

# File Synchronization with Distributed Version Lists

*Bachelor's Thesis*

Alon Dolev of Zurich, Switzerland  
07-922-115 alon@dolev.ch

*Supervised by*

Dr. Matthew Cook  
Institute of Neuroinformatics  
ETH Zurich, Switzerland



Submitted to the Institute of Informatics  
University of Zurich, Switzerland  
Prof. Dr. A. Bernstein

August 20, 2012



## Abstract

Many modern computer users have multiple storage devices and they would like to keep the most up-to-date versions of their documents on all of them. In order to solve this problem, we require a mechanism to detect changes made to files and propagate the most preferable one: A file synchronizer. Many existing solutions need a central server, depend on constant network connectivity, can only synchronize in one way and bother the user with already-resolved version conflicts. We present a novel algorithm which allows for an optimistic, peer-to-peer, multi-way, asynchronous and optimal file synchronizer. It thus allows for changes in disconnected settings, does not require a central server, may synchronize any subset of the synchronization network at any time and it will not report false-positive conflicts. The algorithm improves on the well-known concept of version vectors presented by Parker *et al.* by allowing for conflict-resolution propagation. We do so by storing an additional bit of information for every version vector element. It is a more space-efficient solution to this propagation problem than the “vector time pairs” presented by Cox *et al.* and further, it is not restricted to one-way synchronization. We additionally present a novel user interface concept allowing for convenient handling of synchronization patterns. Based on these ideas we developed the file synchronizer McSync in order to show the feasibility of our approach.



## Zusammenfassung

Viele Computer-Nutzer sichern ihre Daten heutzutage auf mehreren Geräten und möchten diese stets auf dem neusten Stand halten können. Die Lösung dieses Problems benötigt einen Mechanismus, welcher Änderungen in Daten erkennen kann und dann jeweils nur jene auch aktualisiert. Viele solcher Daten-Abgleichungs-Tools, auch Filesynchronizer genannt, benötigen einen zentralen Server, sind netzabhängig, können nur in eine Richtung abgleichen und melden wiederholt bereits gelöste Versionskonflikte. Wir haben einen neuen Algorithmus entwickelt, welcher ein optimistisches, peer-to-peer, multidirektionales, asynchrones und optimales Filesynchronizing erlaubt. Ein solcher Filesynchronizer lässt Änderungen im unverbundenen Zustand zu, benötigt keinen zentralen Server, ist jederzeit in der Lage auch nur Teile der Daten abzugleichen und meldet keine falschen Versionskonflikte. Unser Algorithmus verbessert das bekannte Konzept der Versionenvektoren, welche von Parker *et al.* eingeführt wurde, dadurch, dass es bereits gelöste Versionskonflikte übertragen kann. Dies wird mit Hilfe eines zusätzlichen Bits per Vektoreintrag erreicht. Die existierende Lösung dieses Problems - die Vektoren-Zeit-Paare («vector time pairs») - benötigt mehr Speicher und ist zudem beschränkt auf unidirektionale Abgleichung. Zusätzlich präsentieren wir ein neues Konzept für Benutzerschnittstellen eines solchen Filesynchronizers. Basiert auf den eben vorgestellten Ideen haben wir den Filesynchronizer «McSync» entwickelt und so die Machbarkeit der hier neu vorgestellten Konzepte gezeigt.



## Acknowledgments

I would like to thank my supervisor Matthew Cook for sharing his ideas with me and for all his support, patience and encouragement: *Matthew, you are a great teacher and I have very much enjoyed working with you!*

I would also like to thank my co-supervisor Abraham Bernstein for all he has done for me and particularly for enabling me to work on my project of choice at an external Institute.

Last, but certainly not least, I would like to thank all those people who are always there for me, especially my family and Sarah, for all their love and support!



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement . . . . .	2
1.2	Distributed Conflict Detection . . . . .	4
1.3	Our Solution . . . . .	10
<b>2</b>	<b>Algorithm</b>	<b>1</b>
2.1	Definitions . . . . .	1
2.1.1	Machines . . . . .	1
2.1.2	Storage Devices . . . . .	1
2.1.3	Files and Aspects . . . . .	2
2.1.4	Scans . . . . .	3
2.1.5	Histories and Version Lists . . . . .	3
2.1.6	Identification . . . . .	4
2.1.7	Guidance . . . . .	5
2.1.8	Noticing . . . . .	6
2.1.9	The Virtual Tree . . . . .	7
2.1.10	Graft and Prune Points . . . . .	8
2.1.11	Preferability . . . . .	9
2.1.12	Instructions . . . . .	14
2.1.13	Gossip . . . . .	14
2.1.14	Synchronization . . . . .	18
2.2	Summary . . . . .	18

<b>3</b>	<b>Implementation - The File Synchronizer McSync</b>	<b>21</b>
3.1	The Build Process . . . . .	21
3.2	Core Architecture . . . . .	22
3.2.1	Configuration - The Device Folder . . . . .	22
3.2.2	Actors . . . . .	23
3.2.3	Communication . . . . .	24
3.3	User Interface . . . . .	30
<b>4</b>	<b>Conclusion and Outlook</b>	<b>35</b>
	<b>Bibliography</b>	<b>40</b>





# Chapter 1

## Introduction

Many modern computer users have multiple devices and they would like to keep the most up-to-date versions of their documents on all of them. It is very common that they will try to solve this problem by simply copying their files manually, using only the functionalities provided by the operation system. This approach is labor-intensive and error-prone.

It is obvious that outsourcing the task to a computer program would have many benefits.

There are for example tools which help the user to get the copying right, by avoiding writing over newer files with older ones (using file modification dates). One known tool, originally developed for UNIX systems, is `rsync` [1]. There are problems associated with these kind of tools. For example, when a user makes different changes to the same file on multiple machines without synchronization. Relying only modification dates will lose one change for the other. Another problem is the renaming of files on UNIX systems, which does not change the modification date of the file itself but of its parent directory.

To avoid such problems, the user should make use of a file synchronizer. Examples for such programs include Unison [2], Rumor [3] and Tra [4]. Compared to simple file copy programs, like the one described above, they keep additional information about the files. They then use this data to make better decisions regarding the preference of file versions. Usually, such a program will also recognize when no autonomous decision should be made. This situation is called a *conflict* and will typically be solved by the user.

There has been innovation in the field lately: So called “cloud storage and backup solutions” allow user-friendly file synchronization using commercially run central servers. Some of the most widely known services include Dropbox<sup>1</sup>, iCloud<sup>2</sup> and Google Drive<sup>3</sup>. Unfortunately, not a lot is known about the algorithms involved, which makes a scientific discussion impossible. Further, there are fundamental problems associated with these kind of services, concerning for example file ownership, protection, dependence on network connectivity and privacy.

---

<sup>1</sup><http://www.dropbox.com/>

<sup>2</sup><http://www.icloud.com/>

<sup>3</sup><http://drive.google.com>

We developed a new peer-to-peer file synchronizer called “McSync”, which allows the user to synchronize independent of network connectivity, with full user control over synchronization patterns and complete independence from third parties. McSync introduces two fundamental innovations:

1. An optimal, space-efficient synchronization algorithm
2. A novel user interface concept for the convenient handling of synchronization patterns

First, we will state the problem we have been trying to solve in section 1.1. In section 1.2 we will discuss work which has already been done in the field. Next, we will introduce our own algorithm in chapter 2. The inner workings of our implementation of said algorithm, the file synchronizer McSync, will be explained in chapter 3.

## 1.1 Problem Statement

The exact definition of what a file synchronizer does and what correct synchronization must accomplish is a topic of ongoing research. According to Balasubramaniam and Pierce [5] the overall goal of a file synchronizer is to “detect conflicting updates and propagate non-conflicting updates”. They specify formal requirements for a file synchronizer and they derive a general algorithm from those requirements. This algorithm was later implemented in the Unison file synchronizer [6] [2].

The approach of Balasubramaniam and Pierce [5] is based on definitions of the file system states before and after synchronization. Ramsey and Csirmaz [6] argue that “this state-based approach leads to an unnecessarily narrow view of conflicts” and further “Balasubramaniam and Pierce (1998) actually build the conflict-resolution policy into the specification, making it unclear how to implement an interesting class of conflict-resolution policies.”. They propose an algebraic approach to file synchronization and formulate synchronization based on the operations performed at each replica instead of the respective state. They conclude that “by reasoning about an algebra of operations instead of states, we have shown that there can be family of specifications for file synchronizers, each of which could be considered correct.”

With no intention of taking part in this fundamental discussion about what file synchronization is supposed to do, we will now state the properties we designed our synchronizer to have:

- *Peer-to-peer and multi-way*

As already mentioned in the introduction, our file synchronizer is a peer-to-peer system which does not rely on a central server for operation. The reason for this design choice is, that in today's world most users have more than two devices they would like to store their data on (devices may include for example USB-Sticks), but they do not have or want to have a server constantly running to allow synchronization.

Further, the devices may not necessarily be available at the same time. It is therefore desirable to allow partial (multi-way) synchronization with only a subset of the currently available devices.

- *Optimistic*

We should ask ourselves how we want to handle updates to multiple copies of the same file. Computer networks, for example the Internet, are considerably more reliable than they were a couple of decades ago. Further, availability and reliability of mobile connectivity is on the rise. Nevertheless, we always have to account for problems with the network and should therefore allow for asynchronous operation of our file synchronizer. This means that we need to be able to recognize *mutual inconsistency* or *conflicts*: independent changes to copies of the same file on different replicas thereof. Such a file synchronizer is called “optimistic” as opposed to a “conservative” one, which prevents all concurrent updates.

- *Optimal*

It is desirable to concern the user with as few conflicts as theoretically possible. This means that there should be a minimal amount of false positives: Conflicts which have actually been already resolved, but are still reported. A synchronizer which reports minimal conflicts is called “optimal”.

- *Non-immediate*

We want to give the user full control over which files should be synchronized when. As opposed to immediate (online) file synchronization, this approach is also less resource intensive. It does not need constant monitoring of the file system. Further, the synchronizer does not need to run as a daemon or to be triggered automatically on file system operations, allowing operation by non-privileged users.

The problem we have been trying to solve thus is: How can we build a file synchronizer with the properties listed above? For the algorithmic part, this translates into the problem of optimally detecting conflicts and propagating non-conflicting updates in distributed settings.

## 1.2 Distributed Conflict Detection

Early work on the distributed conflict detection problem was done by Parker *et al.* [7]. Under the assumption that nothing is known about the file semantics, they recognize two types of possible conflicts:

### 1. Name Conflicts

File names are not a good identifier in a distributed system. The reason is that the creation and renaming of files on multiple devices might lead to situations where files have the same name, but are unrelated. As a solution, Parker *et al.* propose so called “origin points”, which are used as a unique identifier for files. As an example, on OS X systems we could use the inode and btime as origin points (assuming no changes to the system clock during the life time of the system and sufficient time resolution). If we now have two files, they are derivation of a common ancestor if and only if they have a common origin point. When two replicas have a the same name but different origin points, a “name conflict” has occurred. This means that the files have been created independently and a special form of conflict-resolution is required.

### 2. Version Conflicts

A version conflict occurs when two versions of a file (with the same origin point) have been modified in an incompatible way. In order to detect such conflicts, Parker *et al.* [7] introduce a “modification id”, a unique identifier for each change of a file for each device and time relative to that device. A “modification history” for a version of a file is the set of modification ids corresponding to the modifications of that version of the file which have occurred. Two modification histories are called “compatible” if they are the same or if one is an “initial history” of the other, and are “incompatible” otherwise. A version conflict occurs when two versions of the same file (with the same origin point) have “incompatible modification histories” [7].

In order to formalize and illustrate the version conflict detection problem, Parker *et al.* [7] introduce the concept of the “partition graph”, a directed acyclic graph defined as  $G(f)$  for any file  $f$ . Such a partition graph is shown in figure 1.1. The nodes represent versions of a file. The letters inside the nodes each represent a device. All devices which have the same version (or mirror) of a file, at a given point in time, are shown together and inside of the same node. The directed edges, or arrows, indicate that the two connected nodes share a partition (the arrows are pointing away from the ancestor). The letter “M” next to a node means that this partition was modified. The letters “U” and “X”, shown on top of a node, stand for the type of update which had to be made when recombining multiple partitions into said node. This is either an update without conflict (“U”) or a conflicting one (“X”). Please note again that a partition graph is always just considering the versions of a single file. We can see that at the beginning all devices have the same version of a file. The file is then modified in the AB partition (a time when only A and B were kept in sync). Then, B and C are synced, which propagates the AB-partition to C, creating a BC partition, which is modified again. In the meantime the AB version on A was modified again, creating a new partition A. The BC change then propagated to D, creating BCD. When we try to

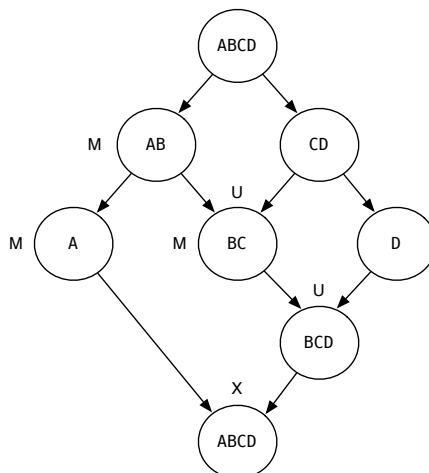


Figure 1.1: Example of a partition graph  $G(f)$  for a file  $f$ , redundantly stored on four different devices A, B, C and D. It has been directly taken from Parker *et al.* [7], but using the refined notation as proposed by Cox *et al.* [4], and our own deviation thereof.

synchronize A with BCD we get a conflict, because of earlier independent modifications (the one made to the A and the other one made to the BC partitions).

