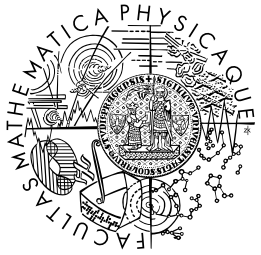


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Filip Štědronský

A decentralized file synchronization tool

Department of Applied Mathematics

Supervisor of the bachelor thesis:	Mgr. Martin Mareš, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

To Medvěd, my supervisor, a great friend and one of the best teachers I know. Someone who has way more answers than one person should. Whenever I encounter a problem from pretty much any field, the first subconscious impulse usually is: "Let's ask Medvěd, he will figure something out." But I learn to resist this impulse and instead try to acquire some of the tricks of his trade – relentless curiosity being one of the most important. Do not stop with a half-baked kinda-sorta answer. Think things through. Experiment. Poke. Change assumptions. Ask nagging questions.

To Karry, one of the few close friends I have ever had. An endless source of amazement, sometimes rumoured to have supernatural powers. She actually managed to get two master's degress almost faster than I will (hopefully) get my bachelor's! An inspiration to dare do (not try to do, simply do) more seemingly impossible things. You have made my life better in more ways than you can imagine.

To my dad, who is always supportive, even though he often thinks I'm crazy.

To all the random happenstances of evolution that gave us the ability to write bachelor theses and do a lot of other interesting stuff.

Title: A decentralized file synchronization tool

Author: Filip Štědronský

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., Department of Applied Mathematics

Abstract: We explore the problem of file synchronization, with the goal of improving on the efficiency, scalability, robustness, flexibility and security of current file synchronization tools. We solve several important subproblems that may help this, especially in the areas of filesystem change detection (both online and offline) and peer-to-peer synchronization of file metadata. We show techniques to make scanning a file system for changes faster and more reliable. We extend the Linux kernel's 'fanotify' filesystem change notification API to report more events, especially renames. We present several original solutions to the set reconciliation problem and its variants and apply them to metadata synchronization.

Keywords: file synchronization set reconciliation fanotify

Contents

Introduction

This thesis describes the design and implementation of a decentralized file synchronization tool called Filoco[1]. Filoco intends to be an alternative to commercial tools like Dropbox or Google Drive but one specifically tailored to advanced users, computer enthusiasts, matfyzák's[2], the paranoid and everyone else with specific needs not met by mainstream tools.

A prototypical Filoco user has a laptop, a backup laptop, a home computer, a work computer, a bedroom computer, several phones and tablets, 8 terabytes' worth of external hard drives, a home server, a NAS, and a VPS. A prototypical Filoco user has on the order of a few million files scattered across all these places: software, music, movies, books, audiobooks, notes, scripts, configuration files...

At different times, they need to use the same files on different devices. They usually transfer them between storage locations on an as-needed basis using ad-hoc methods such as thumb drives, rsync, scp, e-mail, personal git repositories, a `tmp` directory on their web server... This leads to several copies of each file, some of them temporary, some of them serving as backups, not all of them kept regularly up to date.

With such a setup, it is easy to lose track of all the places where a file is stored, let alone which of these places contains the current version. It is also easy to get to situations where you need a specific file, which is currently only stored for example on your home computer (currently turned off and far away) because you forgot to copy it to a server.

File synchronization tools try to alleviate these issues by automatically copying files between machines and keeping these copies up-to-date. However, most common synchronization tools have at least some of the following limitations, which make them less suitable for the user group described:

- These tools usually try to sync everything everywhere. This is a problem for users that have much more data in total than any single one of their computers or disks can hold.
- They often have limited scalability, especially with regard to the total number of files. There are many technical reasons for this that will be discussed in further chapters.
- They often require using proprietary software and/or cloud services (where files are often stored unencrypted). This can be hard to accept for people with a bit of healthy mistrust and paranoia or anyone unhappy about sharing their data with American three-letter agencies.
- Synchronization often must be performed via a centralized server in the cloud, even when devices are able to communicate directly. This causes problems if one wants for example to synchronize their phone with their laptop while travelling, with only a slow and/or expensive mobile internet connection available.
- It is usually not possible to use external drives as synchronization replicas.

[1] Short for File Locomotive (because it pulls your files around).

[2] The term *matfyzák* is colloquially used to refer not only to the students of Faculty of Mathematics and Physics but also to anyone bearing the personality traits typical for such students. In that sense, one does not become a matfyzák, one is born one. [1]

A more detailed survery of existing tools is given in chapter **??**.

Filoco tries to overcome these limitations. Its basic task is to synchronize data among a set of *stores*. A store is simply a directory containing ordinary files plus some additional Filoco-specific metadata. A store can be physically located on a desktop computer, mobile device, server, external drive or anything else with a file system. We call all the stores that are synchronized among each other a *realm*.

Filoco follows the philosophy of *global metadata, distributed data*. This means that while each store has copies of only some files, it has information about all the files in the realm. This metadata describes a single logical directory tree containing all the files in the realm that is consistent across all the stores.

The metadata also contains information about where each file is physically stored (an in what version). Thus that when a file is not available locally, Filoco knows where to fetch it from. If this is an offline store (e.g. an external drive or a powered down laptop), Filoco can ask the user to connect it and/or turn it on.

Upon request, any two stores can be synchronized, either via network or locally if the other is on an external drive connected to the same computer. All stores are equal, there is no special master store. The user can configure which files should be kept by which stores.

Apart from the basic concept outlined above, Filoco has the following design goals (in order of importance):

- *Scalability and efficiency.* We shall optimize specifically for the common case when the user has a lot of data, most of which changes only infrequently. Small incremental updates should be fast even when the total number of files is large (a few million). Ideally, the time complexity of most operations should not depend on the total number of files at all, only on the number/size of changed/affected files. This will not always be possible but we should try to get close to this ideal.

- *Robustness.* This means not only that it should not eat your data but also resilience to things like interrupted transfers, power failures or race conditions with other processes accessing the files managed by Filoco.

- *Flexibility.* Rather than a one-size-fits-all solution, Filoco should be a framework that each user can adapt to fit their unique needs and workflows. It should be both configurable and easy to integrate with shell scripts. Where possible, the user should be put in control. We should make as little policy decisions as possible.

- *Security.* This of course includes transport encryption and mutual authentication during network communication (nowadays taken for granted). It also includes the ability for designating *untrusted stores*, which (1) only store and exchange encrypted data (and metadata), without ever having access to the cleartext, (2) can only relay updates made by trusted stores (and cryptographically signed by them), not make their own changes to the data. Any (meta)data received from and untrusted store must be cryptographically verified to have been originally created by a trusted store.

  Otherwise, untrusted stores should be able to participate in normal synchronization, exchanging encrypted (meta)data with other stores, both trusted

and untrusted. This exchange should ideally be as efficient as (or close to) the unencrypted exchange between two trusted stores, including efficient incremental updates to encrypted files if possible (although this seems like a hard problem).

This would allow using any untrustworthy cheap cloud storage provider for additional storage, or as an intermediary for exchanging data between nodes behind NAT. The untrusted store will not be able to read nor modify your data. The only damage it can do is to delete your data, which can be alleviated by redundant storage on different stores.

Explicit non-goals include fancy GUIs and beginner-friendliness. Users are expected to have at least a basic understanding of Filoco's internals to make full use of it. The same is true for example for git.

Filoco runs only on Linux and there are currently no plans to support other operating systems (with the possible exception of Android, which is basically Linux).

1. Change Detection

The purpose of a file synchronization tool is simple: whenever a change to the synchronized tree is made in one replica, transfer the change to the other replicas and apply it there. From this stems a natural need for a way of detecting filesystem changes efficiently.

There are two broad categories of filesystem change detection methods.

Offline change detection consists of actively comparing the filesystem state to a known previous state. The detection must be explicitly initiated by the application at an arbitrarily chosen time, e.g. regulary (every day at midnight) or upon user request. It can be considered a form of active polling.

But polling is the lesser evil here. The real problem is that the comparison process usually involves recursively scanning the entire directory tree (or a subtree), saving the results and comparing them with the previous scan. This can be quite slow on larger trees, wherefore it cannot be done very often, leading to an increase in change detection latency.

Online change detection, on the other hand, relies on specific operating system features that allow applications to be notified of filesystem changes immediately as they happen. Instead of polling, the application just passively waits for change notifications. However, the notification systems often have many limitations, issues and idiosyncrasies. For example they fail to report some kinds of operations (e.g. renames) or operations done in specific ways (e.g. writes to a file via a memory mapping).

Even with a perfect notification system, we face a serious issue. The application monitoring the notifications must be running at all times. Notifications of filesystem changes made when the application is not running will be missed and forever lost to the application.

Due to both these issues, the application's idea about the state of the filesystem can diverge from reality over time. The only way of fixing this is with a full rescan of the directory tree. Thus while being efficient, online change detection is usually not very robust. In contrast, offline change detection is by definition 100% reliable, because it looks at the actual current state of the file system and updates internal structures accordingly. Actually, that is true only if the filesystem is not changed during the scan, as we shall see later.

There also emerges an interesting middle ground between these two extremes, which we shall dub **filesystem-based change detection**. Some filesystems can store some data about their change history as a part of their on-disk data structures and offer operations that query these structures to return information about filesystem changes. Two examples of this are the `btrfs` find-new and send-receive mechanisms.

This last category seems to offer the best of both worlds: we get reliably and efficiently informed of all changes. Often the comparison operation is fast enough to be run very frequently, for example every minute, effectively replacing online detection. The obvious disadvantages are that most filesystems do not support such operations and the need for a solution specifically tailored to each filesystem that does support change detection (there is no generic API, at least on Linux).

The next sections will survey various ways of doing each kind of change detec-

tion on Linux. Even methods that are not inherently filesystem-based will often depend on the idiosynchracies of different filesystem types. In such cases, we will consider primarily `ext4` and `btrfs`, two commonly used Linux filesystems, while remarking how other file systems may differ. For simplicity, we shall also only discuss change detection in trees contrained to a single filesystem volume (i.e., not containing any mount points within them), therefore also to a single filesystem type.

1.1 Offline Change Detection

1.1.1 The anatomy of linux filesystems

Before diving into change detection, we have to understand a bit about the structure of Linux filesystems and filesystem APIs. If terms like *inode*, *hardlink*, *file descriptor*, and `openat` are familiar to you, you can safely skip this section. Most of what is being said here applies to all Unix-based operating systems, however, some details might be specific to Linux.

Inodes and links

The basic unit of a Linux filesystem is an **inode**. An inode represents one filesystem object: e.g. a file, a directory, or a symbolic link. There are a few more esoteric inode types, which we shall mostly ignore (so-called *special files*: sockets, named pipes and device nodes).

The inode serves as a logical identifier for the given filesystem object. It also holds most of its metadata: size, permissions, last modification time. However, **an inode does not store its own name.**

The names are instead stored in structures belonging to the parent directory. A directory can be thought of as special kind of file whose content is a mapping from names to inodes of its direct children. The elements of this mapping are called *directory entries*.

This implies that an inode can have multiple names if multiple directory entries reference the same inode. These names are usually called *hardlinks* or simply *links* to the given inode.

However, for practical reasons, multiple hardlinks to a directory are not allowed. Thus while the filesystem structure is a DAG rather than a tree, directories form a proper tree. Also, unlike all other kinds of inodes, directories store a reference to their parent (as a special directory entry called "`..`").

This explains many otherwise perplexing (especially for newcomers to the Unix world) facts:

- **Perplexing fact #1:** The syscall used to delete a file is called `unlink`.
  Explanation: It does not in fact delete a file (inode), but merely removes one link to it. Only when all links to an inode are removed, it is deleted.

- **Perplexing fact #2:** It is possible to delete a file that is opened by a process. That process can happily continue using the file.

Explanation: Inodes in kernel are reference counted. Only when all in-kernel references to the inode are gone *and* the inode has no links, it is physically deleted.

- **Perplexing fact #3:** To rename or delete a file, you do not need write permissions (or in fact, any permissions) to that file, only to the parent directory.
  Explanation: These operations do not touch the file inode at all, they change only the parent directory contents (by adding/removing directory entries).

- **Perplexing fact #4:** Renaming a file updates the last modification time of the parent directory, not the file.
  Explanation: Same as above.

We should also clarify that the term *inode* is actually a little overloaded. It can mean at least three related but distinct things:

- A purely logical concept that helps us to talk about filesystem structure and behaviour.
- A kernel in-memory structure (`struct inode`) that identifies a filesystem object and holds its metadata. These structures are kept in memory as a part of the *inode cache* to speed up file access.
- An filesystem-specific on-disk data structure used to hold file object metadata and usually also information about the location of the file's data blocks on the disk. However, some filesystems do not internally have any concept of inodes, especially non-Unix filesystem like FAT.

Each inode (in all the three senses) has a unique (within the scope of a single filesystem volume) identifier called the **inode number** (*ino* for short) that can be read from userspace.

Filesystem access syscalls

Most filesystem syscalls take string paths as their arguments. The inode corresponding to the path is found in the kernel using a process called **path resolution**. The kernel starts at the root inode and for each component of the path walks down the corresponding directory entry. This process is inherently non-atomic and if files are renamed during path resolution, you might get unexpected results. [2]

The most important syscalls include:

- `lstat`(*path*): resolve *path* into an inode and return a structure containing its metadata. Among other things, it contains: type (file/directory/etc.), size, last modification time and inode number.
- `unlink`("*dir*`/`*name*"): resolve *dir* into an inode, which has to be an existing directory, and remove the directory entry *name* from it. *name* cannot be a directory.
- `rmdir`("*dir*`/`*name*"): like `unlink` but removes a directory, which must be empty.

- `mkdir`("*dir/name*"): create a new directory inode and link it to *dir* as *name*.
- `rename`("*orig-dir/orig-name*", "*new-dir/new-name*"): resolve *orig-dir* and *new-dir* to inodes. Then perform the following atomically: remove the *orig-name* directory entry from *orig-dir* and create a new *new-name* directory entry in *new-dir* that refers to the same inode as *orig-name* did. If there was already a *new-name* entry in *new-dir*, replace it atomically (such that there is not gap during the rename when *new-name* does not exist).
- `link`(*orig-path*, "*new-dir/new-name*"): create a new hardlink to an existing inode. Unlike `rename`, this does not allow overwriting the target name if it already exists.

