Syncpal: A simple and iterative reconciliation algorithm for file synchronizers

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Marius Alwin Shekow (M.Sc.)

aus Tettnang, Deutschland

Berichter: Univ.-Prof. Dr. rer. pol. Matthias Jarke
Univ.-Prof. Ph. D . Wolfgang Prinz
Univ.-Prof. Dr. Tom Gross

Tag der mündlichen Prüfung: 22. November, 2019

Eidesstattliche Erklärung

Ich, Marius Alwin Shekow,

erkläre hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktoranddieser Fakultät und Universitätangefertigt;
2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde diesklar angezeigt;
3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;
4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;
5. Alle wesentlichen Quellen von Unterstützung wurden benannt;
6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;
7. Teile dieser Arbeit wurden zuvor veröffentlicht und zwar in:
    - Marius Shekow. “Syncpal: A simple and iterative reconciliation algorithm for file synchronizers”. In: *Distributed Applications and Interoperable Systems - IFIP International Federation for Information Processing, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Lingby, Denmark, June 18-21, 2019, Proceedings.* Ed. by José Pereira and Laura Ricci. 2019
    - Marius Shekow and Wolfgang Prinz. “A capability analysis of groupware, cloud and desktop file systems for file synchronization”. In: *Proceedings of 17th European Conference on Computer-Supported Cooperative Work-Exploratory Papers. The International Venue on Practicecentred Computing an the Design of Cooperation Technologies. European Society for Socially Embedded Technologies (EUSSET). 2019.* DOI: 10.18420/ecscw2019_ep06. URL: `http://dx.doi.org/10.18420/ecscw2019_ep06`.

Datum

Unterschrift

Abstract

File synchronizers are tools with the goal to facilitate collaboration scenarios and data management across multiple devices. They replicate the file system, e.g. from a cloud storage to a device disk, achieving convergence by only transmitting detected changes. A popular variant available in a plethora of widely adopted products are state-based file synchronizers such as Dropbox. They detect operations by computing the difference between a previously persisted state and the respective current state, performing a bi-directional synchronization of two replicas. While users rely on synchronization to run without errors, bugs in industrial synchronizers make this difficult. Users often have to detect and fix synchronization errors themselves, and some errors remain undetected for a long time. This results in cost-intensive iterations in cooperation processes, which should be avoided in academia and industry.

This work identifies three core challenges of state-based file synchronization. The first challenge is the heterogeneity of different file systems, which requires the file synchronizer to detect and handle incompatible capabilities. Second, a synchronizer needs to detect and resolve conflicting operations that result from a group of users working on their replica in isolation. Third, non-conflicting operations computed from state differencing are not immediately suitable for propagation. The operation order is not available and operations may be affected by consolidation, such that important intermediate operations are missing. This problem most notably affects file systems that support the move operation which changes an object's parent folder. The goal of this work is to design and analyze an algorithm and develop an implementation of a file synchronizer that solves these challenges.

To address heterogeneity we analyze existing real-world implementations (such as the NTFS file system) and their compatibility issues, and also examine related academic works. We identify six file system capabilities relevant to file synchronizers, formally define a file system model $\mathscr{F}$ that is in large parts compatible to the various existing definitions and suggest several alternatives for handling incompatible differences. To detect conflicts we perform a precondition analysis of the operations of $\mathscr{F}$. Resolving conflicts is an open problem where the right approach depends on the context. Our related work analysis finds that most academic works present arbitrary resolution methods that lack a rationale for their decisions. To determine a reflected conflict resolution approach we design a four-step framework which starts with an informal definition of consistency properties and iteratively refines it to a set of formal and detailed steps for resolving concrete conflicts.

Apart from $\mathscr{F}$ and a conflict resolution approach the main contribution of this work is Syncpal, an iterative algorithm that reconciles two divergent file systems, solving all of the above challenges. It first handles conflicts, one at a time, such that resolving one conflict does not negatively affect others. Whenever possible, conflicts are avoided. It then finds a valid propagation order for the remaining non-conflicting operations, breaking cyclic dependencies if necessary. The iterative nature of Syncpal reduces the overall complexity and the probability of bugs. The technical evaluation of our implementation of Syncpal includes its complexity analysis, automated testing and a comparison with five industrial-grade file synchronizers. We find that our algorithm improves the handling of file system heterogeneity and synchronizes changes from long offline periods correctly, where other implementations fail and may even cause data loss. Our implementation has been in operation by 30 users over a period of over 18 months, providing valuable insights for further research regarding usage patterns and practical requirements.

Zusammenfassung

File Synchronizer sind Tools, die Kollaborationsszenarien und Daten-Management über mehrere Geräte hinweg vereinfachen sollen. Sie replizieren Dateisysteme, z. B. von Cloud-Speichern auf Festplatten. Dateisystem-Konvergenz wird erreicht, indem nur Änderungen übertragen werden. In der Praxis haben sich zustandsbasierte File Synchronizer wie Dropbox oder ähnliche Produkte durchgesetzt. Sie erkennen Operationen via Differenzbildung aus einem zuvor persistierten und dem aktuellen Zustand zweier Dateisystem-Replikate und erreichen dadurch deren bidirektionale Synchronisation. Benutzer sind darauf angewiesen, dass die Synchronisation fehlerfrei abläuft. Marktübliche File Synchronizer erschweren dies jedoch, sodass Benutzer Synchronisationsfehler oft selbst erkennen und unter großem Zeitaufwand beheben müssen und manche Fehler lange Zeit unerkannt bleiben. Dadurch entstehen im Kooperationsprozess kostenintensive Iterationen, die es in Wissenschaft und Wirtschaft zu vermeiden gilt.

Diese Arbeit identifiziert drei zentrale Defizite der zustandsbasierten Dateisynchronisation. Erstens bedingt die Heterogenität verschiedener Dateisysteme, dass File Synchronizer inkompatible Eigenschaften oft nicht erkennen und behandeln. Zweitens muss ein Synchronizer Konflikte erkennen und lösen, die durch den isolierten Zugriff mehrerer Benutzer auf Dateisysteme entstehen. Drittens sind konfliktfreie Operationen, die via Differenzbildung der Zustände erkannt wurden, nicht unmittelbar für die Übertragung geeignet. Hier ist die Reihenfolge der Operationen unbekannt und diese wurden evtl. konsolidiert, sodass wichtige Zwischenoperationen fehlen. Dieses Problem betrifft insbesondere Dateisysteme, die Verschiebe-Operationen unterstützen, die den Überordner eines Objekts ändern. Ziel dieser Arbeit ist es, diese Defizite durch einen neuartigen, verbesserten File Synchronizer zu beheben.

Zum besseren Verständnis der Heterogenität analysiert diese Arbeit bestehende wissenschaftliche Arbeiten und reale Dateisystem-Implementierungen wie NTFS und deren Kompatibilität. Der Autor identifiziert sechs Dateisystem-Fähigkeiten, definiert ein formales Dateisystem-Modell $\mathscr{F}$, das zu bestehenden Definitionen größtenteils kompatibel ist, und schlägt mehrere Alternativen für den Umgang mit inkompatiblen Eigenschaften vor. Zur Konflikt-Identifizierung wird eine Vorbedingungsanalyse der Operationen von $\mathscr{F}$ durchgeführt. Das Lösen von Konflikten ist diffizil, da der geeignete Ansatz vom Kontext abhängt. Ein Großteil wissenschaftlicher Arbeiten löst Konflikte willkürlich und ohne Begründung der konkreten Entscheidungen auf. Zur Findung eines reflektierten Konfliktlösungsansatzes hat der Autor ein vierstufiges Framework entwickelt, das eine informelle Definition von Konsistenzeigenschaften iterativ zu einer Menge von formal definierten, detaillierten Konfliktauflösungsschritten verfeinert.

Neben $\mathscr{F}$ und dem obigen Konfliktlösungsansatz ist der Hauptbeitrag dieser Arbeit der iterative Algorithmus *Syncpal*, der die Abweichungen zweier Dateisysteme synchronisiert und die genannten Defizite löst. Konflikte werden nacheinander behandelt, ohne dass sich die Lösung eines Konflikts negativ auf andere auswirkt. Wenn möglich werden Konflikte vermieden. Anschließend werden eine valide Übertragungsreihenfolge der verbleibenden, konfliktfreien Operationen ermittelt und eventuelle zyklische Abhängigkeiten aufgelöst. Dieser iterative Ansatz reduziert die Gesamtkomplexität und die Wahrscheinlichkeit von Programmierfehlern. Die technische Evaluation der Syncpal-Implementierung umfasst eine Komplexitätsanalyse, automatisierte Tests sowie einen Vergleich mit fünf weiteren, etablierten File Synchronizern. Die Ergebnisse zeigen, dass Syncpal die Dateisystem-Heterogenität besser bewältigt und Änderungen aus langen Offline-Phasen korrekt synchronisiert. Andere Implementierungen scheitern hier häufig, was bis hin zu Datenverlust führen kann. Die Verwendung der Syncpal-Implementierung durch 30 Benutzer über einen Zeitraum von über 18 Monaten liefert für weitere Forschungsfragen wertvolle Einblicke bzgl. Nutzerverhalten und Anforderungen aus der Praxis.

Acknowledgements

This work would not have been possible without my employer, Fraunhofer FIT, who provided the freedom to work on the thesis alongside day-to-day project work. I specifically want to thank my department heads Dr. Leif Oppermann and Prof. Wolfgang Prinz for encouraging me to turn the file synchronizer from being just project work into a PhD thesis. They enabled a PhD-friendly environment on the job by giving me the chance to work on projects that contextually overlap with the thesis. Fraunhofer FIT has also been supportive financially by sponsoring technical writing programs and paying for conferences.

Many thanks go to my thesis advisors Prof. Matthias Jarke and Prof. Wolfgang Prinz who not only helped shape this work by giving valuable input, but also considerably improved several conference papers. I'd also like to express gratitude to my colleague Martin Kretschmer for the extensive discussions, his input and proof reading of many theoretical sections of this work. I would like to thank all colleagues of the CSCW and MARS group at Fraunhofer FIT for the vivid exchange regarding file synchronization. They have been using my file synchronizer implementation for over two years and their input has been invaluable to identify and fix programming errors and to design new software features based on real user requirements.

To my parents I'm grateful for their kindness and support over the years, not only during the thesis but also any other period of my life. I finally want to thank my partner Julia for being loving, understanding and supportive at all times, reminding me that there more things to life than the thesis.

Marius Alwin Shekow
July 2019

Contents

Chapter 1

Introduction

Since decades file-based tools, such as word processors, have been a core component in the daily workflow of users of computing devices. Users create large parts of their data in the form of files, which are stored and distributed on multiple devices and storage systems in a hierarchical *file system*. Tasks such as retrieving a specific file are already challenging when working only on the local disk [RSK04]. Difficulty increases tremendously if a user wants to find the correct version of a file, when it could be located on her mobile phone, personal computer, work laptop, company cloud storage, file server, or group ware system [DP08; JOO15; SW13; Voi+06].

To avoid that users need to manually locate and copy files, one approach to ease this data management problem is to consolidate all storage systems to a single, logical namespace. Variant #1 is to *map* each storage into a local directory on the device. However, the configuration is challenging for novice users, there are security risks and not all storage systems can be assumed to be "online" at every point of time[1]. Variant #2 is to employ *replication* of the data. Programs called *file synchronizers* [BP98] keep one or more local directories synchronized with other storage systems. They automatically recognize conflicts and either automatically solve them or allow the user to choose a resolution.

With the increased availability and affordability of cloud services [Yan+16], cloud storage services like Dropbox, Google Drive and OneDrive have become popular over the last ten years, indicated by the high number of their users [Kol16; Pri17; TBM13]. Once the user installs the corresponding file synchronizer of the cloud storage provider on every computing device, it will maintain the up-to-date versions of all files on all devices, by continuously performing a bi-directional, pair-wise synchronization between each device and the cloud storage system. This also facilitates *collaborative* scenarios involving *several* users, because files are no longer located on an isolated system such as the local device storage. Instead files are transparently shared with others, thus supporting a group to develop documents in a cooperative way.

1.1 Motivation

We wrote this thesis for two reasons. First, we found that file synchronizer literature is very sparse in academia., s.t. this work can enrich the corpus, in particular in the area of CSCW and cooperative information systems. Second, we observed multiple issues (detailed below) with industrial file synchronizers. By providing up-to-date and thorough research, we hope to inspire file synchronizer developers in academia and industry alike. We envision that future file synchronizer releases make fewer intransparent decisions, cause less frustration for users, and avoid subsequent, manual repairs of directory structures and file contents. Due to the large (and still growing) user-base of these tools, their improvement does have considerable practical impact.

Some of the issues we observed first hand with existing implementations include:

[1] For instance, while it would be technically possible to map the work laptop's storage into a directory on the personal computer, that directory would be inaccessible while the work laptop is turned off.

- The synchronization process gets stuck or shows an error in an "offline scenario", when the synchronizer reconnects to the storage system after the user worked on the local file system without connectivity for an extended time period.
- Lack of documentation of conflict detection or resolution behavior. When a conflict happens it is automatically resolved, but synchronizers provide little to no visual feedback to the user. As a consequence they make intransparent decisions, because they silently alter or undo operations which the user thought were successful. Conflict resolution behavior also differs between different implementations. Academic works do not provide reflection or recommendations regarding conflict resolution.
- The synchronized file systems each have a slightly different definition (in other words: they are slightly incompatible with each other). For instance, the BSCW groupware allows directories to be linked into several parent directories, while desktop file systems (Windows, macOS) do not. There are many other issues, such as varying sets of reserved names or characters. For this reason, file synchronizers are *heterogeneous* data synchronizers [AC08; Fos+07]. Heterogeneity of file synchronizers is not discussed in academia, and industrial synchronizers handle some aspects incorrectly, causing synchronization errors.
- When a user requires synchronization with *several* storage systems, she has to install several file synchronizers, because often each synchronizer is proprietary to a specific storage system. This wastes computing resources and shows the need for a synchronizer that can abstract and synchronize different storage systems. Such tools (e.g. Goodsync or Syncovery[2]) do exist, but come with a complex user interface which is challenging to use for average users.

1.2 Goals and research questions

A heterogeneous synchronizer is typically *state-based* because alternative *operation-based* approaches require operation logs that are not ubiquitously available. The term *state-based* means that the synchronizer detects operations from comparing the current file system *state* with a persisted state of the previous synchronization. This allows the synchronizer to be operational even when shut down while the user changes the file system. However, detected operations lack information about their exact order which makes their synchronization challenging in certain situations.

The goal of this work is to build a *state-based*, heterogeneous, near real-time file synchronizer geared towards non-technical end-users. It synchronizes operations detected in two file system replicas as they happen, typically in a client-server setup. It should support several storage system implementations with *different* file system definitions, without the need to modify the storage systems themselves. Consequently, the synchronization logic is entirely run on the client. This *client-side reconciliation* concept is also found in previous works [MT94].

The research questions are as follows:

- **RQ1 - File systems:** What different kinds of file system definitions exist in academia and practice? Which criteria are relevant for file synchronizers? How should a file synchronizer's internal, abstracted file system model look like, which incompatibilities exist and how can they be handled?
- **RQ2 - Operation order:** As operations detected during state-based update detection lack order, but not all operations are commutative, how can a valid order be detected and propagated by a synchronizer? This question was already answered for file system models which do not support *move* operations, but remains open for file systems which do support them.
- **RQ3 - Conflicts:** What sets of operations applied to two disconnected file systems are conflicting? How do conflicts depend on the file system model? Can multiple conflicts be combined? What are possible solutions for resolving individual conflicts and conflict combinations? Which conflicts are relevant in practice? How can conflicts be explained to the user?

[2]See `https://www.goodsync.com` and `https://www.syncovery.com`, retrieved July 21, 2019.

1.3 Research methodology

We start with literature review of academic file synchronizer works and works of related systems, as well as manuals of real-world systems (e.g. file system APIs) to gain a deep understanding of file systems, conflicts and related algorithms, and to identify shortcomings in existing works. We use that knowledge to design our own synchronization algorithm. Because we focus on *practical* use, we implement and distribute the synchronization algorithm as an executable program to real users, immediately after designing it. This allows us to discover unexpected real-world issues and to iteratively improve the algorithm. We verify the correctness of our algorithm via theoretical proofs. We also test our *implementation* for correct behavior by a large number of hand-written and automatically generated software tests. To compare the behavior of our work with other academical and industrial implementations we perform controlled experiments.

1.4 Contributions

The main contribution is the development of a state-based reconciliation algorithm called *Syncpal*, which synchronizes two disconnected file system replicas. The algorithm was published in [She19].

Since real-world file systems are heterogeneous, we present an analysis of the heterogeneous traits of file systems in chapter 3, resulting in a unified model compatible with a large set of real-world models. We refer to [SP19] for the corresponding publication. The individual steps of the Syncpal algorithm terminate provably, and synchronization can be interrupted at any time without causing side-effects. Our approach finds a suitable order of the detected operations of the unified file system model (section 4.2.7), which also supports *move* operations, which complicates the analysis. Syncpal also features a new conflict resolution approach (chapter 5), which includes graceful handling of conflict *combinations* and avoids conflicts wherever possible. We discover a lack of guidance in academic works w.r.t. conflict handling and develop a reflected philosophy, which discusses several alternatives for resolving conflicts.

We implemented the algorithm as an executable application for the operating systems Windows and macOS. It has been used by 30 users over a period of over 18 months, which provides insights into daily life conflicts (section 8.3) and helped discover various other issues in practice (section 7.5). We implement an extensive evaluation framework in section 8.2 that benchmarks our implementation, Dropbox, Google Backup and Sync, Microsoft OneDrive and NextCloud, discovering numerous issues in all other implementations.

1.5 Thesis structure

We begin with a survey of related works in chapter 2, which unravels the *distributed systems* research topic and puts our work into context. We identify several characteristics of file synchronizers which help guide the development of our approach. With our goals from section 1.2 in mind, we then focus on *file systems* in chapter 3, where we first analyze existing implementations and their compatibility issues, followed by extracting and formally defining the file system model we use for the remainder of this work. This formal definition allows to discuss update detection for state-based synchronizers, which is done in chapter 4. Before we present the overall synchronization algorithm in chapter 6, we first take a deep dive into conflicts in chapter 5, starting from the generic concept (file system model-independent) of what conflicts are and what *achieving consistency* means, down to the concrete approaches for finding and resolving conflicts in the concrete file system model from chapter 3. In chapter 7 we present the implementation of our algorithm, with its software architecture, project background, and unforeseen issues we encountered in practice. We evaluate our algorithm on a *technical* level (not with users) in chapter 8, and conclude and present future work in chapter 9.

Chapter 2

Related Work

This chapter discusses related work to identify research gaps, find approaches to base our algorithm on, and to put our work into context. We follow a top-down approach, starting with generic concepts and narrowing them down to our specific field: file synchronizers.

Because file synchronizers are programs that achieve *consistency* in replicated, *distributed systems*, we provide an introduction to these concepts in section 2.1. We identify *optimistic replication systems* as the suitable sub-field, for which we present important characteristics and generic data synchronization approaches in section 2.2. We then examine file synchronizer works specifically in section 2.4, but also look at intricacies of their underlying data structure, file systems, in section 2.3. Finally, we mention related fields and technologies in section 2.5 and conclude in section 2.6, where we make suitable choices regarding our own work for each characteristic found in section 2.2. The research questions that result from gaps in related work are discussed in sections 2.3 (RQ1) and 2.4 (RQ2+3) respectively.

2.1 Distributed and replicated systems

From a high-level perspective, a file synchronizer is an application realizing a *distributed system*. The main characteristic of a distributed system is that its software (the middleware or applications on top of it) is installed on *networked* computers which exchange asynchronous messages (see [CKD10], p. 1). The system *appears* as a *single* system to the user, even though it is distributed. The main advantage (over a software system that runs on a single, isolated computer) is *sharing* of resources like data or computing power. On the downside, distributed systems have to deal with many challenges, including heterogeneity, openness, security, scalability, failure handling, concurrency or transparency [CKD10]. To understand how a file synchronizer fits into the research area we provide an ontology of distributed systems and relevant sub-variants shown in figure 2.1. Bold boxes highlight the domain to which the kind of file synchronizer belongs which we develop in this work.

There are various models that describe and categorize distributed systems. For instance, *architectural* models like the client-server or peer-to-peer model are *“concerned with the placement of [the system’s] parts and the relationships between them”* [CKD10, Chap 2]. Different variations of those models exist, see [CKD10, sect. 2.2.3], such as extending a “multiple client, one server” model to “multiple clients, multiple servers”.

One important variant is the use of *replication* [CKD10, sect. 15]. Here data is replicated (i.e., copied) fully or in part[1] between computers that serve them. Replication offers several advantages [SS05]: it improves availability, because data can still be accessed even when some replicas are unavailable. It also improves performance in terms of lower latency (users can work on nearby replicas) and increased throughput (better bandwidth to nearby replica, and multiple replicas can serve data simultaneously). We will further discuss replication in section 2.1.3. A very important topic in replicated distributed sys-

[1] Partial copies with limited life-span are often referred to as *cache*.