Parker *et al.* [7], define the conditions required for a version conflict as follows:

Let  $P$  be a node in  $G(f)$ . A version conflict had to be reconciled at  $P$  if there are backwards paths from  $P$  to distinct nodes  $P1$  and  $P2$  in  $G(f)$ , such that:

1. an update to  $f$  and/or a version conflict reconciliation for  $f$  occurred at both  $P1$  and  $P2$ , and
2. there is no ancestor node of  $P$  having two backward paths to both  $P1$  and  $P2$

It is simple to design a mechanism which detects all possible version conflicts (marked with “U” and “X” in the figures). The difficulty is to only detect real conflicts, denoted with “X”. To solve this problem, Parker *et al.* [7] propose the concept of *version vectors*. For each copy of each file such a vector is kept. When partitions are merged, these vectors are compared and if they turn out to be *incompatible*, a conflict is marked. They define version vectors formally as following:

A version vector for a file  $f$  is a sequence of  $n$  pairs, where  $n$  is the number of sites at which  $f$  is stored. The  $i$ th part ( $S_i : v_i$ ) gives the index of the latest version of  $f$  made at site  $S_i$ . In other words, the  $i$ th vector entry counts the number  $v_i$  of updates to  $f$  made at site  $S_i$ . We will use letters A, B, C, etc. . . to designate site names, and vectors will be written as  $\langle A:9, B:7, C:22, D:3 \rangle$

A set of vectors are *compatible* if one vector is greater than or equal to in every respective component of the others. Otherwise, they are *incompatible* and thus in *conflict*.

For example the *version vectors*  $\langle A:3, B:0, C:1, D:8 \rangle$  and  $\langle A:3, B:1, C:1, D:10 \rangle$  are *compatible*. The second one *dominates* the first one.

The *version vectors*  $\langle A:2, B:1, C:2, D:0 \rangle$  and  $\langle A:1, B:2, C:2, D:0 \rangle$  are in conflict, as neither dominates the other.

Every time there is an update to a file, which originates at a site  $S_i$ , the  $S_i$ th component of said file's version vector is incremented.

Figure 1.2 augments figure 1.1 with version vectors. We can see that at the beginning all components are zero. Then the file was modified twice at site A, while it was in sync with site B. This partition is then synchronized back with the C partition. There is a non-conflicting update, because AB's vectors dominates C's. The same is true, when the newly created and modified (note, that the C counter was incremented) BC partition is merged with the D partition. When the BCD is merged with a modified A, there is a conflict: Neither version vector dominates the other.

Unfortunately, file synchronization with *version vectors* is not optimal. Consider the situation shown in figure 1.3. We modified the file on A, while it was in sync with B, and independently on C, while it was in sync with D. Then we synchronize B and C (The BC-node). There is a *version conflict*, as both files had independent changes made to them. In this case, the user has decided to prefer the version provided by B. In the meantime A and D were also changed. Now, we want to synchronize A and B, as well as C and D. Using version vectors, two *conflicting updates* will be shown: No version vector dominates the respective other. An optimal algorithm would detect only a *conflicting update* between C and D, because the user has already chosen B over C in the earlier conflicts, which means that A's change should be accepted as preferable over B. Remember, B is transitively just an earlier version of A.

Cox *et al.* [4] solve this problem by the introduction of *vector time pairs*. In addition to the modification times, they propose keeping a vector of *synchronization times*. The latter keeps track of the synchronization events in the life of a file, or as they put it: "Informally, the modification time tracks 'which version we have,' while the synchronization time tracks 'how much we know' about a file " [4] [8]. Please note, that their approach is based on one-way synchronization only.

In order to figure out which of two versions is preferable, they compare the modification vector of one version with the synchronization vector of the other. The comparison between modifications and synchronization vectors is equivalent to the version vector comparison defined above. Consider two versions of a file  $F_A$  and  $F_B$ , on two devices A and B, with modification vectors  $m_A$  and  $m_B$ , and synchronization vectors  $s_A$  and  $s_B$ . We are synchronizing in the direction of A to B.

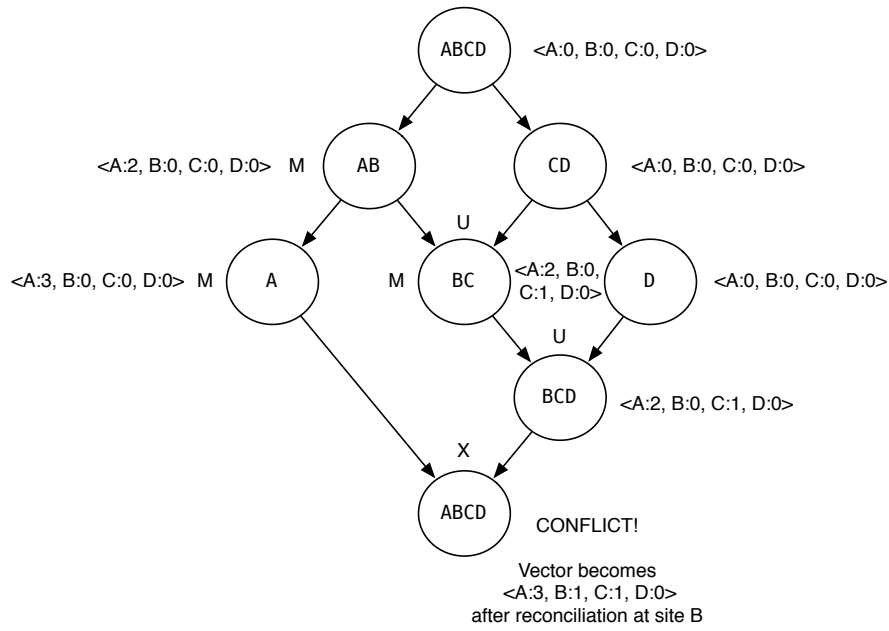


Figure 1.2: Example of a partition graph  $G(f)$  for a file  $f$  with version vectors effective at the end of each partition. It has been taken from Parker *et al.* [7], but is shown using our notation.

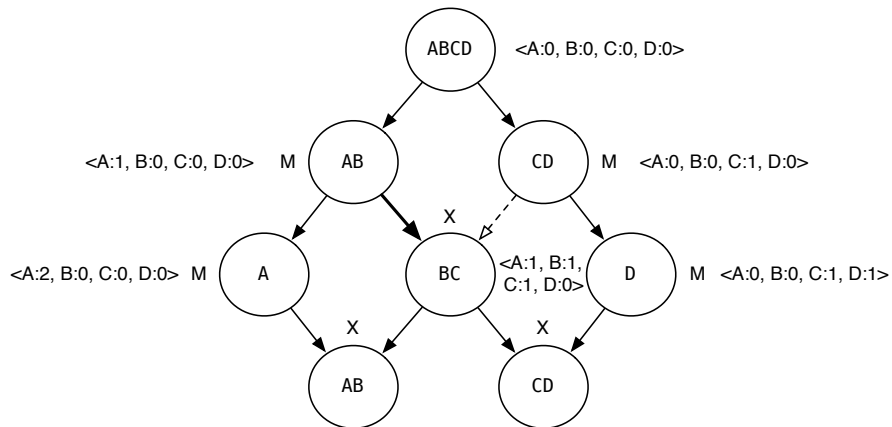


Figure 1.3: Example of a false positive version conflict. It was caused by the missing propagation of conflict-resolutions in systems using only version vectors. This figure has been taken from Cox *et al.* [4], but is shown using our notation.

The following algorithm determines what action needs to be taken [8]:

- If  $m_A \leq s_B$  then  $F_B$  is equal to or derived from  $F_A$ . B knows about all of A's updates, so we do not have to copy anything. B's synchronization vector is set to the element-wise maximum of both  $s_A$  and  $s_B$ .
- Else, if  $m_B \leq s_A$  then  $F_A$  is derived from  $F_B$ . A knows about all of B's updates and B does not know about some of A's updates. We need to copy  $F_A$  to B.
- If neither is the case, report a conflict. A does not know about some of B's updates and B does not know about some of A's updates. We can not make an autonomous decision without losing updates (the user has to specify which changes are preferable).

If there is a conflict, the user may chose one version over the other. This simply means replacing the modification vector of the unwanted version with the one from the preferable version. Like in the non-conflicting synchronization, the synchronization vector is set to the element-wise maximum of all versions involved in the sync. There is the possibility to merge both versions, by incrementing B's clock and then setting the synchronization vector to the element-wise maximum of the new entry and A's.

Figure 1.4 shows a similar situation as figure 1.3 using a different notation. The reason for the introduction of this new notation is that partition graphs are not as informative when considering directional synchronization. They were designed for describing distributed file systems with partially failing network connectivity (as opposed to non-immediate file synchronizers). Using this figure, it is easy to show that vector time pairs do propagate conflict-resolutions.

We can see that the user made two changes to the file on A and C. The respective modification vectors are updated, the device-times are incremented and the synchronization vectors adapted. Then, the user synchronizes  $A \rightarrow B$  as well as  $C \rightarrow D$ . We can see that in both cases, there is a non-conflicting update: A's version will be copied to B, C's version will be copied to D. B and D each adapted the modification vector from A or B respectively. Their synchronization vector was set to the the element-wise maximum of the synchronization vectors and those of A or C respectively.

The user then changes the file again on A at  $t_3$ .

At time  $t_4$ , the user wants to synchronize  $B \rightarrow C$ . Because B's modification vector does not dominate C's synchronization vector and C's modification vector does not dominate B's synchronization vector, there is a conflict. The user choses version B as preferable. The result is that C's modification vector is replaced with B's and both update their synchronization vectors to the element-wise maximum, in this case:  $\{A \mapsto 1, C \mapsto 1\}$ .

When the user now synchronizes  $C \rightarrow D$ , there is no conflict. Using the synchronization vector we can tell that even while C is having A's value, D is on older version of C (D's modification is contained in C's synchronization vector). The same is true for the synchronization  $A \rightarrow B$ .

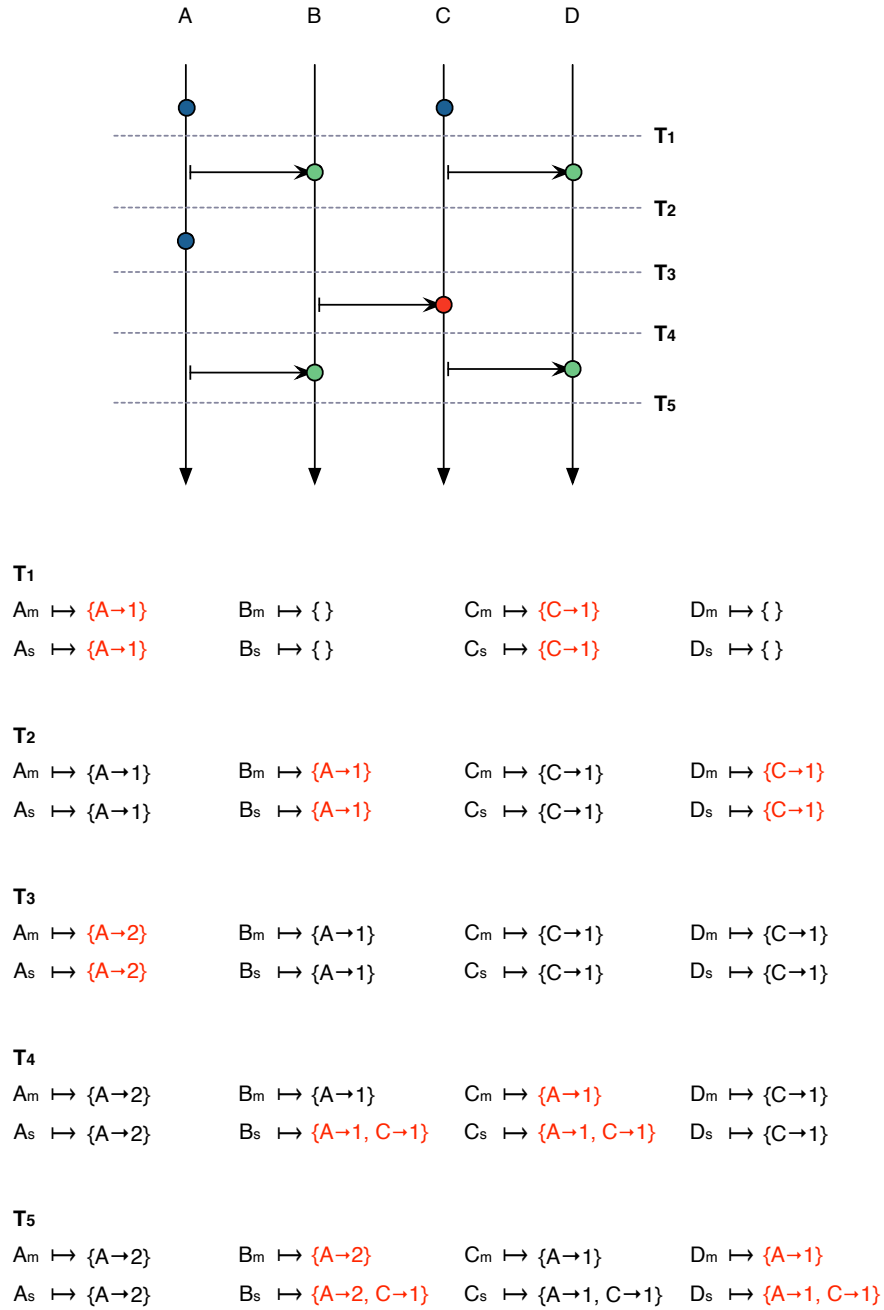


Figure 1.4: Synchronization graph for a file redundantly stored on four devices A, B, C and D. The vertical arrows represent increasing time. Horizontal arrows represent the one-directional synchronizations that occurred. Blue dots are changes made by the user to the file at this point in time. Green dots are non-conflicting updates. Red dots are conflicting updates. The bottom part of the figure shows the vector time pairs stored on every device for specific snapshots in global time. Changes relative to the last snapshot are red.

If we would synchronize the other way round ( $B \rightarrow A$ , or  $D \rightarrow C$  respectively), nothing would be copied. There are no false positives in this scenario anymore.

We now ask ourselves whether the restriction to unidirectional synchronization is necessary and further, whether the “vector-time pair” data structure presented by Cox *et al.* [4] [8] is the most space-efficient option for an optimal synchronization algorithm (solving the conflict-resolution propagation problem described above).

### 1.3 Our Solution

We developed a novel data structure for optimal distributed conflict-resolution. It is more space-efficient than vector time pairs and it is not restricted to unidirectional synchronization (it is multi-way). We call this data structure a *distributed version list*. Instead of storing two vectors on each device for synchronization times and modification times, we only store modification times together with a single bit called *is-same*. The preferable version or a conflict are determined with the help of a voting algorithm. The data structure, and the voting algorithm are explained in section 2.1.11.