When desiring to access the *content* of inodes (e.g. read/write a file or list a directory), you must first *open* the inode with an `open`(*path*, *flags*) syscall. `open` resolves *path* into an inode and creates an **open file description** (OFD, `struct file`) structure in the kernel, which holds information about the open file like the current seek position or whether it was opened read only. The OFD is tied to the *inode* so that it points always to the same inode even if the file is renamed or unlinked while it is opened.

The application gets returned a **file descriptor**, a small integer that is used to refer to the OFD in all subsequent operations on the opened file. The most common operations are `read`, `write` and `close`, with the obvious meanings, and `fstat`, which does a `lstat` on the file's inode without any path resolution.

One can also `open` a directory and obtain a file descriptor referring to it. Apart from listing directory contents, this file descriptor can be used as an anchor for path resolution. To this end, Linux offers so-called **at-syscalls** (`openat`, `renameat`, etc.), that instead of one path argument take two arguments: a directory file descriptor and a path *relative to that directory*. Such syscalls start path resolution not at the root but at the inode referenced by the file descriptor. Thus userspace applications can use directory file descriptors as "pointers to inodes". This will later prove crucial in elliminating many race conditions.

1.1.2 Change detection in a single file

Let's start off with something trivial: detecting changes in a single file. First we need to decide what to store as internal state. Against that internal state we shall be comparing the file upon the next scan. One option is to store a checksum (e.g. MD5) of the file's content. However, this makes scans rather slow, as they have to read the complete contents of each file. This is unfortunate as today's file collections often contain many large files that rarely ever change (e.g. audio and video files).

A more viable alternative takes inspiration from 'quick check' algorithm used by the famous `rsync` file transfer program. [3] It consists of storing the size and last modification time (*mtime*) of each file and comparing those. This may be unreliable for several reasons:

- It is possible to change mtime from userspace (possibly to an earlier value) and some applications do so.
- mtime might not be updated if a power failure happens during write.

- mtime updates might be delayed for writes made via a memory mapping.
- While most modern file systems store mtimes with at least microsecond granularity, some older file systems store mtimes with only second granularity. This means that if the file was updated after we scanned it but in the same second, we wouldn't notice it during next scan. We can compensate for this in several ways: for example if we get an mtime that is less than two seconds in the past, we wait for a while and retry.

Most of these problems should be fairly unlikely or infrequent and the massive success of `rsync` attests that this approach is good enough for most practical uses.

Moreover, size and mtime can be acquired atomically while computing checksums might give inconsistent results if the file is being concurrently updated. We can still store checksums for consistency checking purposes but it is sufficient to update them only when the (size, mtime) tuple changes. And we do not even have to recalculate the checksums every time a file is changed. Instead, we can simply remember that a file has pending changes and delay actual checksum calculations to make them less frequent. This is discussed in sec. 2.1.4.

1.1.3 Scanning a single directory

For a single directory, we can simply store a mapping from names to (size, mtime) tuples as the state.

To read a directory, an application calls the `getdents` syscall (usually through the `readdir` wrapper from the standard C library), passing it a directory file dectriptor and a buffer. The kernel fills the buffer with directory entries (each consisting of a name, inode number and usually the type of the inode). When the contents of the directory do not fit into the buffer, subsequent calls return additional entries.

We can hit a race condition in several places if entries in the directory are renamed during scanning:

- Between two calls to `getdents`. The directory inode is locked for the duration of the `getdents` so everything returned by one call should be consistent. However, a rename may happen between two `getdents` calls. In that case, it is not defined whether we will see the old name, the new name, both or neither. [4] The last case is particulary unpleasant because we might mistakenly mark a renamed file as deleted.

  This can be mitigated by using a buffer large enough to hold all the directory entries. This could be achieved for example by doubling the buffer size until we manage to read everything in one go. However, trying to do this for large directories could keep the inode locked for unnecesary long.

- Between `getdents` and `lstat` (or similar). Because `getdents` returns only limited information about a file, we need to call `lstat` for each entry to find size and mtime. Between those to calls, the entry might get renamed (causing `lstat` to fail) or replaced (causing it to return a different inode). Both cases can be detected (the latter by comparing inode number from `lstat` with inode number from `getdents`, which is unreliable because inode numbers can be reused).

However, instead of problematic workarounds for specific issues, there is one simple solution to *all* directory-reading race conditions. The key is that directories have mtime, just like files. The directory mtime is, as you would expect, the last time a directory entry was added to or removed from the directory. The solution is now obvious: we remember the directory's mtime at the beginning of the scan. After we have enumerated all the directory entries, we once again look at the mtime. If it is different, the directory has been concurrently updated and the scan results may be unreliable. In such case, we simply throw them away and retry after a delay. The same caveats about mtime granularity apply as were mentioned above for files.

There is one other problem: when a file is renamed, it would be detected as deletion of the original file and creation of a new on with the same content (or just similar, if it was both renamed and changed between scans). Unless the data synchonization algorithm can reuse blocks from other files for delta transfers, this would force retransmission of the whole file.

The problem gets even more serious when renaming a directory, perhaps one containing a large number of files and subdirectories. Unless we can detect that this is the same directory, we would have to recreate the whole subtree under the new name on the target side instead of just renaming the directory that is already there.

To correctly detect renames, we would need a way to detect that a name we currently encountered during the scan refers an inode that we know from earlier scans, perhaps under a different name. For this to be possible, we need to be able to assign some kind of unique identifiers to inodes that are stable, non-reusable and independent of their names.

1.1.4 Identifying inodes

Inode numbers

The first natural candidate for an inode identifier is of course the inode number. But inode numbers can be reused when an inode is deleted and a new one is later created. This happens farily often, for example this simple experiment quite reliably reproduces inode number reuse on an otherwise quiet ext4 filesystem:

```
$ echo "first file" >first
$ ls -i
12 first
$ rm first
$ echo "second file" >second
$ ls -i
12 second
```

Both files got inode number 12 despite being completely unrelated. In other filesystems (e.g. btrfs), the inode number is simply a sequentially assigned identifier and numbers are not reused until necessary (usually never, because inode numbers can be 64-bit so overflow is unlikely).

Thus at least on some filesystems, including ext4, one of the most common filesystems in the Linux world, inode numbers cannot be used to reliably match inodes between offline scans. Is there a better way?

Enter filehandles

There is an alternative way of identifying inodes, created originally for the purposes of the Network File System (NFS) protocol. NFS was designed to preserve the usual Unix filesystem semantics (e.g. that a file can be renamed while open) over the network. It was also designed to be *stateless* on the server side. This entails that a client should survive not only reconnection after a network outage but even a full reboot of the server noticing nothing but a delay.

For example, the client must be able to continue using files opened before the reboot as if nothing happened. Even if the files were renamed in-between. Extreme case: open file on client, disconnect network, reboot server, rename the file on server, reboot server, reconnect network, client can continue using the renamed file.

To accomplish this, the concept of **file handles** was created. A file handle is simply a binary string identifying an inode. But unlike inode numbers, a file handle can never be resused to refer to a different inode. When a client tries to use a handle referring to an inode that has been deleted, the server must be able to detect that and return a "stale handle" (`ESTALE`) error.

A file handle should be treated simply as an opaque identfier, its structure depends on the filesystem type used on the server side. Many file systems (including ext4) create file handles composed of the inode number and a so-called **generation number**, which is increased every time an inode number is reused. Such pair should be unique for the lifetime of the file system.

Not all filesystems support file handles. Those that do are called **exportable** (*exporting* is the traditional term for sharing a filesystem over NFS). Most common local filesystems (e.g. ext4, btrfs, even non-Unix filesystems like NTFS) are exportable. On the other hand, NFS itself, for example, is not.

File handles are usually used by the in-kernel NFS server. But they can also be accessed from userspace using two simple syscalls: `name_to_handle_at` returns the handle corresponding to a path or file descriptor. `open_by_handle_at` finds the inode corresponding to the handle, if it still exists, and returns a file descriptor referring to it. If the inode no longer exists, the `ESTALE` error is reported. These syscalls were created to facilitate implementation of userspace NFS servers. We shall (ab)use them in rather unusual ways. [5]

Being non-reusable, file handles seem like a good candidate for persistent inode identifiers. However, there is a different problem. The NFS specification does not guarantee that the same handle is returned for a given inode every time. [6, p. 21] I.e., it is possible for multiple different handles to refer to the same inode, which prevents us from simply comparing handles as strings or using them as lookup keys in internal databases. Most common file systems (including ext4 and btrfs) have stable file handles. However, just for the fun of it, we will show a solution for the general case.

The best of both worlds

We propose a reliable inode identification scheme that combines the strengths of both inode numbers (stability) and file handles (non-reusability). It works as follows: for every known inode, we store both its inode number and a file handle referring to it in our internal database, with inode number usable as a lookup

key.

Whenever we encounter an inode during a scan, we look up its inode number in our database. If a record is found, we fetch the stored handle and try to open it with `open_by_handle_at`. If that succeeds, the original inode still exists and thus its inode number has not been reused. At this point, we can be sure that the inode we encountered during scan corresponds to the record just found in our database. If we found it at a different path than last time, we can record this as a rename.

On the other hand, if we get an `ESTALE` error, we know that the original inode has been deleted and thus we can remove it from our database. We can then proceed with inserting a new record with a new handle for the inode encountered.

Storing file handles has other benefits, too. For example the stored handle allows us to open the inode corresponding to an internal record in our database at any time (e.g. when synchronizing file data) free from the race conditions of path resolution.

We have solved the inode identification problem for two broad classes of filesystems: exportable filesystems and filesystems that do not reuse inode numbers. This covers most common file systems that a Linux user encounters, with the exception of (client-side) NFS. That is rather unfortunate as it is common practice for users to have NFS-mounted home directories in schools and larger organizations. This issue should certainly be given attention in further works but it seems likely that it will require kernel changes.

Extended attributes

Another possibility is to use POSIX extended attributes (xattrs) [7] to help identity inodes. Extended attributes are arbitrary key-value pairs that can be attached to inodes (if the underlying file system supports them; most moder Linux file systems do). Because they are attached to inodes, they are preserved across renames.

This offers a simple strategy: store a unique inode identifier as an extended attribute. Whenever we encounter an inode without this attribute, we assign it a new randomly-generated identifier and store it into the xattr.

However, we consider the handle-based scheme superior for several reasons:

- Not all file systems support extended attributes (probably less than support file handles).
- The size of extended attributes is often severely limited. For example on ext4, all the extended attributes of an inode must fit into a single filesystem block (usually 4 kilobytes). While our identifier would be rather small, we cannot predict how much data other programs store into extended attributes.
- We use file handles for several other purposes, such as a race-free way of accessing inodes and to speed up directory tree scans (as described in sec. 1.1.5.
- Some programs copy all extended attributes while copying a file. This would create two inodes with the same identifier, which is asking for trouble. We could partially work around this by also storing the inode number in the xattr and trusting its value only when it matches the real inode number.

- Extended attributes cannot be attached to symlinks. This seems harmless at the first glance, we do not need rename detection for symlinks because they are cheap to delete and recreate. However, rename detection on symlinks will prove crucial in a surprising fashion when implementing a feature called *placeholder inodes* (sec. 2.1.5).

1.1.5 Scanning a Directory Tree

Internal state

When scanning directory trees, we definitely do not want to store the full path to each object. If we did and a large directory was renamed, we would need to individually update the path of every file in its subtree... and probably transfer all those updates during synchronization, unless additional tricks were involved.

Instead, we will choose a tree-like representation that closely mimics the underlying filesystem structure. The internal state preserved between scans consists of:

- A list of inodes, each storing:
  - A so-called **IID**, a random unique identifier assigned upon first seeing this inode.
  - Inode number and filehandle as dicussed above, with fast lookups by inode number possible.
  - Last modification time, for files also size.
- For every directory inode, a list of its children as a mapping from names (without path) to IIDs.

This way, when a large directory is renamed, it suffices to remove one directory entry from the original parent and add one directory entry to the new parent, requiring a constant number of updates to the underlying store.

Speed

Scanning large directory trees is slow, especially on rotational drives like hard disks. The main contributor to this is seek times. We are accessing inodes, each several hundred bytes in size, in essentialy random order. That is actually not true as file systems contain many optimizations that do a good job at clustering related inodes together but these are far from perfect and seek times are still a major concern.

This problem is aggravated by the structure of the ext4 file system. In ext4, the disk is split into equally-sized regions called **block groups**. Each block group contains both inode metadata and data blocks for a set of files. [8]

Fig. 1.1 shows the on-disk block group layout. The dark bands represent areas storing inodes, the white are data blocks. Also note that this picture is quite out of scale. The default block group size is 2 GB,[1] so on a 1 TB partition there will

[1] The default was 128 MB for ext2/3. Acutally, ext4 block groups are still 128 MB by default but they are grouped into larger units called **flex groups** (16 block groups per flex group by default), with inode metadata for the whole flex group stored at its beginning.

Figure 1.1: ext4 block group layout (not to scale)

be approximately 500 block groups. This makes inodes literally scattered all over the disk.

This layout improves performance for most of the normal filesystem access patterns (by improving locality between metadata and data blocks). However, scanning the whole file system is not one of them.

Not all filesystems are like this. For example, NTFS keeps all file metadata in one contiguous region called the Master File Table (MFT) at the beginning of the partition. This allows the existence of tools like SwiftSearch[2] that read and parse the whole raw MFT in several seconds (bypassing the operating system) and then allow instantaneous searches for any file by name, no previous indexing required.

Nothing like this can be done for ext4. Just reading all the raw inode regions will include a lot of seeks and takes tens of seconds to minutes.

In ext4 and many other filesystems, the inode number directly corresponds to the location of the inode structure on disk. Because of the block group structure, the mapping is not linear but it is monotonic. Therefore, if we access inodes in inode number order, the access will be sequential inside each block group (with perhaps only a few gaps for recently deleted inodes).

We can for example do the scan using a breadth-first search with a priority queue ordered by inode number. We know the inode number from `getdents` without `stat`-ing the inode itself

Faster rescans For the second and further scans, we can do even better. Linux stores a modification time for directories as well as for files. The modification time of a directory is the last time a direct child is added to it or removed from it. Thus we can simply iterate over the all inode records in our database, files and directories alike, in inode number order. We open each of them using the saved file handle, which encodes the inode number and thus the location of the inode on disk. We can then `fstat` the opened directory, which directly accesses this location, without any path resolution steps that would require the kernel to look up directory entries in parent directories.