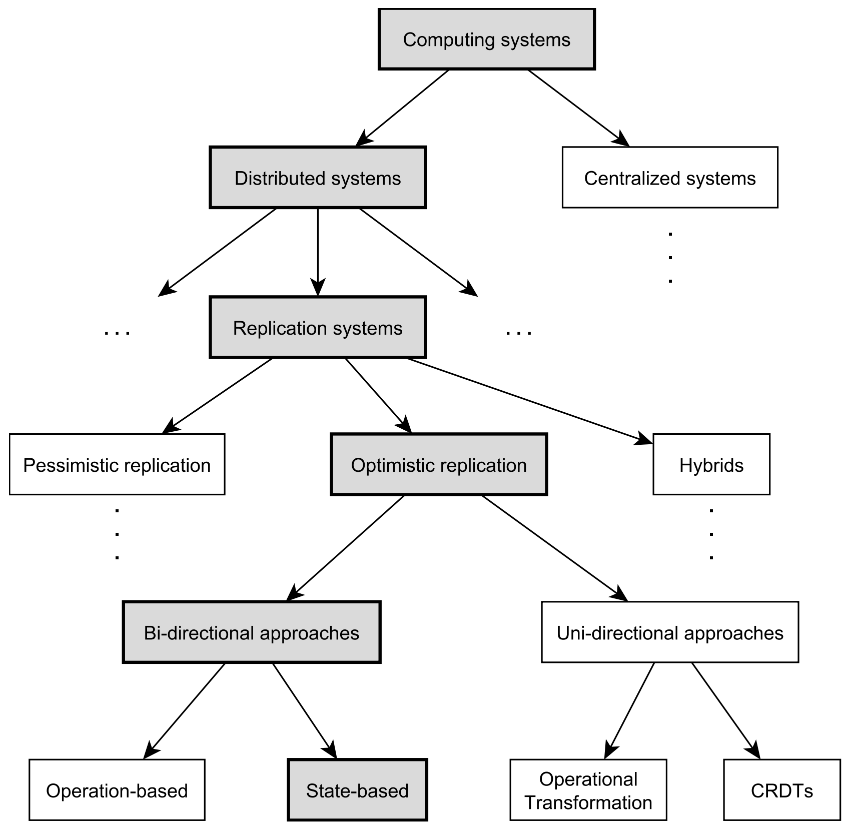


Figure 2.1: Ontologoy of distributed and replicated systems

tems is *consistency* which is *"the property that the state of replicas stay close together"* [SS05], which we discuss in the next section.

2.1.1 Consistency

Ideally the whole system's state should be *consistent*, i.e., behave as if it consisted of a *single* data repository that is in a specific state. Since replication systems distribute data over multiple computers this means that the state of *all* replicas should be equal at any point of time. As we will see in section 2.1.3 replicated systems vary in the degree of how and how fast they achieve consistency. Consistency is in fact a large research field. During the last decades over 50 different notions of consistency have been formulated in the form of *models*, see [VV16; AT16] for a survey. These models specify rules that make the behavior of the system predictable, without specifying how to implement a consistency mechanism. Two models often found in literature are *strong consistency* (or: *linearizability*) [HW90] and *eventual consistency* [Ter+94]. The former resembles the ideal conditions described above by simulating *sequential* access to a single replica. When data is written in one replica, once the write operation has finished, the written data must be available for reading at *all* replicas immediately. In contrast, *eventual consistency* provides a much weaker guarantee. Here all replicas will *eventually* converge to the same state, given that no new write operations are generated at any replica.

2.1.2 Consistency-availability trade-off

The two models presented above are placed on rather opposite ends of the consistency spectrum. From their short description one can already derive the implications on *availability* in the event of a *partition*, that is, when one or more replicas have become disconnected. In such an event distributed replicated systems using the *strong consistency* model do practically *not* offer availability for the read and write operations. Since these systems implement the model using *coordination*, when a write operation is executed at one replica, *all* other replicas have to confirm the receipt of that operation before it is actually committed. Consequently, the write operation would lock up the system during the time of the partition, because the confirmation messages of one or more replicas are missing.

In contrast, *eventually consistent* replication systems offer high degrees of availability. Read and write operations are executed immediately at the replica from which they originate, and are then propagated to other replicas in the background whenever possible. The downside is that this allows the state of replicas to diverge. *Conflicts* may arise which the system needs to resolve, with or without the involvement of the users.

This trade-off between consistency and availability has been formally studied. Fischer's impossibility result [FLP85] shows that reaching perfect consensus in a distributed, asynchronous system is impossible if just one process can fail. Determining the presence of faulty processes is difficult as well. One cannot reliably distinguish a faulty process from one that is slow or rule out that the network is the culprit. More recently, Brewer has established the CAP theorem [Bre00; Bre12], where *C* refers to strong consistency, *A* to availability and *P* to partition tolerance, that is, the ability to detect start and end of a partition and to keep operating during a partition, possibly with degraded functionality[2]. The CAP theorem states that during a partition, replication system designers have to choose between C and A[3]. However, the decision is not binary for the entire system, but can be made in a fine-grained way, choosing different trade-offs for each operation the system offers. For instance, a calendar management application could choose to favor C over A for *write* and *update* operations, while favoring A over C for *read* operations. Different approaches have been formally studied to achieve such *hybrid* consistency models, see e.g. [Bal+16; Ter+13]. In summary, designers of replication systems are faced with the challenge of choosing a C-A trade-off for each operation such that the choice minimizes the negative effects of a partition and incorporates the users' willingness to wait for a response as well as their tolerance of data inconsistencies [Bre12].

2.1.3 Optimistic and pessimistic replication systems

A common subdivision of replication systems is the one into *optimistic* and *pessimistic* replication systems [SS05]. Pessimistic systems bear that name because they are pessimistic about the *lack* of conflicts or the consequence in case a conflict occurs. They prefer strict concurrency control and strong consistency, avoiding conflicts altogether. Optimistic systems are situated on the opposite end. They assume conflicts happen rarely in practice - and if they do, they can be solved after the fact. Therefore little concurrency control exists, together with rather weak consistency guarantees (such as eventual consistency). Classifying a concrete system as one or the other depends on the chosen C-A trade-off. A system that chooses C over A for all operations is a fully pessimistic system, a system that always chooses A over C is a fully optimistic one. Systems in-between employ a Hybrid approach.

In the following, the advantages and disadvantages of optimistic and pessimistic systems are discussed.

The main advantage of *pessimistic* approaches is strong consistency. The disadvantage is that this consistency model is achieved via coordination between all participating nodes, which makes it hard to deploy such systems in scenarios where nodes may only offer ad-hoc connectivity (e.g. tablet computers only turned on occasionally) or in a *geo-distributed* setting, where computers are connected via *wide-area networks* such as the Internet. Such networks have low reliability, increasing the risk that one or more replicas are unreachable, blocking the operation from succeeding. Since communication is unreliable, pessimistic systems also don't scale well, as adding more replicas further increases the chance of communication failure, degrading availability and throughput. These disadvantage inhibit the use of pessimistic approaches in the area of cooperative work. Pessimistic systems are also unsuitable in scenarios such as software control management (SCM), where individuals need the ability to work on

[2] In general, forfeiting *P* is not an option because in practice partitions can and will happen (even in high-availability datacenters). In such an event, a system that has not implemented *P* loses *both* C and A.

[3] While the system is *not* partitioned, *both* C and A can be provided.

files in isolation, not locking them for other users. In this case users prefer A over C, accepting the risk of conflicts due to concurrent changes.

Optimistic systems solve the disadvantages of pessimistic systems. They are suited for (and scale well in) wide-area networks, by offering high availability. Their disadvantage is the weak consistency model, which increases the chance of conflicts.

2.2 Achieving consistency in optimistic replication systems

A system that achieves consistency in optimistic replication systems is often referred to as *data synchronizer*. In literature the term *synchronization* appears frequently and is, unfortunately, used with two different meanings, depending on whether it is applied to *processes* or *data*. *Process* synchronization deals with issues that arise when concurrent processes (or threads) need to occasionally join and agree on certain actions, and to avoid race conditions. In this work we refer to *data* synchronization, which deals with consistency and integrity of disconnected data replicas. We consider the definition of [AC08] suitable who define that *"synchronization is the process of enforcing consistency among a set of [replicated] artifacts and synchronizers are procedures that automate - fully or in part - the synchronization process."*

We start by presenting several *characteristics* of data synchronizers in section 2.2.1. We consider the *directionality* characteristic the most distinct one, and thus present corresponding approaches in sections 2.2.2 and 2.2.3.

2.2.1 Characteristics of optimistic replication systems

We have compiled a selection shown in figure 2.2 from related works such as [SS05] and [MT94] and will elaborate on them in the following subsections. It should be noted that various other characteristics exist, such as *syntactic vs. semantic scheduling*, or *single-master vs. multi-master* [SS05].

2.2.1.1 Directionality

Directionality refers to the communication pattern of a synchronizer.

An *uni*-directional synchronizer determines the data to be transmitted and sends it to other replicas, without further concern what these replicas will do with it. This data might be the current state of the replica or a list of executed operations since the last synchronization.

Bi-directional synchronization[4] indicates that information flows in both directions, that is, data is exchanged via a full-duplex channel, making the consistency algorithm aware of the current state (or operations) of both replicas at once. Traditionally such systems have a *synchronization phase* where, conceptually, all replicas take part in a roundtable meeting with the agenda to achieve consistency. The process works as follows:

1. First, all available replicas establish a connection with each other and *lock* access to their data, to avoid that the user can modify data during the synchronization phase.

2. Replicas exchange data (state or operations) and the merge algorithm (which may be distributed or centralized) determines consistency, given the data from all replicas. It determines what the final, consistent state should be, or which operations need to be propagated to each replica to achieve global consistency. This includes *finding* conflicts and possibly resolving them, e.g. automatically or by asking the user.

3. All replicas exchange states or operations data, apply it to their own replica and finally unlock access to their data again.

[4]Replace *bi-directional* (which applies to *star* communication topology) with N-directional in case of a *peer-to-peer* communication topology.

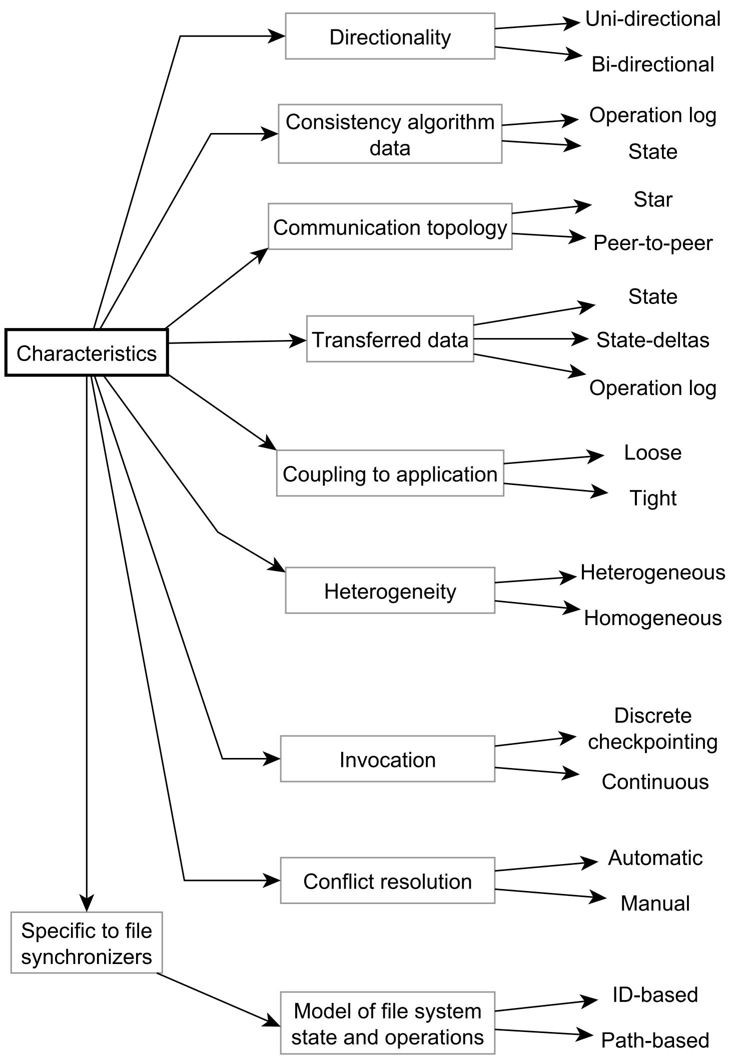


Figure 2.2: Characteristics of optimistic replication systems

In practice such approaches work best when synchronization involves only *two* replicas at a time, because the likelihood of failure increases for N-directional ($N > 2$) synchronization when using unreliable communication channels (such as the Internet).

A number of issues have been identified for systems that use uni-directional synchronization. Here conflicts are resolved on the receiving replica. The sending replica does not immediately know about detected conflicts or their resolution. In the event of a conflict, the sending replica, which is still oblivious of that conflict, may apply additional operations on objects affected by the conflict. These update operations need careful handling. In [CJ05] the authors approach this problem with vector time pairs, whereas [TRN15] solve it by creating a special conflict resolution function that can redirect the updates happening after a conflict update to the correct targets.

2.2.1.2 Consistency algorithm data

An algorithm whose job is to achieve consistency of two or more replicas needs data. Typically, this data comes in the form of complete *operation logs* or *state*. The former requires tight integration of the consistency mechanism with the application that manages the data (see *coupling*, section 2.2.1.5). The logs describe *all* operations that took place at each replica since the last time consistency was achieved. In the latter case the entire state of the application's data is used. In some cases the algorithms, given the current and a previous state, first compute operations from state deltas.

We note that the data *processed* by the algorithm does not necessarily have to coincide with the data that is being *transferred* from one replica to another [Csi16], see *transferred data* characteristic in section 2.2.1.4.

2.2.1.3 Communication topology

Communication topology (or *network topology*) describes the way networked computers are connected to each other. Optimistic replication systems often use the *star* or (arbitrarily connected) *peer-to-peer* topology, but many others exist, such as *bus* or *ring*. The choice of topology has a strong influence on aspects like availability, load balancing and speed. *"At one end of the spectrum, the star topology boasts quick propagation, but its hub site could become overloaded, slowing down propagation in practice; it is also a single point of failure. A random topology, on the other hand, is slower but has extremely high availability and balances load well among sites."* [SS05] Systems with a peer-to-peer topology are often *uni-directional*.

2.2.1.4 Transferred data

The transferred data can, again, be the application's state, state deltas or operation logs. To illustrate that *transferred* data and *processed* data don't have to coincide, consider the following two examples:

1. A file synchronizer may achieve consistency between two replicas by sending the entire state from one replica to another. When state data is received, operations are computed from performing a *diff* (or *delta*) of the received state with a persisted state from the last successful synchronization. These operations are then used by the consistency algorithm.
2. A file synchronizer whose consistency algorithm operates on *state* data may choose to *transfer* only the *initial* state from replica X to Y. X's state data is cached at Y and subsequently X only transfers *operation logs* to Y. Y applies the received operations to the cached state, producing an updated version that is supplied to its consistency algorithm.

2.2.1.5 Coupling to application

Coupling describes the degree of integration of the consistency algorithm into the application, from a software development point of view. For example, the application might be a calendar management software. We find that coupling typically ranges from *tight* to *loose*. *Tight* coupling means that achieving consistency amongst multiple replicas is built into the application itself, or added as a plug-in in case the application allows for extension of functionality this way. Typically, consistency algorithms based on *operation logs* are tightly coupled, because the application needs to provide these logs to the consistency algorithm [Fos+07].

On the other end, *loose* coupling means that the application is not aware of replication or its consistency. Typically, *state-based* approaches such as [Fos+07] are loosely coupled. They are implemented as a 3rd party program that extracts the entire state from the application on each replica and uses that state to achieve consistency, without the application's awareness.

We note that when we classify a system as *loose* later in this chapter, we refer to the coupling between the synchronizer application and the replica it directly works with. There may still be *intermediate* systems with *tight* coupling. For instance, consider a calendar application where a loosely coupled synchronizer achieves consistency between *N* client replicas. The synchronizer is loosely coupled to the calendar application, but may be tightly coupled to additional servers (and their software) used to exchange data or meta-data.

2.2.1.6 Heterogeneity

A *heterogeneous* synchronizer [AC08; Fos+07; PSG04; Her+12] allows data on different replicas to be stored using varying schemas or languages. For instance, an address book application may store contacts as a comma-separated value spreadsheet on one replica, but may use a proprietary database engine on another replica. Another example from software engineering, is the model of a program which may exist as source code but also as UML model. Algorithms that achieve consistency need to convert the representations of the data accordingly and handle incompatibilities. In contrast, *homogeneous* synchronizers are built on systems that store data in the same format on all replicas. *Uni-directional* replication systems, such as Operational Transformation-based systems, are typically homogeneous.

2.2.1.7 Invocation

Invocation determines the points of time when synchronization is triggered. Synchronizers may follow a *continuous* approach that attempts to automatically synchronize whenever users update their replica, given that other replicas are available for synchronization. Another solution is *discrete check-pointing* [MT94] where the user needs to explicitly trigger the synchronization process.

Each approach has advantages and disadvantages. While continuous synchronizers do their work as often as possible at the expense of high bandwidth usage and lack of control for the user, it lowers the risk for conflicts and avoids divergences of replicas that happen simply because the user may have forgotten to trigger the synchronization. Discrete check-pointing suffers from increased risk for conflicts but saves bandwidth and allows for workflows where users may not actually want automatic synchronization, e.g. because they want to be in control over when updates of their replica are propagated (say, to avoid synchronization of incorrect, intermediate file states, or to avoid high charges for bandwidth) [How93]. Most *uni-directional* systems use continuous invocation.

2.2.1.8 Conflict resolution

When users make concurrent updates in different replicas in isolation these may be incompatible with each other and cause a conflict. The list of incompatible updates depends on the replica's data structure. For example, when considering a file system, a conflict occurs if a specific file is deleted in one replica while its content is updated in another replica. Related work distinguishes two general approaches for resolving conflicts [RC01]:

1. *Manual* conflict resolution: the conflict is visually presented to the user who is then given multiple resolution options to choose from. Concrete synchronizers differ in whether they can propagate all non-conflicting updates first and present the remaining conflicting updates to the user at the end of the process, or whether conflict resolution is a necessary first step.
2. *Automatic* conflict resolution: the synchronizer chooses a resolution for the user and applies it automatically. If users need specific conflicts to be resolved differently, they need to apply the fix manually afterwards. This approach is typically used in synchronizers that run as a transparent background service.

Which approach is superior to the other one is still an open question in the domain of file systems [RC01]. We elaborate on conflict-related issues later in this work in chapter 5 on page 65.

2.2.1.9 File system state and operations model

While the general idea is that a file synchronizer is a synchronizer that works on file systems, each work defines the state and operations of a file system differently. The most notable difference is whether a file system is modeled as a set of paths without identity, or whether identities are also part of the model. There are several other differences further examined in section 2.3.

2.2.2 Bi-directional approaches

As described in section 2.2.1.1 the algorithm of a bi-directional approach is given data from *all* replicas and decides how to achieve convergence. We sub-divide them further by the type of *data* used by the consistency algorithm, which is either *state* or *operations*. The following two subsections further elaborate on the respective approaches.

2.2.2.1 State-based

Approaches used in state-based synchronization vary strongly, depending on the supported data structures (and its operations) and other characteristics such as listed in section 2.2.1. We now describe a few exemplary systems.

Non-incremental systems are one of the most simple approaches. They do not store any historic information but only know about the *current* state of a replica. These systems resort to comparing the respective current states, typically between *two* replicas at a time. RSync [TM+96] follows this approach. Such approaches are also used in *heterogeneous* synchronizers (see section 2.2.1.6) where replicas whose state is encoded in different formats are synchronized by *converting* the *current* state of one replica to the format of the other one, followed by equalizing the two states by computing the delta/diff between them [AC08].

Three-way merging (3wm) is a technique often used in *incremental* systems [AC08] that store historic information. It refers to a family of algorithms that are given a base data set d_b (e.g. the state at the last synchronization) and two independently modified data sets d_1, d_2 (which are both based on d_b). These algorithms then find (or are already given) the differences between d_1 and d_b (and d_2 and d_b, respectively) and produce a merged data set d_m that contains a merge of both differences and d_b. 3wm is more powerful than two-way merging because the historic state solves the *create/delete ambiguity* [GPP93]. Thus, 3wm can reliably conclude whether the absence of an object is due to deletion, or the existence of an object is due to creation. When conflicts are detected, they may be resolved automatically or with user involvement.

The algorithms strongly depend on the data structure and the operations they provide, s.t. merging two text documents is very different from merging sets of tuples (e.g. databases), trees, or other application-specific structures. Solving 3wm for *text* data is feasible with limited complexity. In [Mye86] the authors describe how to detect the shortest sequence of edit operations (*edit script*) that transforms an input text file to an output file, considering only line insertions and deletions as operations. After computing the edit scripts $e_1 = d_1 - d_b, e_2 = d_2 - d_b$, applications such as *diff3*, perform the 3wm of d_b, e_1, e_2 [KKP07]. 3wm has been prominently used in practice for merging software source code, in spite of the risk for syntactic or semantic merge conflicts [Ape+11].

For *tree*-based data structures *differencing* (which includes *matching* of nodes) and *merging* is considerably more complex, as surveys such as [Lin01, section 4.4] and [Bil05] demonstrate. Research is frequently driven by concrete domains, such as *merge conflict resolution* [Ape+11; Men02] of source code or merging UML models [PB03]. Generic synchronizers also exist, such as [Lin04], a framework that applies 3wm to XML documents. 3wm has also been applied to *heterogeneous* systems, see [Fos+07]. Here the authors built a universal, heterogeneous synchronization framework called "Harmony" for *tree*-based data. Developers who want to use Harmony synchronize an application's state need to implement two *transformation functions* (called “lenses”) that transform data structure to and from Harmony's expected, internal tree format, to which 3wm is applied. Developers also need to specify a synchronization *schema*, which defines invariants of the data[5].

We note that conventional 3wm requires the state of both replicas to remain *static* during the synchronization. In systems where updates occur frequently, such as collaborative text editors where multiple key strokes per second are common, locking the data during synchronization would severely impact the

[5]For instance, a phone book application would have the invariant that states that a person can have at most one Work number.

usability of the application. [Fra09] proposes an extension to 3wm that improves handling of concurrency, allowing the user to manipulate data even while the system is performing a synchronization.

For our work, 3wm is relevant, but not for merging divergent *file content*, because earlier (base) versions of files are typically not available. Instead we consider the file system's *structure* as a document. 3wm can be used to detect and merge changes on the structural level, which is also done in [LKT05], see section 2.4.2.