Further, we decrease the number of possible conflicts by introducing *file aspects*. Instead of keeping *distributed version lists* for every file, we divide a file into several aspects such as its name, modification time or content (in this case, by just using a fingerprint of the file’s contents). This means that there is no conflict if (for example) the user independently changes only the permissions one device and the contents of a file on a different one.

A different problem of peer-to-peer file synchronizers is related to human computer interaction. The authors of Tra [9] have stated that it is difficult to design comprehensible user interfaces for partial synchronization and arbitrary synchronization patterns. As a solution to this problem, we introduce the concept of a *virtual file tree*. It gives an illusion of a single file system while in reality it is distributed amongst many of the user’s devices.





## Chapter 2

# Algorithm

In this section we will first introduce and define all the elements involved in our algorithm and then show how all the parts play together in the summary. The information presented in this chapter is based on an unpublished technical report written by Matthew Cook (the supervisor of this thesis).

### 2.1 Definitions

#### 2.1.1 Machines

Physical *machines* with CPU's are not central to the algorithm. Nevertheless, they should be defined here because it is important to differentiate them from *storage devices*.

They have three uses:

1. They have CPUs and are capable of running the synchronizer.
2. They can access data on storage devices.
3. They can communicate with other machines at remote locations.

All the data needed for the synchronization algorithm is kept on storage devices. Machines, as long as they are capable of the tasks described above, are interchangeable and not important to the workings of the synchronizer.

#### 2.1.2 Storage Devices

A *storage device* is capable of storing information and is accessible through the normal file system of the operation system. For example this could be a USB stick, a drive with multiple physical storage locations, a computer with one or more hard drives or a machine with multiple physical storage locations mounted.

A storage device has the following information associated with it:

- Its identifier
- Its device-time
- Its histories of tracked files

The identifier (or *device-id*) has to be unique in the setting in which the synchronizer operates. Typically one would use a large pseudo-randomly generated number.

The *device-time* is an increasing integer used in the history and scans. It is the devices own notion of time and is crucial to the working of the algorithm. It will be explained in further detail in the coming subsections.

The *histories of tracked files* contain meta-data on files stored on the devices. They will be defined and explained in subsection 2.1.5.

### 2.1.3 Files and Aspects

A file<sup>1</sup> is no more than a collection of *aspects* like the name, owner, modification time, etc. which can be stored as key-value pairs.

The content of a file is also considered just a regular aspect. To make handling of these unlimited-size elements easier, a finger print is recorded instead of the real contents.

There can be an arbitrary number of aspects and they can be adapted to the underlaying operation system. On UNIX systems, one could consider recording extended attributes (as returned from the `getxattr` function) as additional aspects.

The contents of a directory, i.e. on UNIX systems the list of names and inode pointers, are each treated as aspects of the corresponding plain file, rather than as aspects of the directory itself. Otherwise, there are pointless conflicts when two machines make changes in the same directory. So the directory itself doesn't have any "contents" aspect. It has most other aspects (permissions, ownership, modification dates, etc. . .).

We have three special values for aspects:

- *missing*: Data is deliberately absent, even a complete replica of the file will have nothing stored for this
- *not-visible*: Data may well be there, but cannot be read (a permissions problem, or a mounting problem)
- *unstable*: Data cannot be written (device doesn't store such data, or it would lie in an inaccessible area)

---

<sup>1</sup>Please note that we do not differentiate between directories and files when referring to a "file".

For the algorithm there is no difference between *not-visible* and *unstorable* as it is simply blocked from the data (we cannot read and/or write for whatever reason). For the user and what he/she will do about it there is a difference and thus it is recorded accordingly.

The *missing* value acts like a normal value is and synchronized and compared with other values. Files which have all aspects missing are considered deleted. They are kept as *ghosts* for a while, as if they were “present”, to allow deletions to propagate properly.

#### 2.1.4 Scans

A file or directory can be scanned at any time. This simply means looking at its aspects, and generating a “snapshot” record, which is a timestamped snapshot of every aspect of the file except the large ones, which get a fingerprint. This “snapshot” of all file aspects is called a scan.

If a directory is for some reason set to be unreadable, the unreachable file aspects will be noted as *not-visible*.

The purpose of a scan is to be merged into a history, which means matching up the files as they were present on a device at a previous time with the ones found by the latest scan.

Every scan increments the device-time.

#### 2.1.5 Histories and Version Lists

Each storage device maintains a *history* for each of its tracked files (a file which was set by the user to be synchronized at one point in the past).

The history of a file has for each of its aspects a list of mirrors: The *version list*. These are other files this file has ever had the aspect synchronized with. The list includes the file itself (they are “synchronized” with themselves with every scan).

The version list holds the most recent time that it knows about that the respective aspect changed for the mirror, and whether the current aspect on the local file is untouched since then.

An item in such a version list is defined to hold the following information:

- Device-id
- File tracking number
- Latest-change time
- Is-same flag

Each storage device has its own *device-id* and a file tracking number system. Together these two values make a global identifier for a tracked file.

*Latest-change* is the birthday (first scan time on that device when the new value was noticed) of the latest value of this aspect that we know about on that device). This time is specific to the device only and is increased upon scans and synchronization. The mechanism is similar to the well-known concept of *version vectors* [7].

The *is-same* flag states whether the aspect has changed (even if it changed back) since the time stated in the latest-change field. Any value aspect that a history knows about is always one that is strictly previous to (has been updated to) the current aspect.

Histories do not keep track of alternate versions (which might after all not be the same conceptual file anymore in the user's mind). They keep track of the history that is prior to the present version. They only store the most recent ancestor version on each machine, and even then only the time of adoption of that ancestor on that machine, and whether it is up-to-date or not from this machine's point of view.

An example for a history of a file with tracking number 72, on a device with id 2, can be seen in figure 2.1.

### 2.1.6 Identification

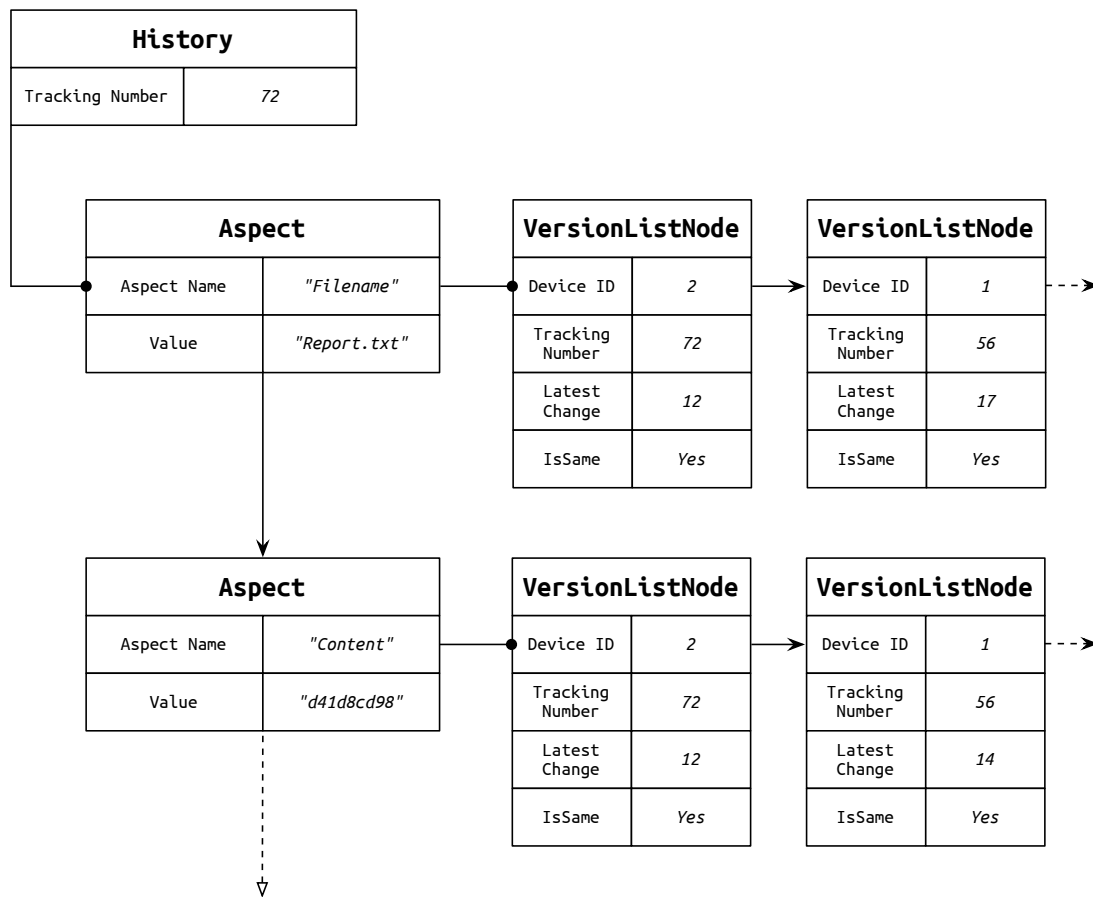
Recording more history for a tracked file (due to changes found by a scan) depends on knowing which file in the history corresponds to which file in the scan.

A history is for a tracked file (with a file tracking number) on a device. Scans present snapshots of files currently present on the device. Files may be moved around and arbitrarily renamed by the user or some program. When we see a snapshot from a new scan, we have to figure out what tracked file (and thus implicitly what previous actual file) it corresponds to. A variety of heuristics can be used for *identification*, which is simply the information about which scanned file is a continuation of which file in the history.

When doing a scan, a list of all currently present files is made. This may bear a close resemblance to information stored in the history. The (often straightforward, but very delicate) task at this point is to figure out which file tracking numbers on this device belong to which currently present files (from the scan). Some new file tracking numbers may need to be created, and some previously tracked files may be missing in the latest scan. The heuristics used can logically be anything, as the algorithm must be prepared for an arbitrary mapping of file tracking numbers to currently present files. In the implementation of our synchronizer for UNIX systems, we consider a scan to be a continuation candidate of a history if one of the following holds:

- The scan and the history have the same inode.
- The parent and the file names are the same.

In some cases, only the user can provide the correct mapping. For example, if a file is copied to a new location and given a new name (something users often do when they are cleaning up their files), then the user will need to indicate that this should be considered



*Identification guidance* allows the user to indicate that a certain file is the new version of a certain old file (thus setting the continuation).

*Preference guidance* is the user's indication of preferred values for aspects. (Not just names and parents, but all aspects.)

For example:

- That one version of a file is preferable to another (considering all aspects)
- That a file should be moved to a different location (file location is also just an aspect)
- That the permissions of a file should be set to 644 (arbitrarily set an aspect)

If a user specifies that an older version is preferable over a newer version, then this is considered as a local edit, changing the file back to its former state. The older version is marked as obsolete, as is the newer version. Otherwise, in later syncs, other versions equal to or deriving from the new version could overwrite the version the user preferred. This is all because the algorithm treats increasing time as synonymous with increasing *preferability* (see section 2.1.11). The user-specified version must be marked so as to be preferable to all known alternatives.

### 2.1.8 Noticing

*Noticing* is the act of updating a history with identification, guidance and a scan.

Once the identification and guidance tell us which scanned file corresponds to which history file, we can update the history. We say the history notices the information.

If the file has never been scanned before, then it gets a new file tracking number and is considered to be its only mirror and every aspect will have a version list with a single entry. The entry will contain the new file's device-id, its tracking number, the current device-time, and is-same set to *true*.

If a history does not have a corresponding scan all its aspects are considered *missing*.

If the aspect value has changed (the aspect value differs between the scan and the one stored in the history), we increment the device-time of the corresponding version list entry and we set the is-same flag of all other version list entries to *false*.

As an example for *noticing* imagine that the user changed the contents of the file from figure 2.1 on device 2. A scan from device-time 15 found a new value for the "content" aspect. Identification and guidance matched this to the history of tracked file number 72. The result of noticing can be seen in figure 2.2: The new content aspect value was incorporated into the history. Further, the entry for device 2 in the version list of the content aspect is updated to store the device-time 15 (the time the new aspect value was detected). Finally, on all other entries of the version list the is-same flag was set to *false*.

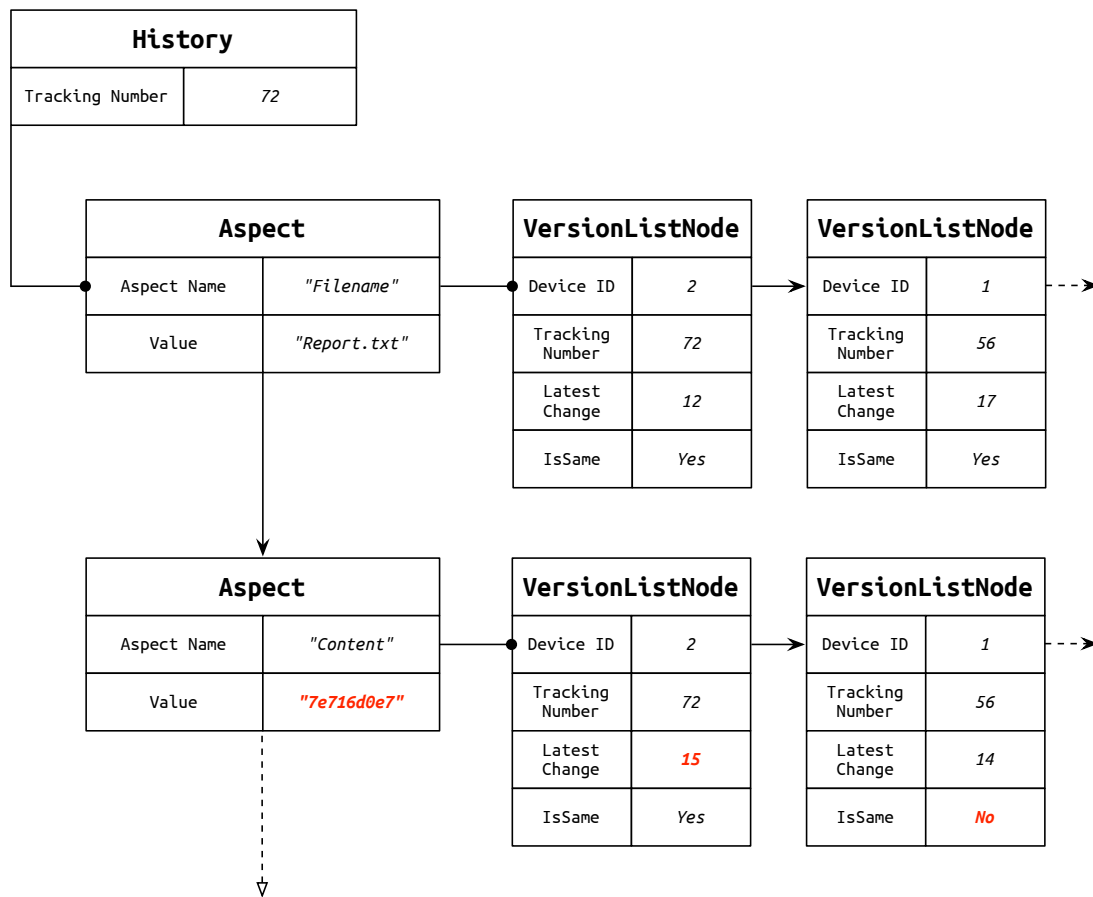


Figure 2.2: Resulting changes to the history shown in figure 2.1 after noticing a scan from the device-time 15. (The changes are marked in red.)