This way, we access only inode metadata blocks (the gray areas in fig. 1.1) and not directory content blocks, which are stored in the white data sections. This further reduces seeking.

[2] `https://sourceforge.net/projects/swiftsearch/`

Table 1.1: Scan times (mm:ss) and throughputs (inodes/min) for different access strategies.

Order	Access by	All inodes		Files only		Dirs only	
		time	inodes/min	time	inodes/min	time	inodes/min
inode	handle	1:40	1.4 M	0:34	3.5 M	1:37	125.9 k
	path	1:56	1.2 M	2:08	943.5 k	1:22	149.6 k
scan	handle	4:41	490 k	0:52	2.3 M	4:33	44.8 k
	path	4:23	530 k	4:36	438.6 k	1:56	105.1 k
find	path	4:41	490 k	4:27	46 k		
random	handle	> 1 h		> 1h			
	path	> 1 h		> 1h			

Tbl. 1.1 shows times necessary to `lstat` all the inodes on a filesystem for different access orders and access methods on a real-world ext4 filesystem with approximately 2 million inodes (10 % of which were directories).

The experiment has been performed as follows: first, we performed a normal depth-first scan of the directory tree to obtain a flat list of all the inodes in the file system containing inode number, file handle and full path of each inode (this is similar to what Filoco metadata would look like, only we use paths instead of parent/child relationships).

Then we the complete inode list was loaded into memory and sorted in one of the following ways:

- *inode* is ascending inode number order
- *scan* is the original order in which we encountered the inodes during the recursive scan (that is, DFS order where children were visited in the order returned by `getdents`)
- *random* is a completely random shuffle of the inode list

Then we clear the filesystem cache (using the `sysctl -w vm.drop_caches=3` command) to prevent it from unpredictably distorting the results. Only after that, we start measuring time. Then we try to `stat` all the inodes in the given order in one of two ways:

- *handle* means an `open_by_handle_at` syscall on the saved handle followed by an `fstat`.
- *path* means simply `lstat`-ing the saved path, which triggers the path resolution process in the kernel.

For comparison, there is a row labelled *find*, which shows how long usual DFS traversal of the directory tree (i.e. intermixed `getdents` and `lstat` calls) would take (as performed by the `find -size +1` command; the `-size` argument is needed to force `stat`-ing every inode).

The results confirm our predictions:

- Accessing inodes in inode number order is faster (about two times) than accessing them in DFS order.

- Accessing inodes using handles may be faster than using paths, as shown by the "files only" case.

However, it showed a few rather surprising results:

- Scanning only directories takes almost as long as scanning all inodes even though there is ten times less of them. Contrarily, scanning only files using handles only is several times faster than scanning only directories, despite there being 9 times as many files as directories. The times remain more or less the same when we leave out the `fstat` and only open the handles. From this we can conjecture that the `open_by_handle` operation is for some reason significantly slower on diretories than files. Perhaps the kernel performs some additional checks? This would definitely benefit from further investigation.

All of this applies to an ext4 or similar file system on a rotational hard drive. For btrfs and SSD, the differences will probably be negligible if any. The results were produced using scripts in the `experiments/treescan2` directory in attachment 1.

We experimented with several other techniques, for example massively parallelizing the scan in the hope that the kernel and/or hard disk controller will order the requests themselves in an optimal fashion. However, most of these attempts yielded results worse than a naive scan so they would not be discussed further.

Race Conditions

Tree scanning presents numerous opportunities for race conditions. Some were already discussed in sec. 1.1.3. But the most serious threat is a file being moved from one directory to another during the scan. To be more precise, from a directory that we have not yet scanned to one that we have. We would completely miss such a file from the scan and might mistakenly consider it deleted.

As the whole scan may take several minutes, it is quite easy for this to happen.

It cannot be detected or mitigated in any easy way with offline techniques alone. However, we can use an online detection mechanism during the scan. Then, if any changes to the filesystem happen while scanning, the kernel will tell us about them and we can for example rescan the few affected directories.

Even if we are not interested in long-term realtime change monitoring, it pays to set up online change detection even if only for the duration of the scan. It is the only way we know of of mitigating such race conditions without support of the filesystem (e.g. in the form of atomic snapshots).

1.2 Online Change Detection

1.2.1 Inotify

Inotify[9] is the most widely used Linux filesystem monitoring API. It is currently used by virtually all applications that wish to detect filesystem changes: synchronization tools, indexers and similar.

When using inotify, a process must first create a **watch list** – a list of inodes that it wants to monitor. Inodes are added to the watch list using paths (file descriptors may be added using the `/proc/self/fd` trick) but once added, the kernel keeps a direct reference to the inode.

Inotify supports reporting all the usual filesystem events (writes, creations, renames, unlink) and several less usual ones (opens, closes and reads). Events are generated when anything happens to any inode on the watch list or a direct child of a directory on the watch list. This holds true even for events that do not touch the directory inode, like writes to a file inside a watched directory.

However, the watching is not recursive. Thus if we want to watch a whole directory tree, we need add all the directories in the tree to the watch list one by one. Experiments in the previous section have not shown much efficient ways of doing that because opening directory inodes, which we now need, is for some reason much slower than for files.

Inotify assigns a unique cookie called the **watch descriptor** to every inode on the watch list. This watch descriptor is then returned with events concerning this inode. In case of directory-changing events (creations, renames and unlinks), a basename of the affected file is returned alongside the watch descriptor of the directory. We can simply keep a mapping from watch descriptors to IIDs or some other kind of internal identifiers. This also gives us access to the file handle if a race-free access to the affected inode is necessary.

Another consideration is that the inotify watch list and the watched inodes (which cannot be dropped from the inode cache because they are referenced by the watch list) consume non-swappable kernel memory. This would not be a problem for most users as the amount is approximately 0.5 kB per directory. For our example file system with 200 000 directories, this would constitute 100 MB of wasted memory.

Inotify can efficiently be used as an anti-race-condition aid during offline scans (as discussed in sec. 1.1.5). Because we have to visit all the directories during the scan anyway, we can add inotify watches to them at little extra cost (except for memory usage). To prevent all kinds of races, we have to first add the inotify watch for a directory and only then read its contents.

1.2.2 Fanotify

Fanotify [10] is the newest change notification API added to the Linux kernel. Like inotify, it supports watching individual inodes but unlike inotify, it also supports watching whole mount points. Note that this is not the same as watching one filesystem volume because (1) one volume may be accessible via several mountpoints, (2) a mount point may show only a part of a volume's directory tree. For example, after invoking the commands:

```
mount -t ext4 /dev/sdb1 /mnt/hdd
mount --bind /mnt/hdd/music/bob /home/bob/music
```

there are two mount points:

- `/mnt/hdd`, which shows the whole directory tree of the file system on the `/dev/sdb1` device

- `/home/bob/music`, which shows the `/music/bob` subtree of the file system on `/dev/sdb1`

When you create a fanotify watch for the `/home/bob/music` mount point, you get events for filesystem changes made via this mount point. For example, if some program writes to `/mnt/hdd/music/bob/test.mp3`, you will not get an event, even though the file `/home/bob/music/test.mp3` now has different content than before.

This could be (ab)used to make fanotify watch only a given directory subtree. For example when you issue the command `mount --bind /home/alice /home/alice`, the directory `/home/alice` becomes a separate mount point (although it contains the same files as before), which can be separately watched by fanotify. However, this has a drawback: it is not allowed to move files between mount points, even if the two mount points refer to the same filesystem volume.

Another interesting property of fanotify is that you get a file descriptor to the affected inode along with any event. From it we can determine inode number and file handle and look up the corresponding object in our internal database.

However, fanotify has two important limitations:

- Its use requires root permissions (because otherwise there is no easy way for the kernel to determine which events a user should be allowed to see).
- More importatnly, it does not report directory-changing events (creates, renames, and unlinks).

1.2.3 The `FAN_MODIFY_DIR` kernel patch

We have implemented an extension to fanotify that enables it to report directory change events. This extension is available as a series of two kernel patches (currently against Linux 4.10, but they should apply to any 4.x version with trivial modifications) in the `src/fanotify/` directory in attachment 1.

Such an extension is useful not only in context of file synchronization but also for example to filesystem indexers.

There are two main reasons why fanotify currently does not support directory events. Let's look at how we deal with each of them.

Directory event semantics

The first problem is that it is not clear how to represent directory events and what semantics should they have in order to be useful.

Inotify reports them in a rather complicated way that involves passing watch descriptors of parent directories and string names of their children. Because of race condition, by the time you receive the event, these names may already refer to a completely different inode than the one the event was about. There are also issues with regard to what is or is not guaranteed about ordering of these events, especially in cases such as concurrent cross-directory renames. In general, inotify directory events are hard to interpret correctly.

In contrast, the fanotify event interface is beautifully simple. You get a fixed size structure with one event type, one file descriptor, no need to allocate space for any strings and no need to worry about what they mean.

Our solution to this conundrum lies in the filesystem-watching wisdom that we have already encountered several times: names are useless (and paths are even more useless), inodes and file descriptors are great. So instead of passing any names to userspace, we generate a simple event called `FAN_MODIFY_DIR` every time the contents of a directory are changed in any way (a directory entry is added or removed), i.e. exactly when the directory's mtime would be updated.

As with all fanotify events, you get a file descriptor – one referring to the modified directory, i.e. the parent of the created, renamed or unlinked file. This makes directory modification events completely analogous to file modification events. In case of a cross-directory rename, you get two `FAN_MODIFY_DIR` events for both the old and new parent directory.

This scheme is based on a suggestion made on the Linux kernel mailing list back in 2009. [11] Since then, nobody has attempted to implement it.

This scheme has one more advantage (pointed out to me by kernel developer Jan Kára in personal communication): when there are more events of the same type queued for an inode before they are read by userspace, kernel automatically merges them. So for example when moving 100 files from one directory to another, you would get only a few events instead of 200.

We have expanded this idea into a more general trick. You can actually purposefully stall reading fanotify events, to (1) give kernel more chance for merging repeated events, (2) read events in larger chunks to reduce number of context switches, even when there is a little delay between them. The realization is rather simple, as shown in algorithm 1.1.

Algorithm 1.1 fanotify event grouping

```
1: repeat
2:     wait 5 seconds
3:     wait for a fanotify event to be available
4:     read all pending events into one large buffer and process them
```

This has to be done carefully because waiting too long might cause the kernel queue to overflow and events to be dropped.

Which mount point?

The second problem lies in the fact that fanotify watches are tied to a specific mount point. Thus to generate a fanotify event in reaction to a filesystem operation, we need to know through which mount point the operation was performed. Two important in-kernel structures are relevant to understanding this:

A `struct dentry` represents one directory entry. It contains the following information:

- A reference the inode to which the directory entry refers (the child)
- The name of the entry
- A reference to the parent dentry. It really is the parent dentry, not the parent inode; this allows reconstructing full paths by walking the dentry parent chain. However, there can also be so-called *disconnected dentries* that do not know their parent or name, so they should be rather considered to represent an inode than a directory entry. These can be created for

example when opening file handles, because in that case the inode is directly accessed without path resolution so its parents cannot be known.

A `struct path`, despite its name, does not represent a string path but rather the result of path resolution. It contains references to:

- The dentry represented by this path.
- The mount point to which the original path belongs.

The kernel's open file description structure stores a `struct path` representing the path using which the file was opened. This allows, among other things, (1) showing full paths of files open by a process by tools such as `lsof` or `ls -l /proc/<pid>/fd`, (2) generating correct fanotify events because the mount point is known.

However, most kernel-internal filesystem APIs, including the ones dealing with directory changes, operate on inodes and dentries and do not get the mount information contained in a `struct path`.

Here is how an `unlink` syscall is currently processed in the Linux kernel:

1. The syscall implementation (`SYSCALL_DEFINE1(unlink)` in `fs/namei.c`) gets string path from userspace.
2. It calls helper function to resolve the path into a `struct path`.
3. It passes the dentry from the `struct path` to a kernel-internal function `vfs_unlink`.
4. `vfs_unlink` carries out the operation and generates an inotify event for the parent inode. It does not generate a fanotify event because it does not know the mount point.

The `vfs_unlink` function (and other similar functions like `vfs_rename`) is a stable kernel API that is used on many places in the kernel so it is not easy to change its signature.

Instead, we opted to generate the fanotify event directly in the syscall code, and many other places that call the `vfs_*` functions, for example the in-kernel NFS server. Scattering fanotify calls at several places across the kernel is probably not a good long-term solution but practically it works.

Our patch has been submitted to the Linux kernel mailing list as a RFC but it sparked little interest at the time. [12]

1.2.4 Amir Goldstein's fanotify patches

Another solution to the fanotify directory events problem has appeared recently, in parallel with ours. [13]

Amir Goldstein's patches are a much more comprehensive (and complex) solution to the directory event reporting problem. They offer the following features:

- Report separate fanotify event types for the individual kind of directory entry manipulations (create, rename, unlink).
- Optionally report names of the affected directory entries in addition to the parent directory file descriptor. When this flag is disabled, the result is rather similar to our patch.

- Allow attaching fanotify watches to filesystem volumes as a whole as opposed to specific mount points.
- Allow reporting events about an arbitrary directory subtree of a file system, although this is subject to reliability issues. Specifically, as it filters events by walking dentry parents, it does not report events for disconnected dentries (because the kernel simply does not know whether they belong to a given subtree).

The last point seems particularly interesting because this might in the future allow file systems to generate fanotify events from within, which are not related to any specific mount point. This could be used for example by distributed or network file systems to report server-side changes.

2. Metadata Synchronization

2.1 Metadata Model

As stated in the introduction, in Filoco, every store keeps a complete copy of the metadata about all files in the realm but only stores actual data of a subset of the files.

This concept was used for example by git-annex [14], where, as the name suggests, metadata is stored in a git repository (with actual file contents stored externally in a distributed fashion).

The user can configure which files should be replicated to which stores – either on a per-file basis or using filters depending on file name, path, type or size. This allows them to choose a compromise between storage requirements, redundancy, and availability.