2.2.2.2 Operation-based

Operation-based approaches which use a complete log (sometimes also referred to as "trace" or "journal"), such as [SRK00; Ker+01; Ter+95; CH06], follow a common pattern. At each replica the application collects a complete log of all operations performed by the user. When synchronization is triggered, replicas connect to each other in a synchronization phase. The application locks access to the user, logs are transmitted and all replicas *roll back* to a consistent state, i.e., a point in the log where all replicas's logs coincide. Then the reconciliation algorithm produces a *schedule*, that is, a total order of operations that interleaves the operations of all replicas, including conflict detection and resolution - which may involve the user. Once a valid schedule is found, all replicas apply it and finally unlock access to the user. [Qia04; SRK00] Approaches to find a schedule, such as [SRK00; Ker+01] model operations in a way s.t. dependencies between operations are made explicit using *assertions*. Assertions consist of pre- and post-conditions often formulated as first-order logic (FOL). These assertions represent *constraints*. Finding a valid schedule therefore becomes a constraint satisfaction problem, where the goal is to maximize (rather than minimize, which would be the goal for solving *planning* problems) the number of operations in the schedule - a NP-hard problem. There are also distributed variants of such algorithms, see [CH06].

These log-based approaches suffer from multiple issues. In case operations occur frequently, the log grows quickly and may become very large, especially if a replica is disconnected for longer periods. This causes increased storage requirements, transfer time and time required for finding a schedule, which in turn causes the replica to be locked for extended time periods. [Qia04] In addition, the log collection must be 100% reliable. If there is a chance that executed operations are missing in the log, the replica's state will begin to diverge over time [Fra09].

2.2.3 Uni-directional approaches

The merge algorithms of uni-directional approaches are able to achieve consistency over time, without the requirement to know about the data of all replicas at the point of the merge. Conflicts are detected and resolved without the need to coordinate with other replicas. To achieve this, conflict resolution is typically fully automatic and deterministic, without user involvement[6]. In the following sections we introduce two popular approaches, Operational Transformation and Commutative and Convergent Replicated Data Types.

2.2.3.1 Operational Transformation

Operational Transformation (OT), presented in the seminal work [EG89] and follow-up paper [SE98], is an approach with a large amount of academic works and industrial use-cases[7]. OT has been used in different types of collaborative applications, such as groupware [EG89; SE98], document editing [Sun+04], CAD design [Agu+08], XML [DSL02; Ost+06a] file systems ([NS16; Mol+03], see section 2.4.3) and many others[8].

[6] There are exceptions, such as [CJ05], where users can resolve conflicts non-deterministically.

[7] OT was popularized by Google Wave, which became Apache Wave `http://incubator.apache.org/projects/wave.html`. It lives on in web-frameworks such as SwellRT `http://swellrt.org/` or ShareDB `https://github.com/share/sharedb`, and is used in industry products such as Google Docs `https://www.google.com/docs/about/` or Codox Wave, `https://www.codox.io/`. URLs retrieved July 21, 2019.

[8] See sections 1.2+1.3 of `http://www3.ntu.edu.sg/home/czsun/projects/otfaq/` for further applications, retrieved July 21, 2019.

The general idea of OT is the following: every operation performed by the user causes an OT operation to be generated, executed and sent to all other sites. The operation is given the context in which it was generated (e.g. operation *Insert("e", 1)* and context being a document's state, *"tst"*, such that the *Insert* operation transforms the document to *"test"*). Whenever a site receives an OT operation, it transforms it against already-executed operations to account for their effect on the state. OT systems are generally divided into two parts, the generic *control algorithm* (sometimes referred to as *integration algorithm*) and a set of application-specific *transformation functions*. The control algorithm is independent of the application and is in charge of processing incoming OT operations and maintaining a buffer of executed (applied) operations, calling the transformation functions against the correct, buffered operations, in the correct order. Many different variants were developed over time, such as dOpt [EG89], GOT [Sun+98], SOCT2 [SCF97], TIBOT [LSL04], or COT [SS09]. Some rely on a central server to determine a global operation order, some allow for arbitrary peer-to-peer arrangements. The application developer then needs to develop one transformation function for each possible operation pair and ordering[9].

2.2.3.2 CRDT

A CRDT[10] [Sha+11b; Sha+11a] is a mutable object, replicated to each site, which offers an interface to the application that uses it to store data. CRDTs can be divided into *state-based* CRDTs (Convergent Replicated Data Types, CvRDT), and *operation-based* CRDTs (Commutative Replicated Data Types, CmRDT). CvRDTs can be emulated by CmRDTs and vice-versa. Modifying an object immediately affects the object's local state. CvRDTs then asynchronously distribute the whole data type's state to other replicas (or just state-*deltas* [ASB15; vLP16; Ene+18] to reduce network usage), whereas CmRDTs log and send only the operation itself, including its parameters. A core concept of CRDTs is *commutativity*, which applies to the merge procedure of both operations and states. Whenever a site of a state-based CRDT receives another replica's state, it *merges* that received state with its local one. CvRDTs build on the concept of a join-semilattice, which is a partial order with a defined least upper bound (LUB) [DP02]. By its definition, the LUB has commutative, idemptotent and associative properties, and the merge function computes the LUB of the local and received state, which ensures eventual convergence of all replicas. In operation-based CRDTs operations are enriched with additional meta-data to make them commutative.

A considerable number of basic CRDTs have already been specified, such as counters, sets, registers, maps, graphs [Sha+11a], lists [Roh+11] and JSON [KB17] which are used as building blocks in more complex applications, such as collaborative rich text editing [Pre+09; Nic+16], 3d modeling [TIH19], CAD [Lv+17; Lv+18] and file synchronization [TSR15; AMU12]. CRDTs have also been increasingly used in industry applications, such as Amazon DynamoDB [DeC+07], Riak [Bas18] and many others[11].

OT and CRDT research is still actively pursuing the holy grail of building *truly* provably correct replicated systems with optimal space and time complexities (i.e., less than quadratic). While there is some effort to unify these two theories [Meh+14] we also observed heated separative discussions where each "camp" defames solutions of the other, in the style of "your proof is not a real proof" or "our implementation is n times faster than yours", see [Sun+18] and the corresponding online discussions[12]. It remains open which theory turns out to be superior.

2.3 File systems

The core of any data synchronizer is the data model the synchronizer works on. For file synchronizers, the data model is the file system. We reviewed related *academic* file synchronizer works to understand how they model the file system (see section 2.3.1) and their general approach (see section 2.4). As our results from section 2.3.1 show, file system definitions vary strongly between each work, different models are not compared against each other, and some works don't even provide any formal model. We would

[9] For instance, for a text editing system with insert operation i and delete operation d, the transformation functions $T_{ii}, T_{dd}, T_{id}, T_{di}$ need to be developed.

[10] See [Pre18; Rij18] for an overview. The originating work is WOOT [Ost+06b].

[11] See `https://github.com/ipfs/research-CRDT/issues/40`, retrieved July 21, 2019.

[12] See `https://news.ycombinator.com/item?id=18191867`, retrieved July 21, 2019.

have expected that the file system model is formally defined, including its operations and invariants. Since the analyzed works don't consider *several* models, their differences or incompatibilities are not discussed either. While there is some literature regarding the heterogeneity of data models for *generic* data synchronizers (as presented in section 2.2.1.6), there is a lack of discussion for *file systems*. This caused us to ask research question 1:

RQ1 - File systems: What different kinds of file system definitions exist in academia and practice? Which criteria are relevant for file synchronizers? How should a file synchronizer's internal, abstracted file system model look like, which incompatibilities exist and how can they be handled?

The answers are found in sections 2.3.1 (for *academic* file system definitions) and 3.1 for practical models. Section 3.1 also discusses how to handle incompatibilities, providing guidance to researchers and developers of file synchronizers.

The following sections discuss short-comings in related file synchronizer works regarding the file system and operations model.

2.3.1 File system specifications

When reviewing related works we found that the degree of *formalism* varies strongly. Some works don't provide any file system specification [CJ05; EYL13], some [Bao+11; Li+12a; LKT05; Mol+03; TSR15; UFB10; Bjø07] only provide a partial description, e.g. of their internal state database records or operations, and some synchronizers [BP98; NS16; RC01; Csi16] formally specify *a* file system (includings its operations and invariants). There are considerable differences in the specifications[13], which affects how the synchronizers work internally. By comparing the file system models used in related works, we found the following characteristics:

- *Identity- vs. path-based model*: as discussed in [TSR15, section 3] the file system and its operations can be modeled using the *identity*-based approach where each object is identified by a unique ID, or by a *path*-based approach ("namespace-based" in [TSR15]) where objects are only identified by their path. ID-based approaches include [TSR15; Bao+11; Li+12a; Li+12b; LKT05; Mol+03; Bjø07], for path-based approaches, see [BP98; NS16; Csi16; TH10].
- *Hardlink*[14] support for files: a file system may support that a specific *file* is linked into multiple directories (the name of each link may vary), or is linked multiple times under different name into a specific directory. This typically affects only files, not directories, because that would allow to form cyclic namespaces. File systems without hardlink support only allow each file and dir to be linked once, which allows a diff algorithm to detect *unambiguous* move operations. For file systems with hardlink support *move* operations can still be detected, but only for *directories*. For *files* we instead have the *link* and *unlink* operation, and move operations become ambiguous once the *before state* or *after state* link count for a file is different than 1.
- *Directory* support: while all file systems we encountered support directories (as otherwise they would not be hierarchical), some file synchronizers limit the support for directories. In [Qia04, Definition 2.3.1 + section 2.4.4] the authors model their file system as a set consisting only of *file* paths and their identities. The Git VCS [TH10] does include directories as *tree* objects in its internal commit files, but does not support *empty* directories. Although dropping support for directories reduces the complexity of the synchronization algorithm, it also reduces the efficiency and intention preservation[15]. We assume that for this reason the majority of synchronizers do support directories.

[13]We note that modeling file systems is far from trivial once all its features (such as permissions and concurrency) are considered [Ntz16]. Many real-world specifications such as POSIX have only informal definitions and works exist [Rid+15] that iteratively extract a formal model from existing file system implementations.

[14]We do not cover *soft* or *symbolic* links because they do not influence the file system model. They are separate files (with their own identity) which the file system APIs can treat in a specific way (following them).

[15]If the user renames or moves a directory that contains sub-files, this causes the synchronizer to instead apply move operations to the other replica for each sub-file separately. This is neither efficient nor do the operation-intentions of the two replicas match.

Category	Short-hand	Description	Related works
Directory support, but **n**o **h**ardlink support	NH-MD	Full support of operations: create, edit, **m**ove, **d**elete.	[LKT05], [Bao+11], [Bjø07], **our work**
	NH-M	Supports create, edit, **m**ove. The delete operation becomes a move to a garbage directory.	[Mol+03]
	NH-D	Supports create, edit, **d**elete. Move becomes a sequence of delete + create.	[BP98], [Li+12a; Li+12b], [RC01; Csi16]
	NH-RD	Supports create, edit, **r**ename, **d**elete. Rename operations do not change the object's parent directory. Move operations that do change the object's parent directory become a sequence of delete + create.	[NS16]
Directory support, with **h**ardlink support	H-All	Supports edit, link, unlink, move. Move exists only for directories. Unlink of the last instance of an object represents the *delete* operation. Link represents *create*.	[TSR15]
No (**e**mpty) **d**irectory support	NED-All	Supports createfile, edit, deletefile, movefile.	Git [TH10], [Qia04, Definition 2.3.1 + section 2.4.4]

Table 2.1: Operation support of file synchronizers

Overview of hardlinks, directory and operations supported by related works. Note that additional file system interpretations are possible (e.g. NED-D, where *move* is modeled as *delete+create*), but we found no corresponding works using those interpretations.

- *Operation* support: while all file synchronizers we found support create and edit operations, support for other operations varies.

Table 2.1 presents an overview of the above characteristics. If an implementation wants to detect *move* or *rename* operations, approaches that compute operations from state should use an identity-based model, because path-based models do not allow to reliably detect moved objects. Operation-based approaches such as [Mol+03; NS16] can be based on either identity-based or path-based models, because the implementation is sure to know the exact order of operations.

Aside from the large diversity of definitions we found that most works fail to mention on which file system their *implementation* is actually executed. It is often unclear whether the authors implemented their own file system from scratch, use an abstraction layer on top of an existing file system, or whether the implementation operates on an existing real-world file system directly (e.g. ext3, NTFS, etc.).

Finally, we note that the choice of a file system model influences the set of *conflicts* that exist for the chosen model. We examine this effect more closely in section 5.6 on page 91, after having discussed the concept of conflicts.

2.3.2 Real-world file system operation logs

We found that approaches based on *operation logs* (such as [NS16; Mol+03]) do not discuss how this log is obtained. Our conclusion is that a complete log of file system operations can only be obtained when the information is either provided by the user, or when the synchronizer is tightly integrated into the file system implementation, e.g. by hooking into the file system implementation, or writing the file system from scratch. We back this observation in chapter 4 where we investigate the capabilities of obtaining a complete log from the most wide-spread end-user file systems on Windows and macOS computers. We find that without extending the file system implementation in some way, any solution is best-efforts, i.e., the complete log *cannot* be obtained in all scenarios, and the synchronizer needs to fall back to computing operations from state.

2.4 File synchronizers

This section presents selected file synchronizer works. Our overview excludes the following two related areas:

- Systems that distribute files on *multiple* cloud-storage backends, see e.g. [Han+16; Tan+15; Cel+16]. The core contribution of these works is the distribution aspect.
- Papers related to *(remote) file synchronization*, see [SM02]. This research topic discusses how a specific file which exists on two remote sites can be efficiently transferred, s.t. after the transfer both sites have the same file content, without transmitting the whole file.

An overview of the related works is presented in table 2.2 . After explaining the basic differences of state- and operation-based approaches in section 2.4.1 we briefly describe each work, presenting *state-based* approaches in section 2.4.2 and *operation-based* ones in section 2.4.3. Details regarding conflict detection and resolution are deferred to chapter 5, where we analyze the approaches of related work in section 5.1 and propose solutions. In summary the analysis exhibits several shortcomings in related works, such as inadequate formalism, lack of rationale regarding conflict resolution approaches, or missing discussions how to resolve situations where *several* conflicts apply to one specific file at once. For this reason the third research question considers different conflict-related aspects:

RQ3 - Conflicts: What sets of operations applied to two disconnected file systems are conflicting? How do conflicts depend on the file system model? Can multiple conflicts be combined? What are possible solutions for resolving individual conflicts and conflict combinations? Which conflicts are relevant in practice? How can conflicts be explained to the user?

Answers are provided in chapter 5.

2.4.1 State- vs. operation-based approaches

This section presents the basic differences between state- and operation-based approaches for file synchronizers. Figure 2.3 provides an overview for a two-replica scenario.

For *operation-based* approaches the concept we discussed in section 2.2.2.2 for *generic data* synchronizers also applies to *file* synchronizers. The file synchronizer's merge algorithm is given a list of operations and uses *only* this operation list for the reconciliation of updates and equalizing the replica's states. The operation lists may have been retrieved from the file system directly, or may have been deduced by a preprocessing step that compares the current file system *state* to a previous state.

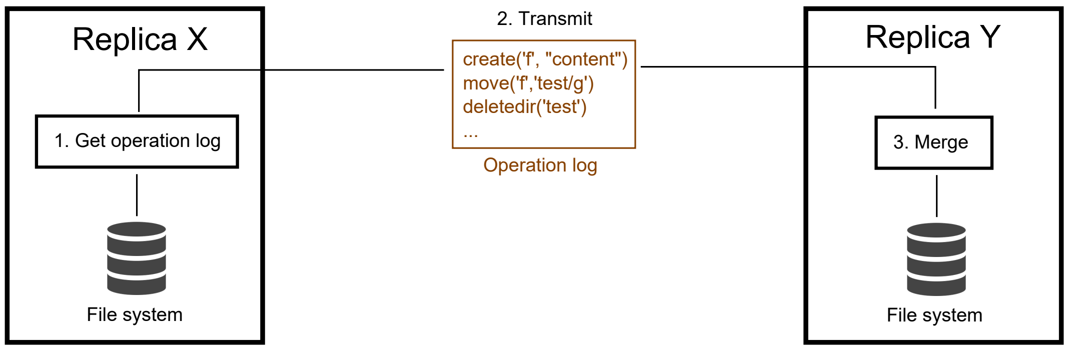
The merge algorithm of *state-based* approaches uses structures derived from the state of the synchronized replicas, although some works also describe the *additional* use of computed operations from state deltas. As presented in the seminal work [BP98], most state-based file synchronizers use a three-stage process that consists of *update detection*, *reconciliation* and *propagation* stage. During *update detection*, which is performed for each replica separately, the differences between the state of the previous

	[BP98]	[LKT05]	[Bjø07]	[UFB10]
Directionality	Bi-directional	Bi-directional	Uni-directional	N-directional
Coupling	Loose	Loose	Loose	Loose
Merge approach	Analysis of dirtyness predicate	Variant of 3wm	Unclear	Unclear
Communication architecture	Client/Server	Client/Server	Peer-to-peer	Peer-to-peer via dynamically elected master
Consistency algorithm data	State	State	State	Operations computed from state
Conflict resolution	Manual	Semi-Automatic (depends on operation)	Automatic	Manual, Automatic
Transferred data	State	State	Unclear (State?)	State
ID/Path-based file system model	Path-based	ID-based	ID-based	Path-based

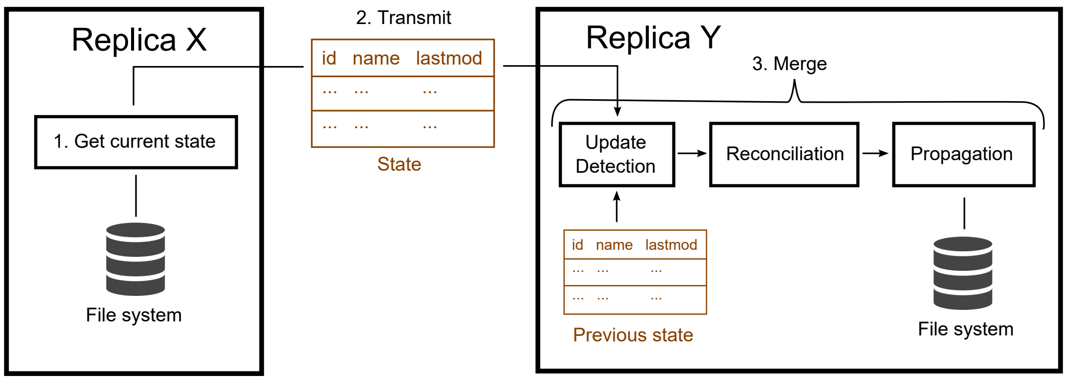
	[Bao+11]	[CJ05]	[Li+12a; Li+12b]	[RC01; Csi16]
Directionality	Uni-directional	Uni-directional	Uni-directional	N-directional
Coupling	Loose	Loose	Loose	Loose
Merge approach	Unclear	Vector time pairs	Version vectors	Command algebra
Communication architecture	Client/Server	Peer-to-peer	Client/Server	Unclear
Consistency algorithm data	Operations computed from state / logged on server)	State	Operations computed from state	Operations
Conflict resolution	Automatic	Manual	Automatic	Manual
Transferred data	Operations	Operations	Operations	Operations
ID/Path-based file system model	ID-based	Unclear	ID-based	Path-based

	[Mol+03]	[NS16]	[TSR15]	[Naj16]
Directionality	Uni-directional	Uni-directional	Uni-directional	Unclear
Coupling	Loose	Unclear	Unclear (Loose?)	Unclear
Merge approach	Operational Transformation	Operational Transformation	State-based CRDTs	Synchronization phase for move operations, CRDTs for all other operations
Communication architecture	Client/Server	Client/Server	Peer-to-peer	Unclear
Consistency algorithm data	Operations	Operations	State	Unclear (State?)
Conflict resolution	Automatic	Automatic	Automatic	Automatic
Transferred data	Operations	Operations	State	Unclear
ID/Path-based file system model	ID-based	Path-based	ID-based	ID-based

Table 2.2: Overview of file synchronizers



(a) Operation-based



(b) State-based

Figure 2.3: Overview of state- vs. operation-based file synchronizers (2-replica scenario)

synchronization and the replica's current state are determined. The *reconciliation* stage uses the available states and the detected updates to schedule a list of operations that eliminate the divergences of all replicas, including conflict detection. Finally, the *propagation* stage executes these operations.

Some state-based approaches, such as *three-way merging* discussed in section 2.2.2.1, allow the synchronizer to build a merged state (during reconciation) and then *atomically* replace each replica's diverged state with the merged state (during propagation). However, this is typically not possible for file systems. Modifying file systems is expensive. Atomically replacing a replica with 30'000 files with another state that consists of, say, 32'000 files is typically not supported. Consequently a state-based synchronizer needs to manipulate each diverged replica in *incremental* steps, using the available file system operations. This is challenging because states (and operations computed from state deltas) do *not* indicate the order of operations. An example is shown in figure 2.4. Most works we present in section 2.4.2 ignore this detail. Only the authors of [RC01; Csi16] discuss it, but their work is limited to a file system that does not support *move* operations, whose support is, however, very desirable. For this reason our second research question is:

RQ2 - Operation order: As operations detected during state-based update detection lack order, but not all operations are commutative, how can a valid order be detected and propagated by a synchronizer? This question was already answered for file system models which do not support *move* operations, but remains open for file systems which do support them.

We answer this question in section 4.2 on page 47.