### 2.1.9 The Virtual Tree

The *virtual tree* is a user interface concept for helping users to keep an understanding of what file are to be synchronized with what other files.

It is a merged view of all files from all devices which are kept synchronized. Actual files which are considered “the same” are mapped to exactly one virtual file in the tree, hiding the fact that they are actually multiple files, distributed on different storage devices. It thus gives the illusion of a single file system while in reality it is distributed. The mapping of actual files from storage devices onto the virtual tree are called *grafts* and are defined in section 2.1.10.

All file manipulations by the user (e.g. preference and identification guidance) are formulated using the virtual tree abstraction.

### 2.1.10 Graft and Prune Points

A *graft* point is where an actual file path is mapped to a virtual file path.

A *prune* point belongs to a graft, and indicates a position within the subtree where the mapping ceases. Prune points may be patterns, such as “any directory containing a file ending in .svn”.

An actual file can be grafted to multiple points on the virtual tree, and multiple stored files can be grafted to the same point on the virtual tree. However, no chain of nested subdirectories, hopping back and forth between actual and virtual, may be cyclic. In other words, the result of an ideal synchronization must be finite.

An example of a virtual tree with some graft and prune points is given in figure 2.3:

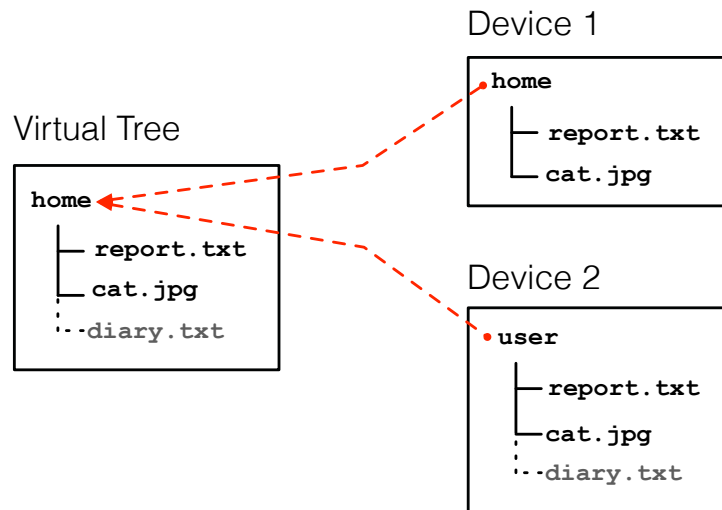


Figure 2.3: Example of grafts from two devices onto the same virtual node. The red lines represent grafts. Both, the folders “home” on device 1 and “user” on device 2, are “grafted” onto the virtual node “home”. This means their contents will be kept synchronized. The graft from device 2 has a prune point “diary.txt” specified (shown in gray). This will prevent its propagation onto devices with a graft pointing to virtual “home”. In order to keep up the illusion of a single file system the prune point is also visible in the virtual tree.

### 2.1.11 Preferability

An updated version of an aspect of an (actual) file is to be preferred over an old version. What a synchronizer does is to replace non-preferred versions of files (or in our case, file aspects) with preferred versions. When a user edits an actual file, we consider this action to demonstrate the user’s preference for the new version of the corresponding aspect of the file. Preferences are discovered by observing changes on disk (via a scan), and they can also be indicated explicitly by the user (by *preference guidance*).

After *identification* and *noticing*, we want to give suggestions to the user as to which version of a file’s aspect is preferable. We do this by comparing the *version lists* of the aspect-candidates from the different devices.

Consider two version lists  $\alpha_A$  and  $\alpha_B$ , for some aspect  $\alpha$  (e.g. “permissions”) of some file (e.g. “report.txt”) stored redundantly on two devices A and B:

$$\begin{aligned}\alpha_A &\mapsto ((A, 6, =), (B, 2, \neq), (C, 20, \neq), (D, 10, \neq)) \\ \alpha_B &\mapsto ((A, 4, =), (B, 2, =), (C, 23, =), (D, 11, \neq))\end{aligned}$$

We introduce a different notation for version lists than the one used in the figures so far: They are shown as a list of triples. The triples are the *version list entries* (or “nodes” as shown in figure 2.1). In the first field we use a single capital letter as global identifier for a file (like naming a pair of device-id and file tracking number). The middle field is the latest-change field. The last field is the is-same flag: “=” means “the same” (*true*), “ $\neq$ ” means “not the same” (*false*).

A comparison between any two version lists  $V$  and  $W$  (for example  $\alpha_A$  and  $\alpha_B$ ), can have one of the following four outcomes:

- $V$  is preferable
- $W$  is preferable
- $V$  and  $W$  are the same
- No opinion

The comparison of two version lists  $V$  and  $W$  is defined as the combined result of the element-wise comparison of their entries which belong to the same conceptual file (have the same first entry). In the example above, we would compare  $(A, 6, =)$  with  $(A, 4, =)$ ,  $(B, 2, \neq)$  with  $(B, 2, =)$ , etc...

Each such comparison “votes” on which outcome the overall version list comparison should have (one of the four listed above) by using the scheme shown in table 2.1. If there is an entry for a file which exists only in one list (no matching triple with the first entry equal to it in the other version list), then the result is always “no opinion”.

	$v_t < w_t$	$v_t = w_t$	$v_t > w_t$
$v = \wedge w =$	$W$	$\equiv$	$V$
$v = \wedge w \neq$	$W$	$W$	$\emptyset$
$v \neq \wedge w =$	$\emptyset$	$V$	$V$
$v \neq \wedge w \neq$	$\emptyset$	$\emptyset$	$\emptyset$

Table 2.1: Results of the comparison of two corresponding version list entries  $v$  and  $w$ . The top captions label the columns which belong to the result of comparing the latest-change field of the two. The left captions label the result depending on the is-same flag configuration of the involved entries, for example “ $v = \wedge w \neq$ ” means,  $v$  has is-same set to true and  $w$  has it set to false. The  $\emptyset$  stands for “no opinion”. The  $\equiv$  stands for “they are the same”.

Finally, all the votes are tallied. Any list entry comparison which voted “no opinion” is ignored, unless everybody voted “no opinion”, then this is the result of the comparison. The remaining votes must all match, or else there is an error (not a *version conflict*, this situation should not happen).

As an example, we compare the version lists  $\alpha_A$  and  $\alpha_B$  (from above):

$\alpha_A \mapsto$	( $A, 6, =$ )	( $B, 2, \neq$ )	( $C, 20, \neq$ )	( $D, 10, \neq$ )
$\alpha_B \mapsto$	( $A, 4, =$ )	( $B, 2, =$ )	( $C, 23, =$ )	( $D, 11, \neq$ )
	$\alpha_A$	$\alpha_A$	$\emptyset$	$\emptyset$

The comparisons result is that  $\alpha_A$  is the preferable version of the aspect  $\alpha$ .

Every time there is a synchronization, any number of storage devices might be involved. We pairwise compare all of them in the same way we just compared  $\alpha_A$  and  $\alpha_B$  and collect the results. It helps to think of this process as building a graph as shown in figure 2.4. We solve the problem of finding the most preferable version by using a disjoint-set data structure. In the beginning all versions are considered preferable sets. We union the sets of versions who are “the same”. Every obsolete version renders the whole set it is contained in obsolete. Comparisons resulting in “no opinion” are ignored. In the end, if we have a single preferable set - we have a non-conflicting update (the set contains the preferable version of the aspect considered). Otherwise, there is a *version conflict*.

Figure 2.6 shows how distributed version lists solve the conflict-resolution propagation problem. In the beginning, all storage devices have the same version  $V_x$  of an aspect. Then, the user changes the aspect value on A and C. If we could theoretically look at them in  $T_2$ , we would see that A and C have incremented their device-time, set their own version list entry to this time and set all other entries to “not the same”. This would of course only

happen after a *scan*, *identification* and then *noticing*, which typically only happens right before a synchronization. Next, we synchronize A with B and C with D, with the effect that there is a non-conflicting update from A to B and C to D. All devices increment their device-time. A and C just adopt the latest time of B and D respectively and set them to “the same”. All devices which are not involved in the respective synchronization are set to “not the same”. (If you want further explanations on how the new version lists is formed after each synchronization, please look at section 2.1.14.)

In the next step, the user changes the aspect value on A again, which is the same procedure described above and they synchronize B with C. The comparison yields the following result:

$$\begin{array}{cccc}
 \alpha_B \mapsto ( & (A, 4, =) & (B, 2, =) & (C, 20, \neq) & (D, 10, \neq) & ) \\
 \alpha_C \mapsto ( & (A, 3, \neq) & (B, 1, \neq) & (C, 21, =) & (D, 11, =) & ) \\
 \hline
 & \emptyset & \emptyset & \emptyset & \emptyset & 
 \end{array}$$

No version is preferable, and they are not the same: We have a *version conflict*! This is correct: The user has made independent changes to two mirrors of the aspect. The user now needs to give *preference guidance* and they choose mirror B as preferable. In the next synchronization we can see that the conflict-resolution propagation works. Let’s look at how the synchronization between C and D will lead to the result in time  $T_5$ :

$$\begin{array}{cccc}
 \alpha_C \mapsto ( & (A, 4, =) & (B, 2, =) & (C, 23, =) & (D, 11, \neq) & ) \\
 \alpha_D \mapsto ( & (A, 3, \neq) & (B, 1, \neq) & (C, 22, =) & (D, 11, =) & ) \\
 \hline
 & \emptyset & \emptyset & \alpha_C & \alpha_C & 
 \end{array}$$

The *voting algorithm* shows that  $\alpha_C$  is the preferable version of the aspect value.

In the last step, there is a synchronization involving all four storage devices. This multi-way synchronization will, after pair-wise comparison between the mirrors, lead to the graph shown in figure 2.5. We can see that A and B are the same while C and D are both obsolete.

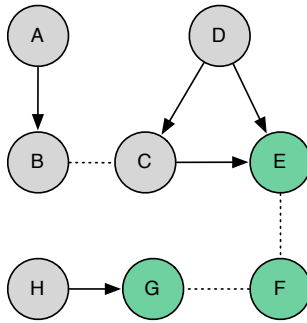


Figure 2.4: Visualization of the results of comparing version lists of the same aspect of a file redundantly stored on multiple devices. The nodes represent the mirrors. When a comparison results in “no opinion” nothing is drawn. When there is a preference, an arrow is drawn pointing from the obsolete mirror to the new one. Two nodes are connected with dotted lines when they are the same. By following the arrows, we can see that the gray nodes are obsolete, while the green nodes have the most preferable version.

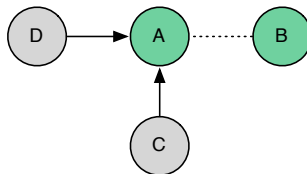
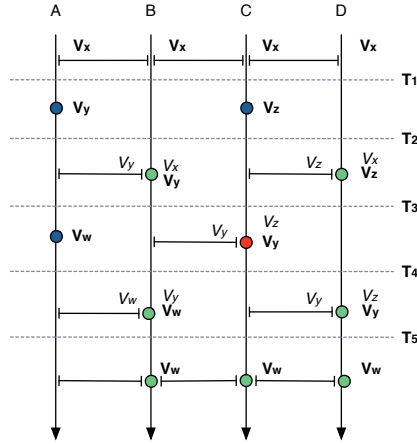


Figure 2.5: The result of the last synchronization from the scenario in figure 2.6.