For example you can configure some small, important or often-used files (emails, writings, notes, own source code) to be always replicated everywhere, while bigger and less important files (movies you will probably never watch again) will have only one copy distributed among several slow external hard drives in your closet.

Currently, you have to manually configure which old movies should be stored on which slow external drives. In the future, there should be the option of automatically distributing a given set of files over a given set of stores. So you could classify five stores as *movie disks* a thousand files as *movies* and Filoco would automatically spread the files across the drives. There could be even more advanced option, for example configure some files to be stored at two out of four backup drives and one out of two server stores.

On the other hand, global metadata allows you to always keep track of all your files, no matter where they are stored, even if it is an external hard drive in your safe deposit box. The synchronized metadata contain a complete directory tree (i.e. file/directory names and parent-child relationships) of all the files in the realm, which is shared among all the stores.

This means that is is not possible to have a file stored under a different name in one store than in another. If a file or directory is moved or renamed on one store, this change is replicated to all stores, even those not hosting the file's content. The rename can even be initiated from such a store.

You can completely reorganize your directory hierarchy from the comfort of your laptop, even though some of the files are physically located only on offline external drives. The next time you connect such drive and perform synchronization, these renames will be applied there.

Apart from the directory tree and some basic metadata like file sizes, the centralized metadata contains two important pieces of information:

- Data location information, that is, a list of stores that have the file's content (and in what version, as described below). This allows you to ask Filoco for the content of a file and if it is not locally available, it will either fetch it from a reachable store that has it or at least inform you on which stores host the file. You can then take the right external disk out of your closet or turn on your secondary laptop as neccesary.

- Data checksum. This allows to detect media failure or tampering and when detected, use another replica if available.

2.1.1 Detailed metadata structure

Now we shall examine the structure of the globally replicated metadata in more detail. Metadata is modelled as a set of immutable *objects* of several different types (described below). Each object has a unique 128-bit identifier, generated etither pseudorandomly or using a cryptographic hash function from the object's content. The result of complete synchronization between two stores is always the set union of their objects, although partial synchronization is also possible (e.g. restricted to a directory subtree).

How can we represent changing entities (e.g. files with changing content) using immutable objects? In exactly the same ways as git commits are organized [15]. We create a new object for each version of the file, which contains references to the parent version(s).

As synchronization never deletes objects, we are currently forced to keep indefinite metadata revision history (just as is the case for git). A cleanup mechanism might be introduced in the future.

The following types of objects currently exist:

- A **filesystem object (FOB)** is the basic unit Filoco works with. It represents a single file or directory (other inode types, including symlinks, are currently not supported, though support should be trivial to add). It serves primarily as an identifier for the filesystem object that is stable across renames. It also carries immutable metadata like inode type (file or directory).

  A filesystem object has three important conceptually mutable properties: (1) content hash (files only), (2) location in the directory tree, (3) storage information (a list of stores that host the file's content). As suggested above, the values of these properties are not stored inside the FOB object but instead as separate version objects (FCVs, FLVs and SRs) described below.

- A **file content version (FCV)** contains information about one version of a file's content. It stores the ID of the relevant FOB and, similartly to a git commit, the content hash and a list of parent versions (FCVs) of the given file. The parent list is used to establish an ordering on the versions. This is necessary because the FCVs are stored in an otherwise unordered object set. It also helps conflict handling. See sec. 2.1.2 for a precise explanation of parent version semantics.

- A **FOB location version (FLV)** describes the location of a filesystem object as a versioned property. Location is represented by the tuple (*parent*, *name*), where *parent* is the ID of the parent FOB. This format was chosen for three reasons: (1) It allows us to efficiently rename or move directories that contain a large number of files and subdirectories (which would be impossible if we stored full path for each file). Each such move costs only one new FLV for the directory being moved. (2) While maintaining a list of child FOBs for each directory would also allow for efficient renames and

would be closer to Unix tradition, a parent pointer is a scalar value whose versioning is conceptually much easier than trying to define semantics for versioning child lists. (3) It corresponds to my personal intuition that name and parent directory are logically properties of the file (for name it should be quite clear, directory could be considered a kind of category tag attached to a file). Similarly to FCVs, a FLV carries a list of parent FLVs.

- **Storage records (SR)** describe storage events. A storage event consists of a store beginning or ceasing to host a given FCV. The fields of a SR are (1) store identifier, (2) FCV ID, (3) event type (start or end of object hosting) and (4) a list of parent SRs, just as with other versioned objects. To determine whether a store has the contents of a FCV available, one has to look at the event type of the last (by parent-child ordering) SR for the given FCV and store ID (while remembering that this information is not necessarily up to date, so we have to be wary about deleting a file independently on two stores because each of them thinks the other has a copy).

There are also a few attributes common to all the object types:

- An identifier of the store which created the object.
- A creation timestamp.

Please note that the versioning of FOB properties is there only to facilitate synchronization, conflict resolution (see below), and auditing. We do not try to systematically keep the content of old file versions. Except for when conflicts occur, each store only keeps the content of the newest version of any file. Because of synchronization delays, old versions can be present in the realm for quite some time but this is a byproduct and users should definitely not rely on that. However, the architecture is intentionally designed such that (optional) versioning can be implemented in the future.

2.1.2 Versioning and conflict resolution

Wherever there is bidirectional synchronization, there looms the threat of conflicts. Imagine that two stores A and B have the same version v of a file. Then the user makes changes to the file in store A (perhaps on a laptop), creating a new version w_A. Later they modify the file in store B, which still has the old version v (perhaps it is on their work computer, because they forgot the laptop at home). They make some other, independent changes, creating a new version w_B.

When they synchronize A with B later, both stores will have both versions w_A and w_B in their metadata database. But which of these versions should be considered "current", which version of the file should be *checked out* (i.e., written to the file system)? Clearly, it is incorrect to replace w_A with w_B on A (even though w_B has a newer timestamp), because the changes made from v to w_A would be lost. It is also incorrect to just keep w_A and ignore w_B, for the same reason.

This situation is called a *(version) conflict* and is familiar to most readers from revision control systems like git. While in some simple scenarios, conflicts

can be resolved automatically using techniques such as three-way merge[1] or git's recursive merge, they often require user intervention.

In Filoco, we decided to leave all conflict resolution up to the user, for three reasons:

- Conflicts should be much less common than in revision control systems. Most RCS conflicts are caused by multiple people working on one project simultaneously. Because our primary focus is managing personal data, we usually expect only one person making changes to files in a Filoco realm. But conflicts certainly can happen, e.g. because of delayed synchronization and offline stores, as suggested by the scenario above.

- We are not limited to source code or plain text and have to handle all kinds of files including binary (LibreOffice documents, images, archives, databases...). There is no universal conflict resolution strategy for such a wide variety of file types.

- As we do not systematically keep the content of old file versions, the common parent of the two conflicting versions is not guaranteed to be available at the time of resolution, which precludes using most classical conflict-resolution strategies based on three-way merge and it variants.

Thus when a conflict occurs, we simply present the user with both the conflicting versions and they have to somehow merge their content, either manually or by using some specialized tools.

The following additional requirements have been set for conflict handling in Filoco:

1. Conflicts must be automatically and realiably detected, so that we can apply all non-conflicting changes without user intervention on the one hand and inform the user of any conflicts on the other.

2. The user should not be forced to resolve conflicts immediately (e.g. as a part of the synchronization process). When a conflict occurs, the synchronization should finish completely, synchronizing all the other changes, conflicting or not. The user should be able to resolve any conflict locally at a later time (for example when the user wants to access the affected file).

3. Conflicts should not impede further synchronization. For example, if store A has conflicting versions v_1, v_2 of a file and later synchronizes with a store C that has neither, it should transfer both versions there. The user can then resolve the conflict in any of the stores.

4. Once a conflict has been resolved in one store, the resolution should spread to all other stores. This makes the previous requirement much more useful. Of course, if there were independent changes that were not part of the resolution, this can create more conflicts.

[1] This technique is now virtually ubiquitous. It originated in the GNU `diff3` program developed by Randy Smith in 1988[16].

5. It should be possible to rename or move a file in one store and edit it in another without this being considered a conflict.

We shall present a simple solution that fulfills ale these requirements. It is in large part based on how branching and merging works in git. First, we shall look into content versioning and then briefly mention location versioning and storage record relationships.

Each FCV has a list of parent FCVs. Usually (except for when resolving conflicts), this list contains just a single item: the logically preceding version. When you have a version v of a file on your file system and modify it, a new version w is created with a single parent v. The parent-child relationship signifies that w is based on v, that it incorporates all the content from v that the user did not purposefully remove, that it supersedes v. Whenever a store has version v checked out (meaning that the contents of the corresponding file in the user's local file system corresponds to version v) and acquires version w through synchronization, Filoco automatically replaces the checked out version with w.

The parent-child relation (or more precisely, its transitive closure) describes a partial ordering on the versions. As long as you keep your replicas up to date and always edit only the chronologically newest version of the file, the ordering is linear (the version graph is a path) and there is a unique maximum ("newest verson").

However, when you make changes to an older version of the file (in a store that is not up to date) and later synchronize them, the history branches, as shown in fig. 2.1). Now the version ordering has multiple maximal elements (we call these *heads*). This we shall consider the definition of a conflict state, which will be announced to the user.

After the user resolves the conflict, a new version (marked z in the figure) is created with all the previously conflicting versions as parents. Now there is again a unique head and thus no conflict. After another resynchronization, the resolution is spread to B, which automatically checks out z instead of either w_1 or w_2.

As noted above, this is very similar to git branching and merging, with several differences:

- Versioning is done per file instead of the whole repository. This allows resolving conflicts individually and leaving some unresolved for a later time.
- Branching is implicit. It works as if whenever you were trying to do a non-fast-forward push in git, instead of the remote rejecting it, a new unnamed branch would be automatically created. This allows synchronization in the presence of conflicts and delayed conflict resolution.

File locations are versioned independently from content, so that one can edit the file in one store and rename it in other without this constituting a conflict (this fullfills requirement #5).

Two kinds of conflicts can arise when dealing with FLVs:

- An **identity crisis** conflict happens when the FLV graph for a given FOB has multiple heads (i.e., we try to assign multiple different locations to a file). This is similar to a FCV conflict but less severe because it cannot lead

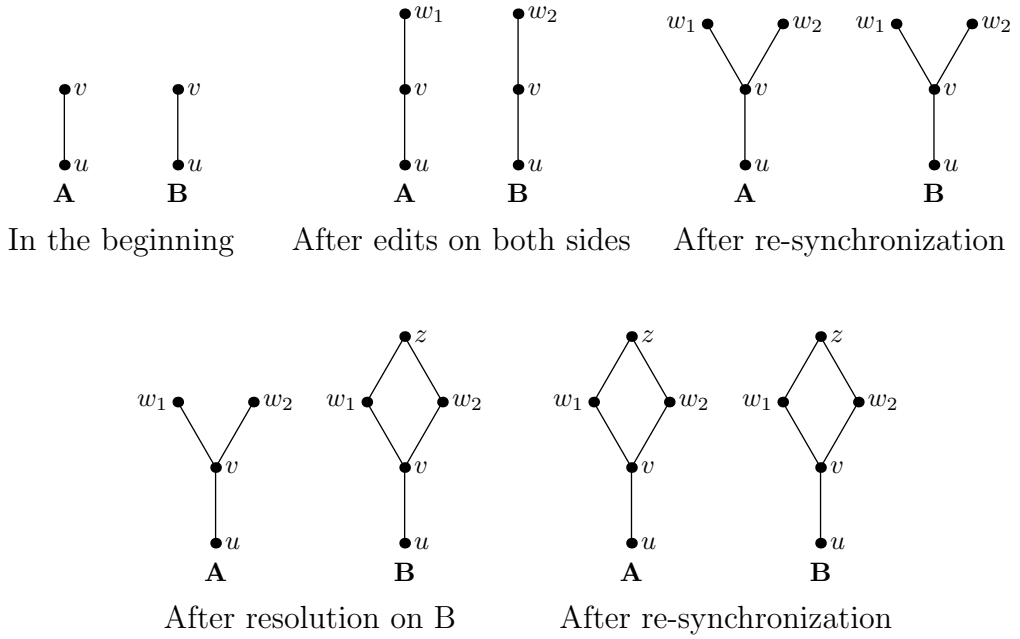


Figure 2.1: History branching during a conflict

to data loss. Currently we just give precedence to the FLV with the newest timestamp and output a warning.

- A **pigeonhole conflict** happens when head FLVs for different FOBs try to claim the same location. This is currently resolved by appending a unique suffix to each of the file names.

Storage records use the same parent-version mechanism but with differrent semantics. Whenever a new SR is created, its parents are all the current SR heads of a given FOB (regardless of from which store they are). This gives a partial ordering on the SRs. This is purely for informative purposes. SR heads have no special meaning, multiple SR heads are not considered a conflict or in any way an unusual state.

2.1.3 Alternative versioning: vector clocks

As an alternative to explicit git-like parent version pointers, we could use *vector clocks* for partially ordering versions. This is a now almost universally known mechanism for versioning in distributed systems, discovered independently by two teams in 1988 [17][18].

Their main advantage is that we do not need to maintain information about previous versions. Instead, it suffices to remember a vector of s integers (where s is the number of stores in the realm) for each head version. The partial ordering between any two versions can be determined by just looking at their vectors, without any additional information. As we expect s to be small and infrequently changing, this seems to be fairly efficient.

The main reason to use for an explicit version graph is to keep a permanent record of changes made to a file for auditing purposes. This is useful when dealing with potentially compromised stores. When a file contains unexpected data, you can look up which stores modified it and when. The version graph can be made

into a Merkle DAG (which works exactly the same as a Merkle tree [19], only it is a generic DAG instead of a tree) to prevent anyone from rewriting history. This is exactly the same thing that git does with commits. [15]

2.1.4 Working versions

Creating new FCVs is expensive. Not only additional versions increase metadata storage requirements but we also have to compute a hash of the file's content, which is slow and creates unnecessary I/O and CPU load on the system. If some process writes a few kB into a 4 GB file every second (think disk images and large databases), we definitely do not want to read the whole file and compute a hash every time. Not to mention that computing a hash of a file that can change at any moment is riddled with race conditions, which have to be handled, increasing the price even more.