2.4.2 State-based approaches

[BP98] is among the most-cited works for file synchronizers. Additional details are available in a tech report, see [PV04] and an unfinished manuscript [JPV02]. Its algorithm is implemented in *Unison*, a user-level program available to end users[16]. Unison synchronizes two replicas upon the user's request. The authors formally specify their file system model, but they don't mention operations explicitly. Their model is path-based and excludes multiple hard-links, or symbolic links. Their algorithm consists of the three stages *update detection*, *reconciliation* and *propagation*, which is a typical approach for many bi-directional file synchronizers. In the *update detection* stage, both update detectors (one for the local, one for the remote replica) collect the current state of the file system, consisting of a set of paths with associated meta-data, such as the inode and the last-modified timestamp. The *current* state is compared against an *archive* version taken from the most recently completed synchronization, to compute a dirtyness-predicate, an upward-closed boolean function that returns *true* for a path that has changed[17] between the archived and current state, *false* otherwise. The *reconciliation* stage is given the both replica's dirtyness-predicates, computes a list of all non-conflicting changes and finds conflicting ones, which are presented to the user for resolution. Finally, the *propagation* stage executes the changes determined during reconciliation, to achieve convergence among both replicas. Unison is known to work on end-user file systems on Windows, macOS and Linux. Its main limitations are (1) the assumption that the file system is not changed by the user during any of the three stages, and (2) file system objects have no identity, s.t. a move operation is not detected but instead synchronized via a delete + create operation.

[LKT05] also performs a pair-wise synchronization of two replicas. The authors approach the task by representing the scanned file system hierarchy as an XML document. The XML elements contain a globally unique ID (GUID) for each object[18], but the authors don't provide any further formalization of the file system or its operations. Using a persisted base document T_0 (from the last successful synchronization) and the current documents T_1, T_2 from each replica they apply their own tree-based three-way merge algorithm *3dm* [Lin04] to compute the merged tree T_m that represents the final outcome, including the resolution of conflicts. To make both replicas consistent their algorithm finally iterates over T_m's nodes and compares them to the ones of T_1 and T_2 respectively to determine the necessary *create*, *move* and *delete* operations required to achieve consistency[19]. Their work additionally contributes a method to speed up the repeated scanning of the replica's file system hierarchy. Their approach is called *bubbling modification timestamps* (BMT). They hook into the Linux file system API functionality, such that whenever a file is modified, their hook updates the *last-modified* timestamp of not only the immediate parent directory, but *all* ancestor directories. When subsequently scanning a directory (to obtain T_1 or

[16]Available at `http://www.cis.upenn.edu/~bcpierce/unison/`, retrieved July 21, 2019.

[17]The specific change could be the creation of a new object, a change in an existing object, or the deletion of an existing object. In case an object is moved, both its source and destination paths are marked dirty.

[18]The paper mentions the usage of UNIX *inodes* to uniquely identify file system objects, which does not make sense because inodes are system-dependent, i.e., *not* globally unique. We can only assume that the implementation maintains a 1:1 mapping for each object's inode to the corresponding GUID.

[19]Missing from this list of operations is the *update* operation (i.e., updated files) which their algorithm deals with in a final step after all other operations are executed.

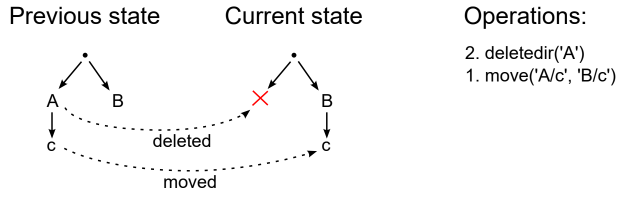


Figure 2.4: Example for non-commutative operation order

Applying the two operations *move('A/c', 'B/c')*, *deletedir('A')* detected in one replica to the other replica is not trivial. Choosing the wrong order, i.e., first deleting "A", would also delete "A/c" on the other replica, causing replica states to diverge.

T_2) their algorithm can then check whether the last-modified timestamp from T_0 matches the one from the scan, in which case it can skip recursively scanning that directory.

In [Bjø07] Microsoft's distributed NTFS file system, *DFS-R* for Windows Server, is presented. Their approach synchronizes N replicas, arranged in an arbitrary (peer-to-peer) topology. Their state-based approach is divided into *two* synchronization phases, a *local* and a *global* one, which take turns. The *local* synchronization synchronizes a directory on a NTFS disk with a machine's local database. Details of local synchronization are not discussed by the authors. They also do not describe the (NTFS) file system or its operations formally, but mention that they support *create*, *delete*, *rename* (which includes *move*) and *file-update* as operations, and the file system to resemble an arborescence where each node is uniquely identified by a local ID (c.f. *inode*) and appears exactly once. The database, however, is formally described. It contains *version vectors* and *globally* unique IDs (as well as the local IDs) for each object. Deleted objects remain in the database, with a tombstone-flag. A garbage-collection mechanism that reclaims tombstones is also briefly discussed. The *global* synchronization, which is briefly discussed, then synchronizes the databases of two machines at a time, using version vector information to properly detect concurrent (conflict) changes.

In [UFB10] the authors also present an approach suitable for an environment where the goal is to achieve efficient synchronization of N replicas, where the devices on which the replicas reside only provide ad-hoc connectivity via a (wireless) local area network. A common approach is *cloud-based file synchronization* [Zha+14] such as Dropbox, which is a pair-wise synchronization between a replica and a server, over the Internet. The authors want to avoid downsides of cloud-based file synchronization, such as poor performance due to slow upload bandwidths, costs, security and privacy concerns, by keeping the content of files only on the devices, transmitting the content only within the local area network. Whenever multiple devices which are concurrently connected to the local network and the Internet want to synchronize, they connect to a *cloud service*. This service stores only the meta-data of the synchronized objects and is in charge of coordinating the synchronization process. It chooses one device to act as master, which then performs two rounds of pair-wise synchronizations with all other local devices which are currently online. After the first round, the master is sure to have collected the changes of all replicas, which it then distributes to all other replicas during the second round. This work does not provide sufficient detail to fully understand the pair-wise synchronization protocol. Neither the file system model nor the operations are modeled formally. The update detection stage seems to compute the set of operations from the archived and the current state of a replica. It is unclear whether the states (and computed operations) use unique IDs or paths. It is also unclear how the reconciliation stage determines the list of operations and how they are propagated.

[Bao+11] present a cloud-based file synchronization approach with an architecture similar to Dropbox. Consistency is achieved by clients performing a pair-wise synchronization of their file system with a central server. While clients communicate with a single end-point using a HTTP-REST interface, in the cloud the load is distributed among a *control* server (providing the REST interface), *meta-data* server and *storage* server. The authors specify their ID-based internal database record structure of the file system state, but they don't formalize the file system or its operations. Their synchronization protocol achieves bi-directional synchronization by two consecutive uni-directional synchronizations. In the upstream synchronization the client regularly scans its file system and detects operations. Since the authors don't explicitly explain the process, we assume that the client stores a persistent snapshot of the file system's state locally and computes a difference of the persistent snapshot and the current scan to derive the list of operations. The client sends the list of operations to the control server (including payload in case of created or modified files), which performs the reconciliation, updating both the storage- and meta-data server. The latter keeps a historic *log* of operations of all users. The downstream synchronization is achieved by clients regularly querying the control server, requesting the list of operations that happened since the previous query. The client provides a timestamp in this query and the server serves the log of operations (again, including file payloads) that the client then merges.

In [CJ05] the authors introduce synchronization using *vector time pairs*. Their algorithm internally uses two (instead of one) version vectors, which fixes several of short-comings of version vectors, like the inability to record conflict resolutions or the necessity for *global* consensus to garbage-collect deletion notices. The work presents how the algorithm works for synchronizing the *modifications* of *individual files*. Whenever a file's content has changed[20] the corresponding logical clock entry is increased by one in the respective vectors. The authors don't formally specify their file system model or its operations. Although the work presents pseudo-code for the synchronization of *directories*, it is unclear how files and directories of two replicas are associated with each other. We assume that their approach is path-based where files and directories have no identity, because *move* or *rename* operations are not mentioned at any point. The authors make a Linux implementation available[21] which appears to work on existing, real-world file systems.

Another cloud-based file synchronization approach (similar to [Bao+11]) is presented in [Li+12a; Li+12b]. The authors specify their ID-based internal database record structure of the file system state, but they don't formalize the file system or its operations. Even though files and directories are modeled on IDs, they propagate *move* operations as *delete* + *create* operations. The basic synchronization is done in two phases, where each phase reflects a uni-directional synchronization: In phase 1 (client-to-server) the client sends locally detected operations to server. The server checks for conflicts, if any exist, none of the operations are executed but the conflicts are returned to the client. The client then converts the conflicting operations to resolve the conflicts and then retransmits the operations to the server, until no more conflicts are reported. Then the server applies the client's operations. In phase 2 the client receives the list of updates from server. If there are local conflicts, the client first transforms the operations and sends them to the server (the cycle from phase 1 is applied again). When there are no conflicts anymore, all server updates are applied on the client.

In [TSR15] the authors use state-based CRDTs for file system synchronization. They formally specify their file system model, but don't mention operations explicitly. Their model is ID-based and supports multiple hard-links for files. Their work describes conflicts and their automatic resolution in detail on a conceptual level, but omits the intricate details of their implementation as state-based CRDT due to space constraints.

In the thesis [Naj16] (and the later paper [NSE18]) a lot of *theoretical* work was done using CRDTs on various applications, including file systems. The authors formally specify a file system including its operations and their pre- and post-conditions in first order logic. They use CRDT *sets* to store the file system state and design the merge function to solve different conflicts that can occur. Using CISE [Got+16], an SMT-based solver, they find that it is not possible to use CRDTs to build a *fully* asynchronous file system (where *all* file system operations can be executed asynchronously). The reason is that *move* operations (of *directories*, for which only one path or link may exist at a time) executed concurrently in different replicas can break the invariant of a file system. The invariant dictates that file systems always form an arborescence from graph theory. When directory X is moved to Y in one replica but X is moved to Z in the other (or when X is moved into Y in one replica and Y into X in the other) then the invariant is violated[22]. To solve this problem, the authors then specify a *mostly* asynchronous file system, which is a hybrid approach where *move* operations affecting directories have to be *synchronized*, while all other operations can be dealt with by CRDTs. This model was verified to be provably correct using CISE. As the approach was not actually implemented, various characteristics such as coupling, topology, directionality and others are unknown.

2.4.3 Operation-based approaches

In [RC01] and the follow-up work [Csi16] the authors formally model their path-based file system and present a formal *algebra* of commands for the different file system operations. To reduce the complexity of the problem, the *move* operation is replaced by *delete+create*. The authors provide a table of algebraic laws which explore various properties of the commands, such as *commutativity*. Their synchronization algorithm follows the general paradigm of [BP98]. First, all replicas determine their list of commands

[20]Deletions also count as modification.

[21]See `https://swtch.com/tra/`, retrieved July 21, 2019.

[22]In [TSR15] the authors work around the issue by translating two conflicting move operations into creating two *copies* of the directory, including all its children.

(operations) by computing the difference of their previous and current file system state. The updates are sent to one chosen replica which performs reconciliation, including conflict detection. The notable difference to [BP98] is that the number of participating replicas is $n \geq 2$ (not forced to $n = 2$). Non-conflicting updates are propagated, while conflicting ones need to be solved by the user. The work does not discuss implementation details, such as propagation.

The following two works apply Operational Transformation to file systems. [Mol+03] is the first work to our knowledge. They use SOCT4 [Vid+00] as control algorithm. The authors don't specify their file system model but provide a semi-formal definition of the operations (*create* files or directories, *move* objects, deletion is modeled as a move operation of the object into a garbage directory), which are based on IDs instead of paths. When files are text or XML files the OT algorithm can be applied to propagate changes to other replicas or merge conflicting updates. The synchronizer was developed as part of the *LibreSource* collaboration platform[23]. The implementation is a command-line tool with commands[24] similar to SCM tools such as SVN or Git. In [Mol+03] the authors don't discuss update detection, but examining the workflow in LibreSource's manual reveals that operations are computed by diffing states. One issue in this work is that the transformation functions are provably wrong [NS16] because the author's definition of the *move* operation misses the pre-condition that states that the path of the parent directory of the destination may not begin with the source path[25]. This allows for concurrent changes to become unsynchronizable.

[NS16] is a more recent OT-based work using the COT [SS09] control algorithm. The authors provide a formal specification of their file system as a graph. They also specify the operations that create or delete files or directories, update file contents or *rename* objects. *Move* operations that move an object to another parent directory are *not* supported and are replaced by *create* + *delete* operations. Their file system model is path-based. The provided OT transformation functions, apart from dealing with conflicting and non-conflicting operations, also handle issues such as the *adaptation* of *paths* in case directories are renamed on a specific replica[26]. Although the work mentions a prototype implementation, it is unclear how operations (updates) are detected and how their implementation is coupled to existing file system implementations.

2.4.4 Summary

Of the 12 surveyed systems, three are operation-based and nine are state-based, indicating that state-based systems are more common. While the merge approach of operation-based works is either OT- or CRDT-based, the merge algorithm of each state-based work differs, and is sometimes not explained at all, see [Bjø07; UFB10; Bao+11]. The merge algorithms vary because the data foundation, the file system model, is defined differently, and because some synchronizers desire certain characteristics such as peer-to-peer synchronization. Our survey provides us with an understanding of the challenges and trade-offs made in the presented works. With these findings we selected a suitable merge algorithm for our implementation. Details of our decisions are found in section 2.6.

2.5 Related fields and technologies

2.5.1 Version Control Systems

Version control systems (VCS) are software systems that manage and document changes to artifacts, such as source code or other kinds of files. Different variants exist, e.g. *centralized* client/server systems like Concurrent Versions System (CVS) or Subversion (SVN), and *decentralized* systems such as Git.

[23] `http://dev.libresource.org`, retrieved July 21, 2019.

[24] See `http://dev.libresource.org/home/doc/so6-user-manual/manuals/commandline.html` and `http://dev.libresource.org/home/doc/so6-user-manual/manuals/reference/commands.html`, retrieved July 21, 2019.

[25] For example, *Move("A", "A/x")* must fail.

[26] For instance, if an already synchronized directory "x" is renamed to "y" by one device, while on another device the user creates a new object at "x/z", the create-path is adapted to "y/z".

Each user has a local replica of the entire data set. It is divided into a *workspace* directory (which resides on the normal file system, representing a specific revision or commit) and a proprietary database that stores a complete history of all files and their versions, tracing their creation, deletion and content modification over time. Thus, VCSs are optimistically replicated systems, because users can work freely on their workspace copy in isolation. The synchronization process is triggered manually by the user, by pulling new changes from the server, merging those with the user's own changes. Most systems support automatic, syntactic merging of text files using *three-way merging* (see section 2.2.2.1). They are often deployed in software development and other text-data-focused environments. While file systems (and file synchronizers working on those) only store the latest version of each object (such that after deleting a file it is typically lost), the monotonically growing database of a VCS, which may consume an excessive amount of disk space over time, is considered a feature.

2.5.2 Techniques for detecting causality and concurrency

When optimistically replicated systems exchange updates, various techniques exist to detect conflicting vs. non-conflicting updates and how to merge them. When a system allows replicas to be arranged in an *arbitrary* topology (e.g. peer-to-peer), the update detection (and merging) mechanisms need to be able to detect whether any two updates are *truly* conflicting, i.e., it has to reject *false conflicts* that arise because updates were received out-of-order. This requires knowledge about *causality*, i.e., which updates precede others. Attaching a single time stamp to updates u_i, such as a real-time clock value or even Lamport's logical clock [Lam78] is not sufficient, because $timestamp(u_1) > timestamp(u_2)$ does not imply that u_2 causally precedes u_1 [SM94]. Numerous approaches exist to determine causality from attaching meta-data to transmitted operations or state, each with different advantages and disadvantages. Examples include Version Vectors [Par+83] (and derivative works such as Concise version vectors [MT07]), Vector clocks [Mat+89], Interval Tree Clocks [ABF08], Vector time pairs [CJ05], Hash histories [KWK03] and many others.

We note that the file synchronizer we are building in this work does not require above mechanisms, for there are only *two* replicas being synchronized. We use a star topology with one server and many clients. The merge procedure run in the client is unaware of the identity of other clients. Instead, the updates detected in the server replica are a serialization of the updates of *all* other clients. Because we use state-based differencing, each artifact can only be affected by a specific type of update at most once (e.g. a file can be detected as *moved* only 0 or 1 times). Thus, causality is implied and does not need to be explicitly recorded.

2.5.3 Distributed File Systems

A distributed file system (DFS) is defined as a system that allows *different* (several) machines to share a *common* filesystem [LS90]. DFSs such as AFS [Mor+86] and its descendant Coda [Sat+90], Locus [Wal+83] and descendants such as Rumor [Guy+99] and Ficus [Rei+94] store large amounts of data, structured in a hierarchical name space, distributed among multiple servers. To users who are connected to one or more servers accessing the namespace is done transparently, i.e., clients (and users) are unaware of the distribution and actual server topology. Typically support for this kind of transparency is integrated into the client's operating system kernel. The goal is not that client machines like desktop workstations have a complete replica of the data, since they aren't considered to be fault tolerant nor are their disks large enough [Sat+90]. Some works offer caching functionality to clients on which they can operate, but DFSs typically consider such modes of disconnected operations to be undesired. Therefore, synchronization plays a role not so much between client and server, but between servers. Servers run special, homogeneous DFS-software with *tight* coupling between the physical file system on the server and the DFS, making it possible to lock the DFS during synchronization. In conclusion, DFSs differ from typical file synchronizers because: (a) they don't necessarily fully replicate the data on the clients, (b) they are not fully optimistic in disconnected mode (between client and server), and (c) the client OS is extended on kernel or driver level.

2.5.4 Distributed databases

Database systems, such as relational databases or NoSQL key-value stores, also benefit from optimistic replication and thus need to handle concurrent updates and resulting conflicts. Because the data model and set of operations is typically limited to rows and columns (relational) or keys (NoSQL) with *insert*, *delete* and *update* operations, the set of conflicts is smaller than for file systems. However, in practice complexity is large, because databases have many additional operations, such as index-operations or transactions, which complicate concurrency issues.

Industrial and academic systems are divided into two camps, where one focuses on applications that require *strong consistency* (here database engines attempt to maximize performance/scalability), while the other foregoes strong consistency for availability (CP vs. AP, see *CAP theorem*, section 2.1.2). Database systems favoring consistency provide replication mechanisms that either use a *multi*-master[27] approach with *pessimistic* concurrency control (e.g. Two-Phase Commit), or use a *single* master that optimistically (i.e., asynchronously) replicates data to multiple slaves. To improve scalability techniques such as *sharding*[28] are applied. Examples for these kinds of systems are MySQL[29], MongoDB[30] and Postgres-R [KA10]. In the field of traditional relation databases there are also exceptions, such as Oracle[31] which offers *optimistic*, *multi*-master replication. However, Oracle's conflict resolution is best efforts, without guarantee for convergence. If conflicts occur that cannot be automatically resolved, it is the job of the database administrators to resolve them (man-made consistency). Aside from relational systems, several NoSQL key-value stores have emerged which are specifically designed for optimistic replication, such as Riak [Bas18] (based on CRDTs), CouchDB[32], Amazon Dynamo [DeC+07] and Cassandra [LM10]. These systems use variants of causal histories or version vectors (see section 2.5.2) to detect concurrent, conflicting updates. Conflict resolution either happens within the database engine, or is delegated to the application. In the former case, only simplistic resolution options are available, such as Last-writer wins (LWW), where each update is assigned a timestamp and the update with larger timestamp wins [JT75]. In the latter case the database stores multiple conflicting versions and defers conflict resolution to the application. For instance, the read operation may indicate that there are multiple conflicting versions to the first client that reads data. The client can then solve the conflict, with or without involving the user.

2.5.5 SyncML / OMA DS

SyncML (Sync Markup Language, known as OMA DS[33] since 2002, Open Mobile Alliance Data Synchronization) is an open standard for data synchronization. It specifies both the syntax and protocol for a pair-wise synchronization of several types of data-collections between a client and a server. While originally designed to synchronize a *mobile* device with a server, OMA DS is nowadays also used for other kinds of devices, such as desktop PCs. Different synchronization types are supported, e.g. two-way or one-way synchronization, as well as various data types such as email, contacts, calendars, etc. The OMA DS protocol specifies how to uniquely identify data objects and how to detect which objects need to be transferred (via synchronization anchors). The synchronization is triggered by a client, sending its changes to the server and requesting changes that happened on the server. Conflict detection and resolution is done on the server, but OMA DS does not specify any particular approach.

While synchronization of files and folders is possible since version OMA DS 1.2[34] the support is optional

[27]Master nodes are allowed to perform write operations, slaves only read operations.

[28]In simplified terms, sharding in a relational database refers to storing some rows (data objects) in one master node, and some other rows in another master node. Thus, there are *multiple* master using asynchronous replication, but they are in charge of different data ranges, which avoids conflicts.

[29]See `https://dev.mysql.com/doc/refman/8.0/en/replication.html`, retrieved July 21, 2019.

[30]See `https://docs.mongodb.com/manual/replication/`, retrieved July 21, 2019.

[31]See "Oracle Advanced Replication" `https://docs.oracle.com/cd/E18283_01/server.112/e10706/toc.htm`, retrieved July 21, 2019.

[32]See `http://docs.couchdb.org/en/stable/replication/index.html`, retrieved July 21, 2019.

[33]See `http://www.openmobilealliance.org/wp/` for further details, retrieved July 21, 2019.