**T<sub>1</sub>**

(A, 3)  $\mapsto$  ((A, 3, =), (B, 1, =), (C, 20, =), (D, 10, =))  
 (B, 1)  $\mapsto$  ((A, 3, =), (B, 1, =), (C, 20, =), (D, 10, =))  
 (C, 20)  $\mapsto$  ((A, 3, =), (B, 1, =), (C, 20, =), (D, 10, =))  
 (D, 10)  $\mapsto$  ((A, 3, =), (B, 1, =), (C, 20, =), (D, 10, =))

**T<sub>2</sub>**

(A, **4**)  $\mapsto$  ((A, **4**, =), (B, 1,  $\neq$ ), (C, 20,  $\neq$ ), (D, 10,  $\neq$ ))  
 (B, 1)  $\mapsto$  ((A, 3, =), (B, 1, =), (C, 20, =), (D, 10, =))  
 (C, **21**)  $\mapsto$  ((A, 3,  $\neq$ ), (B, 1,  $\neq$ ), (C, **21**, =), (D, 10,  $\neq$ ))  
 (D, 10)  $\mapsto$  ((A, 3, =), (B, 1, =), (C, 20, =), (D, 10, =))

**T<sub>3</sub>**

(A, **5**)  $\mapsto$  ((A, 4, =), (B, **2**, =), (C, 20,  $\neq$ ), (D, 10,  $\neq$ ))  
 (B, **2**)  $\mapsto$  ((A, **4**, =), (B, **2**, =), (C, 20,  $\neq$ ), (D, 10,  $\neq$ ))  
 (C, **22**)  $\mapsto$  ((A, 3,  $\neq$ ), (B, 1,  $\neq$ ), (C, **21**, =), (D, **11**, =))  
 (D, **11**)  $\mapsto$  ((A, 3,  $\neq$ ), (B, 1,  $\neq$ ), (C, **21**, =), (D, **11**, =))

**T<sub>4</sub>**

(A, **6**)  $\mapsto$  ((A, **6**, =), (B, 2,  $\neq$ ), (C, 20,  $\neq$ ), (D, 10,  $\neq$ ))  
 (B, **2**)  $\mapsto$  ((A, 4, =), (B, 2, =), (C, **23**, =), (D, **11**,  $\neq$ ))  
 (C, **23**)  $\mapsto$  ((A, **4**, =), (B, **2**, =), (C, **23**, =), (D, 11,  $\neq$ ))  
 (D, 11)  $\mapsto$  ((A, 3,  $\neq$ ), (B, 1,  $\neq$ ), (C, 22, =), (D, 11, =))

**T<sub>5</sub>**

(A, **7**)  $\mapsto$  ((A, 6, =), (B, **3**, =), (C, **23**,  $\neq$ ), (D, **11**,  $\neq$ ))  
 (B, **3**)  $\mapsto$  ((A, **6**, =), (B, **3**, =), (C, 23,  $\neq$ ), (D, 11,  $\neq$ ))  
 (C, **24**)  $\mapsto$  ((A, 4, =), (B, 2, =), (C, 23, =), (D, **12**, =))  
 (D, **12**)  $\mapsto$  ((A, **4**, =), (B, **2**, =), (C, **23**, =), (D, **12**, =))

Figure 2.6: Synchronization graph of the distributed conflict-resolution propagation problem. Blue dots represent user change, green dots are non-conflicting updates and red dots are conflicting updates.  $V_x, V_y, \dots$  are versions of the aspect. A bold version letter means this version was adopted, the ones printed in *italics* represent the possible version choices at this merge. The pair which maps onto the version lists contains the device-id and the device-time. All synchronizations are multi-way (in this case we have 2 and 4-way synchronizations.)

### 2.1.12 Instructions

When a comparison and/or discussion with the user have determined that a new value for an aspect is preferred over the old value, then instructions are given for an update. The instructions consist of the new value of the aspect, and a new version list. These instructions are sent to the target storage device, and hopefully directly executed.

### 2.1.13 Gossip

*Gossip* is simply the propagation and incorporation of histories. It is the only way history information is spread amongst storage devices.

The histories of two tracked files might include a common ancestor that one of them matches. If a replica feels superior or equivalent to another (current) replica that it hears about, then it can add any new information that the other replica has acquired since the fork (such as that version having propagated to other devices).

Say we have an aspect of a history, on a device with id 1 and tracking number 23, as shown in figure 2.7. We will call this aspect mirror A.

Consider now different possible mirrors of the aspect, on an additional device with id 2, shown in figure 2.8. Please note the aspect value. We will call these possible mirrors B1 to B7.

We can see that A and B1 to B7 were never synchronized, because they have no respective entry in the version list. What can be seen though is that all mirrors, including A, were synchronized with a third mirror on a device with id 3 (file tracking number 54). We will call it mirror C.

By comparing the version list node for device 3 from mirror A with the respective entry in the list of mirrors B1 to B7 we can tell their order transitively.

Mirrors B1, B2 and B3 are all preferable to mirror A. We can tell, because the version list entries for device 3 either have a higher device-time than A has for this device, or the same device-time but are not the same (is-same is “No”). From this we conclude, that mirrors B1, B2 and B3 can incorporate A’s entry into their version list, as shown in figure 2.9. Please note that the new entry has is-same set to “No”. Mirror A can’t add any of these mirrors into the version list.

Compared to B4, A is preferable, and it can incorporate B4’s entry into the version list (analogous to the history incorporation described above). B4 cannot add anything.

In the case of B5, if we have no other information, then neither A nor B5 are preferable. This is called a conflict and no histories will be exchanged.

It is impossible to have a mirror like B6 if the aspect values are not the same (note that they are not in this case). This should not happen.

The history entries of B7 and A show that they are the same. Both have the same aspect value (as opposed to B6). They can incorporate each others entries in the version list. In

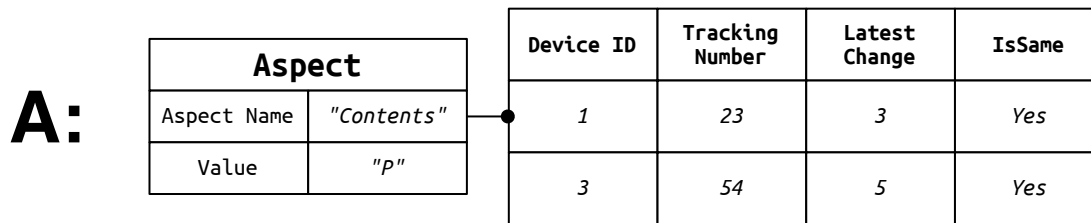


Figure 2.7: Aspect with version list from a history on device 1 with file tracking #23.

---

this case the is-same flag will be set to “Yes”, as can be seen in figure 2.9.

When not choosing the algorithmically preferred value, then the value chosen by the user is considered to be an update over the algorithm’s preferred value. We give an example of mirror A’s version list after a user has specified it as being preferable in figure 2.10.

Gossip can help to resolve conflicts. Say two devices A and B (unrelated to the examples above) are synced to one version of a file and devices C and D are synced to a conflicting version. You sync A and C, specifying that A is better. Now if you sync B and D, they don’t know that you have specified that A is better, and you will have to specify it again. But if you sync A and B first, then even though the file is fine (the same version is on A and B), A can send B the gossip that its version is better than the one adopted by C and D, and so then the sync between B and D will not exhibit a conflict.

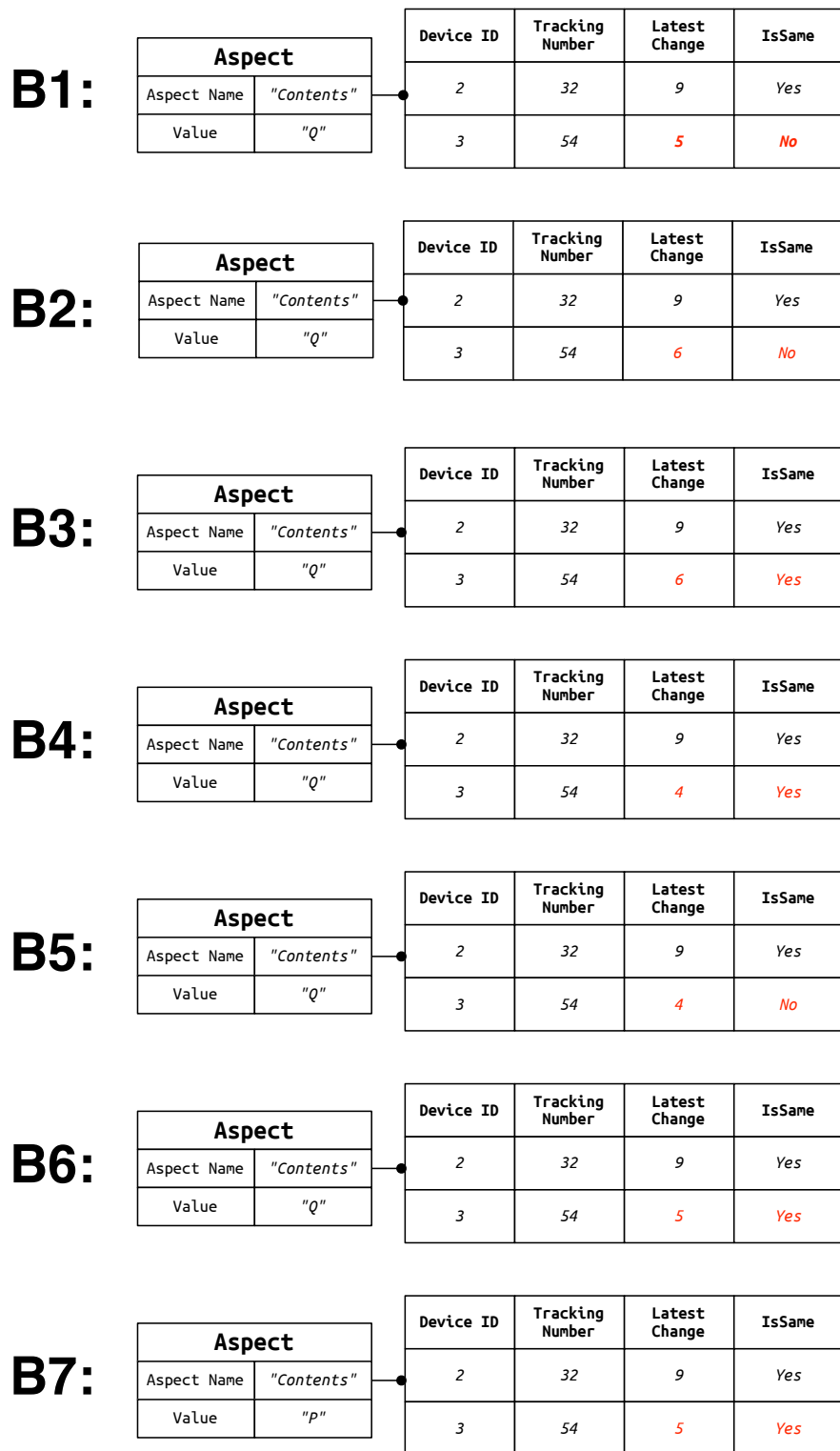


Figure 2.8: Possible version lists for a history of an aspect on device 2 with tracking #32.

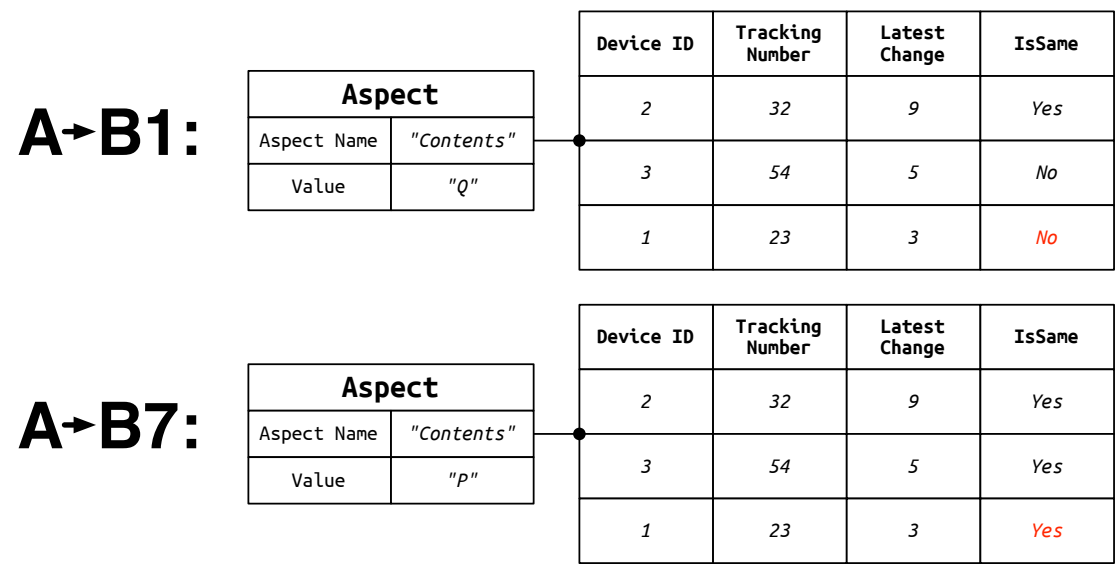


Figure 2.9: The history of mirror A from figure 2.7 incorporated into mirror B1 and mirror B7. Note the different handling of the is-same flag. B1 and B7 can be seen in figure 2.8.

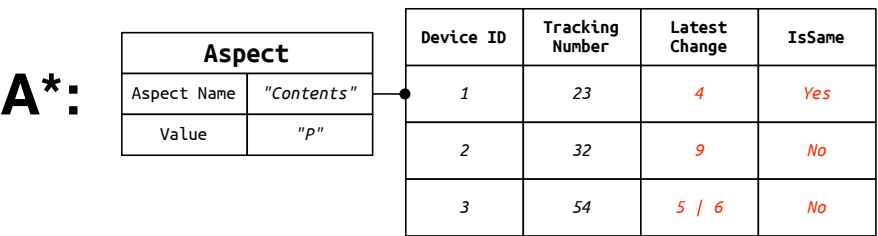


Figure 2.10: A’s version list is shown after it was chosen to be preferred over either B1, B2 or B3 (from figure 2.8). The version list is changed to contain the maximum of all entries. For device 3 this might either be 5 or 6, depending on which one of the possible mirrors of B is considered.

### 2.1.14 Synchronization

A *synchronization* consists of *instructions* being followed.

When we synchronize two devices A and B, and A was determined to have a preferable version of an aspect over B, then A's value is copied onto B.

If the update is successful, the histories are updated.

The history at B is updated by taking all of A's version list together with B's version list with everything marked to obsolete, and B itself marked as equal since now, using only the most recent record for each storage device. The history at A is updated (as soon as B has confirmed it has adapted the change) by adding any of B's mirrors that are missing from A's version list, marking them as obsolete. B itself is marked as equal since now. The resulting version list is the same at both A and B.

If B's aspect value (checked right before synchronization) no longer matches the one being overwritten (it has been changed even since the latest scan), then the update is canceled and marked unsuccessful. Further, if permissions or hardware failures block the copy, the update is unsuccessful.

We have several types of synchronizations:

- regular, non-conflicting (a newer version propagates over an older one)
- regular, non-conflicting (a newer version is the same as an older one)
- conflict, one version marked as preferred
- conflict, arbitrary value marked as preferred

In every case, when we synchronize a file aspect on a storage device, we update the version list based on the version lists of all devices considered when choosing the value. The new list will have the most recent time T shown for all list entries with the same global file version identifier (storage device-id together with file tracking number). We only set the is-same flag to *true*, if it was true in each history where it appeared with time T, and the preferred value matches this value.

On the storage device where we are changing the value (or leaving it unchanged), we mark it as same since now (or since whenever it got this value). If the user wants the new value not to be considered the same as the identical most-recently-seen value on this storage device, it can be indicated by setting its is-same set to *false*.