To overcome this, whenever a local change to a file is detected, a so-called *working FCVs* is created. This is a special FCV with the content hash field left empty. This version normally participates in metadata synchronization, to let the other stores know you have a new version of the file.

Whenever you want to synchronize the contents of the file with another store (see chapter 3 for details on that), a full FCV is created with the working FCV as a parent. This is rather cheap because during data synchronization, we have to read the whole file and deal with race conditions anyway.

Storage records are never created for working FCVs. The only store that can ever have the data for a working FCV is the one that created it. Whenever the data is transferred to another store, a full FCV is created to represent the transferred version of the file.

A working FCV never has another working FCV as a parent. When the local head already is a working FCV and the file is further modified, no new FCVs are created, the current working FCV is simply re-used to represent the newer modified version. A consequence of this is that one cannot reliably determine the file's last modification time from a working FCV timestamp because the FCV is created upon the first of a series of local modification. While a last modified timestamp would be a nice information to have in the metadata, we consider this a small price to pay for less version bloat.

2.1.5 Placeholder inodes

One of the major goals of Filoco is to present the user with an unified view of their data, no matter where they are physically stored. This means first and foremost a unified directory tree. This begs the question of how to represent files for which we have no data in the local file system.

We could omit them completely and offer some specialized tools (called perhaps `filoco-ls`, `filoco-tree`, etc.) to list the locally missing files. However, this seems rather inconvenient. We opted for a different method, and that is to represent them with a special kind of inode. The best choice seems to be a broken symlink, i.e., one with a nonexistent target (`/!/filoco-missing` in our case).

This has several advantages:

- The user can see the missing files with all the filesystem access tools they are used to, from CLI tools to graphical file managers, search tools, shell scripts, etc. All of them will give the same consistent view of the global directory tree.
- The user can manupulate (especially move, rename and delete) locally missing files using any tools of their chosing: command-line `mv`, file managers, mass rename tools, shell scripts or custom programs in any language.
- Many programs visualize broken symlinks in a way that symbolizes the concept of "missing". `ls` shows it in red, some GUI programs will show a cross mark or warning icon, etc.
- When a program shows symlink targets (as `ls -l` does, or some file managers in the status bar), the user sees the informative string "filoco-missing".
- When trying to access the file programmatically, one gets the correct error code, namely `ENOENT` ("No such file or directory"), the same error as returned for nonexistent file names.
- The chosen target `/!/filoco-missing` offers one more advantage: the `/!` directory is unlikely to exist on anyone's file system. Thus when one tries to open the symlink for writing (e.g. using `echo x >some-missing-file`), they also get an error because the target cannot be created. If we used a relative target such as `filoco-missing`, a file named `filoco-missing` would be silently created upon the write attempt, turning the symlink into a non-broken one.

2.2 The Set Reconciliation Problem

Our metadata is modelled as a set of immutable objects identified by unique IDs. In order for two repositories A and B to synchronize their metadata, A should send to B exactly the objects A has but B does not (an vice versa, if we want bidirectional synchronization). If O_A is the set of object IDs possesed by A (analogously O_B for B), A should transmit the set difference $O_A \setminus O_B$.

The only problem is, A does not know O_B. In a cenralized client-server setup, the client can keep track of which objects it has already sent to a server and send only new ones during synchronization. In this case, the client is essentially keeping track of the intersection $O_C \cap O_S$ without knowing the whole O_S, which is enough for computing the set difference. The server can do the same for every client.

This is indeed what most centralized synchronization tools do. However, such approach does not translate to a distributed setting. For example, assume node A has a lot of new objects compared to node C and keeps track of this. Now we synchonize node A with another node B and then B with C. Now C has all the extra objects from A but A is not (and cannot) be aware of that. If we synchronize A with C at this point, it will send all those objects all over again. We will call this the *indirect synchronization problem.*

Instead, we will use a stateless approach. We want a protocol that allows two nodes to efficiently compute the intersection $O_A \cap O_B$ without any prior mutual information (A knows only O_A and B knows only O_B at the start of the exchange).

This is a known problem called the *Set Reconciliation Problem* [20]. It could be formally stated as follows. Let $U = \{0,1\}^\ell$ be a universe of ℓ-bit strings for

some fixed ℓ. Alice has an arbitrary set $A \subseteq U$. Bob likewise has a set $B \subseteq U$. At the beginning, they know nothing about each other's sets. We want to find a protocol that allows Alice to compute the set difference $A \setminus B$ and Bob to compute $B \setminus A$.

For our use case, we shall assume that both sets A and B similar in size: $n := |A| \approx |B|$, and significantly larger than their respective differences, which we shall for the sake of simplicity also consider similar in size: $n \gg c := |A \setminus B| \approx |B \setminus A|$. The asymmetric case is not much more interesting. The number c represents the number of "changes" (represented by new objects being created) made on one node that need to be synchronized to the other.

There are several ways of measuring the efficiency of different protocols, all expressed as a function of n, c, ℓ, and any parameters of the protocol.

- Communication complexity, i.e., the total number of bits transferred in both directions.
- Number of rounds of communication. This is important because it determines the number of network round trips required. And especially in mobile networks, latency is often a greater concern than bandwidth – the RTT on a 3G connection with suboptimal reception can be 500 ms or more.
- Computational time on each side. Without any precomputation, this would have to be at least $\Omega(n)$ because of the need to at least read the input sets. As n is presumed to be large compared to c and the sets will probably be stored on disk, we would prefer to have a data structure that can efficiently answer queries about the set needed by the reconciliation protocol – ideally in a time dependent only on c and not n (or maybe on something like $\log n$ at worst).

We will be primarily interested in the expected (as opposed to worst-case) values of these complexities. This is because the elements in our sets are random (either pseudorandomly generated or cryptographic hashes) and we only communicate with authorized peers so we do not have to worry about adversarial inputs.

2.2.1 Divide and conquer

A rather obvious solution to the set reconciliation problem and one of the first described [20, alg. 3.1] is a simple divide-and-conquer approach. First, let's assume that the elements in the sets to be from a uniform probability distribution. If they are not, we first process them by a hash function and apply the rest of the protocol on the result.

First, we need a way to compare two sets X and Y possessed by Alice and Bob, respectivelly. This is simple: Alice computes a value $\textsc{Digest}(X)$ representing the set. This value should be the same for equal sets and with high probability different for inequal sets. A simple implementation of $\textsc{Digest}$ would be to compute a cryptographic hash of the concatenation of all the elements of X. Now simply Alice sends $\textsc{Digest}(X)$ to Bob and Bob sends $\textsc{Digest}(Y)$ to Alice. If they get a value equal to what they sent, the sets are the same.

From this, a divide-and-conquer reconciliation algorithm is glaringly obvious (alg. 2.1).

Algorithm 2.1 Basic divide-and-conquer algorithm for set reconciliation

```
1: procedure RECON1(A, i = 0)
2:     D_A ← DIGEST(A)
3:     SEND(D_A)
4:     D_B ← RECV()                                  ▷ The other side's digest
5:     if D_A = D_B then
6:         return ∅
7:     else if A = ∅ then
8:         return ∅
9:     else if D_B = DIGEST(∅) then
10:        return A          ▷ Other side's set is empty, need to send everything
11:    else if i = ℓ then
12:        return A
13:    else
14:        A_0 ← {x ∈ A | x_i = 0}          ▷ All the elements with i-th bit zero
15:        A_1 ← {x ∈ A | x_i = 1}
16:        return RECON1(A_0, i − 1) ∪ RECON1(A_1, i − 1)
   =0
```

This can be easily visualized if we look at the strings of each side as an (uncompressed) binary trie. If v_s is a vertex of the trie representing the prefix s, let A_s and U_s denote the subsets of A, resp. U restricted to elements with this prefix.

Recursion then simply walks this trie. Both parties start in the root v_ε. If $A_\varepsilon = B_\varepsilon$, the sets are the same and algorithm ends. If A_ε or B_ε, one side's set is empty and the other party has to send the whole set. In this case, recursion also stops at both sides. If A_ε and B_ε are nonempty and different, both sides recurse to v_0 and v_1. The same is repeated for every vertex visited. Only in case of a leaf, no recursion is done because each set contains at most one element so the set difference can be computed trivially.

From this description it is also clear that the recursion tree looks exactly the same on both sides: Alice and Bob visit the same trie vertices in the same order; Alice recurses exactly when Bob recurses and stops recursion if and only if Bob stops recursion. Because of this, it is sufficient to send only the subset digests in the vertex visit order, without any further labelling.

Complexity

Communication complexity How does the protocol fare on the different complexity measurements? We recurse from vertex v_s iff (1) there is at least one new leaf under this vertex in Alice's trie, (2) there is at least one leaf of any kind (new or old) under this vertex in Bob's trie (or vice versa). These events are independent. Let's examine the probability of the first condition $p_1 := \mathrm{P}[|A_s \setminus B_s| \geq 1]$. Because leaves are uniformly distributed, the expected number of new leaves under v_s is $\mathrm{E}[|A_s \setminus B_s|] = c \cdot |U_s|/|U| = c/2^d$, where d is the depth of the vertex. By Markov's inequality, $p_1 = \mathrm{P}[|A_s \setminus B_s| \geq 1] \leq \min(\mathrm{E}[|A_s \setminus B_s|], 1) = \min(c/2^d, 1)$.

Similarly, we can estimate $p_2 := \mathrm{P}[|B_s| \geq 1] \leq \min(n/2^d, 1)$. Therefore, the probability of recursing from a vertex is $p \leq 2p_1p_2 \leq 2\min(c/2^d, 1)\min(n/2^d, 1)$.

We multiply by two because the new leaf can be on either side and we use the union bound.

For the first $\lg c$ levels of the tree (which we shall call *slice I*), the estimated value of p is 2, which we shall cap to 1. We expect the recursion tree in this slice to be very close to a full binary tree. The total expected number of vertices recursed from in the slice thus is $\mathrm{E}[K_\mathrm{I}] \leq 2^{\lg c+1} = 2c$.

For the next $(\lg n - \lg c)$ levels (slice II), our estimate is $p \leq 2c/2^d$. The expected number of vertices visited on each of these levels is $\mathrm{E}[k_d] = 2^d \cdot \leq 2^d \cdot 2c/2^d = 2c$. Thus in total we expect to recurse from $\mathrm{E}[K_\mathrm{II}] \leq 2c(\lg n - \lg c)$ vertices in total on these levels.

For the remaining $\ell - \lg n$ levels (slice III) at the bottom of the tree, we estimate $p \leq 2cn/2^{2d}$. Thence again, $\mathrm{E}[k_d] \leq 2cn/2^d = 2c/2^{d'}$, where $d' := d - \lg n$ is vertex depth measured from top of the slice. Totalling over the slice we get $\mathrm{E}[K_\mathrm{III}] = 2c(1 + 1/2 + 1/4 + \dots) < 4c$. In this slice even elements common to A and B are becoming increasingly sparse so any recursion soon dies out because it hits an empty set on the other side.

The total expected number of vertices recursed from is simply $\mathrm{E}[K] = \mathrm{E}[K_\mathrm{I}] + \mathrm{E}[K_\mathrm{II}] + \mathrm{E}[K_\mathrm{III}] \leq 2c + 2c(\lg n - \lg c) + 4c = 6c + 2c(\lg n - \lg c)$. The total number of vertices visited is simply twice this number, i.e., $12c + 4c(\lg n - \lg c)$ and the total number of bytes transmitted is $cg(3 + \lg n - \lg c)$, where g is the digest size (we send two $g/8$-byte digests per visited vertex).

Communication rounds Now we would like to estimate the number of communication rounds. If the algorithm were implemented as described in algorithm 2.1, each visited vertex would cost us one round. However, the algorithm can be easily modified to perform a breadth-first traversal of the original recursion tree. Then we can send digests from all active vertices on a given level in a single round and the number of rounds needed is exactly the depth of the recursion tree.

This modification is shown as algorithm 2.2. It should be easy to see that this algorithm straightforwardly maps to the original.

We know an upper bound on the expected number of vertices $\mathrm{E}[k_d]$ visited on each level of the tree. From this, we can once again use Markov's inequality to estimate the probability as least one vertex is visited on that level. The expected number of rounds is then simply the expected number of levels on which ve visit at least one vertex. We will do this again per slice.

For slice I, we expect to visit all levels, i.e. $r_\mathrm{I} \leq \lg c$. For slice II, $\mathrm{E}[k_d] = 2c > 1$, so again we expect to visit all levels, $\mathrm{E}[r_\mathrm{II}] \leq \lg n - \lg c$. With slice III, we are finally getting somewhere. We have shown that $\mathrm{E}[k_d] \leq 2c/2^{d'}$, where d' is vertex depth relative to the top of slice III. Thus the probability of visiting at least one vertex on a level is bounded by $\min(2c/2^{d'}, 1)$. For $d' \leq 1 + \lg c$, this bound is equal to one. For all the subsequent levels, the probabilities form the geometric sequence with sum $1 + 1/2 + 1/4 + \dots < 2$. Thus the expected number of levels visited $\mathrm{E}[r_\mathrm{III}] \leq 3 + \lg c$.

When we put this together, we can bound the expected number of communication rounds by $\mathrm{E}[r] = \mathrm{E}[r_\mathrm{I}] + \mathrm{E}[r_\mathrm{II}] + \mathrm{E}[r_\mathrm{III}] \leq 3 + \lg n + \lg c$.