[34]See sections 6.10.3 and 6.11 of `http://www.openmobilealliance.org/release/DS/V1_2_2-20090319-A/`

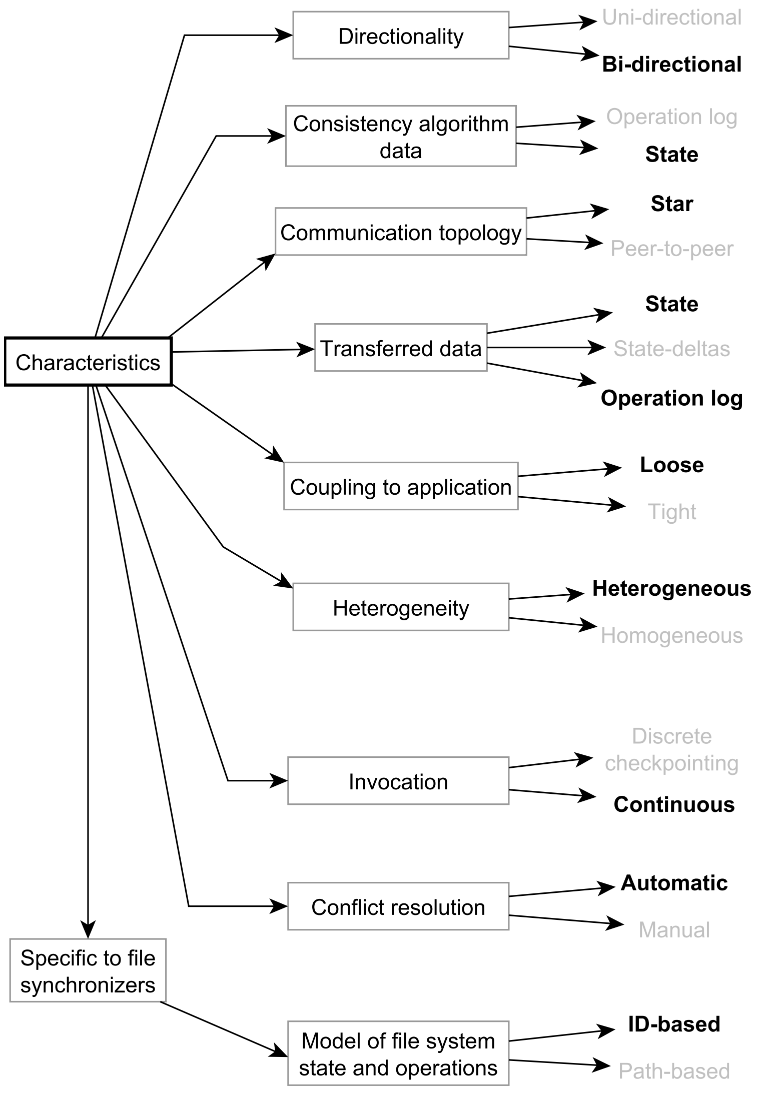


Figure 2.5: Characteristics chosen for our work

and only offered by few client and server implementations[35]. Just like with any other data type, *file* data is embedded into the XML-encoded messages, using approaches such as *base64*, which adds ~ $\frac{1}{3}$ of the file size. The size of each SyncML message is also limited (to avoid that parsing of a, say, 100 MB XML file becomes problematic due to memory limitations), typically to 10s or 100s of kilobytes. Consequently, larger files are split into a potentially large number of messages, which leads to a slower, inefficient synchronization.

2.6 Conclusion

The goal of this thesis is to build a file synchronizer for end-users that can synchronize a variety of storage systems without the need to modify their code-base. From our related work analysis we now con-

OMA-TS-DS_Protocol-V1_2_2-20090319-A.pdf, as well as http://www.openmobilealliance.org/release/DS/V1_2_2-20090319-A/OMA-TS-DS_DataObjFile-V1_2_2-20090319-A.pdf and http://www.openmobilealliance.org/release/DS/V1_2_2-20090319-A/OMA-TS-DS_DataObjFolder-V1_2_2-20090319-A.pdf, retrieved July 21, 2019.

[35]See "Files" column of implementation tables on https://en.wikipedia.org/wiki/SyncML, retrieved July 21, 2019.

clude which trait our synchronizer should have for each of the characteristics we built in section 2.2.1. An overview is shown in figure 2.5.

- **Directionality:** we choose a *bi-directional* approach, because it avoids issues of uni-directional approaches discussed in subsection 2.2.1.1, and because it fits well to using the *star* communication topology (see below).
- **Heterogeneity:** as per our requirement, the synchronizer we build is a *heterogeneous* synchronizer that transforms implementation-specific file system models into a slightly different, internal model that is compatible with several specific models.
- **Consistency algorithm data:** We adopt the approach of related work such as [UFB10; Bao+11; Li+12a; Li+12b]. They use *state* together with a computed list of operations determined from comparing a cached state with the currently determined state of the file system. As chapter 4 will show, the *state*-based approach is more feasible in practice, because operation logs are not available on many real-world storage systems, and because well-understood mechanisms such as three-way-merge can be used. From the seminal work of [BP98] we adopt the generic three-stage processing pipeline: update-detection, reconciliation and propagation.
- **Communication topology:** the goal is to support synchronization for a large number (that is, two or more) of replicas/users. We choose the *star* topology (over peer-to-peer) with a central replica for multiple reasons:
  - Pair-wise synchronization between a client and a central replica is typically easier to implement than peer-to-peer based approaches. For instance, the synchronizer does not need to implement membership management, as this is already offered by the storage system APIs if necessary[36]. Also, peer-to-peer approaches have to establish causality between updates, e.g. using *version vectors* (or similar mechanisms as discussed in section 2.5.2), dealing with modifying these vectors in case new members are added.
  - Peer-to-peer systems require maintenance of additional meta-data about files and directories (e.g. version vectors), which consume storage space.
  - With the increased popularity and availability of cloud storage, average users are now well familiar with setting up a pair-wise synchronization between their local disk and a cloud storage. Setting up pair-wise synchronization requires less expertise than setting up peer-to-peer systems.
  - The central replica resides on a highly reliable server that is always online (assuming network connectivity). This allows two devices *A* and *B* to synchronize their changes even when they are not online at the same time. In contrast, even peer-to-peer systems that support *epidemic propagation* where device *A* can get its changes to device *B* via device *C*, require that such a device (*C*) must have been online and connected to A and B *coincidentally* during the synchronization of A <-> C and B <-> C.
- **Coupling:** our synchronizer has a *loose* coupling to the file system, because it is a *heterogeneous* synchronizer. The file system is not aware of its synchronization.
- **Transferred data:** we use a *hybrid* approach. As will be shown in section 4.2.1, all storage systems provide transferring state information. However, always requesting and transferring state is expensive. Therefore we use online or offline change detection for those storage systems that support it.
- **Invocation:** we prefer the *continuous* mode over discrete checkpointing because it resembles a "set it and forget it" approach suitable for end-users. This is also the prevalent mode in today's industrial file synchronizers.

[36]For example, a WebDAV-based file system offered by ownCloud would require the user to authenticate over HTTP's authentication mechanism. The backend providing the WebDAV API, here: ownCloud, already provides membership management functionality, such as registration.

- **Conflict resolution:** we choose *automatic* over manual resolution because it is in line with the automatic *invocation*, and because automatic resolution saves time and effort in case the resolution was appropriate. We refer to chapter 5 on page 65 for more details.
- **Model of file system state and operations:** as shown in section 2.3 on page 14 there are a number of choices to make for a file system model used internally by a file synchronizer. As chapter 3 will show, we will choose a model where objects are unique identified by an ID, are linked into exactly *one* parent directory with one name, supporting the operations *create*, *delete*, *update* and *move*.

The next chapter will further clarify why our choice for the file system state and operations model is suitable.

Chapter 3

File systems - analysis and definition

In section 2.2.1.6 we provided a *general* introduction to heterogeneity. In this chapter we take a closer look at heterogeneity in the context of *file systems*. Just like many other industrial file synchronizers, we aim to build a synchronizer that supports the file system APIs of popular end-user operating systems it runs on, as well as the APIs of popular remote file systems, maximizing practical use. This enables users to continue using existing file systems, without the (expensive) migration to a homogeneous system. The disadvantage of such a file system agnostic approach is that we need to build an *internal* file system model used in the synchronizer that is as compatible as possible with every file system the synchronizer aims to support. This is challenging because no two file systems are exactly equal.

We start in section 3.1 where we analyze file system capabilities relevant to file synchronizers. We use the term *capability* for a specific characteristic of a file system, such as namespace limitations or the way object relationships are modeled. Their traits may be different (heterogeneous) between any two file systems. If the synchronizer developer ignores or overlooks a capability, this impairs the usability of the system because of bad side effects that occur during synchronization. For example, if a developer overlooks that a file may not be named “aux” on Windows, the Windows implementation will run into unexpected loops or errors while trying to synchronize such a file, which was synchronized successfully by the macOS implementation. We have observed several instances of such side effects in practice in leading industrial synchronizers. The result is either just a divergence of the file systems, or worse, data loss. From our analysis we formalize the internal file system $\mathscr{F}$ using First Order Logic in section 3.2, which we use in the remainder of this work.

3.1 Capability analysis

This section corresponds to our published work [SP19]. It provides an in-depth analysis of six file system capabilities relevant to file synchronizers, which we discovered while implementing and technically evaluating our own file synchronizer implementation. We discuss one capability per subsection. We first state its significance for the user, followed by an analysis, then extract similarities that manifest in the file synchronizer’s internal model and finally give advice how file synchronizers can handle incompatibilities, if applicable, with the goal to avoid data loss whenever possible.

To find capabilities we sample different *types* of file systems, selecting representative implementations often used in today’s computing landscape. We examine **Windows** version 7-10 (NTFS) and **macOS** version 10.11-10.13 (HFS+ and APFS) APIs because these are the most widespread end-user operating systems at the time of writing. Our findings also transfer to UNIX and therefore to both file servers (e.g. network-attached storage) and mobile devices such as smartphones. We consider **WebDAV** [Dus07] which is widely available as interface for proprietary as well as open-source Internet (cloud) storages. **Dropbox** (HTTP API v2 [Dro17]) is chosen as a representative for widespread cloud storages. **BSCW Social** [Orb18] is a representative for groupware systems commonly found in academia, a system that originates from the CSCW community [BHT97; JP14]. We note that many more file systems exist in each category, such as ownCloud [own19] and OneDrive [Mic19] for cloud storages or CDMI [Sto15] for

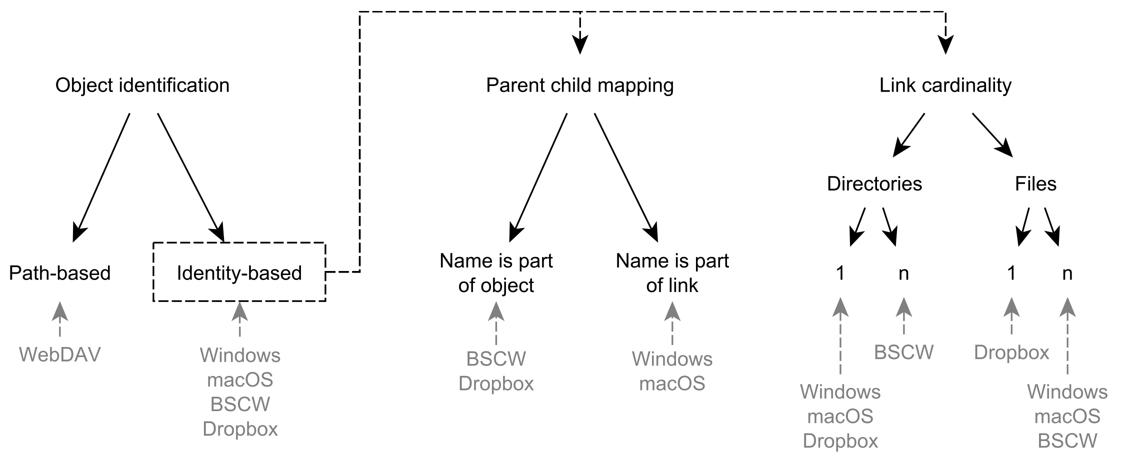


Figure 3.1: Analysis of object identification and namespace to object mappings

generic interfaces. However, they are either less used than the systems we analyzed, or their degree of use is not documented. Our decision to reduce the number of examined systems to a representative set improves the clarity of the results.

We note that the comparison done in this section is *informal*, as this allows the provision of immediate results for a large selection of file systems. Some online resources such as [Cra08; Wik17] also provide informal comparisons. Apart from [JPV02], an unfinished manuscript by the authors of [BP98], there is no related scientific literature to the best of our knowledge that provides an in-depth discussion of the capabilities of file systems. Providing a *formal* comparison, while interesting, is hindered by the fact that real-world file system specifications, such as POSIX, are only formulated informally. While there are a few academic works such as [Rid+15] which extract exhaustive formal specifications for real-world implementations, most main stream file systems are not covered yet.

3.1.1 Physical object & namespace mapping

The *namespace* is the user-facing side of a file system. It consists of a hierarchical set of *paths*, where a path is a notation for addressing a specific *object*. A path is a sequence of *names*, where names are simple strings. Hierarchy levels of a path are separated by a *separation character*, such as '/' or '\'. File system implementations differ in their approach how objects are identified, physically stored and how the mapping between namespace and objects works.

3.1.1.1 Significance

From the user's perspective the synchronizer translates a prefix of the synchronized namespace between the local and the remote replica, e.g. *'C:\SyncFolder'* to *'https://server.com/synced'*. Users expect that the local disk's and the server's namespace match exactly. However, due to technical limitations (analyzed below) this is not always possible. A synchronizer that is aware of incompatibilities should find a suitable way to inform the user about namespace mismatches [Dou96].

3.1.1.2 Analysis

An overview of the analysis is shown in figure 3.1.

We first classify whether file system objects (files, directories, etc.) can be identified uniquely (e.g. after moving them) by a persistent identity, or whether only the path is available [TSR15]. Windows provides the *file index*, and macOS or UNIX systems provide *inode* numbers. The WebDAV protocol specification leaves it up to the implementation how to identify objects beyond their path. Some implementations may internally provide an ID for each resource, but a HTTP client cannot rely on its exposure[1]. BSCW and Dropbox identify each object by a unique object ID, assigned upon creation by the server.

For identity-based systems, two further classifications are appropriate, because an object with a specific ID may be accessible from one or more paths. In practice the cardinality varies per object type, s.t. Windows or macOS forbid more than one link to a *directory* to prevent cycles to occur in the tree[2]. Some systems model the parent child relationship s.t. each directory has a list of $(name, id)$ tuples of its immediate children (name of the objects is part of the *link*), whereas others store the name as part of the object and each directory maintains a simple list of immediate child IDs.

Two more aspects not covered in figure 3.1 are that the invariants of each file system need further examination. A file system may or may not allow two sibling objects to have the same name, and it may use a case-sensitive or case-insensitive comparison while enforcing this invariant (case-sensitivity is further discussed in section 7.2.1 on page 135).

3.1.1.3 Derived unified model

To derive the internal file system model we suggest the following approach:

- If one or more file systems are *path-based*, either let the internal model be path-based too, or emulate IDs by generating them on the client, setting IDs as custom meta-data, if the file system API supports it (e.g. WebDAV PROPPATCH, see section 9.2 of [Dus07]).
- When the parent child mapping varies, let the name be part of the object.
- If link cardinality varies, use the smaller (1) cardinality.
- When invariants vary, enforce the one that is most strict.

Applying these guidelines to the set of examined file systems yields an *ID*-based file system where each object is linked exactly once, the name is part of the object and sibling nodes may not have the same name, being case-insensitive.

3.1.1.4 Advice for handling incompatibilities

When a file synchronizer encounters an incompatible mapping at run-time, e.g. if a specific file exists at multiple paths but the internal model limits file cardinality to 1, we suggest the synchronizer either stops synchronizing, asking the user to fix the situation, or to automatically add the affected paths or IDs to an *ignore list*. Numerous industrial synchronizers provide such an ignore list that users can fill with paths to files or directories they want to exclude from synchronization. We suggest that this list can also be manipulated by the reconciliation algorithm automatically to handle compatibility issues, notifying the user in such an event. For certain traits, workarounds may be possible. For example, junctions (Windows) and symbolic links (macOS) may be used to allow a N-cardinality for directories. The synchronizer needs to choose one link as primary one and use junctions or symbolic links for all other paths, updating them in case the primary link (and its path) changes.

[1] As an example, our analysis has shown that *ownCloud's* WebDAV implementation provides the unique ID of an object, while Apache's *mod_dav* does not.

[2] Cycles cause problems for programs that iterate over the file system namespace, such as backup or synchronizer tools. If multiple links for directories were allowed, these tools wouldn't be able to easily detect them. To detect them anyway, they would have to keep a *visited* list, containing IDs and corresponding paths, which would be memory-intensive for large namespaces. In case of a backup tool, the inability to detect directory links would artificially inflate the number of files and dirs copied to the backup medium. In the worst case, this number becomes *infinitely* large, if links formed a *cycle* in the namespace. To enable users to conveniently access a directory from multiple other locations, they can instead use symbolic links or junctions, which are discussed in subsection 3.1.2.

	Files	Directories	Device files	Symbolic links	Others
Windows	✓	✓	✓ [3]	✓	Link files, junctions
macOS	✓	✓	✓	✓	–
WebDAV [Dus07]	✓	✓	X	X	–
BSCW Social [Orb18]	✓	✓	X	X	Plethora of object types other than regular files and dirs, such as: contact list, forum, calendar, URL, poll, voting, appointment scheduling, project, ...
Dropbox [Dro17]	✓	✓	X	X	–

Table 3.1: Supported object types per file system

3.1.2 Supported object types

Files and directories are the two object types offered by all examined file systems. [JLP13] show that even in groupware systems such as BSCW which offer many additional object types, the majority (90%) of user interaction takes place with these two object types. A file system may also support other object types that are incompatible with other systems.

3.1.2.1 Significance

When an object available on one file system is unavailable on the other one, its omission in the namespace, which is a loss of information, will confuse the user.

3.1.2.2 Analysis

While all examined file systems offer files and directories, there are several other types supported by just a subset of file systems, e.g. device files or symbolic links on macOS and Windows, or special types like contact lists, calendars or URLs on BSCW. Table 3.1 provides an overview. Some elements in table 3.1 are elaborated below:

- **Device files**: files mapped into the namespace that allow programs to communicate with devices. Examples are block or character devices (terminals, printers or other physically installed devices) or communication channels such as named pipes or sockets. Since UNIX systems have the mantra that “everything is a file” [4], the file system’s namespace is used to list and work with these devices.
- **Symbolic links**: special files that link to a target path in the file system. The target is typically a (real) file or directory. The file system APIs (and file managers) automatically interpret these files and follow the redirection to the target path. Symbolic links differ from hard links in multiple points:
  - They are distinct files with their own ID. Programs working with the file system can identify when a path is a symbolic link. Consequently, tools such as a backup programs can avoid backing up the same physical files and dirs multiple times, by detecting symbolic links and refusing to follow them.

[3] Unlike for macOS, devices aren’t mapped in ordinary file system namespace, but are accessible from a dedicated namespace, such as \\.\DEVICENAME

[4] See `https://web.archive.org/web/20120320050159/http://ph7spot.com/musings/in-unix-everything-is-a-file`, retrieved July 21, 2019.

  - They can link to any target on any volume (whereas hard links pointing to a specific file always exist on the volume of that file).
  - When the target moves or gets deleted, the symbolic link breaks.

- **Link files**: similar to symbolic links: binary files with the .lnk extension that point to a user-defined target path. File managers automatically follow the redirection, while file system APIs typically don't!

- **Junctions**: a special type of symbolic link that causes an existing, empty directory to redirect to another target directory. They are implemented via *reparse points* [5], a mechanism that attaches meta-data to the source directory, causing file system APIs and file managers to automatically follow the redirection to the target.

3.1.2.3 Derived unified model

By taking the intersection set of the available object types of each file system, the internal model should consist only of files and directories. We suggest to ignore other object types because they are specific to that file system and cannot be meaningfully viewed or manipulated on other systems that do not support them.

3.1.2.4 Advice for handling incompatibilities

We propose a similar handling as for mapping issues (section 3.1.1) where the synchronizer either stops or adds affected objects to the ignore list automatically, notifying the user about this action. A workaround is to create proxy objects, such as '.url' files, that allow the user to see the existence of the corresponding objects, redirecting the user to the respective location on the other file system in case she opens the proxy object.

3.1.3 Operations and atomicity

File system APIs offer many operations to both *query* the current state of the file system (e.g. listing a directory's content) or to *manipulate* it. In the update detection stage a state-based file synchronizer relies on the *query* operations to extract the current state. At the final *propagation* stage, the synchronizer needs to transform the scheduled abstract operations (which equalize both file systems) to concrete *manipulation* operations of each file system. This is challenging because the exact operations, their preconditions and their degree of atomicity[6] vary.

3.1.3.1 Significance

A user expects that operations she applied to her local file system are consistently applied to other file system by the synchronizer. Users also expect the synchronizer to avoid inconsistent states while synchronization is active or was interrupted. Not handling related issues causes confusion (e.g. attempting to open a partially transferred file) or additional work (such as manually cleaning up inconsistent files and directory structures) for the user.

3.1.3.2 Analysis

Each examined file system offers operations to *query* the current state. This allows to list the names of immediate children of a directory and to retrieve both system-generated and arbitrary meta-data information about objects, such as their ID or the timestamp of last modification. There are slight variations in the query operation signatures between each file system, but these are merely an implementation detail. When considering *manipulation* operations, all file systems offer operations to create or delete empty directories, or to move an object. However, there is significant variation in the availability and

[5]See `https://docs.microsoft.com/en-us/windows/win32/fileio/reparse-points`, retrieved July 21, 2019.
[6]We refer to *atomicity* as known from database systems, see also section 1.3.4 of [EN15].

	Windows	macOS	WebDAV	BSCW	Dropbox
Create empty dir	✓	✓	✓	✓	✓
Create file	Initially empty	Initially empty	With content	With content	With content
Move file or dir	✓	✓	✓	✓	✓
Copy file	✓	✓ (UNIX: X)[7]	✓	✓	✓
Copy dir (with sub-objects)	X	✓ (UNIX: X)	✓	✓	✓
Delete file	✓	✓	✓	✓	✓
Delete empty dir	✓	✓	✓	✓	✓
Delete non-empty dir	X	✓ (UNIX: X)	✓	✓	✓
Create file hard-link	✓	✓	X	✓	✓
Create dir link	X	X	X	✓	✓
Create other file types	Symlinks, Junctions	Symlinks	X	Calendar, contact list, URL, ...	X
Mount volume	✓	✓	X	X	X

Table 3.2: Namespace manipulation operations per file system

atomicity of operations used to create or update files, or to delete non-empty directories. For instance, BSCW allows to atomically create non-empty files or delete non-empty directories, while Windows does not. Another observation is that desktop file systems like Windows and macOS offer *mount* operations which create a mount point that establishes a transition between volumes. An overview is shown in table 3.2.