## 2.2 Summary

We will now show how all the parts defined in the previous section play together. An overview of what is described in this section can be seen in figure 2.11. We will assume that the user has already *grafted* some folders from different devices onto virtual nodes

using the *virtual file tree*. The first thing to happen when the users wishes to synchronize is the scan of the *grafted* folders on each device. The next step, is mapping the scans of each device to the histories of the same device, this is the *identification* step. While there is a heuristic of determining which scan belongs to which history, this process might require help from the user, in the form of *identification guidance*. As soon as the mapping is complete, the histories *notice* the changes determined by the scans: A new file may have been added, or a tracked file may have been deleted, modified or is still the same. They adapt or create the version lists accordingly. Using this information, and the intra-device mapping information given by the user by *grafting*, we algorithmically determine the preference of each aspect of every tracked file from all devices. The user may interfere using *preference guidance*, which includes the possibility of setting arbitrary values on aspects. The last step is *synchronization*, which includes generating *instructions* for the devices to follow and the sending of *gossip* on success. This two stage *synchronization* helps in case there is a problem with the instructions, so that eventual problems with instructions may be resolved.

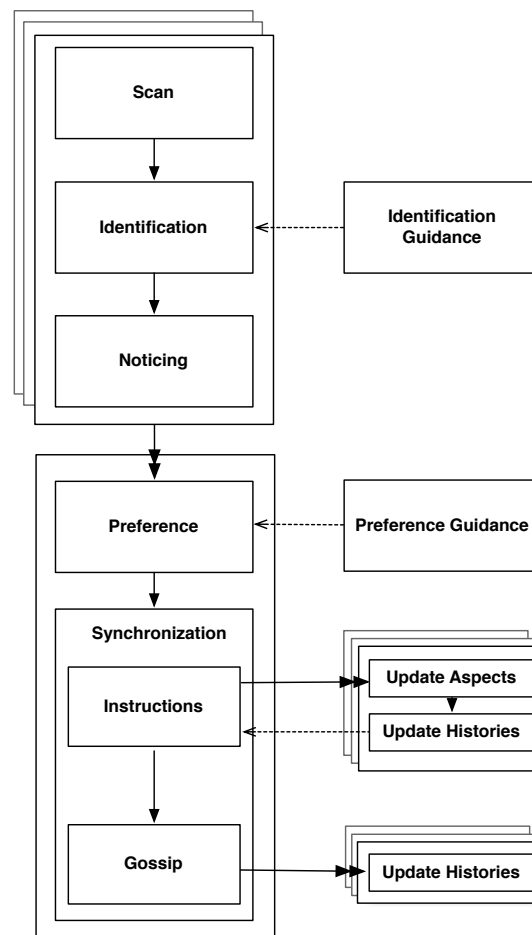


Figure 2.11: Our algorithm visualized. The stacked boxes are multiple storage devices.



## Chapter 3

# Implementation - The File Synchronizer McSync

We developed the file synchronizer McSync based on the ideas presented in chapter 3. It was implemented using ANSI C99 [10] and specifically designed for UNIX systems. In this chapter, we will introduce the challenges encountered during development, their solutions and an overview of the internal architecture of our program.

### 3.1 The Build Process

Usually, the build process is not considered part of a program. We decided that it should be possible to update McSync, by using McSync. As a consequence, the installation of McSync may only consist of copying its device application folder. Executables are system-dependent and need specific compilation. This means we need to propagate the code using McSync and integrate the build process into the regular application startup. Further, we must not use any non-standard libraries which cannot be distributed with McSync. The build process is thus important to McSync's propagation mechanism and has to be considered part of the program.

In our implementation, we provide a shell script which is the only way a user should start McSync. The script checks the modification dates of the code and availability of the corresponding system-dependent executable. If needed, it will recompile McSync before starting it. This process is transparent to the user.

McSync depends on the ncurses<sup>1</sup> library, it needs a c compiler and the SSH user-level binaries (usually found in */usr/bin/ssh* and */usr/bin/scp*). The availability of these dependencies is mostly granted in the modern UNIX world. Time pressure during implementation lead to the addition of rsync [1] as alternative to *scp* for efficient data transfers. It is planned to replace the binary dependencies with a single SSH library and implement specific data transfer algorithms for McSync.

---

<sup>1</sup><http://www.gnu.org/software/ncurses/>

## 3.2 Core Architecture

### 3.2.1 Configuration - The Device Folder

McSync stores all its operational data in a single folder: The device folder. Every device in a McSync synchronization network has such a folder stored on it. We call the path to this folder, together with an IP address pointing to the device, a “device location”.

Devices may have many locations with different IP addresses. In order to be recognized as the same device, they all have to lead to the same device folder. There is a special “IP address” called “local” which tells McSync to use local file system manipulation instead of SSH. This is more efficient, but may not be available for all devices (A device may be a local USB stick or a remote server).

We call this information on devices “the global network configuration”. It is stored in a file called “specs”. Specifically, the file contains for each device:

- Its id (generated by reading 16 bytes from `/dev/random`)
- Its nickname (a human readable identifier)
- Its graft and prune points
- Whether it is *linked*, see section 3.2.3

The configuration of McSync consists of editing this specs file, using the Textual User Interface (TUI) further explained in section 3.3. The TUI is not needed for configuration but convenient.

Other device specific configuration files, which must not be manipulated by the user, are:

- The id file (storing the unique device-id)
- The time file (storing the device-specific time)
- The histories and scans from known devices (including the device itself, stored in folders named with the corresponding device-id)

As mentioned in section 3.1 we would like to be able to update McSync using its own synchronization mechanism. For this to be possible, special treatment of the configuration files is needed.

In order to allow a global specs file, the user can specify in the TUI which address of a device to use. In this way, a device may have local and remote addresses. It is up to the user to decide how McSync should try to connect to a device. This means we can copy the specs file from one device to another one with no problems, as the user “knows” which address he/she may use.

The recreation of the other device-dependent configuration files are handled automatically and in run-time. On every data access to the id or time file, the inode and device-id are compared to respective values stored in those files themselves. If they do not match, McSync will “install” itself by initializing the device-time and id.

### 3.2.2 Actors

We built McSync with concurrency in mind and decided to implement the actor model [11]: Several threads, with specific responsibilities (different types of actors), are executed simultaneously. They communicate by sending typed messages to each other. The threads act on the types of the messages they receive, using the attached data as parameters. We have the following types of actors:

- Command (CMD)

Gives input to headquarters. It is basically an interface to the user. All configuration, such as the list of available devices or their locations, is handled by this actor. Every possible user-action, such as requesting scans, connecting to remote devices or giving guidance, is enabled by this actor.

Initially, we implemented two kinds of CMD: A command line interface (CLI) and a textual user interface (TUI). Because of time pressure, we dropped the CLI. The TUI, as described in section 3.3, is currently the only type of CMD actor available. In the future, we consider adding more types of CMD, for example a graphical user interface (GUI) or a batch mode. The latter would send predefined guidance and allow for quick synchronizations without user intervention.

- Headquarters (HQ)

HQ handles all user actions requested by CMD, either by executing them or by delegation to other actors. Its main task is to run the synchronization algorithm. For this, it communicates with workers about scans, histories and instructions. Further, HQ has an important role in setting up workers for freshly connected devices. This is done via the recruiter actor.

- Recruiter (RCR)

If CMD wishes to connect to a device it tells HQ. Each device has a dedicated worker actor. HQ thus needs to create a new worker for the requested device. This setup may require connecting to remote locations, which means we have to account for problematics of computer networks. We do not want to block or endanger HQ, the central actor, unnecessarily. For this reason, HQ outsources this task to the recruiter. Depending on the type of requested connection, RCR starts a local actor thread for the device (e.g. a local USB stick) or begins the remote peer reaching procedure. The latter involves the setup of an SSH connection to the remote peer. RCR is the only actor allowed to create new worker instances after the regular McSync startup. It does so, by having synchronized access to the underlying actor system (the routing thread, explained below).

- Worker (WKR)

Every storage device has a dedicated worker. It does file-system scans, handles the device histories and follows instructions (creating, deleting and modifying files locally on behalf of HQ). Further, it handles the device-id and time files. WKR are an abstraction, hiding the fact that a device might be remote.

We implemented the underlying actor system using the POSIX thread library [12], with a custom built in-memory message passing system. Every thread has two message queues associated with it: An incoming mailbox and an outgoing one. There is a special kind of thread, called *router* which is in charge of the message-routing. It is not an actor, as it does not receive or send messages. Its purpose is to simply copy outgoing messages to the respective incoming box of the receiving thread. Rarely, and only in special cases it does react to messages of certain types. For example, it peeks on the messages to see if they are of the *exit* type. This message is sent to actors that should halt execution. In such a case, the router will not forward any more messages to this address.

### 3.2.3 Communication

In this section we will document how McSync's functionalities are provided by the actors collaboration. The following subsections each handle a specific functionality, and describe the communication involved.

#### Connecting to Devices

At the beginning of every device action, a connection has to be established. CMD sends a *connectdevice* message to HQ with the device-id as parameter. After validation of the request (e.g. avoid connecting to already connected devices), HQ sends a request to RCR for a new connection. The message type used is *newplugplease*. RCR then stops the internal message routing mechanism and inserts the new workers in and outgoing mailboxes (bundled together in a structure called a *plug*, used as synonym for *connection*). Successful completion of said task is confirmed to HQ using the same message type (*newplugplease*), with the sent device-id as reference and unique identifier for the newly created plug.

HQ then sends a *recruitworker* message to RCR with the unique plug number and the address that the worker should be contacted at. RCR will decide, depending on the address, whether to start a local worker thread or the remote peer reaching procedure. The latter will start a SSH process (using UNIXs' *fork* and *execl* functions) and connect to the remote device. Communication with the SSH process is achieved using UNIX pipes.

Upon connection, RCR will start McSync in slave mode, which means it will operate using only a single working agent, two special parent-communication threads and a router thread (no HQ, RCR and CMD).

RCR then verifies the remote execution by exchanging a predefined message with the slave McSync. If successful, RCR sets up two dedicated local threads which simulate a local worker: The message forwarder and message receiver.

The message forwarder monitors the local outgoing mailbox: Any message added to this queue will be collected and sent over the network to the corresponding remote McSync. The message receiver listens on the connection and puts incoming messages into the outgoing queue of the corresponding agent.

Both, message forwarder and receiver, use machine independent serialization, and error checking methods, to ensure correct transmission of the messages.

Using this system, the router can just treat remote workers as if they were local agents.

On the slave McSync side, the router will forward messages targeted at locally unknown agents to the parent (master) McSync. The two special parent-communication threads mentioned above are the same message forwarding and receiving mechanisms as used on the master McSync.

If any of the actions regarding the worker recruiting is unsuccessful, RCR sends a *failedrecruit* message to HQ. After some memory clean-up, HQ requests a removal of the now unneeded plug, by sending a *removeplugplease* message to RCR. As a result, the routing is paused and the plug is removed as requested. HQ then sends a *disconnected* message to CMD with the corresponding device-id.

As soon as a worker (remote or local) was successfully started, it will try to send a *workerisup* message to HQ. The receipt of this message at HQ is considered a successful recruit. HQ will then report to CMD by sending a *connected* message (, including the device-id).

Please note that CMD and HQ always refer to devices (using the ids). HQ and RCR communicate using plug numbers. This separation of device-ids and plug numbers (connections) is important to the device identification mechanism described in section 3.2.3.

An explanatory example of McSync's internal network structure, after successful connection to a local and a remote device, is given in figure 3.1.

## Device Identification

As soon as the worker of a freshly connected device is up and running, it will send the *workerisup* message to HQ. In the distributed setting, HQ will not know which device it reached. It then replies with an *identifydevice* message, additionally sending the id it expects this device to have. The worker then opens the id file stored in its local device folder. Before reporting any found id, it will check whether the file it reads is really its own. It compares the inode of the file with the one stored in the file itself. If they differ, we consider the device file as "foreign". Further, there is a McSync magic cookie associated with all McSync files. If it is missing, something went wrong.

All errors regarding the device-id retrieval, except the case of a foreign or missing id file, are fatal. This means, the worker will stop the execution of the slave McSync after sending a final *goodbye* to HQ. Headquarters will then disconnect the device in the similar way it usually does, which is explained in section 3.2.3. Any action without proper device identification will lead to wrong synchronization and should be avoided at all cost. In case of a missing id file, the worker will create a new one. It will then contain the device-

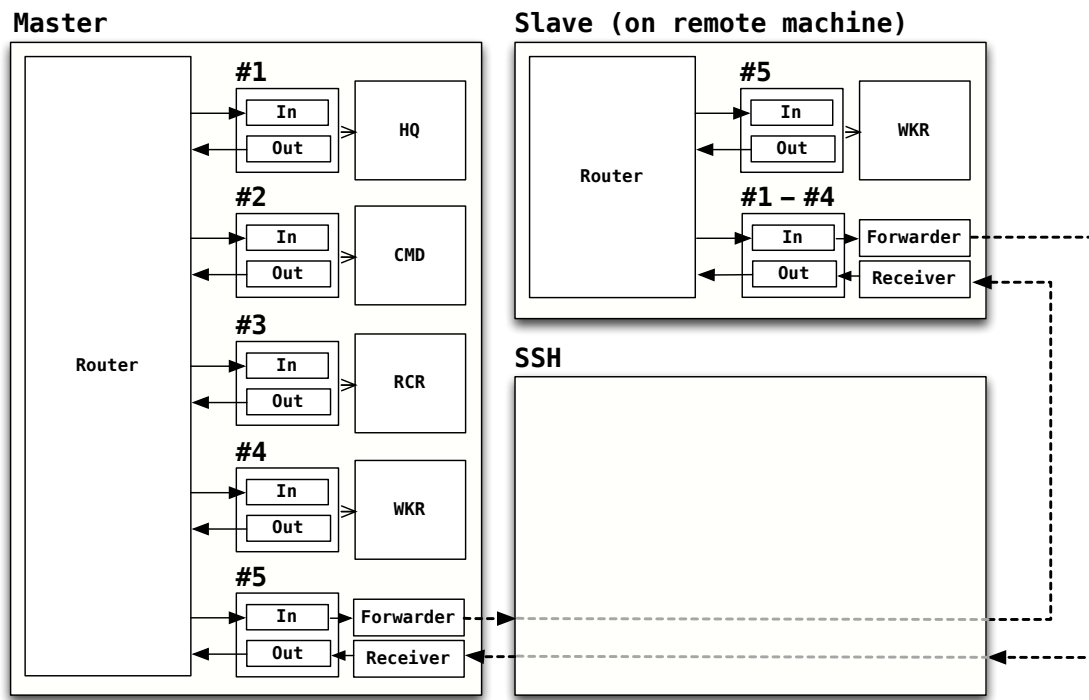


Figure 3.1: Example of a McSync network. Three UNIX processes are shown. The boxes with numbers on top of them represent plugs, containing in and outgoing message queues. The numbers are the respective plug ids. The master McSync has HQ, CMD and RCR actors running. There is a locally connected WKR (with plug id 4) and there is a connection to a remote WKR with plug id 5. The messages sent to this address will be forwarded and received through an external SSH process. The dashed arrows represent UNIX pipes. The slave has no HQ, CMD or RCR running. All messages addressed for these actors are routed back to the master McSync.

id suggestion the worker got from HQ. If a foreign id file was found, a backup is made beforehand.