Note that our protocol is not a request-response protocol. Instead, communication in both directions happens at the same time. The message we recieve in round i is not a reply to the message we sent in round i but the one we sent in

Algorithm 2.2 Breadth-first modification of the divide-and-conquer reconciliation algorithm

```
1:  procedure RECON1-BFS(A)
2:      active ← [ε]                          ▷ ordered list of active vertices on cur. level
3:      C ← ∅                                 ▷ the local changes (A \ B)
4:      while active ≠ [] do
5:          d_A ← [DIGEST(A_s) | s ∈ active]
6:          SEND(‖ d_A)                       ▷ concatenation of all active vertices' digests
7:          d_B ← RECV() split into digest-sized chunks
8:          next ← []
9:          for 0 ≤ i < |active| do
10:             if d_A[i] = d_B[i] then
11:                 do nothing
12:             else if A_s = ∅ then
13:                 do nothing
14:             else if D_B = DIGEST(∅) then
15:                 C ← C ∪ A_s
16:             else if i = ℓ then
17:                 C ← C ∪ A_s
18:             else
19:                 append s ‖ 0 and s ‖ 1 to next
20:         active ← next
21:     return C
```

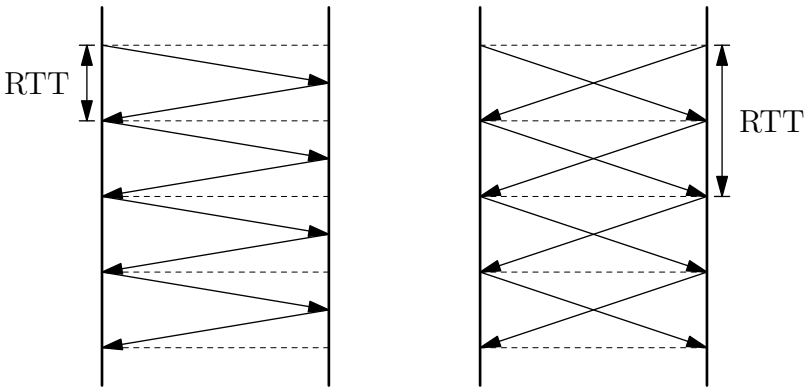
round $i-1$. This means that the number of network round trips required is half the number of cummunication rounds, as shown in fig. 2.2.

We should also realize the importance of the breadth-first optimization here. The naive recursive implementation would require as many rounds as vertices visited, $12c+4c(\lg n-\lg c)$. This would require hundreds to thousands of roundtrips for moderate values of c, which would result in a total time of several seconds to several minutes(!) depending on network quality.

Computational time If we use the naive digest function suggested in above, computional complexity will be simply too horrendous to be even worth estimating, definitely at least $\Omega(n)$. Instead, we can make the trie on each side into a Merkle tree [19]: we define the digest of any nonempty set A_s corresponding to vertex v_s as a cryptographic hash of the digests of two child sets in the trie ($\|$ is the string concatenation operator):

$$\textsc{Digest}(A_s) := \begin{cases} 0\cdots 0 & \text{if } A_s = \emptyset \\ \textsc{Hash}(\textsc{Digest}(A_{s\|0}) \,\|\, \textsc{Digest}(A_{s\|1})) & \text{otherwise} \end{cases}$$

We can store digests for all non-empty vertices on disk. This allows us to get any digest in $\mathcal{O}(1)$ expected time if we use a hashed store or $\mathcal{O}(\lg n)$ worst-case time if we use a tree-based structure (e.g. a typical SQL database with B-tree based indices, which is the case for the SQLite database used by Foloco). For an tree-based database, we get total computational time $\mathcal{O}(c\lg^2 n)$ for one



(a) request-response (b) "criss-cross"

Figure 2.2: Communication rounds vs network roundtrips

reconciliation. When adding a new object to the set, we must update the hashes of all of its ℓ ancestors, which can be done in time $\mathcal{O}(\ell \lg n)$.

2.2.2 Divide and conquer with pruning

From the estimates given in the previous section, we can infer that the recursion tree looks approximately as shown in fig. 2.3 for $c = 4$ (4 new objects on each side, 8 changes total).

Slice I is (close to) a full binary tree, slice II consists of mostly of separate non-branching paths (except for dead-end side branches that immediately terminate because they contain no changes), one for each change. Slice III contains short tails of these paths (expected length bounded by a constant) before recursion terminates.

This seems rather wasteful. Most of the algortihm is spent walking along the paths in slice II, always comparing digests of sets that differ by only one element.

What we would like is to be able to immediately detect that two sets differ only in one element and ideally also reconstruct that element. The XOR function immediately springs to mind. We can define the digest as

$$\textsc{Digest}(\{a_1, \ldots, a_k\}) := h(a_1) \oplus \cdots \oplus h(a_k),$$

where h is a cryptographic hash function. We need h because if we XORred the original strings (which determine trie location), digests of neighbouring nodes would be highly correlated.

Now when we have two sets A and B such that $A \triangle B = \{e\}$ then $\delta := \textsc{Digest}(A) \oplus \textsc{Digest}(B) = h(e)$. However if $|A \triangle B| > 1$, the δ a useless number. We need to determine which of these cases ocurred. The party with the extra element can simply look up δ in a reverse lookup table $h(x) \to x$.

However, this might yield a false positive. What is the probability of that happening? Because we presume values of h to behave as independent uniformly distributed random variables, the digests of any two sets differing in at least one element should behave as independent random uniformly distributed random variables. Thus the probability of a accidental collision of δ for a nontrivial difference with one specific element is close to $1/2^g$, where g is the digest size.

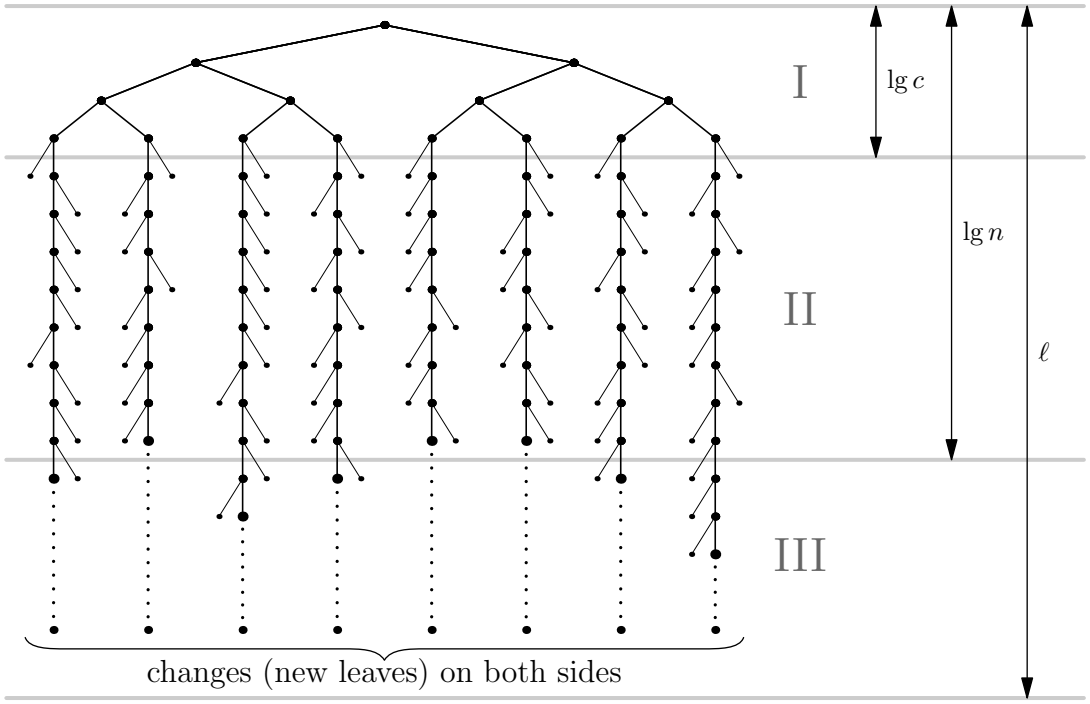


Figure 2.3: Recursion tree of RECON1 (algorithm 2.1)

The probability of collision with any element can be estimated using the union bound as $p \leq n/2^g$. If we want this to be as collision-resistant as a g-bit hash function, we need to use a longer hash, specifically one with $g' := g + \lg n$ bits.

If the extra element is on the other side, we must recurse for now and the other party will inform us in the next round that we should stop any further recursion.

There is an alternative to simply using a longer hash function, and that is to add a checksum to each element's hash as follows:

$$\text{ELEMDIGEST}(x) := h(x) \parallel h(h(x)),$$

$$\text{DIGEST}(\{a_1, \ldots, a_k\}) := \text{ELEMDIGEST}(a_1) \oplus \cdots \oplus \text{ELEMDIGEST}(a_k).$$

This brings the same level of fake positive resistance (probability $1/2^g$ per comparison) at the cost of more extra bits (g instead of $\lg n$). However, now both parties can independently detect that $|A \triangle B| = 1$ by checking if $h(\delta_1) = \delta_2$ (where δ_1 and δ_2 are the two halves of the δ string) and stop recursion immediately. This saves one roundtrip and simplifies implementation. It is not clear which approach is better, both have their mertis.

The second variant (with another checksum hash) is summarized as algorithm 2.3 and implemented in Filoco.

A similar approach has been independently discovered earlier by Minsky and Trachtenberg [21]. They use a scheme based on polynomials over finite fields for pruning branches where the symmetric difference is small. [22] Our solution achieves comparable asymptotic bounds and practical results (even though perhaps with worse constant factors) and is much simpler both conceptually and to implement. They also discuss using XOR for the case of $|A \triangle B| = 1$. [22, protocol 1] However, they XOR the original bitstrings from the set instead of

Algorithm 2.3 Divide-and-conquer set reconciliation with pruning

```
1: procedure DIGEST(A)
2:     return h(A) ‖ h(h(A))
3: procedure RECON2(A, i = 0)
4:     D_A ← DIGEST(A)
5:     SEND(D_A)
6:     D_B ← RECV()
7:     δ ← D_A ⊕ D_B
8:     split δ into two halves δ_1 and δ_2
9:     if D_A = D_B then
10:        return ∅
11:    else if A = ∅ then
12:        return ∅
13:    else if D_B = DIGEST(∅) then
14:        return A          ▷ other side's set is empty, need to send everything
15:    else if h(δ_1) = δ_2 then                                  ▷ |A△B| = 1
16:        if ∃x ∈ A with h(x) = δ_1 then          ▷ we have the extra element
17:            return {x}
18:        else                                    ▷ they have the extra element
19:            return ∅
20:    else if i = ℓ then
21:        return A
22:    else
23:        A_0 ← {x ∈ A | x_i = 0}          ▷ All the elements with i-th bit zero
24:        A_1 ← {x ∈ A | x_i = 1}
25:        return RECON2(A_0, i − 1) ∪ RECON2(A_1, i − 1)
```

their hashes, which makes this technique unsuitable for branch pruning because all the elements under a vertex with depth d have the first d bits in common and therefore the first d bits of the XOR are all ones or all zeroes, depending on the number of elements.

Complexity

Intuitively, pruning should cut off all the boring branches in slices II and III and leave us with $\mathcal{O}(\lg c)$ expected depth of the recursion tree, which corresponds to communication complexity $\mathcal{O}(c \lg c)$. Let's prove that.

Communication complexity We will have to use a slightly different estimation method. The recursion tree has at most c leaves, with one change under each. For each change w (a trie leaf present on one side but not other), we shall estimate the length of the recursion branch leading to that node. We shall consider all the trie ancestors of the change and for each of them compute the probability that we recursed from that vertex.

We recurse from a vertex only if there are at least two changes (in total on both sides) underneath it. There are a few other conditions (for example, recursion stops if the subset on one side is empty, even if the other party has two changes), which we shall ignore because we are doing an upper bound. This means that at least one of the $2c-1$ changes other than the one we are currently examining must lie under this vertex. Because we consider changes to be independent and uniformly distributed, the probability of this happening can be easily estimated, once again using Markov's inequality:

$$p_d := \mathrm{P}[|A_s \triangle B_s| - \{w\}] \leq \min(\mathrm{E}[|A_s \triangle B_s| - \{w\}], 1) = \min(\mathrm{E}[|A_s \triangle B_s|] - 1, 1) =$$

$$= \min(c/2^d - 1, 1) \leq \min(c/2^d, 1),$$

where d is the depth of the ancestor.

Let's look at the values of p_d by slice. In slice I, $2^d \leq c$ and the estimate maxes out at 1. For slices II and III, we get $p_d \geq c/2^d = 1/2^{d'}$, where $d' := d - \lg c$ is depth relative to the top of slice II. Now we can easily estimate the length of a recursion tree branch:

$$\mathrm{E}[L_w] \leq \sum_{d=0}^{\ell} p^d = \lg c + 1 + 1/2 + 1/4 + \cdots \leq 2 + \lg c.$$

The expected number total number of vertices recursed from is bounded by the sum of the recursion branch lengths (we count many vertices several times), which we can estimate using linearity of expectation:

$$\mathrm{E}[K] \leq \mathrm{E}\left[\sum_{w \in A \triangle B} L_w\right] = \sum_{w \in A \triangle B} \mathrm{E}[L_w] \leq 2c \cdot (2 + \lg c) = 4c + 2c \lg c.$$

This corresponds to $2gc + gc \lg c$ transferred bytes for a $g - bit$ hash (we send two hash values per vertex, there are at most twice as many vertices visited as recursed from).

Communication rounds In a similar manner, we can estimate the number of communication rounds, again presuming this algorithm is first transformed to a breadth-first version in a manner similar to algorithm 2.2. The modified version is not shown here but can be found in attachment 1, both as a standalone experiment (in the file `experiments/mdsync/hybrid.py`) and as a part of Filoco proper (classes `TreeMDSync` in `mdsync.py` and `SyncTree` in `store.py`).

For each level of the trie, we will examine the probability recursion gets to this level. We have at most c recursion branches, each of them traversing level d with probability $p_d < c/2^d$. Using the union bound, the probability of at least one branch traversing this level is $q_d < c^2/2^d$. For $d < 2 \lg c$, this bound is larger than one. For further levels, it forms a geometric sequence. Thus the expected depth of the recursion tree is $\mathrm{E}[r] = 1+2 \lg c+1+1/2+1/4+\cdots < 3+2 \lg c$. This is the number of communication rounds required by the breadth-first variant of algorithm 2.3.

Computational time As for computational time, we can once again organize the digests into a Merkle-like tree stored on disk and incrementally updated. Only this time each vertex computes a XOR instead of a cryptographic hash. Thus we get the same $\mathcal{O}(\lg n)$ query time and $\mathcal{O}(\ell \lg n)$ update time. The total computational time is then $\mathcal{O}(c \lg c \lg n)$.

As a further optimization, we notice that if the ℓ is larger than $2 \lg n$, the bottom levels will be rarely ever used during synchronization. We can thus further optimize by only storing the to $\alpha \lg n$ for an empirically chosen constant $1 \leq \alpha \leq 2$. The missing levels can be computed on-the-fly (for example if the set items are stored in a SQL database that supports range queries, we can simply enumerate all the elements under a vertex because they form a contigous segment).