3.1.3.3 Derived unified model

A user would expect a file synchronizer to be capable of a set of operations the user also knows from using the file manager. An exemplary list could be as follows:

- *createdir(path)* creates an empty directory at *path*
- *deletefile(path)* deletes the file at *path*
- *deletedir(path)* deletes the directory and all its children at *path*
- *move(source, dest)* moves an existing object from *source* to *dest*
- *transfer(source, dest)* transmits a *file* located at *source* on the source file system to *dest* on the destination file system, to create a new file or update an existing one

[7] On a UNIX storage, files are copied by manually copying data-chunks by reading from the source file and writing to the destination file.

To not leave either file system in an inconsistent state, every operation is expected to succeed or fail atomically. Optionally, a *copy file* operation can be used to copy a file on the destination file system in case it is feasible to detect exact copies of files on the source file system, e.g. by using checksums.

3.1.3.4 Advice for handling incompatibilities

All discrepancies we found between concrete file system operations and the ones presented above result from varying degrees of atomicity, which can be solved in the following ways:

- *deletedir(path)*: if a file system does not offer an atomic, recursive implementation, we suggest to first call *move(path, temp)* where *temp* is a path outside of the synchronized namespace, but on the same volume. This move operation succeeds (or fails) atomically and *appears* as an atomic delete operation to the synchronizer. Next, perform a *post-order* traversal of *temp*'s sub-namespace, deleting first files then directories.
- *transfer(source, dest)*: if the destination file system's operation is not atomic, we propose to execute *transfer(source, temp)*, i.e., write transferred data to a temporary location *temp* that is outside the synchronized namespace but also on the same volume. Once finished, perform *move(temp, dest)* on the destination file system.

Finally, file synchronizers which detect move operations via the object's ID should be aware of *mount points* within the synchronized namespace. IDs are only unique within a volume. However, a mount point establishes a transition between volumes. When the user performs a conceptual *move(source, dest)* operation where *source* is on volume *A* and *dest* on volume *B*, the synchronizer will incorrectly detect a *delete* operation for *source* and a *create* operation for *dest*. We therefore suggest that synchronizers detect mount points[8] and either reject them (by stopping synchronization) or automatically adding them to the *ignore list*.

3.1.4 Namespace limitations

Although the general namespace allows each name to consist of an arbitrary sequence of *Unicode* characters, a file system may pose limitations on the namespace, affecting paths or the names of a path, usually for technical or historical reasons.

3.1.4.1 Significance

When a user attempts to create an object with a name that violates a namespace limitation, the file manager (or web interface) prevents the creation and provides *immediate* feedback how to fix the name. When using file synchronization, the chosen name may be accepted by the source file system API, but may violate a limitation of the destination API. The file synchronizer discovers this issue after a (possibly large) delay which surprises the user, because to her the creation of the object initially appeared to be successful. Furthermore, users will be confused if objects exist on one system but not the other one due to a limitation that affects only the latter system.

3.1.4.2 Analysis

Table 3.3 illustrates which limitations each file system's API is affected by. The following list provides a

[8]This can be achieved by querying the *device ID* of an object on UNIX or macOS, or retrieve the *volume serial number* of Windows files.

	Windows	macOS	WebDAV	BSCW	Dropbox
Case-sensitive[9]	X	X (default) / ✓	Depends on impl.[10]	✓	X
Unicode normalization sensitive	✓	X	Depends on impl.	✓	Requires NFC input
Reserved characters	**/**[11], <, >, :, “, /, \\, | ?, *	**/**, :[12]	Only **/** (when other characters are encoded*)	See WebDAV, Windows’ reserved characters	See WebDAV, Windows’ reserved characters
Reserved names	**.**, **..**, ’ ’, CON, PRN, AUX, NUL, COM1-COM9, LPT1-LPT9[13]	**.**, **..**	–	Windows’ reserved names	**.**, **..**
Max. path length	32767	1016[14]	–	–	–
Max. name length	255	255	–	–	255
Other limitations	Names ending with ’.’ or space[15], Short file names	Violates Unicode normalization preservation	–	–	–

Table 3.3: Namespace limitations per file system

*WebDAV character encoding: since all objects are addressed by URLs, subsection 2.2 of the URL specification (RFC 1738) addresses that in URLs there is a set of characters that are always *safe* to use, a set of characters that are always *unsafe*, and that each *scheme* may have a set of *reserved* characters (in designated positions of the URL) with special meaning. *Safe* characters can be used in a name as is. It is necessary to *encode* all *unsafe* characters as well as those *reserved* characters used outside of their normal context of the used scheme. In our case, the HTTP(S) scheme is used, therefore the “?” character (whose typical meaning is to start a *query string*) needs to be encoded when it should be used without that meaning, but, say, as part of a dir that should be created. The encoding process converts ASCII characters into a %<ascii hex representation> (e.g. *#* is encoded as *%23*), non-ASCII characters (such as arbitrary Unicode characters) are converted to a UTF-8 byte sequence, of which each byte is encoded as %<byte value> (e.g. the character “ö” with UTF-8 code position U+00F6 has a hexadecimal byte representation of *c3b6* and is thus encoded as *%C3%B6*).

summary of our findings. We refer to the respective file system documentation for further details[16].

- A file system may reserve a set of *characters* from being used in object names, either at any position, or only in specific positions. *Forward slashes* are forbidden in all examined systems, as they separate names in a path. Windows reserves the most characters, and other systems such as BSCW or Dropbox have adopted Windows' set of reserved characters and names for compatibility reasons.
- Similarly, some systems reserve a set of *names*, such as "." or "..". Windows reserves a large set of names such as "CON" or "PRN" for historical reasons and also reserves *short file names* [Mic18b] in case a longer file name already exists (for example, given a directory named "project report", creating an object at "projec~1" is forbidden on volumes with short file name creation enabled).
- Many systems impose a maximum length of names and paths. Often names are limited to a length of 255 characters. Shorter path lengths (such as macOS with 1016 characters) also cause issues, e.g. deep directory hierarchies being inaccessible.
- While all examined systems use the Unicode alphabet with some form of encoding (e.g. UTF-8), not all systems *preserve* the *normalization form* (such as NFC or NFD[17]) of characters. For instance, the HFS+ file system on macOS does not preserve a large set of input characters but converts them to an NFD-like form.
- *Case-sensitivity* may vary between two file systems. By default, the *examined* systems are all case-insensitive. However, others such as the UNIX file system, are case-sensitive! We found all systems to be case-*preserving*.
- In rare instances the file system APIs behave deceptively. They accept a name, seemingly execute successfully, but actually change the name internally. This is problematic for file synchronizers, as the next update detection phase will find an unexpected name and assume that the object was moved by the user. One example is the Unicode normalization conversion of HFS+ volumes mentioned above, another is Windows which silently strips trailing spaces/dots from a name during execution.

3.1.4.3 Derived unified model

For each limitation of file systems *A* and *B* we propose to take the one that is stricter (i.e., provides a more *narrow* set of characters and names) and let the file synchronizer apply it to the file system with the weaker limitation. For *reserved characters* or *names* this means to apply the *union* of the character/name sets to both *A* and *B*. For *length* limitations, the shorter length is stricter. Regarding case-sensitivity, case-insensitivity is stricter than case-sensitivity.

[9]Storage systems with X are case-insensitive. However, *all* of them are case-*preserving*, i.e., when creating a file "/aX", it is also stored with that upper/lower-casing.

[10]According to subsections 5.1 and 5.2 of [Dus07] , the namespace can be case-sensitive or case-insensitive, as chosen by the implementation.

[11]Forward slashes are generally not allowed in file or dir names, because they separate the names within a path.

[12]The : character in a file/dir's name is shown as forward slash in Finder, but the UNIX layer stores it as :, which can be verified using the *ls* command.

[13]Extensions thereof are also reserved, like "CON." or "CON.ext"

[14]This value was determined experimentally. The official documentation and other sources incorrectly state that there is no limit.

[15]Windows APIs do accept paths end with dots or spaces, but they are stripped automatically. For instance, creating a new dir "test.." will create a dir named "test".

[16]See e.g. [BMM94], [App04], [App17] or [Mic18b].

[17]See `http://unicode.org/reports/tr15/`, retrieved July 21, 2019.

3.1.4.4 Advice for handling incompatibilities

We suggest a file synchronizer takes one of the following approaches when encountering paths that are incompatible w.r.t. the unified limitations:

1. Stop synchronization, ask the user to manually rename objects
2. Automatically rename objects to establish compatibility
3. Automatically add incompatible objects to the ignore list

While approach (1) is easy to implement, it is labor-intensive for the user. In case the stopped synchronization goes unnoticed, and if it remains in that state for extended periods of time, this increases the chance for conflicts. Approach (2) mitigates this problem, but automatic renaming can cause issues when the affected objects belong to a naming scheme of a third-party application. Such applications may stop working once these files and directories no longer correspond to the expected naming scheme. The last approach fixes the issues of the two ones but requires the implementation of the aforementioned ignore list.

3.1.5 Meta-data

Meta-data provides further information about objects. It is not stored as part of the object, but at a separate location.

3.1.5.1 Significance

When meta-data stored on one file system is incompatible with the other file system, a synchronizer must skip their synchronization or perform a conversion. This type of data loss negatively affects the user, because she cannot access meta-data available only on the remote file system during an offline period.

3.1.5.2 Analysis

Each file system provides a diverse set of meta-data. Some meta-data are *attributes* managed by the file system, others can be changed by a client application, such as a file synchronizer. Some systems offer one or more APIs to write *custom* meta-data, e.g. *Extended Attributes* and *Alternate Data Streams* on Windows, or *xattr* and *Resource forks* on macOS. Appendix A.1 provides further details. The following meta-data is available on *all* file systems:

- Object type (file, directory, ...)
- File size (for files)
- Timestamp of creation and last modification

3.1.5.3 Derived unified model

All file systems support the retrieval of meta-data that is necessary to extract their state, such as the object's type or the last-modified timestamp. In case a file synchronizer models the file system using IDs, all file systems except for WebDAV automatically generate and provide unique IDs. For WebDAV we propose that the *file synchronizer* generates globally unique IDs (GUIDs) when creating objects on a WebDAV file system, assigning the GUID via the *PROPPATCH* command.

3.1.5.4 Advice for handling incompatibilities

Some meta-data, such as *attributes*, are system-specific and often lose meaning when copied to another file system, especially when it is of different type or located on a different operating system or machine. For instance, synchronizing the *compressed* attribute of a Windows file to the corresponding file on a macOS file system defies any purpose. We find that bypassing meta-data synchronization largely facilitates a file synchronizer's implementation. This also applies to *authorization* mechanisms, such as UNIX *permissions* or the more powerful *Access Control List* entries, which can also be considered to be meta-data, with varying availability and heterogeneity.[18]

The *last-modified* timestamp is an exception. We suggest to synchronize it because it is typically available on each file system, has the same meaning everywhere and users are aware of it when using the file manager. Windows, macOS and Dropbox support overwriting this timestamp. WebDAV-based implementations (including BSCW) protect the timestamp from being modified. Here we propose to set the timestamp as a custom meta-datum instead. A caveat developers need to consider is the variety of resolutions and formats of timestamps.

3.1.6 Locking

Locking allows one user to *exclusively* modify an object on a file system, while all other users are prevented from modifying their own replica of that object.

3.1.6.1 Significance

Locking is an important mechanism that introduces *pessimistic* concurrency control in situations where users expect that conflicts are likely to happen. It avoids conflicts or lost updates. In an example scenario, a user locks a document she exclusively wants to work on for an hour. During this time, other users should be unable to concurrently modify this file, and should be *aware* of this lock while it is set. The information about the lock's existence can be propagated by the synchronizer to other users while they are online. In practice we have not observed locking to play a role for files stored on *local* disks. However, this feature is frequently used in groupware systems such as BSCW, and the transparent handling and awareness of locking behavior is an early requirement for CSCW systems as described in [BR94].

3.1.6.2 Analysis

We analyzed the file systems' locking capabilities to determine whether a file synchronizer can safely protect an object from modification by the local user, because a different user locked the object. We found that some systems such as Dropbox do not offer any locking mechanism. WebDAV and BSCW provide an elaborate locking model, including lock meta-data such as the owner and expiration time.

The locking mechanisms of Windows (*read-only* attribute, *file handle* locking) and macOS (*immutable* attribute, advisory locks via *fcntl*[19] API) are less elaborate. They each work differently and protect other aspects of modification. For example, the read-only attribute on Windows does not protect objects from being moved or renamed, while the immutable attribute on macOS does.

We think that this diversity stems from the fact that each mechanism has a different purpose. On Windows and macOS the read-only/immutable file attribute or handle-based locks were not designed for a multi-user locking scenario. It is our understanding that they exist to allow users (and programs) to protect objects from modification *on the same device*, not across multiple devices. Handle-based locking suffers from *volatile* characteristics[20]. On macOS, handle-based locking is designed for a set of *cooperative* programs and not intended to prevent third-party programs from modifying files. On Windows,

[18]As an example for heterogeneity, macOS and Windows both support *Access Control Lists*, but their implementations vary considerably. Additionally, synchronization of authorization data would require to also synchronize *authentication* data, i.e., user accounts, which introduces additional challenges.

[19]See `http://man7.org/linux/man-pages/man2/fcntl.2.html`, retrieved July 21, 2019.

[20]When the program that owns the handle to an object terminates, the lock is automatically cleared.

handle-based locking has more wide-spread effect than just locking the object itself. It works on a “first come, first served” basis. Even just opening a file for reading already locks it, protecting it against move or delete operations. A file synchronizer may fail to obtain a lock, or inadvertently lock the path of any parent object, which is not desired. In addition, reliable recursive locking of a *directory* is not possible with the mechanisms offered by Windows and macOS.

3.1.6.3 Derived unified model

In case a pair-wise synchronization targets two file systems of *equal* type, such as two WebDAV systems, lock synchronization is feasible. In any other scenario we advise to ignore lock synchronization due to the strong differences in their implementation, making it impossible to meaningfully map one lock type onto another one.

3.1.6.4 Advice for handling incompatibilities

Not synchronizing locks does not necessarily mean that the synchronizer completely ignores locks. Some systems like WebDAV allow the *discovery* of locks (*before* the attempt of modifying a locked resource). Assume a scenario where a synchronizer detects that user 1 updated file f locally, while f is locked on the remote replica by another user 2. The synchronizer may then skip synchronizing f and notify user 1 about the lock’s existence. With additional implementation effort, a synchronizer may also monitor the local user’s opened files and warn her in case she opens a file that is locked by other users on the remote replica. We also suggest to convey the existence of locks by the use of *overlay icons* in the file manager.

If lock discovery is unavailable we propose to treat failed operations like any other permission-related failures, such as failures resulting from prohibitive ACL entries or UNIX permissions. The synchronization may be stopped or the affected object could be skipped. The user should be notified about the problem in either case and be provided with as much available information as possible to fix the problem.

3.1.7 Summary

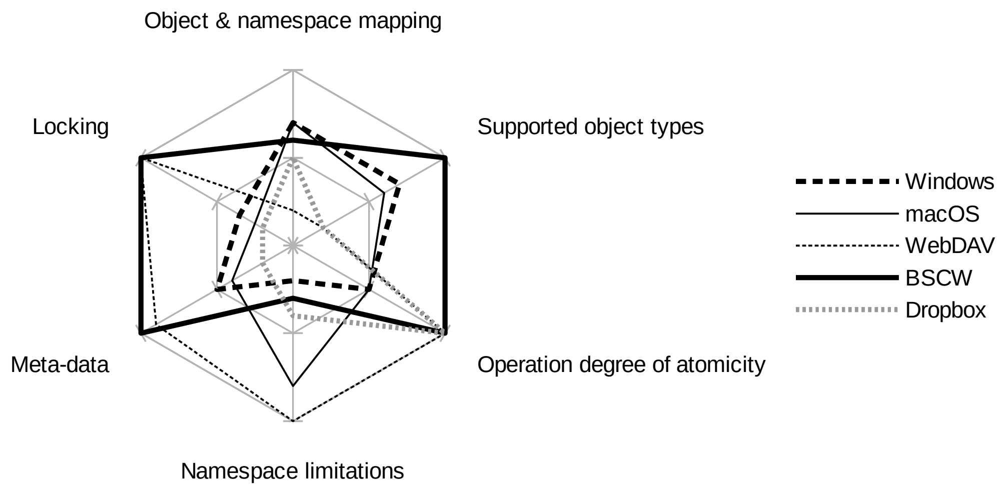


Figure 3.2: File system capabilities overview

A summary of the capabilities of each file system is shown in figure 3.2. This radar chart depicts a rough estimate of the degree of power for each capability from 0% (center) to 100%, based on a technical evaluation that includes information from above sections and appendix A.1. Smaller values indicate less powerful namespace mappings, fewer supported object types, stronger namespace limitations, smaller level of locking, etc. We chose 20% as minimum value only to improve readability. By intersecting the areas of the file systems a synchronizer supports we can derive the degree of limitations of the synchronizer's internal model[21].

3.2 File system model definition

In this section we formally define the file system model $\mathscr{F}$ using First Order Logic (FOL). We built $\mathscr{F}$ based on the *NH-MD* file system from section 2.3.1, while also considering the analysis from section 3.1. We provide exemplary alternative specifications for the *H-All* and *NED-All* file system models in appendix A.2. The rationale of each decision is summarized below:

- Identity-based (rather than *path*-based) because IDs allow to reliably discern *create*, *delete* and *move* operations when using a *state-based* update detection approach. Also, the majority of examined file systems are identity-based.
- Operation support: we detect *delete*, *move*, *create* and file *edit* operations, and we expect their execution to be *atomic*. We consider it an important aspect to preserve and synchronize *move* operations instead of replacing them with *delete* and *create*, for three reasons [RC01]:
  - Improved performance: a *move* operation completes quickly and atomically, while deletion and creation may incur significant overhead, such as payload data.
  - Retention of meta-data, such as access permissions, previous versions or an event history.
  - Improved usability: a user would be confused if she checked the log of the other replica and found that her *move* operations are not shown, but are replaced by *delete* and *create* operations.
- Support for directories (even empty ones). We think that users expect directory support, because directories are also provided by the file managers they regularly use.
- No hardlink support: examining the physical object and namespace mapping from section 3.1.1, we see that to be most compatible with all of the examined real-world file systems, $\mathscr{F}$ must limit each object to exist exactly once in the tree. We choose to model the name as part of the object, rather than of parent-child relation (this choice is arbitrary).
- The supported, synchronized object types are limited to files and directories, because they are common to all examined file systems.
- We do *not* enforce any *namespace limitations* in $\mathscr{F}$'s invariants or operations directly. This allows to build *internal* file system states during *update detection* stage which accurately reflect the underlying, real file system's state (which may have no namespace limitations). As explained later in section 7.2.1 on page 135, our implementation still enforces a set of namespace limitations L, but defers the evaluation and correction of object names until the *reconciliation* stage. L dictates that name-comparisons are case-*in*sensitive and Unicode normalization-*in*sensitive, name lengths are limited to 255 characters and various limitations from Microsoft Windows are applied to *all* file systems, such as reserved characters and names.

3.2.1 File system constituents

We define $\mathscr{F}$ as a set of tuples where each tuple represents an object with a unique ID $i \in I$, parent directory ID $p \in I$ (where I is the set of unique IDs), type $t \in T$ (with $T = \{file, dir\}$), name $n \in \Sigma^+$ (with $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$), *lastmodified* meta-datum $l \in L$ (where L is the set of all valid meta-datum values, e.g. $\mathbb{N}$) and content $b \in B$ (where B is the set of arbitrary byte sequences, including ϵ). That is, $\mathscr{F} \subset I \times I \times T \times \Sigma^+ \times L \times B$, with tuples $(i_k, p_k, t_k, n_k, l_k, b_k)$ where the following invariants hold:

[21]The only exception is *atomicity* where, instead of accepting the union of all limitations, we suggested to *emulate* a higher level of atomicity.

Function	Description
$name(i)$	Returns the name n_k of the tuple where $i_k = i$, or $error$ if no such tuple exists.
$type(i) \rightarrow \{dir, file\}$	Returns type t_k of the tuple where $i_k = i$, or $error$ if no such tuple exists.
$list(i)$	Returns the set of IDs of the *immediate* child nodes for the node with ID i, i.e., the set of IDs of all tuples for which $p_k = i$ holds.
$content(i)$	Returns the binary data b_k of the tuple where $i_k = i \wedge type(i) = file$, or *error* if no such tuple exists.
$lastmodified(i)$	Returns the *lastmodified* meta-datum for the node with ID i.
$id(i,n)$	Returns the ID of the tuple where $p_k = i$ and name $n_k = n$, or $error$ if no such object exists.

Table 3.4: Functions for working with file system IDs

$$t_k = dir \implies b_k = \epsilon \tag{3.1}$$

$$\forall i,j \in I: i \in list(j) \implies type(j) = dir \tag{3.2}$$

$$\forall i \in I: i \notin list(i) \tag{3.3}$$

$$\forall i,j,k \in I: j \neq k \wedge i \in list(j) \implies i \notin list(k) \tag{3.4}$$

$$\exists i_{root} \forall i \in I: i_{root} \notin list(i) \tag{3.5}$$

$$\forall i \in I \setminus \{i_{root}\}: name(i) \neq e \iff type(i) \neq e \iff lastmodified(i) \neq e \iff ancestor(i_{root}, i) \tag{3.6}$$

$$\forall i,j,k \in I: j \neq k \wedge j \in list(i) \wedge k \in list(i) \implies name(j) \neq name(k) \tag{3.7}$$

where helper functions are defined in table 3.4 and e is a shorthand for $error$ in equation 3.6. We additionally define the predicate

$$ancestor(i,j) = \begin{cases} true & j \in list(i) \\ true & \exists k \in list(i): ancestor(k,j) \\ false & \text{otherwise} \end{cases}$$

to express whether the object with ID i is an ancestor of the object with ID j. $\mathscr{F}$ is an arborescence rooted in the well-known object $i_{root} \in I$ with $type(i_{root}) = dir$, where each object exists exactly once. Equation 3.6 expresses that each object must be reachable from i_{root}, while equation 3.7 states that no two siblings may have the same name.