As soon as the worker has successfully determined or accepted a device-id, it sends back a *deviceid* message to HQ. It includes the id HQ has sent and the local id it has found itself. If a new file was created, these two ids will be equivalent.

HQ now has the id it sent to the worker and the one it got back. It has to decide whether it has reached the correct device. The problem is: All HQ knew, when it connected to the device, was the way it can be reached (its location) and which device-id the user will expect there. What if the user connects two devices using the same location (e.g. sequentially mounting USB sticks to the same location)? What if we connect to a device we have never seen before? What if the user wants to add a new device to the network? What if he wants to add an existing one?

	$id_{sent} = id_{received}$	$id_{sent} \neq id_{received}$
<i>linked</i>	Reached intended device	Reached unknown device
$\neg$ <i>linked</i>	Added new device	Added existing device

Table 3.1: Scheme of device identification used by HQ. The compared device-ids are the one HQ read from the specs file and the one the worker has sent in. The latter was read from the respective device folder. The linked bit is also stored in said file.

A solution to these problems is storing an additional bit of information about devices, the *linked* bit. When the user adds a device using CMD, even when they mean an already existing device, it is initialized to a pseudo-randomly generated id and a false linked bit. This will help HQ to decide what to do with the device-id it gets back from a worker.

HQ checks the id it has sent to the worker against the one it gets back.

If they are the same, there are two options: Either we have reached the device we intended to or we have reached a new device. The latter can be the case when the worker can not find an id file or it finds a foreign one, and thus accepts and stores HQs id suggestion. Those two cases are differentiated using the linked bit. If the device was linked before, we have reached the correct one. If it has not been linked before, the linked bit is set to true and we have thus introduced a new device to the network. The id was already written to the device folder by the worker, thus it *named* the device with the id randomly generated back at CMD.

If they are not the same, and the linked bit is set to false, we have reached a device which already existed in some McSync network. This means HQ will change the randomly generated id to the one it got back from the worker and set the linked bit to true. In case the linked bit was true, we have reached some other unknown device and not the one the user intended to. The way we have implemented it in McSync, this unknown device will be added to the current network and set to connected. The device the user wanted to connect to will be shown as not successfully connected.

Table 3.1 gives an overview of the described decision-making scheme.

### Disconnecting Devices

In order to disconnect a device, CMD sends HQ a *disconnectdevice* message. HQ then resolves which plug was used by said device. It will request RCR to remove this plug by sending a *removeplugplease* message, including the plug number.

RCR will then send the worker of the disconnecting device an *exit* message. When the worker receives said message it replies with *goodbye*, waits a couple of seconds and eventually aborts execution of the slave McSync.

There is a problem specific to the workings of the actor system involved here: RCR has to give the worker some time to react to the message. Further, it should not wait forever.

Usually, an actor spends a lot of time in a loop checking its plug for incoming messages. When it reacts to a message, as RCR is doing when it sends the *exit* messages, it can not receive further messages. This means the time RCR spends on waiting for disconnecting devices is lost. To tackle this problem, we developed a function-callback mechanism and integrated it into our message system where needed. When an agent wants to react to a message received at a later time, it registers is with the *waitformessage* function. The arguments include which functions to call in case of the message arrival and the timeout. Time is measured in “ticks”, which is just a count of how many times we call the function *callbacktick*. In our system, the callback functions are ticked in the message receive loop. As soon a message of the stated type arrives or there is a timeout the callback will be executed. Whether it was received or not is given in an obligatory boolean argument of the callback function.

In any case, RCR will try to cancel the threads associated with the devices worker. This includes the forwarder and receiver threads of remote workers, or the agent thread itself for local worker.

There are specific problems with the POSIX threads library concerning stopping the execution of threads. Especially when they are involved in blocking I/O as they do in our case. We could not find a satisfactory solution in the given time. Currently, we send UNIX signals (SIGUSR1) to the involved threads. On receipt, they stop their execution hard (instantly, with no regard for the current code execution involved). We have arranged for special ways of storing references to heap memory used by the threads. These will be freed later, so that memory leaks are avoided. While there are no leaks, we can not avoid “hogged” memory (referenced, but not freed memory, which accumulates). When disconnecting a remote device “unflushed” I/O buffers remain in memory. This results in 16 KB “hogged” by McSync. There is no better solution at this point. All other ways of stopping POSIX threads are conflicting with our way of receiving and sending messages. This should be fixed in the future.

## Scanning

CMD communicates the virtual node, which should be scanned, to HQ using the *scan-virtualdir* message. After sanity checks (e.g. whether the received path is also a known virtual file), HQ figures out which of the grafts of the currently connected devices need to be scanned. It then sends each of the corresponding workers a *scan* request, also transmitting the list of prune points and the root scan path.

The workers first check whether they are still representing the device HQ thinks they are by rereading the id file on disk. (We have to remember, a device at any given location might change, for example if we are using USB sticks.) Next, the worker resolves any “ ” symbol in the path into the local home folder. Then it proceeds to recursively collect information on the files it can find in the path sent over by HQ (as long as they are not prune points). It stores this information in the *fileinfo* tree-like structure corresponding to the scanned directory tree. A node in this structure contains information on all file aspects, as well as the version lists and tracking number of each file.

We use fileinfos to represent both, scans and histories. Depending on the function, more fields defined on structure are used (for example the version lists for histories). To differentiate between the two, we check whether the fileinfo has a file tracking number assigned. Histories are just serialized fileinfos stored on disk. They are organized in a “one file per graft” scheme and kept in the corresponding device data folders.

The workers periodically send status updates to HQ which forwards them to CMD. Currently, the workers only communicate whether they are in the process of *scanning* or *writing*. The latter refers to the serialization and persistence required to send the scan data back to HQ. From experience we know that scan data may be too big to transfer efficiently using the actors in-memory messaging system. For this reason, the worker only sends HQ the local path of the scan and corresponding history file. This information is sent in a *scandone* message from the workers to HQ.

HQ figures out which one of its local device specific folders belongs to the worker reporting the scan. Next, it starts the download of both, scan and respective history, into said folder. For this it uses the *scp* or *rsync* programs for remote files (depending on the configuration) or the *cp* program for local ones. In case of a remote file download it will reuse the already established SSH session.

## Identification

After HQ has successfully downloaded the scan and history data from the worker, it will fold it into the virtual tree. This means, it will try to identify which scan file is the continuation of what history and how they relate to fileinfos from other devices. The identification which is done by HQ is merely stored as suggestions, we call them *continuation candidates*, on the history fileinfos. CMD will later decide, by issuing identification guidance, which of the continuation candidates is the right one.

HQ declares a scan as a continuation candidate of a history if one of the following holds:

- They have the same inode.
- They have the same parent and filename.

Fileinfos from different devices are matched with the same virtual node, if there they have the same name and virtual parent.

We implemented these matching mechanisms using two hash tables. The first one maps file names and the respective virtual parents onto virtual tree nodes. It is used for the inter-device matching of fileinfos (which fileinfos belong to the same virtual node). The second maps inodes and device-ids onto fileinfos. It is used to find continuation candidates by inode. To find continuation candidates by name, HQ uses the result which can be obtained from the first hash table: It simply checks all children of the returned virtual node for a given name and device-id. We have not noticed any impact on performance because of this naive approach, but we still consider replacing it with a third hash table in the future.

## Noticing

Noticing starts as soon as CMD sends HQ a *notice* message. It means that the process of *identification guidance* is done. HQ will compare all histories and scans. If a history has no corresponding scan, all its aspects will be set to *missing*. If there is a scan with no corresponding history, this is a new file, and HQ will initialize an empty *version list* for each of its aspects. In case of matched histories and scans, HQ will compare each aspect and if they differ, update the *version list*: The *latest change* time will be set to the current device-time and all other entries in the *version list* set to “not the same”.

## Synchronization

After *noticing*, it is possible to give aspect preference suggestions, by applying our history voting algorithm. HQ determines for each aspect a preferable value and stores it in the virtual node. The algorithm works exactly as described in section 2.1.11.

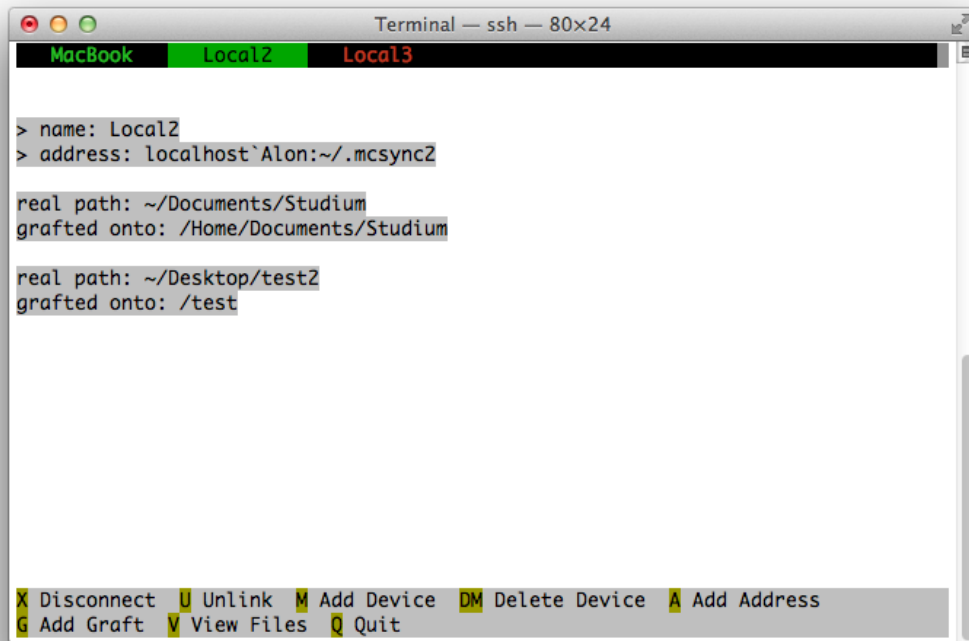
CMD now may give *preference guidance*, by acting on the virtual tree nodes and if finished, sends *sync* message to HQ. In the next step, HQ generates *instructions*, which include the new aspect value they should adopt and the new *version list* they should incorporate, and sends them to the workers. They will then try to follow the *instructions*. If successful, they will increment and adapt their local history and send HQ a *syncsuccess*. Upon receipt, HQ will generate *gossip* and send it back to the workers and forward the *syncsuccess* message to CMD, which then for example, can show it to the user.

## 3.3 User Interface

There is the possibility to implement CMD in several ways. As a first version, we introduce a CMD implemented as textual user interface (TUI), built on top of the ncurses library (see `ncurses`). We have started development of a command line interface (CLI), but had to stop because of time pressure.

For our TUI, we created two main screens: The device view and the virtual file tree view. They are shown in figure 3.2 and figure 3.3 respectively.

We further provide an example of how identification guidance can be given using our TUI, shown in figure 3.4.



```
Terminal — ssh — 80x24
MacBook Local2 Local3

> name: Local2
> address: localhost`Alon:~/mcsync2

real path: ~/Documents/Studium
grafted onto: /Home/Documents/Studium

real path: ~/Desktop/test2
grafted onto: /test

X Disconnect  U Unlink  M Add Device  DM Delete Device  A Add Address
G Add Graft  V View Files  Q Quit
```

Figure 3.2: The device view of McSync’s textual user interface. The top row represent devices. The shown id is only a “nickname” provided by the user. We introduced it, because working with the real id (a very long number) is inconvenient for humans. The algorithm only uses the numeric id. Connected devices are shown in green, disconnected devices in red. The list in the middle includes the nickname, all possible locations (shown here as “addresses”) and a list of all grafts associated with the device. Of course, the user can create, delete and edit all of these by using the commands conveniently listed at the bottom part of the screen (the “help”).

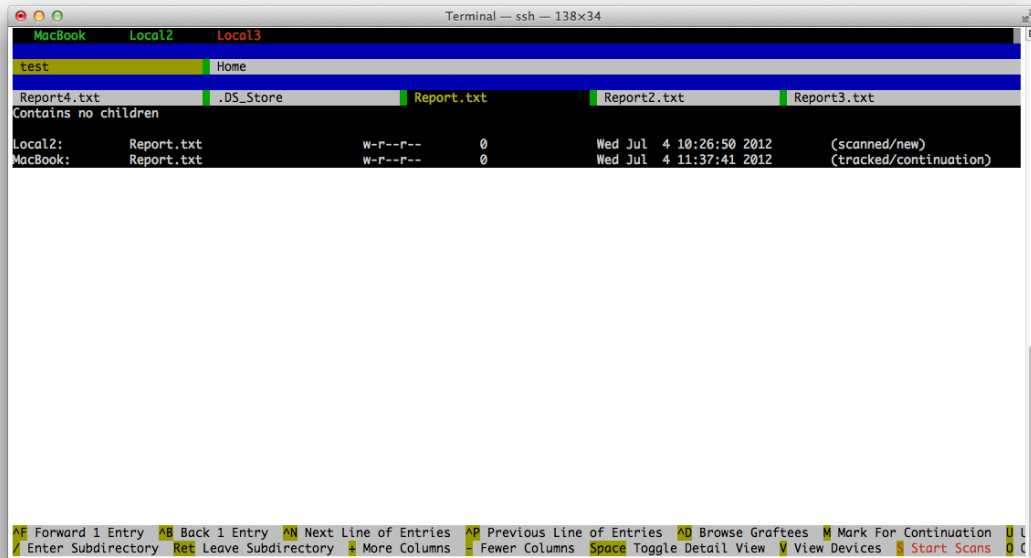


Figure 3.3: The virtual tree view of McSync’s textual user interface. Every row represents a level in the tree’s hierarchy. For the currently selected file a list of all *graffees* is shown: The files from the device currently matched to this virtual node. Specifically we can see that both, MacBook and Local2, have a file called “Report.txt”. Local2 has just seen this file for the first time. MacBook had a matching history for the scan. These are indicated by the “scanned/new” and “tracked/continuation” tags at the end of the line. Using this interface the user gives identification and preference guidance, using the specific virtual-tree shortcuts shown at the bottom in the “help”.

