ i+i_C}}}_{\substack{(dir.\ data\ conflict)\\ merged(A+B)+i_C}}.$$

Pour i est un fichier, la résolution de ce conflit génère de nouveaux fichiers pour stocker le met à jour la partie donnée, en conservant la dernière mise à jour fichier original.

$$\underbrace{\underbrace{(data \oplus data)}_{\substack{(file\ data\ conflict)\\ deleted+i_A+i_B}} \oplus names}_{\substack{(file\ data\ conflict)\\ deleted+i_A+i_B+i_C}} = data \oplus \underbrace{\underbrace{(data \oplus names)}_{\substack{(file\ data\ conflict)\\ deleted+i_B+i_C}}}_{\substack{(file\ data\ conflict)\\ deleted+i_A+i_B+i_C}}.$$

Cas 6: Le conflit dans ce cas est entre deux mises à jour simultanées à la partie noms du nœuds d'index, et une mise à jour de sa partie donnée. Pour i est un répertoire, notre résolution de conflit génère de nouveaux répertoires à stocker les mises à jour de la partie noms du répertoire, tout en Inode d'origine pour stocker la mise à jour dans la partie donnée.

$$\underbrace{\underbrace{(names \oplus names)}_{\substack{(dir.\ naming\ conflict)\\ deleted+i_A+i_B}} \oplus data}_{\substack{(dir.\ naming\ conflict)\\ i_A+i_B+i}} = names \oplus \underbrace{\underbrace{(names \oplus data)}_{\substack{(dir.\ naming\ conflict)\\ i_B+i}}}_{\substack{(dir.\ naming\ conflict)\\ i_A+i_B+i}}.$$

Pour i est un fichier, notre résolution de conflit génère un nouveau fichier pour stocker le met à jour la partie donnée du fichier, tout en conservant la fusion des autres mises à jour dans un autre fichier.

$$\underbrace{\underbrace{(names \oplus names)}_{\substack{(file\ naming\ conflict)\\ i}} \oplus data}_{\substack{(file\ data\ conflict)\\ i+i_C}} = names \oplus \underbrace{\underbrace{(names \oplus data)}_{\substack{(file\ data\ conflict)\\ i+i_C}}}_{\substack{(file\ naming\ conflict)\\ i+i_C}}.$$

Cas 7: Dans ce cas, les sessions simultanées mettent à jour la partie noms du nœuds

d'index. Pour i est un répertoire, cela provoque deux consécutifs conflits donnée-donnée de répertoire. La résolution de ce cas est similaire à l'affaire précédente, sauf dans ce cas, notre résolution de conflit génère de nouveaux répertoires pour tous les mises à jour.

$$\underbrace{\underbrace{(names \oplus names)}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_A + i_B}} \oplus names}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_A + i_B + i_C}} = names \oplus \underbrace{\underbrace{(names \oplus names)}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_B + i_C}}}_{\substack{(dir.\ naming\ conflict)\\ deleted + i_A + i_B + i_C}}.$$

Pour i est un fichier, les mises à jour dans ce cas provoquent l'appariement conflits noms-noms de fichier. La résolution de ce cas consiste à fusionner tous les noms de chaque mise à jour dans le fichier original.

$$\underbrace{\underbrace{(names \oplus names)}_{\substack{(file\ naming\ conflict)\\ i}} \oplus names}_{\substack{(file\ naming\ conflict)\\ i}} = names \oplus \underbrace{\underbrace{(names \oplus names)}_{\substack{(file\ naming\ conflict)\\ i}}}_{\substack{(file\ naming\ conflict)\\ i}}.$$

B.9 Évaluation

Nous avons implémenté le prototype de notre système de fichiers géolocalisé Tofu dans NodeJS et FUSE.

Nous comparons le comportement de notre approche à celui des systèmes commerciaux, y compris: Dropbox [13], Google Drive [23], et Microsoft OneDrive [37]. Le critère de comparaison est le principes de résolution des conflits (Chapters 6 et 7); ceux-ci comprennent la convergence des répliques et la signification (Rules 1 et 2) des résultats de la fusion. Dans tous les cas, notre prototype a pu résoudre les conflits et produire les résultats souhaités, par rapport à la cible de notre sémantique de fusion, alors que les autres ne l'étaient parfois pas.

Nous avons comparé les performances de résolution de conflit de notre prototype de Tofu contre Dropbox—le système de stockage en nuage public le plus populaire. La performance est exprimée en termes de temps de converger les répliques et le réseau utilisation pour la propagation de mise à jour, avec un nombre variable de fichiers en conflit. Celles-ci les mesures représentent l'efficacité de la résolution du conflit et combien frais généraux il a. Le résultat de cette expérience est que Tofu a utilisé beaucoup moins de temps et de bande passante réseau pour synchroniser ses répliques dans tous les cas par rapport à Dropbox. Plus important encore, nous voyons l'exponentielle aug-

menter à la fois le temps et la bande passante que Dropbox utilisé dans synchronisation comme le nombre de conflits a augmenté.

En général, les expériences montrent que le tofu est hautement évolutif et qu'il entraîne des frais généraux linéaires, améliorant ainsi les systèmes académiques et industriels existants.