3.2.2 File system operations

The file system operations and their pre- and postconditions are specified in table 3.6. As it is often more intuitive to address objects using *paths* rather then IDs, we also indicate the relationship between IDs and paths. A *path* is a notation for addressing a specific file or dir node with ID i in the file system tree. It is built by concatenating the *names* of all traversed nodes, starting from i_{root} until reaching node i, separating the names with / (forward slash). Path-related helper functions are defined in table 3.5. Our preconditions and invariants are equivalent to those defined by [NSE18], which the authors proved to be correct using the CISE SMT solver [Got+16].

Function / Predicate	Description
$split(path)$	Returns a list of the individual name segments, by splitting $path$ at each / character.
$parent(path)$	Returns / for all paths that contain / once, otherwise removes from path all characters from the right-hand side up to (including) the first encountered / character.
$basename(path)$	Returns the last list element of *split(path)*, e.g. *basename('/home/user/foo') = 'foo'*. Formally, if we can split $path = \pi/\omega$ s.t. $\pi = parent(path)$ and $\omega = basename(path)$ then the following holds: $p = \pi/\omega \iff \exists i, j \in I : i \in list(j) \wedge path(j) = \pi \wedge name(i) = \omega$
$id(path)$	Returns the id of $path$ in case traversal was successful, or *error* otherwise. The traversal algorithm starts with $i = i_{root}$, and iteratively updates $i = id(i, n)$ for each n in $split(path)$.
$path(i)$	Performs a *tree search* starting at i_{root}, iterating over *all* nodes until the node n with ID i was found. If no node is found, *error* is returned. Otherwise returns the path from the root node to n, concatenating the respective names of the nodes along the traversal-path.

Table 3.5: Functions for working with paths

3.3 Conclusion

This chapter addresses all aspects of RQ1. We completed the analysis of file system definitions (which we started in the previous chapter for *academic* models) by examining *industrial* file system models for heterogeneity, sampling a set of representative systems. The discovered *capabilities* reflect the *"criteria [...] relevant for file synchronizers"* of RQ1. Most of the characteristics we found for academic file systems, such as *identity- vs. path-based*, different *hardlink support* and varying *operation support* are also present in industrial systems. However, by experimentation and studying manuals we discovered several *additional* heterogeneous capabilities not discussed in academic works which file synchronizers need to address, including namespace limitations, meta-data or locking.

Our general suggestion for building a model with maximum compatibility is to limit its features to the lowest common denominator. As doing so for *all* discovered capabilities would result in a weak model, $\mathscr{F}$ follows this advice only for some capabilities (meta-data, locking, supported object types). For other capabilities, like *mappings* or *operations*, $\mathscr{F}$ demands stronger features to improve performance and usability.

Operation	Description, pre- and postconditions
$createdir(path)$, $createdir(i,p,n)$	Creates a new, empty directory with ID i and name $n = basename(path)$ in parent directory with ID $p = id(parent(path))$. • Precondition: $\neg ancestor(i_{root}, i) \wedge ancestor(i_{root}, p) \wedge type(p) = dir \wedge id(p,n) = error$ • Postcondition: $i \in list(p) \wedge type(i) = dir \wedge lastmodified(i) \neq error$
$createfile(path)$, $createfile(i,p,n)$	Creates an empty file with ID i and name $n = basename(path)$ in parent directory with ID $p = id(parent(path))$. • Precondition: see $createdir(path)$ • Postcondition: $i \in list(p) \wedge type(i) = file \wedge lastmodified(i) \neq error$
$move(source, dest)$, $move(i,u,v,n)$	Moves a file or directory with ID $i = id(source)$ from parent directory with ID $u = id(parent(source))$ to $v = id(parent(dest))$, and/or change the object's name to n. • Precondition: $type(u) = dir \wedge i \in list(u) \wedge type(v) = dir \wedge id(v,n) = error \wedge \neg ancestor(i,v)$ • Postcondition: $i \in list(v) \wedge i \notin list(u)$ Note: precondition $\neg ancestor(i,v)$ ensures that the user cannot move a directory to a destination dir below it, e.g. $source =$ '/A' cannot be moved to $dest =$ '/A/x'.
$deletefile(path)$, $deletefile(i,p)$	Removes the file with ID $i = id(path)$ from parent directory with ID $p = id(parent(path))$. • Precondition: $ancestor(i_{root}, i) \wedge i \in list(p) \wedge type(i) = file$ • Postcondition: $i \notin list(p) \wedge \neg ancestor(i_{root}, i) \wedge lastmodified(i) = error$
$deletedir(path)$, $deletedir(i,p)$	Removes the empty directory with ID $i = id(path)$ from parent directory with ID $p = id(parent(path))$. • Precondition: $ancestor(i_{root}, i) \wedge type(i) = dir \wedge list(i) = \{\}$ • Postcondition: see $deletefile(path)$
$edit(path, op)$, $edit(i, op)$	Changes the byte content of file with ID $i = id(path)$ by performing the operation op (e.g. adding, removing or changing bytes at specific positions within the file). • Precondition: $ancestor(i_{root}, i) \wedge type(i) = file$ Let $l_{pre} = lastmodified(i)$ • Postcondition: $ancestor(i_{root}, i) \wedge lastmodified(i) \neq l_{pre}$

Table 3.6: File system operations

Chapter 4

Update detection for file systems

A central component of every file synchronizer is *update detection*, which is concerned with observing the file system of a replica for changes. It delivers the changes to the reconciliation algorithm. The reconciliation algorithm then decides which operations need to be executed on each replica to equalize their state.

In our work we build a *state-based* file synchronizer that runs in the *background* and continuously synchronizes two replicas, which cannot be protected (locked) against concurrent user activity. Its update detector should ideally have the following characteristics:

1. Deliver *operations* and the *current state* in near *real-time*, to allow the synchronizer to start working in a timely manner. A state-based reconciliation requires the current *state* as foundation for finding updates. Real-time *operations* are helpful to allow the synchronizer to discern concurrently executed operations by the *user* from those executed by the synchronizer itself. This allows the synchronizer to *abort* an on-going synchronization in case the user's concurrent operations are conflicting its own scheduled ones (e.g. if a user locally deleted a file *after* the reconciliator scheduled its upload operation).

2. Put as *little load* as possible on the replica, to allow a large number of concurrent users, e.g. on a server.

In this chapter we look at update detection mechanisms offered by file systems. We identify issues that violate the above characteristics and propose solutions. We start with analyzing available mechanisms in real-world file systems to extract an overview of approaches in section 4.1. In the remaining sections we take a closer look at the specific approaches.

4.1 Analysis and overview

Many data synchronization solutions, such as near real-time collaborative text editors, have a *built-in* replication mechanism, which is thus tightly coupled to the application itself. Unfortunately, heterogeneous file synchronizers are typically loosely coupled. The application, the file system, is not aware of its synchronization. A file synchronizer needs to extract its current state and operations, using APIs offered by the file system. We start with an API analysis in section 4.1.1 and extract a taxonomy from it, presented in section 4.1.2.

4.1.1 File system API analysis

A basic requirement for every file system is the ability to sample the current state, by listing all objects in a directory and extracting their attributes (ID, parent, name, type, lastmodified - see section 3.2.1). Without going into details, our analysis shows that all file systems examined in section 3.1 (Windows,

	Online	Offline
Windows	✓ *ReadDirectoryChangesW*[1]	✓ *Change journal*[2] keeps track of file operations. It is limited in length, only for NTFS
macOS	✓ *FSEvents*[3]	X
WebDAV	(X) See below for more details	(X) See below for more details
BSCW, Dropbox	✓ Via proprietary REST interface	✓ Via proprietary REST interface

Table 4.1: File system APIs for update detection

macOS, BSCW, WebDAV, Dropbox) offer APIs for this purpose. Because *regular sampling* to detect activity is expensive, we further analyze APIs that detect operations in near real-time in table 4.1. We subdivide them into *online* and *offline*, where *online* APIs only provide information while a process (such as the file synchronizer) subscribes to receiving changes, whereas *offline* APIs have an internal persistence mechanism, offering a process to retrieve information at any (later) point of time.

Regarding WebDAV, its RFC [Dus07] does not provide any means for monitoring changes. There is RFC 6578[4], a WebDAV extension that offers a *sync-collection report* that contains changes that took place since retrieving the previous report, with the help of a sync token. As is common with HTTP, the API needs to be called in regular intervals (polling), but it allows to track changes even while the client is not online. To the best of our knowledge, at the time of writing there is no implementation of RFC 6578 for tracking changes of files and directories. There are some implementations which support it for synchronizing changes of calendars and contact information[5].

4.1.2 Taxonomy

From above analysis we build a taxonomy of update detection approaches:

1. **Log-based update detection:** the state-based file synchronizer uses an existing file system API that provides a *complete* log of all operations as a side effect of executing them. There are two sub-categories:

   a) **Online update detection:** a callback mechanism transmits operations to the file synchronizer (while it is active) in the order they happen. Operations taking place while the file synchronizer is stopped will be lost. Examples are the callback-based APIs provided by macOS or Windows for local disks. Section 4.3.3 discusses technical challenges that occur when using such APIs, because the provided operation events often lack important information or are incorrect to begin with.

   b) **Offline update detection:** similar to the online-mechanism, but the log of operations is *persistently* stored by the file system, irrespective of whether the file synchronizer is active. The file synchronizer can retrieve all operations even after a period of inactivity. Only some storage systems such as Dropbox or BSCW offer such a service. The API is typically token-based.

[1] `https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-readdirectorychangesw`, retrieved July 21, 2019.

[2] `https://docs.microsoft.com/en-us/windows/win32/fileio/change-journals`, retrieved July 21, 2019.

[3] `https://developer.apple.com/library/content/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html`, retrieved July 21, 2019.

[4] `https://www.greenbytes.de/tech/webdav/rfc6578.html`, retrieved July 21, 2019.

[5] `http://sabre.io/dav/sync/`, retrieved July 21, 2019.

The file synchronizer calls the API (with a token received from a previous call) and receives a new token together with a list of all operations that took place since the last API call.

2. **State-based update detection:** using the (always available) file system APIs to build a file system *snapshot*, the file synchronizer computes operations from those. By taking snapshots at times t_1 and t_2 and comparing them with each other, it deduces the list of file system operations. Section 4.2 provides further details.

3. **Hybrid update detection:** uses *state-based* update detection as foundation, but ensures that the characteristics from the introduction are maintained, by additionally using *log-based* update detection where available. Section 4.3 provides more details.

As stated in section 4.1.1, approach (2) is available for every file system API. Several academic works explicitly state the lack of availability of log-based update detection mechanisms as rationale for using a similar approach [LKT05; RC01]. In our implementation we use a *hybrid* approach presented in section 4.3.

4.2 State-based update detection

In this section we elaborate on details of state-based update detection. We start with *snapshots* which we describe in section 4.2.1, which are a data structure that stores state information. While snapshots are typically kept in memory and are specific to one replica, *database snapshots* are a persistently stored variant which include two replicas, see section 4.2.2. We describe how operations are computed from two snapshots in section 4.2.3. In section 4.2.4 we describe update trees, which are memory structures used in our reconciliation algorithm with better efficiency than snapshots. Finally, we address the problem of operation consolidation in section 4.2.5, which is specific to state-based update detection, and discuss its side effects and a solution in subsequent sections.

4.2.1 Snapshots

In section 4.1.1 we discussed that obtaining the current state of a replica is feasible in all examined file system APIs. File system states change over time and query operations are expensive, thus we now define the *snapshot* data structure which is kept in memory and represents a replica's *abstracted* state at a specific point of time. It stores the list of objects with their attributes, such as ID, name, parent ID, etc. and can be built using the functions $list(i)$, $type(i)$, $name(i)$, $lastmodified(i)$ from section 3.2.1. Its functions are shown in table 4.2, which are similar to those defined in section 3.2. These snapshot functions also return *error* in case looking up the requested data is not successful. Note that snapshots do not store any file contents, but only checksums of files.

4.2.2 Database snapshots

To compute operations it is necessary to take two snapshots at times t_1 and t_2 and compare them with each other. We present the corresponding algorithm in the next section. In a file synchronizer the computation is always done between two snapshots of the *same* replica, where the first snapshot is the persisted[6] *database snapshot* that expresses the replica's state after the last synchronization, while the second snapshot reflects the *current* replica's state. Because the state of the two replicas and the database is exactly equal once their synchronization has successfully finished, it is sufficient to have *one* database snapshot that stores information of *both* replicas. Only the replica-specific IDs mismatch, thus the database snapshot needs to establish a relationship between them.

A database snapshot $dbsnapshot$ internally stores each object as a tuple

$$\langle i_{db}, p_{db}, name, i_X, i_Y, lastmodified_X, lastmodified_Y, type, checksum \rangle$$

[6]For instance, a database snapshot might be built from a table stored in a relational database.

Function	Description
$id(snapshot, path)$	Returns the id of the path in case traversal was successful, or *error* otherwise.
$type(snapshot, i)$	Returns the type (file, dir) of the object with ID i.
$lastmodified(snapshot, i)$	Returns the *lastmodified* meta-datum for *i*.
$parent(snapshot, i)$	Returns the parent directory ID of the object with ID i.
$path(snapshot, i)$	See $path(i)$ in table 3.5.
$name(snapshot, i)$	Returns $basename(path(snapshot, i))$.
$checksum(snapshot, i)$	Returns the checksum of the content of the file with ID i, or *error* if i is not known or $type(snapshot, i) = dir$.
$paths(snapshot)$	Returns the list of paths contained in *snapshot*.
$ids(snapshot)$	Returns the list of IDs contained in *snapshot*.
$ancestor(snapshot, i, j)$	Returns whether node with ID i is an ancestor of the node with ID j in *snapshot*.

Table 4.2: Functions for working with snapshots

Function	Description
$dbid(dbsnapshot, i, r)$	Returns i_{db} for the provided ID i of replica r if i is known, *error* otherwise.
$id(dbsnapshot, i_{db}, r)$	Returns the replica-specific ID of replica r for the database ID i_{db}, *error* otherwise.

Table 4.3: Functions for working with database snapshots

where id_{db} is an unique (primary key) ID generated by the database engine. It establishes the 1:1 mapping between the IDs i_X, i_Y of the replicas, where replica identifiers are denoted X, Y henceforth. The same principle applies to $lastmodified$. $type$ details whether the path points to a file or directory and p_{db} expresses the database ID of the parent object. This compact representation allows to generate both *dbsnapshot*$_X$ and *dbsnapshot*$_Y$, where e.g. *dbsnapshot*$_X$ is the database snapshot for replica X. To generate *dbsnapshot*$_X$ we only need id_{db}, p_{db}, $name$, i_X, $lastmodified_X$, $type$ from each tuple. All snapshot functions defined in table 4.2 also apply to *dbsnapshot*$_X$ and *dbsnapshot*$_Y$. Additionally, functions $dbid(dbsnapshot, i)$ and $id(dbsnapshot, i_{db}, r)$ defined in table 4.3 also apply.

4.2.3 Operation computation

The function *compute_ops(*`db`*,* `snapshot`*)* shown in algorithm 1 detects an unordered set of operations $O = \{o_1, ..., o_n\}$ that explain the differences between the snapshots `db` and `snapshot`. For each o_i we define the following functions:

- $optype(o_i)$ returns the operation's type: *create, delete, move,* or *edit*
- $id(o_i)$ retrieves the object's ID
- $type(o_i)$ returns the object's type (file, directory)
- $lastmodified(o_i)$ returns the object's *lastmodified* meta-datum (the updated one for edited files)
- $path(o_i)$ returns the path of the affected object, or path-tuple *(source,dest)* if $optype(o_i) = move$. If $optype(o_i) \in \{edit, delete\}$ then $path(o_i)$ returns the path from `db` rather than the one from `snapshot`.

We can also express *compute_ops(db, snapshot)* using FOL to more clearly demonstrate the relationship between the post-conditions of operations introduced in table 3.6 and the detection of these operations:

```
Input: db, snapshot: snapshot structures taken at two different points of time, of the
    same replica
Output: set of detected operations

O = Set()
for i in ids(db):
  current_path = path(snapshot, i)
  if current_path == error:
    o = Operation(delete, i, type(db, i), lastmodified(db, i), path(db, i))
    O.put(o)
    continue

  has_moved = name(db, i) != name(snapshot, i) or parent(db, i) != parent(snapshot, i)
  if lastmodified(db, i) != lastmodified(snapshot, i) and type(db, i) == file:
    o = Operation(edit, i, file, lastmodified(snapshot, i), path(db, i))
    O.put(o)
    if not has_moved:
      continue

  if has_moved:
    path_tuple = (path(db, i), path(snapshot, i))
    o = Operation(move, i, type(db, i), lastmodified(snapshot, i), path_tuple)
    O.put(o)
    continue

for i in (ids(snapshot) - ids(db)):  # consider only IDs in 'snapshot' that are not in '
    db'
  o = Operation(create, i, type(snapshot, i), lastmodified(snapshot, i), path(snapshot, i
      ))
  O.put(o)
```

Algorithmus 1 : Pseudo-code for *compute_ops(db, snapshot)*

- Creation of a file or directory, then a new ID must exist in *snapshot* that did not exist in *db* yet: $\exists o \in O: optype(o) = create \iff id(o) \in ids(snapshot) \wedge id(o) \notin ids(db)$
- Deletion of a file or directory, then the corresponding ID that still exists in *db* is no longer available in *snapshot*: $\exists o \in O: optype(o) = delete \iff id(o) \in ids(db) \wedge id(o) \notin ids(snapshot)$
- Moving a file or directory, then the ID exists in both snapshots, but with different parent ID or name: $\exists o \in O: optype(o) = move \wedge path(o) = (source, dest) \iff id(o) \in ids(snapshot) \wedge id(o) \in ids(db) \wedge [name(db, id(o)) \neq name(snapshot, id(o)) \vee parent(db, id(o)) \neq parent(snapshot, id(o))]$
- Editing a file, then the ID exists in both snapshots, but with different *lastmodified* meta-datum: $\exists o \in O: optype(o) = edit \wedge type(o) = file \iff id(o) \in ids(snapshot) \wedge id(o) \in ids(db) \wedge lastmodified(db, id(o)) \neq lastmodified(snapshot, id(o))$

It is easy to see that the number of detected operations is finite, as the following axiom and lemma show.

Axiom 1. The number of objects (files, dirs) $n = |R|$ in any replica R is *finite*, because each R requires each object to be explicitly created by the user, creation speed is limited to y objects/second, R comes into existence with 0 objects at time t (in the past) with $now - t < \infty$.

Lemma 1. *The number of operations detected by* compute_ops(db, snapshot) *is finite.*

Proof. by contradiction: Let *O=compute_ops(*`db , snapshot`*)*. Let $k = |O|$ be the number of detected operations. For k to be infinite, and because $|db|$ and $|snapshot|$ are finite (as demonstrated in axiom 1), the *compute_ops* algorithm must find an infinite number of operations for some object pair $(x, y), x \in db \vee x = \epsilon, y \in snapshot \vee y = \epsilon$. However, it is trivial to see (and is further demonstrated in section 4.2.5) that *operation consolidation* does not allow for this: for any (x, y) we can only detect one of the following operations: none, create, delete, move, edit, move+edit. I.e., for any (x, y) the number of operations that can be detected ranges from $0 < \infty$ to $2 < \infty$. □

4.2.4 Update trees

To find and resolve conflicts, the *snapshot* structure and operation set O is insufficient. Instead our algorithm converts a *database snapshot* and O of a specific replica into an *update tree*. The update tree is an in-memory tree structure where each node represents a specific file or directory. Once built, it represents the file system in the *current* state of the respective replica, but also contains deleted nodes. This tree improves run-time performance by providing efficient means to *search* for nodes, e.g. using i_{db} or i_X, i_Y and offers efficient traversal both up and down the hierarchy. In contrast to (immutable) snapshots and computed operation sets, the structure of update trees can be efficiently manipulated and easily updated with meta-information by different components of the reconciliation and propagation phase. More specifically, the conflict finder and resolver components update the tree by marking which conflicts were already considered for each node, to avoid searching for conflicts more than once. The propagation component keeps the update tree structure in sync with the changes it applies to the physical replica and to the database. This allows the update trees to serve as shadow copy of the file system's current state and allows to efficiently look up the correct paths during propagation, which is necessary because file system APIs are typically not ID- but *path-based*.