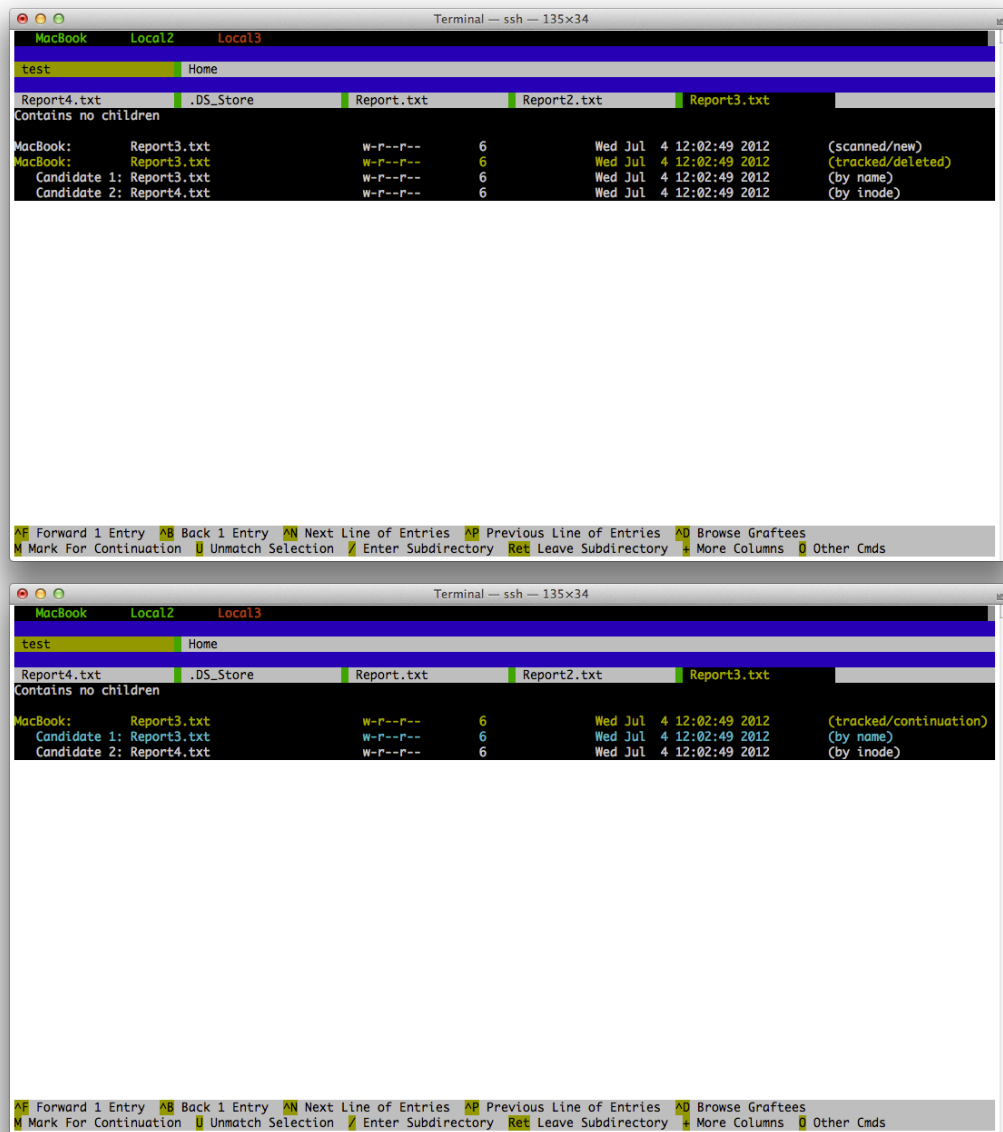


Figure 3.4: Example of Identification guidance. The file “Report3.txt” is missing on local2. MacBook has two continuation candidates for it: One scan matches the history by name and one by inode. The scan with the matching name can be seen right above the history (currently selected, highlighted in yellow). We select the scan and match it to the history. The result is shown in the bottom screenshot. The status is set to “tracked/continuation” and candidate 1 was chosen. This also means that “Report4.txt” will be considered a new file.



## Chapter 4

# Conclusion and Outlook

We have presented *distributed version vectors* as a novel solution to the distributed conflict detection problem. We have shown that it is enough to store an additional *is-same* bit for every version vector entry to fully avoid false-positive conflicts. We further introduced the *history voting algorithm* for multi-way synchronization. In order to allow arbitrary synchronization patterns, we introduced the concepts of *graft points* and the *virtual file tree*, which help the user keep an orderly understanding of what file is going to be synchronized with what other file. Two additional ideas further minimize the conflicts a user has to handle: *file aspects* and *gossip*. The introduced algorithm enables a file synchronization tool which is optimal, non-immediate, optimistic, peer-to-peer and multi-way. In the future, we hope to find additional heuristics to make *identification* better and to explore file deletion propagation in more detail.

We further showed that our ideas can be implemented in a scalable way, by developing the UNIX-specific file synchronizer McSync. The important parts of the algorithm have been integrated, but the overall user-experience should be further optimized. As an example, we would like to mention the separation of the HQ and CMD actors. Ideally, these two actors would only communicate using the messaging system. This would allow easy integration of further CMD actors, which could include graphical user interfaces (GUIs). GUIs are generally regarded as more accessible for “regular” computer users than our textual user interface. Unfortunately, because of time constraints, we had to stop this development and rely on other thread synchronization methods for larger parts of the HQ-CMD communication. In the future this should be fixed. Further, the algorithms for detecting cyclic grafts have not been implemented: A cyclic graft is problematic, because it can lead to never ending synchronization. It would also be beneficial if we could implement our own data-transfer algorithms, optimized for our data structures. Currently, we use rsync, which does a good job, but is an additional dependency which possibility could interfere with the McSync self-installation process. We could additionally consider replacing the mechanism we developed for starting and communicating with an external SSH process with a dedicated SSH library.

# List of Figures

1.1	Example of a partition graph $G(f)$ for a file $f$ , redundantly stored on four different devices A, B, C and D. It has been directly taken from Parker <i>et al.</i> [7], but using the refined notation as proposed by Cox <i>et al.</i> [4], and our own deviation thereof. . . . .	5
1.2	Example of a partition graph $G(f)$ for a file $f$ with version vectors effective at the end of each partition. It has been taken from Parker <i>et al.</i> [7], but is shown using our notation. . . . .	7
1.3	Example of a false positive version conflict. It was caused by the missing propagation of conflict-resolutions in systems using only version vectors. This figure has been taken from Cox <i>et al.</i> [4], but is shown using our notation. . . . .	7
1.4	Synchronization graph for a file redundantly stored on four devices A, B, C and D. The vertical arrows represent increasing time. Horizontal arrows represent the one-directional synchronizations that occurred. Blue dots are changes made by the user to the file at this point in time. Green dots are non-conflicting updates. Red dots are conflicting updates. The bottom part of the figure shows the vector time pairs stored on every device for specific snapshots in global time. Changes relative to the last snapshot are red. . . .	9
2.1	Example of a history of a file with tracking number 72, stored on device 2. The lines ending with filled dots describe “has a” relationships. Arrowed lines mean “links to”. (The aspect and version lists are shown as linked lists.)	5
2.2	Resulting changes to the history shown in figure 2.1 after noticing a scan from the device-time 15. (The changes are marked in red.) . . . . .	7
2.3	Example of grafts from two devices onto the same virtual node. The red lines represent grafts. Both, the folders “home” on device 1 and “user” on device 2, are “grafted” onto the virtual node “home”. This means their contents will be kept synchronized. The graft from device 2 has a prune point “diary.txt” specified (shown in gray). This will prevent its propagation onto devices with a graft pointing to virtual “home”. In order to keep up the illusion of a single file system the prune point is also visible in the virtual tree. . . . .	8

2.4	Visualization of the results of comparing version lists of the same aspect of a file redundantly stored on multiple devices. The nodes represent the mirrors. When a comparison results in “no opinion” nothing is drawn. When there is a preference, an arrow is drawn pointing from the obsolete mirror to the new one. Two nodes are connected with dotted lines when they are the same. By following the arrows, we can see that the gray nodes are obsolete, while the green nodes have the most preferable version. . . . .	12
2.5	The result of the last synchronization from the scenario in figure 2.6. . . . .	12
2.6	Synchronization graph of the distributed conflict-resolution propagation problem. Blue dots represent user change, green dots are non-conflicting updates and red dots are conflicting updates. $V_x, V_y, \dots$ are versions of the aspect. A bold version letter means this version was adopted, the ones printed in italics represent the possible version choices at this merge. The pair which maps onto the version lists contains the device-id and the device-time. All synchronizations are multi-way (in this case we have 2 and 4-way synchronizations.) . . . . .	13
2.7	Aspect with version list from a history on device 1 with file tracking #23. .	15
2.8	Possible version lists for a history of an aspect on device 2 with tracking #32.	16
2.9	The history of mirror A from figure 2.7 incorporated into mirror B1 and mirror B7. Note the different handling of the is-same flag. B1 and B7 can be seen in figure 2.8. . . . .	17
2.10	A’s version list is shown after it was chosen to be preferred over either B1, B2 or B3 (from figure 2.8). The version list is changed to contain the maximum of all entries. For device 3 this might either be 5 or 6, depending on which one of the possible mirrors of B is considered. . . . .	17
2.11	Our algorithm visualized. The stacked boxes are multiple storage devices. .	19
3.1	Example of a McSync network. Three UNIX processes are shown. The boxes with numbers on top of them represent plugs, containing in and outgoing message queues. The numbers are the respective plug ids. The master McSync has HQ, CMD and RCR actors running. There is a locally connected WKR (with plug id 4) and there is a connection to a remote WKR with plug id 5. The messages sent to this address will be forwarded and received through an external SSH process. The dashed arrows represent UNIX pipes. The slave has no HQ, CMD or RCR running. All messages addressed for these actors are routed back to the master McSync. . . . .	26

- 3.2 The device view of McSync’s textual user interface. The top row represent devices. The shown id is only a “nickname” provided by the user. We introduced it, because working with the real id (a very long number) is inconvenient for humans. The algorithm only uses the numeric id. Connected devices are shown in green, disconnected devices in red. The list in the middle includes the nickname, all possible locations (shown here as “addresses”) and a list of all grafts associated with the device. Of course, the user can create, delete and edit all of these by using the commands conveniently listed at the bottom part of the screen (the “help”). . . . . 31
- 3.3 The virtual tree view of McSync’s textual user interface. Every row represents a level in the tree’s hierarchy. For the currently selected file a list of all *grftees* is shown: The files from the device currently matched to this virtual node. Specifically we can see that both, MacBook and Local2, have a file called “Report.txt”. Local2 has just seen this file for the first time. MacBook had a matching history for the scan. These are indicated by the “scanned/new” and “tracked/continuation” tags at the end of the line. Using this interface the user gives identification and preference guidance, using the specific virtual-tree shortcuts shown at the bottom in the “help”. . . . . 32
- 3.4 Example of Identification guidance. The file “Report3.txt” is missing on local2. MacBook has two continuation candidates for it: One scan matches the history by name and one by inode. The scan with the matching name can be seen right above the history (currently selected, highlighted in yellow). We select the scan and match it to the history. The result is shown in the bottom screenshot. The status is set to “tracked/continuation” and candidate 1 was chosen. This also means that “Report4.txt” will be considered a new file. . . . . 33

# List of Tables

- 2.1 Results of the comparison of two corresponding version list entries  $v$  and  $w$ .  
The top captions label the columns which belong to the result of comparing the latest-change field of the two. The left captions label the result depending on the is-same flag configuration of the involved entries, for example “ $v_{=} \wedge w_{\neq}$ ” means,  $v$  has is-same set to true and  $w$  has it set to false. The  $\emptyset$  stands for “no opinion”. The  $\equiv$  stands for “they are the same”. . . . . 10
  
- 3.1 Scheme of device identification used by HQ. The compared device-ids are the on HQ read from the specs file and the one the worker has sent in. The latter was read from the respective device folder. The linked bit is also stored in said file. . . . . 27

# Bibliography

- [1] Andrew Tridgell. Efficient algorithms for sorting and synchronization, 2000.
- [2] Benjamin C. Pierce and Jérôme Vouillon. What's in unison? a formal specification and reference implementation of a file synchronizer. Technical report, 2004.
- [3] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. Technical report, 1998.
- [4] Russ Cox and William Josephson. File synchronization with vector time pairs. Technical Report MIT-LCS-TM-650, MIT CSAIL, 2005.
- [5] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *In Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM) '98*, 1998.
- [6] Norman Ramsey and Elöd Csirmaz. An algebraic approach to file synchronization. In *In Proceedings of the 8th European Software Engineering Conference*, pages 175–185. ACM Press, 2001.
- [7] Jr. Parker, D.S., G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, SE-9(3):240 – 247, may 1983.
- [8] Russ Cox and William Josephson. Optimistic replication using vector time pairs.
- [9] William Josephson Russ Cox. Tra, a file system synchronizer. PDOS Group Meeting, 2001.
- [10] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.
- [11] G.A. Agha. Actors: a model of concurrent computation in distributed systems. 1985.
- [12] Ingo Molnar. The native posix thread library for linux. Technical report, Tech. Rep., RedHat, Inc, 2003.