This changes storage requirements to $\mathcal{O}(n \lg n)$ and update time to $\mathcal{O}(\lg^2 n)$.

For clarity, we summarize the efficiency of both algorithms in tbl. 2.1 (for 128-bit digests).

Experimental results were produced by the `experiments/mdsync/prune.py` script in attachment 1.

The "total roundtrip time" and "total transfer time" are synchronization time estimates based on roundtrip numbers and transfer total from experimental protocol simulations (no actual time measurements were performed). These are computed for a hypothetical low-quality network with 1 Mbps symmetric throughput and 500 ms RTT (for example a 3G connection with subpar reception). The total synchronization time will be probably be close to the maximum of the two estimates. For any network with significantly better parameters the times will become imperceptible.

Table 2.1: Comparison of described set reconciliation algorithms

Metric	Naive D&C	Pruning D&C
total bytes transferred	$128c(3 + \lg n - \lg c)$	$128c(2 + \lg c)$
for $n = 2^{20}$	$128c(23 - \lg c)$	$128c(2 + \lg c)$
for $c = 16$ (theor.)	38 kB (1.2 kB p.ch.)	12 kB (0.3 kB p.ch.)
for $c = 16$ (exper.)	51 kB (1.6 kB p.ch.)	11.1 kB (0.3 kB p.ch.)

Metric	Naive D&C	Pruning D&C
for $c = 1024$ (theor.)	1.6 MB (0.8 kB p.ch.)	1.5 MB (1.5 kB p.ch.)
for $c = 1024$ (exper.)	1.6 MB (0.8 kB p.ch.)	650 kB (0.3 kB p.ch.)
communication rounds	$3 + \lg n + \lg c$	$3 + 2 \lg c$
for $n = 2^{20}$	$23 + \lg c$	$3 + 2 \lg c$
for $c = 16$ (theor.)	27	11
for $c = 16$ (exper.)	8	4
for $c = 1024$ (theor.)	33	23
for $c = 1024$ (exper.)	9	8
computational time	$\mathcal{O}(c \lg^2 n)$	$\mathcal{O}(c \lg c \lg n)$
disk storage	$\mathcal{O}(n \lg \lvert U \rvert)$	$\mathcal{O}(n \lg \lvert U \rvert)$
update time	$\mathcal{O}(\lg \lvert U \rvert \lg n)$	$\mathcal{O}(\lg^2 n)$
total roundtrip time (proj.)		
for $c = 16$	2 s	1 s
for $c = 1024$	2.25 s	2 s
total transfer time (proj.)		
for $c = 16$	0.4 s	0.08 s
for $c = 1024$	12.8 s	5.2 s

In the model situation of $n = 2^{20}$ and $c = 16..1024$, both algorithms seems comparable within a factor of two, both theoretically and experimentally. However, the theoretical bounds are not tight enough to distinguish between the two algorithms for these parameter values, thus we should give more credence to the experimental results.

Please note that all the transfer and time estimates cover only the process of determining which objects each side is missing. After this, we must transfer the serialized objects themselvers; this is not included in our estimates.

Both algorithms seem usable for our application. However, the pruning algorithm performs better, is only slightly more complex and offers much greater scalability because its communication complexity and number of rounds do not depend on n.

2.3 Per-Origin Sequential Streams

Upon further reflection, we actually do not need to solve the fully general set reconciliation problem. Our instance is rather special in one key factor: each object is only ever created once, in one store. Therefore, the assumption of set reconciliation that there is no prior communication between the parties is not true. If two stores share an object, there must have been prior communication between each of them and the object's originating store, albeit possibly indirect.

There are two more important special properties: (1) we always perform full synchronization (unless the synchronization process is interrupted), partial synchronization is not supported; (2) the number of stores in a realm is expected to be small (in the order of tens at most).

If we put all these facts together, we can devise a synchronization scheme both simpler and more efficient than the described reconciliation algorithms.

The idea is simple: instead of considering all the object a store has as one big set, we will split them into several sets based on their originating stores and solve the reconciliation problem for each of these sets separately.

Now we have a different task: several nodes have copies of a set, which they synchronize with each other in a disorganized peer-to-peer fashion. But only one node ever adds new elements to the set! All other nodes must have got their elements from this originating node, directly or indirectly.

This is rather simple to solve: the originating node will assign created objects sequence numbers as they are created. All nodes will keep their sets sorted by these seqence numbers, essentially transforming the problem into one of sequence reconciliation.

Whenever Alice and Bob want to reconcile their sequences, they simply compare their maximum sequence numbers m_A and m_B. If $m_A > m_B$, Alice sends all her objects with sequence number greater than m_B to Bob (who clearly does not have them) in increasing sequence number order (this is important). Bobs appends them to his sequence in the order he recieves them and sends nothing to Alice. If $m_A < m_B$, the same happens in the opposite directions.

We claim this is sufficient to synchronize their sets/sequences. Why? A simple invariant I holds: the sequence of objects possesed by any node is always a prefix of the originating node's sequence. We can prove I by induction. At the beginning, all sequences are empty and I holds trivially. Two kinds of events can happen:

- The originating store adds an element at the end of the seqence. This clearly preserves I.
- Two nodes A and B (for which I holds) synchronize their sequences s_A and s_B. We can assume without loss of generality that $m_A \geq m_B$. At the beginning s_A and s_B are prefixes of the originator's sequence s_O. Because s_B is a shorter prefix, it is also a prefix of s_A. After each object transferred, s_B becomes a longer prefix of s_A, and thus still a prefix of s_O. s_A is unchanged.

This yields a simple synchronization algorithm for complete metadata synchronization:

Algorithm 2.4 Reconciliation using per-origin sequential streams

```
1:  procedure RecvObjects
2:      while other side has not signalled EOF do
3:          o ← RecvSerialized()
4:          add o to the local database (at the end of originator(o)'s sequence)
5:  procedure SendObjects(M_A, M_B)
6:      for every store s present in both M_A and M_B do
7:          if M_A[s] > M_B[s] then
8:              for every object o with originator(o) = s and seq(o) > M_B[s] do
9:                  Send(Serialize(o))
10: M_A = {(id(s), maxseq(s)) | s store}
11: Send(M_A)
12: run SendObjects(M_A, M_B) and RecvObjects() in parallel
```

Thus we can perform synchronization with only one roundtrip and $\mathcal{O}(\#\text{stores})$ bytes overhead in addition to whatever is required to transfer the acual objects missing on the other side.

This leaves the question of why we bother with set reconciliation when a simpler and more efficient solution exists. There are several reasons:

- We actually discovered it much later than the general set reconciliation algorithms. This seems strange because at first sight, the idea seems rather trivial. But it is probably somehow evasive. Not only did we almost miss it; for example the leading open source synchronization tool Syncthing also uses a sequence numbering scheme but one that is slightly different and suffers from the indirect synchronization problem. [23]

- Set reconciliation is an interesting problem by itself. That should be enough reason for anyone. It also has numerous other applications, both within file synchronization an elsewhere. For example, it has been used for delta transfer of files as a replacement of the established rsync algorithm [24, sec. 4.1.2].

- Assigning sequential numbers has some reliability issues described bellow.

Currently, both approaches are implemented in Filoco, with sequence numbering being the default. The main reason is surprisingly not the difference in reconciliation times but the need to keep the reconciliation trie on disk, increasing both storage overhead and slowing down database updates.

2.3.1 The problem with sequence numbers

Any attempt to assign sequential numbers is potentially problematic. It can happen that Alice creates an object o_1, assigns it a sequence number s and transfers it to Bob. Then Alice suffers from a power loss before o_1 has been flushed to disk. After reboot, she creates a completely unrelated object o_2, which nevertheless gets assigned the same sequence number s, because the information about s being already taken has been lost. Now when Alice synchronizes with Bob, their maximum sequence numbers for Alice-originated objects will be the same, namely s. Thus they will mistakenly think their object sets are identical, despite Alice missing o_1 and Bob missing o_2.

Several things can be done about this. The simplest is to flush (`fsync`) local changes to disk before every synchronization.

If we do not want to do that or do not trust the disk to reliably fullfil the request (which is known to happen at times), we can instead check that the common prefix is really the same on both sides.

For example we can store for each prefix of the local object sequence a XOR of its object IDs. Upon synchronization, Alice and Bob exchange XORs of their complete sequences, x_A and x_B, in addition to their maximum sequence numbers $m_A \geq m_B$. Now Alice can compare x_B to her prefix XOR of the corresponding prefix ending with seqence number m_B. If they match, Bob really has the same objects as are in her prefix and it is sufficient to send the remaining suffix.

Otherwise a different synchronization scheme must be used. For example, we could perform a binary search on the sequence numbers to find the longest

common prefix by comparing corresponding prefix XORs. Then both sides can simply exchange the remaining suffixes and merge them into their sequences, updating the neccesary prefix XORs. Presumably the error has occured recently so the suffixes that need to be fixed should not be long.

3. Content Synchronization

After metadata synchronization, each store knows which files need to be updated. Namely any files that the store hosts (or wants to host) and for which there is a head FCV that the store does not have. Storage records will contain information about where the data can be obtained.

When a store that can provide the right data is found and contacted, the transfer itself can begin. There are many ways of transferring incremental file updates over a network; this is a fairly well-researched problem.

The trivial solution is to simply send over the new version of the file. However, that is fairly inefficient for when small changes are done to large files. As with metadata synchronization, ideally, we would like the amount of transferred data to depend more on the size of the change than on the size of the whole file. Techniques to achieve this are generally called *delta transfer* algorithms.

Most delta transfer methods work by somehow splitting the file into blocks on both sides. The blocks may be of fixed or varying size, aligned, unaligned or even overlapping, the splitting may be identical or different on the two sides. Then we the sending party must somehow learn which blocks are already present on the receiver side as part of the old version of the file – or in some cases, even blocks from different files are reused.

That is especially useful in the absence of rename detection because then after a rename, the target file can be reconstructed from blocks of the source file without retransmitting the data over the network. However, since we have reliable rename detection, we opted for doing delta transfers of each file separately, isolated from the others. While cross-file block reuse might still provide some optimizations because file systems often contain similar files, these are less important and given the number of files we have to deal with, considering all the blocks in all the files would be quite hard (although definitely not impossible) to do efficiently.

When it is determined which blocks the receiving party is missing, we simply send them along with any instructions necessary to reassemble the whole file from both old and new blocks.

Now the key questions are: (1) how exactly to split files into blocks, (2) how to determine which blocks the other party already has.

3.1 The Rsync Algorithm

The trivial solution to (1) is to always split the files into fixed-size, aligned blocks. This technique breaks whenever contents is inserted to or deleted from the file. Then, block boundaries shift and none of the blocks will match. Some file snychronization tools nevertheless use this approach, for example the already mentioned Syncthing. [23]

The trivial solution to (2) is for the receiving side to simply send checksums of its blocks to the sending size. This reduces the transfer requirements by at most a constant factor because we need to send Θ(file size) checksums. In practice, however, this is often sufficient, as demonstrated by the success of the rsync algorithm [25], now a de facto standard for delta transfers.

Rsync splits the old file on the receiving side into fixed-size aligned blocks. For

each of these blocks, two different checksums are computed: a "slow" checksum (a cryptographic hash, which is reliable but expensive to compute) and a "fast" checksum (that is unreliable but cheap to compute). Now comes the key trick: the fast checksum is computed using a rolling hash function. That means when we know the hash for a w-byte substring of the file starting at position i, we can efficiently (in constant time) use it to compute the hash for a w-byte string starting at position $i + 1$. This is often called a "sliding window" algorithm: we imagine having a "window" w bytes byte wide that we are moving over the file. Each time we can move it one byte to the right and efficiently recompute the hash of the string now in the window.

This property does not seem useful when computing hashes of aligned blocks on the receiver side. However, the sender uses the sliding window property to compute the fast checksum of w-bytes blocks starting at every possible byte offset in the file. This allows finding shifted and unaligned blocks.

Now the sending party transfers instructions for reconstructing the file. Each instruction is either (a) write a given block from the original file to the new file at a given offset, (b) write these bytes to the new bytes at a given offset (used for parts of the file not covered by any old blocks, these can be of varying sizes from a few bytes to the whole file).

The cannonical implementation of the rsync algorithm is a part of the `rsync` program [3]. However, `rsync` has its own protocol and semantics for establishing connections, authentication, dealing with multiple files, dealing with file paths, etc., that do not fit well into our design. A more promising implementation of the algorithm is available in the `librsync` library. [26] This library implements only the pure rsync algorithm and leaves all the other aspects, including the logistics of network communication and filesystem access, up to the application, which makes it very flexible.

It is the `librsync` library that was intended to be used for implementing content synchronization in Filoco, although the implementation was never finished.

3.2 Set Reconciliation Based Methods

We might notice that the structure of block-based synchronization problem is rather similar to the set reconciliation problem: both sides have some blocks and Alice wants to send Bob exactly the blocks he does not have.

However, in order for them to use set reconciliation, both of them must split the file into blocks *independently* to create the sets to be reconciled. Therefore we cannot use the rsync trick where Alice's splitting is dependent on knowledge of Bob's blocks.

This can of course be accomplished by the already mentioned fixed aligned block splitting, which has numerous problems.

An alternative is to determine block boundaries based not on file offsets but on content. For example we can once again use a rolling hash and make a block boundary whenever the hash value is smaller than some fixed value. Now when the two files share a segment that has at least two block boundaries in it, the block between the boundaries will be split indentically on both sides, reagardless of the offset. If we consider the hash values to be essentially random, this gives us expected block length ℓ/m, where ℓ is the fixed limit and m is the maximum

value of the hash function. To prevent extreme cases, we should also bound the minimum and maximum length of the block and if necessary, cut in non-standard places.

The use of set reconciliation and content-dependent block splitting for file synchronization has been thoroughly examined by Marco Gentili in his bachelor thesis. [24]

3.3 Filesystem Access

A general-purpose file synchronizer, possibly running in the background, has to deal with the file system being concurrently changed. For metadata changes (creates, renames and deletes), this has been already tackled in chapter 1.

However, file contents can also change, and do so in two ways. Some programs write to directly to the destination file, while others first create a temporary file, write the new version to it and then replace the original with an atomic `rename`.

When a file is changed while it is being synchronized, the synchronization will probably not give meaningful results. This is not only because of race conditions involved in the synchronization algorithms (for example we compute a checksum of a block and then the block changes). Even a simple whole-file copy is riddled with possible race conditions. For example, after you have copied half the file, someone may concurrently make one change and the beginning and then one change at the end. Your copy will contain the unchanged beginning and the changed end, a version of the file that was never present in the original.

This is far less far-fetched than it may seem. For example, a program might first change some area in the file's header to remove a pointer to some records at the end of the file and then physically remove the records at the end, perhaps overwriting them with something else. In your copy, the pointer in the header would be still present, now pointing to garbage data.

There is probably no way to recover from such situations. The only way to correctly make a copy of a file is when it does not change during the copy/transfer process. There are two ways this might be achieved: (1) lock the file in some fashion to prevent other programs from accessing it, (2) make the copy in a transactional fashion. The kind of locking needed for (1) is more or less impossible for locking.

As for (2), we can reuse the oldest trick in our book, namely comparing before/after mtimes. First we remember the mtime of the source file, then we perform the copy/transfer into a temporary file, then we check the source mtime again. If it has not changed, we consider the copy correct, otherwise we start over.

When synchronizing over a network, we can perform the complete synchronization protocol reading from the original files, while the receiver saves the result into a temporary file. At the end, if the files on neither side changed according to mtime comparisons, the parties agree to commit the transaction, otherwise they retry the whole protocol again.

A different kind of race condition might occur on the receiver side. Between the moment that we decided the original files have not changed and we should commit the transaction and the moment we actually replace the target file using an atomic rename, the target file might be changed. In this case, we would

replace the target file with a consistent version but lose some recent changes. With a normal `rename`, this cannot be prevented. However, on Linux we can use the extended `renameat2` syscall with the `RENAME_EXCHANGE` flag. This causes the kernel to atomically exchange the temporary file with the target file. If any concurrent modifications were made during the brief window, they will now be available in the location of the temporary file. We could record this as a normal version conflict and store both verions of the file.

4. Implementation

A sketch of Filoco implementation is provided as attachment 1 in the electronic version of this thesis. Due to a limited amount of time, the implementation is in many regards incomplete and currently serves more as a proof of concept of some of the techniquest discussed here than a piece of software that would be actually useful for synchronizing one's files.

Most of the implementation is a rather straightforward application of the algorithms and methods described. Here we will pinpoint only a few interesting technical aspects.

4.1 Metadata Storage

Stores are odinary directories that contain a special subfolder named `.filoco`. This contains all the Filoco-specific metadata. Most of the metadata is stored in and SQLite [27] database called `.filoco/meta.sqlite`. This database is composed of two logical parts:

- Local filesystem metadata, used to store the last known filesystem state to compare against when scanning the file system for changes.
  - The `inodes` table contains information about every inode known to Filoco, including its inode number, file handle (used to identify inodes as discussed in sec. 1.1.5) and the FOB, FLV and FCV currently associated with this inode, if any.
  - The `links` table stores information about directory entries, with parent inode identifier, child parent identifier and name for each.
- Sychronized metadata, that model more or less one-to-one the structure described in sec. 2.1.1.
  - The `syncables` table holds information common to all object types, such as originating inodes and sequence numbers (for per-origin sequence number synchronization) or position synchronization trie key (for set-reconciliation-based synchronization).
  - The `fobs`, `flvs`, `fcvs` and `srs` tables hold information specific to the individual object files.

We heavily rely on SQLite's atomic transaction support. For example, when a file's mtime has changed, we record the new mtime and the newly created working FCV in the same transaction so that a power failure does not cause changes to be forgotten. We also update the synchronization digest trie in the same transaction as inserting a new object if the set reconciliation scheme is used. This ensures that the precomputed subset digests are never out of date.

SQLite is often accused of being "slow". This however depends heavily on the way one uses it. By default, it performs an `fsync` at the end of every transaction, which definitely causes issues. This can be helped by grouping updates to large enough transactions. As most operations we do are bulk anyway (filesystem

scanning, metadata synchronization), it is not a problem to make transactions for say every 5000 scanned/transferred items.

Where this does not help is online change detection, because there changes come separately rather than in bulk. In this case, switching SQLite to the so-called WAL (Write-Ahead Log) mode [28] supported by newer version that implements transactions in a different way and does not require a sync after every signle transaction.

Another big scalability issue is with filesystem scanning, because it has to read the file system and update the metadata database, causing seeks between scanned inodes and database blocks and thus rendering all our precious scan optimizations useless. This can be partially helped by forcing SQLite to cache more changes in memory and increasing its cache size.

However, SQLite's cache management is not perfect and it starts to lose scalability with large enough databases. What we would really want it to interleave periods of pure filesystem scanning with periods of pure database updates, each at least tens of seconds long.

The easiest way to do this is to cache scan results in separate in-program data structures and only give them to SQLite in batches. However, during scan, we also need to read the old inode data from sqlite to compare against, which can also be slow for larger databases.

The easiest solution seems to be to load all inode information from the database to memory on start, then scan the file system and make SQLite updates in batches. This is more or less what our `check_helper` program does.

4.2 Basic Structure

The implementation comprises of several independent programs. They currently have to be run manually every time the user wishes to perform an action such as rescanning the file system or synchronizing two stores. Nothing happens automatically in the background, with the exception of fanotify-based filesystem online filesystem watching in `scan.py`.

All of the programs directly access the underlying SQLite database and though SQLite performs some locking, it is not recommended to run any two of them at the same time (with the exception of `scan.py` in fanotify live watch mode).

These are:

- `init.py` – creates a new Filoco store
- `scan.py` – performs online and/or offline scanning of the file system and updates metadata accordingly
- `check_helper.c` is a helper program in C used by `scan.py` to make incremental rescans faster
- `mdsync.py` – performs metadata synchronization between two stores
- `mdapply.py` – applies metadata updates received from remote stores (moves and renames) to the local file system
- `dsync.py` – performs file content synchronization (unfinished)

More information about installing and using these programs can be found in the `README.md` file in attachment 1.

Conclusion

We set out with a goal of designing and implementing an efficient, scalable, robust, flexible, and secure file synchronization (tool)kit for advanced users. Unsurprisingly, we have stopped quite far from this mouthful of a goal, due to mainly time constraints. We present a collection of solutions to some important subproblems lying on the way to our target rather than a finished piece of software.

Despite the limited scope, we have touched on a fairly broad range of topics, from filesystem architecture to kernel development and randomized algorithms. The following are the most important original contributions of this work, in roughly the order they appear in the text:

- A mechanism for reliable rename detection during file system scanning, based on inode numbers and file handles. This in turn allows efficient synchronization of directory trees when directories with large subtrees are moved or renamed. A proof of concept implementation of this mechanism has been provided.

- A mechanism for speeding up incrementally scanning a file system for changes about 2 to 8 times by accessing inodes using file handles in inode number order, including experimental measurements.

- A patch to the Linux kernel that extends the fanotify change notification interface with the ability to report directory modification events (creating, renaming, moving and deleting directory entries), a feature for which there has been great demand for since the creation of fanotify in 2009 but almost no solution attempts.

- The concept of placeholder inodes to represent files not available locally in a way that allows seeing and manipulating them with arbitrary file management tools. A partial proof of concept implementation has been provided.

- An independently discovered simpler version of the Partition-Recon set reconciliation algorithm (called *divide and conquer with pruning* in our text) first described by Minsky and Trachtenberg in 2002 [21]. Elementary proofs of some complexity bounds, experimental simulations of the algorithm and a full implementation with on-disk storage of the digest tree have been provided.

- A simple algorithm for peer-to-peer synchronization among a small set of nodes with single roundtrip overhead using per-origin sequence numbers.

From among our stated priorities, we ended up focusing mostly on efficiency, scalability and robustness, especially with regard to the change detection and metadata synchronization aspects of file synchronization, while touching only briefly and indirectly on file content synchronization and security.

There is a lot of room for future work. Partly in putting all of the techniques described here together into a complete tool suitable for daily use, partly in future research into areas neglected here, especially content synchronization and security.

Bibliography

1. *Úvod do matfyzáka* [online] [visited on 2017-07-17]. Available from: http://michal.bdnet.cz/matfyzak.html.
2. *The Linux man-pages project*. path_resolution(7) [online] [visited on 2017-07-20]. Available from: http://man7.org/linux/man-pages/man7/path_resolution.7.html.
3. TRIDGELL, Andrew; MACKERRAS, Paul, et al. *rsync documentation*. rsync(1) [online] [visited on 2017-07-20]. Available from: http://download.samba.org/pub/rsync/rsync.html.
4. TS'O, Theodore. *The Linux Kernel Mailing List archive*. Re: readdir loses renamed files [online] [visited on 2017-07-20]. Available from: https://lkml.kernel.org/r/20041025123722.GA5107%40thunk.org.
5. *The Linux man-pages project*. open_by_handle_at(2) [online] [visited on 2017-07-20]. Available from: http://man7.org/linux/man-pages/man2/open_by_handle_at.2.html.
6. CALLAGHAN, Brent; PAWLOWSKI, Brian; STAUBACH, Peter. *NFS Version 3 Protocol Specification* [Internet Requests for Comments]. RFC Editor, 1995. ISSN 2070-1721. Available also from: http://www.rfc-editor.org/rfc/rfc1813.txt. RFC.
7. *The Linux man-pages project*. xattr(7) [online] [visited on 2017-07-20]. Available from: http://man7.org/linux/man-pages/man7/xattr.7.html.
8. *Linux kernel source tree*. ext2.txt [online] [visited on 2017-07-20]. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/ext2.txt?id=v4.10#n85.
9. *The Linux man-pages project*. inotify(7) [online] [visited on 2017-07-20]. Available from: http://man7.org/linux/man-pages/man7/inotify.7.html.
10. *The Linux man-pages project*. fanotify(7) [online] [visited on 2017-07-20]. Available from: http://man7.org/linux/man-pages/man7/fanotify.7.html.
11. LARSSON, Alexander. *The Linux Kernel Mailing List archive*. Re: Issues with using fanotify for a filesystem indexer [online] [visited on 2017-07-20]. Available from: https://lkml.kernel.org/r/1238272705.23703.77.camel%40fatty.
12. ŠTĚDRONSKÝ, Filip. *The Linux Kernel Mailing List archive*. [RFC 1/2] fanotify: new event FAN_MODIFY_DIR [online] [visited on 2017-07-20]. Available from: https://marc.info/?m=148944656130682.
13. GOLDSTEIN, Amir. *The Linux Kernel Mailing List archive*. [RFC][PATCH 0/6] fanotify: super block root watch [online] [visited on 2017-07-20]. Available from: https://lkml.kernel.org/r/1489411223-12081-1-git-send-email-amir73il%40gmail.com.

14. HESS, Joey. *git-annex* [online] [visited on 2017-07-19]. Available from: http://git-annex.branchable.com/.

15. CHACON, Scott; STRAUB, Ben. Git Internals: Git Objects. In: *Pro git*. Apress, 2014. Available also from: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects.

16. KHANNA, Sanjeev; KUNAL, Keshav; PIERCE, Benjamin. A formal investigation of diff3. *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. 2007, pp. 485–496. Available also from: http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf.

17. FIDGE, Colin J. Timestamps in message-passing systems that preserve the partial ordering. 1987. Available also from: http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf.

18. MATTERN, Friedemann et al. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*. 1989, vol. 1, no. 23, pp. 215–226. Available also from: http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/mattern88virtual.pdf.

19. MERKLE, Ralph C. A digital signature based on a conventional encryption function. In: *Conference on the Theory and Application of Cryptographic Techniques*. 1987, pp. 369–378. Available also from: http://link.springer.com/content/pdf/10.1007/3-540-48184-2_32.pdf.

20. MINSKY, Yaron; TRACHTENBERG, Ari. *Efficient reconciliation of unordered databases*. 1999. Available also from: https://ecommons.cornell.edu/bitstream/handle/1813/7432/99-1778.pdf. Technical report. Cornell University.

21. MINSKY, Yaron; TRACHTENBERG, Ari. Practical set reconciliation. In: *40th Annual Allerton Conference on Communication, Control, and Computing*. 2002, vol. 248. Available also from: https://gnunet.org/sites/default/files/practical.pdf.

22. MINSKY, Yaron; TRACHTENBERG, Ari; ZIPPEL, Richard. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*. 2003, vol. 49, no. 9, pp. 2213–2218. Available also from: https://ecommons.cornell.edu/bitstream/handle/1813/5803/2000-1813.ps.

23. *Syncthing documentation*. Block Exchange Protocol v1 [online] [visited on 2017-07-19]. Available from: https://docs.syncthing.net/specs/bep-v1.html.

24. GENTILI, Marco. *Set Reconciliation and File Synchronization Using Invertible Bloom Lookup Tables*. 2015. Available also from: https://dash.harvard.edu/bitstream/handle/1/14398536/GENTILI-SENIORTHESIS-2015.pdf. Harvard University.

25. TRIDGELL, Andrew; MACKERRAS, Paul, et al. *The rsync algorithm*. 1996. Available also from: https://openresearch-repository.anu.edu.au/bitstream/1885/40765/3/TR-CS-96-05.pdf. Technical report. The Australian National University.

26. POOL, Martin et al. *librsync* [online] [visited on 2017-07-21]. Available from: `https://librsync.github.io/`.

27. *SQLite* [online] [visited on 2017-07-21]. Available from: `http://sqlite.org/`.

28. *SQLite documentation*. Write-Ahead Logging [online] [visited on 2017-07-21]. Available from: `https://sqlite.org/wal.html`.

List of Abbreviations

FCV – file content version (sec. 2.1.1).
FLV – FOB location version (sec. 2.1.1).
FOB – filesystem object (sec. 2.1.1).
IID – inode identifier (sec. 1.1.5).
mtime – an inode's last modification time, as reported by the `lstat` syscall
NFS – the Network File System [6]
OFD – open file description, in internal kernel structure describing an open file
RTT – network round-trip time (i.e., what `ping` measures)
syscall – system call, a function implemented by the kernel that can be invoked from user space

A. Attachments

Attachment 1 (`filoco-0.1.tar.gz`) is a part of the electronic version of this thesis. It contains the source code of the Filoco implementation sketch, several experiments and proofs of concepts, and the `FANOTIFY_MODIFY_DIR` kernel patches.