Each node has the following attributes:

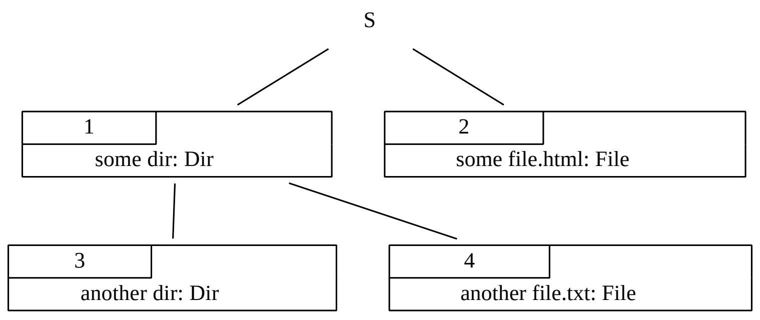
- i_{db}: provided by the database snapshot, is undefined (-1) for *created* files or directories, which are not part of the database yet
- *side*: the identifier of the replica (X, Y)
- *name*: the name of the object
- *type*: file or dir
- *change_events*: list of change-events (more details below)
- *ID* and *lastmodified* meta-data, where *ID* is the replica-specific ID, i_X or i_Y
- *processed*: indicates whether the node was already considered during reconciliation, i.e., while generating operations
- *children*: list of child nodes
- *parent*: pointer to the parent node

Objects that were moved by the user have two additional attributes:

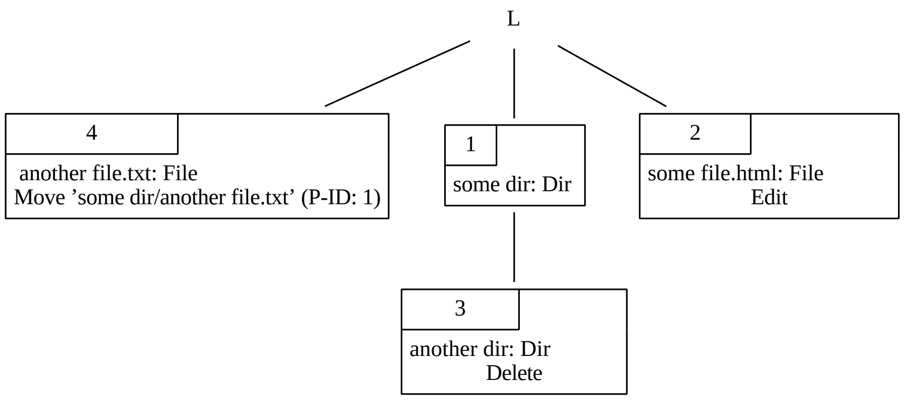
- *move_origin*: the path of the object before it was moved
- *move_origin_parent_id*: p_{db}, that is, the ID of the object's parent directory *before* it was moved

We use n for nodes. Attribute access is denoted with a dot, e.g. $i = n.ID$. The path $p = path(n)$ of a node n is computed by concatenating the nodes' names from the root node to n, with a forward slash separation character.

Figure 4.1 shows two visualizations of an update tree representing a file system before and after the user applied some operations. Efficient search for nodes is implemented by adding look-up tables at each tree's *root* node, mapping from ID to the node object.



(a) Start situation



(b) Changed situation

Figure 4.1: Update tree visualization

Subfigure (a) shows a visualization of an update tree of a file system as a graph. At the top, the label *S* indicates *Synchronized* state (i.e. the local and remote replica have the same structure) - alternatively *L* or *R* indicate that the update tree is of the *Local* or *Remote* replica respectively (see subfigure (b)). *L* and *R* are used synonymously to *X* and *Y* for identifying replicas. In the tree each file or dir is represented by a node. The top left corner indicates the node's i_{db}. The first line of the node indicates its *name* and *type* (file, dir). The line(s) below the name indicate *change-events*, if present. Because subfigure (a) depicts a synchronized file system, there are no change-events. In contrast, subfigure (b) shows a situation in which the user changed the file system from subfigure (a) via multiple operations. "/some dir/another dir" was deleted, "/some file.html" was edited, and "/some dir/another file.txt" was moved to the root level. The line in the node (here: $i_{db} = 4$) that starts with "Move" has the following form:
Move <source path> (P-ID: *n.move_origin_parent_id*)
P-ID is -1 in case the source parent dir is the *root* directory, since the root directory itself is not part of the database and thus has no i_{db} value.

4.2.4.1 Change-events

To efficiently find conflicts or to determine non-conflicting operations, all those nodes affected by the user since the last synchronization are marked dirty during the creation of the tree, using the information from the set of computed operations O. Instead of using a single dirty bit, each node is assigned 0-2 change-events (like a bit *mask*). We define the following events:

- *Create*: indicates that a new file or dir was created, its *ID* is unknown to the database
- *Edit*: for files: indicates that the file's content was changed (or it was deleted and replaced by another created file, see next section for more details)
- *Delete*: indicates that a file or dir was deleted (either directly, or indirectly because a parent node was deleted)
- *Move*: indicates that a file or dir was moved, by the user (changing the name, parent node, or both)

4.2.4.2 Generation

We perform the update tree generation as follows:

1. Create dir nodes for each $o_i \in O$ where $optype(o_i) = move \wedge type(o_i) = dir$, using $path(o_i)$. Assign each node with the *Move* change-event. This step (as well as others) creates missing intermediate nodes if necessary (e.g. if $path(o_i) = {}'some/dir'$ then create an intermediate node for 'some' if it doesn't exist yet).
2. Create file nodes for each $o_i \in O$ where $optype(o_i) = move \wedge type(o_i) = file$, see above for details.
3. Create dir nodes for each $o_i \in O$ where $optype(o_i) = delete \wedge type(o_i) = dir$. Each node is assigned the *Delete* change-event. An important detail is that for any such o_i, $path(o_i)$ points to the path of the `db` snapshot. In case the user applied operations o_1*=move('/dir', '/dirMoved')* + o_2*=deletedir('/dirMoved/somedir')* to a replica, then *path(*o_2*)='/dir/somedir'*! When inserting the node for o_2 into the tree, the insertion algorithm must first attempt to find the possibly existing new path (*'/dirMoved/somedir'* in this case).
4. Create file nodes for each $o_i \in O$ where $optype(o_i) = delete \wedge type(o_i) = file$. Like above, each node is assigned the *Delete* change-event. In addition we also implement the special case where we replace the *Delete* with an *Edit* change-event in case the user deleted the file and created another file with the same name under the same parent. This approach has also been applied in other works, such as [Li+12a; Li+12b; RC01]. In case the replacement is done, we remove the corresponding create-operation o_j with $optype(o_j) = create$ from O. The reason for this decision is that many end-user applications, such as Word processors, apply a "safe" file replacement strategy when updating a file's content. The application *doesn't* change the file by opening it, writing bytes and closing it again (this approach would have preserved the file's ID). Instead, it saves the new version of the file to a temporary location, followed by deleting the original file, followed by moving the temporary file to the original location. *Semantically*, this operation reflects an *Edit* operation, therefore we decide to replace *Delete* and *Create* of a file with *Edit* in the update tree. We think that it is reasonable to assume that, statistically, users edit files more often than deliberately deleting a file and creating a new one with the exact same name, expecting that the synchronizer honors the latter intention verbatim. The incorrect replacement would also only happen during long periods in which no synchronization takes place.
5. Create dir nodes for each $o_i \in O$ where $optype(o_i) = create \wedge type(o_i) = dir$ with the *Create* change-event.
6. Create file nodes for each $o_i \in O$ where $optype(o_i) = create \wedge type(o_i) = file$ with the *Create* change-event.

7. Create file nodes for each $o_i \in O$ where $optype(o_i) = edit \wedge type(o_i) = file$ with the *Edit* change-event.
8. Create missing and update existing intermediate nodes for all *unchanged* files and dirs, by iterating over all objects in the database snapshot. This step serves several purposes:
   a) It makes sure to update all those *intermediate* nodes created during the previous steps, which lack important information, such as i_{db}, the replica-specific ID or the $lastmodified$ metadata, because this data wasn't available for them in the corresponding operation used during their creation.
   b) It creates missing nodes without any change-event.

The update tree generation procedure generates a $updatetree_X$ and a $updatetree_Y$ object.

Axiom 2. The number of change-events in each *updatetree* structure is finite. The change-events are created based on O, where $k = |O|$ is finite, see lemma 1 on page 49. As described above, some *delete* and *create* operations in O may be consolidated to an *Edit* change-event. Whenever such a replacement happens, the number of operations is *decreased* by 1, making it impossible that the number of change-events reach infinity.

4.2.5 Operation consolidation

Operations computed from state are affected by *consolidation*, which refers to the elimination of operations that take place inbetween taking snapshots. Consider taking a snapshot at time t_1. Then a user creates a new file, then moves it to a different path and then deletes it. If we then take another snapshot at time t_2, the computed set O between the two snapshots will not contain any of these operations, because the file's ID is not known in either snapshot. This observation was also made by [RC01] as *simplifying laws* and their follow-up work [Csi16] which refers to this as *minimal sequence or set of commands*. The authors considered file systems without *move* operation support. For $\mathscr{F}$, which does support *move* operations, we define the concept *operation consolidation* where two file system operations (as listed in table 3.6) are consolidated to zero or one operation. We use $\cong$ as notation for a consolidation, such that the left side of $\cong$ indicates the operations done by the user and the right side indicates how such two operations would appear in O. The following list shows all consolidation rules, which we built by examining every possible combination of two operations:

1. $move(i,u,v_1,n_1) + move(i,v_1,v_2,n_2) \cong move(i,u,v_2,n_2)$
2. $createfile(i,p,n) + edit(i,op) \cong createfile'(i,p,n,c)$ (with $lastmodified(snapshot,i)$ corresponding to the timestamp *after* the $edit(i,op)$ operation took place)
3. $create(i,p,n_1) + move(i,p,v,n_2) \cong create(i,v,n_2)$
4. $edit(i,op_1) + edit(i,op_2) \cong edit'(i)$ (with $lastmodified(snapshot,i)$ corresponding to the timestamp *after* the second $edit(i,op)$ operation took place)
5. $create(i,p,n) + delete(i,p) \cong []$
6. $edit(i,op) + deletefile(i,p) \cong deletefile(i,p)$
7. $move(i,u,v,n) + delete(i,v) \cong delete(i,u)$

with $create = createfile \vee createdir$, $delete = deletefile \vee deletedir$. $createfile'$ is a slightly modified variant of $createfile$ which also indicates the file's content, while $edit'$ is a variant of $edit$ that does not indicate the exact operation, as it is not necessary to know the exact content during update detection.

The consolidation rules can be applied iteratively to an operation input sequence, by first examining all operation pairs in O for matching rules 1-4 (until no more of these rules match), followed applying rules

5-7. For example, the sequence $createfile(1,0,'test')+edit(1)+move(1,0,0,'test2')+deletefile(1,0)$ can be consolidated to

$$\begin{aligned}
&\cong createfile'(1,0,'test',c)+move(1,0,0,'test2')+deletefile(1,0)\\
&\cong createfile'(1,0,'test2',c)+deletefile(1,0)\\
&\cong []
\end{aligned}$$

using consolidation rules 2, 3 and 5.

4.2.6 Side effects of operation consolidation

The advantage of operation consolidation is that the operation count is reduced, which improves processing and transmission time and simplifies conflict resolution. However, there are two side effects which we introduce informally in the subsequent subsections. Some of the related works, such as [LKT05; Li+12a], fail to mention these side effects, or handle them incorrectly. For simplicity we restrict the scenario to a *uni*-directional synchronization, where two replicas X, Y are initially equal, the user modifies only replica X and the computed operations O are then applied to Y, see figure 4.2.

4.2.6.1 Lack of serialization order

It is easy to see that *compute_ops()* does not provide the *serialization order* of the computed operations. It is not straightforward to determine the serialization of the *create*, *delete*, *move* and *edit* operations when the goal is to apply them to the other replica, because the synchronization algorithm may not violate the operation's invariants. Consider the following examples:

1. User deletes file "x" and moves "a" to "x": *deletefile* operation has to be applied before the *move* operation
2. User moves file "a" to "b" and creates another file at "a": *move* has to be applied before *createfile*

There are several other examples which are discussed in subsection 4.2.7.1, which also explains how to formally determine these reordering dependencies.

4.2.6.2 Cycles from missing move operations

Consolidation rule (1) can lead to *cycles* when considering the serialization order. Consider the example of two files named "a" and "b" whose names are *swapped* by the user. This can only be done by performing *three* move operations, e.g. *move('a', 'temp'), move('b', 'a'), move('temp', 'b'). compute_ops()* will detect only *two* move operations, *move('a', 'b')* and *move('b', 'a')*. Synchronizing these operations is not straightforward, because the first one has to precede the second one, while the second one also has to precede the first one.

4.2.7 Addressing side effects in a synchronizer

The following subsections address the issue of finding a valid operation sort order. Parts of this section are also found in [She19]. We explain how to find order dependency rules, detect *cycles* and how to break them.

As noted above and explained in figure 4.2 we assume *uni*-directional synchronization. In chapter 6 (section 6.3.2) we will address the specifics of *bi*-directional synchronization for which there is a chance for *conflicts* between concurrent operations. We will show that our approach to address the side effects of operation consolidation still works for bi-directional synchronization, as long as there are no such conflicts (but, at most, *pseudo*-conflicts).

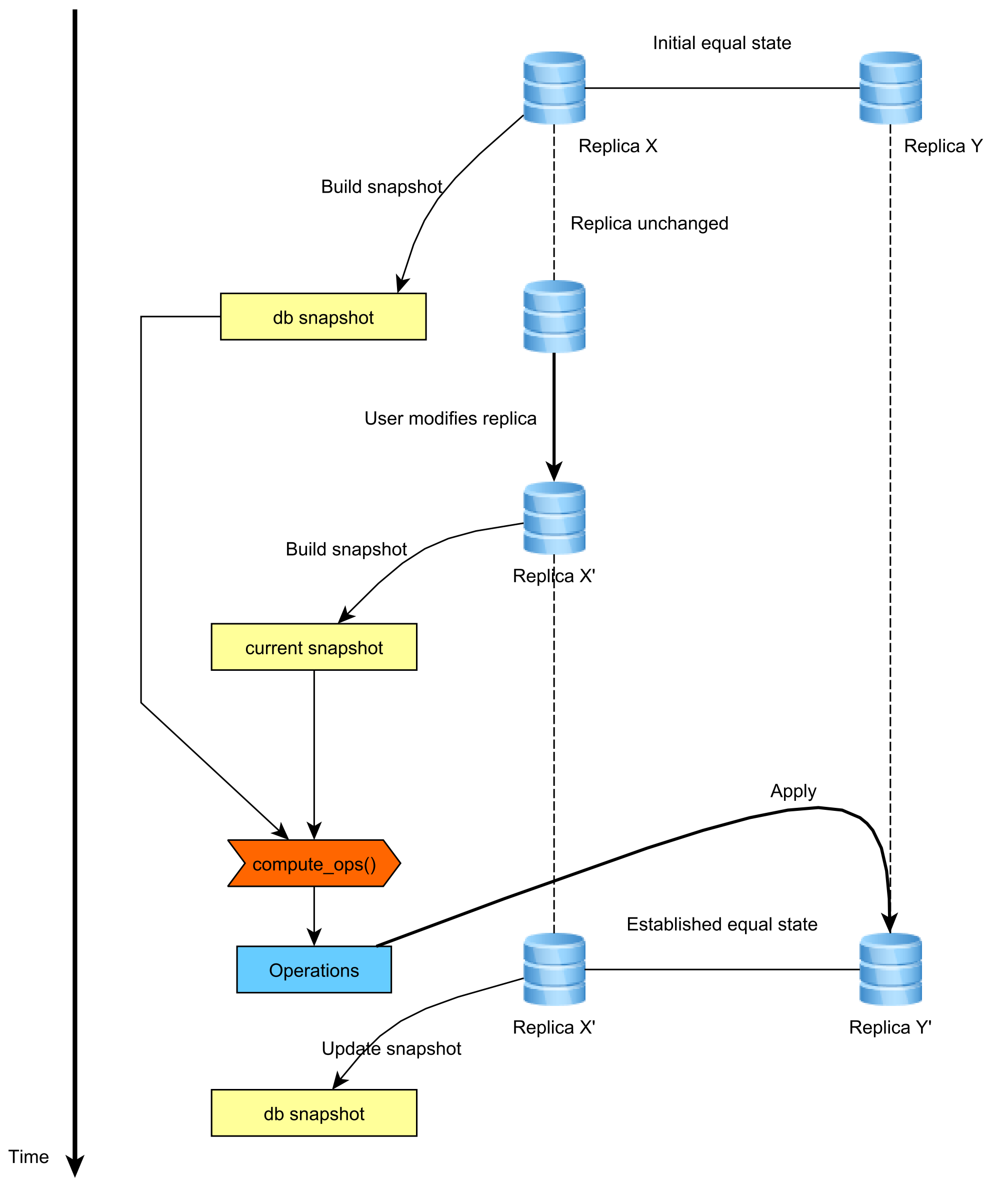
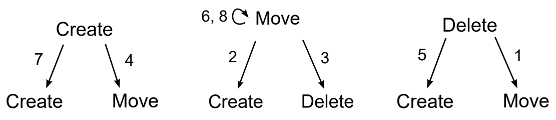


Figure 4.2: State-based synchronization using computed operations

Illustration of a *uni*-directional synchronization from replica X to Y. Both replicas are initially equal (e.g. empty). The file synchronizer maintains a `db` snapshot of X and Y. Whenever replica X was modified by the user, e.g. by *create* or *move* operations, these are detected by taking a current `snapshot` and calling *compute_ops(`db`, `snapshot`)*. The detected operations are then applied to replica Y, after which both replicas are equal again. Afterwards the file synchronizer updates `db` to be able to detect only modifications made by the user to X from this point forward.



(a) Create (b) Move (c) Delete

Figure 4.3: Operation order dependencies

4.2.7.1 Operation sorting

We build an algorithm $\bar{O} = sort\_operations(O)$ which turns an unordered set O to an ordered list $\bar{O}$. This requires an analysis of the operation preconditions because not all operations are commutative. Let OT be the list of considered operation types. For any two types $t_A, t_B \in OT$ we instantiate the respective operations o_A, o_B, detected on replica X. We choose the parameters (i, p, u, v, n for $\mathscr{F}$'s operations) such that applying the sequence (o_A, o_B) to replica Y is feasible, but applying (o_B, o_A) would fail, because a precondition of one of the two operations is violated[7]. We end up with a list of *order dependencies*, where each order dependency contains t_A, t_B (in a specific order) and the violated operation precondition(s). Finally, we examine whether cycles can be built from the *order dependencies*.

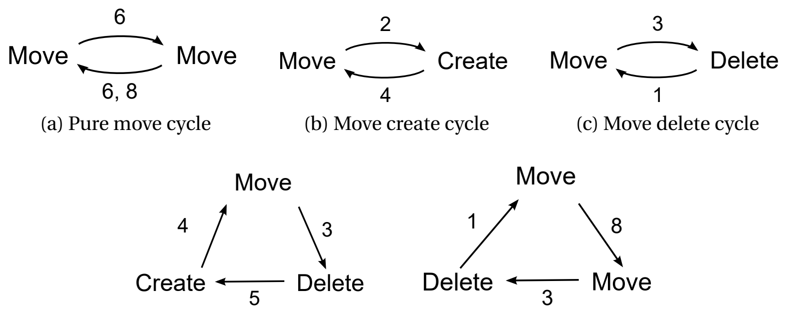
For $\mathscr{F}$ we choose $OT = \{createfile', createdir, move, edit', delete'\}$. The operation $delete'$ summarizes *deletefile* and *deletedir* and *recursively* deletes a directory's children. Figure 4.3 shows an overview of the eight order dependencies we found for the operation types in OT. The arrows are denoted with a dependency number explained below:

1. *delete* before $move$, e.g. user deletes an object at path "x" and moves another object "a" to "x"
2. $move$ before *create*, e.g. user moves an object "a" to "b" and creates another object at "a"
3. $move$ before *delete*, e.g. user moves object "X/y" outside of directory "X" (e.g. to "z") and then deletes "X"
4. *create* before $move$, e.g. user creates directory "X" and moves object "y" into "X"
5. *delete* before *create*, e.g. user deletes object "x" and then creates a new object at "x"
6. $move$ before $move$ (occupation), e.g. user moves file "a" to "temp" and then moves file "b" to "a"
7. *create* before *create*, e.g. user creates directory "X" and then creates an object inside it
8. $move$ before $move$ (parent-child flip), e.g. user moves directory "A/B" to "C", then moves directory "A" to "C/A" (parent-child relationships are now flipped)

We found these ordering rules by analyzing the pre-conditions of the file system operation from table 3.6 on page 44:

- *createdir(i, p, n), createfile(i, p, n)*:
  - $ancestor(i_{root}, p) \wedge type(p) = dir$, in other words, the parent directory has to exist: rule *create before create* (7)
  - $id(p, name) = error$, that is, *name* in p has to be free: rules *move before create* (2) and *delete before create* (5)
- *move(i, u, v, n)*:
  - $i \in list(u)$, i.e., object has to exist: rule *move before delete of parent dir* (3)

[7]For instance, when creating a directory and a file inside it, the order cannot be flipped. That is, (o_A, o_B) is feasible with $o_A = createdir(1, i_{root}, 'dir')$ and $o_B = createfile(2, 1, 'file')$, but (o_B, o_A) would fail, because precondition $type(1) = dir$ of o_B would be violated.



(a) Pure move cycle (b) Move create cycle (c) Move delete cycle

(d) Move delete create cycle (e) Move move delete cycle

Figure 4.4: Operation cycles

Graphical illustration of different types of operation cycles that result from chaining order dependency rules. Subfigures a-c show *minimal* cycles that involve as few rules as possible. For subfigure a we refer to appendix section A.3 that proves that cycles that exclusively consist of move operations connected only by rule (8) are impossible. Subfigures d-e show examples for more elaborate cycles.

- $type(v) = dir$, i.e., destination parent directory must exist: rule *create before move* (4)
- $id(v,n) = error$, i.e., destination name has to be free: rules *delete before move* (1), *move before move occupation* (6)
- $\neg ancestor(i,v)$, i.e., destination parent directory may not be below the destination itself, rule *move before move parent-child flip* (8)

The analysis of *deletefile* and *deletedir* do not yield any pre-conditions that lead to additional ordering rules. We note that the pre-condition $list(i) = \{\}$ of *deletedir* is not an issue in practice, because the implementation, when given the task to delete a directory, will automatically traverse over the sub-tree of all child objects and delete all objects in a depth-first *postorder* order. In general, two *delete* operations cannot be dependent on each other, because either the two paths of the delete operations are *independent* of one another (i.e., not in a parent-child relationship, in which case the delete order is also independent), or they are hierarchically dependent on each other, in which case our reconciliation algorithm will only create *one* delete operation for the highest node. See section 6.3.1 for more details. We note that due to operation consolidation, *edit'* can only co-exist with a *move* operation affecting the same object. Because these two operations are commutative, *edit'* is not part of any order dependencies.

We implemented `fix_op_before_op()` functions to detect and reorder incorrectly ordered operations, see algorithms in appendix A.4 on page 205. Each operation is given the unsorted set O (`ops`) and the (partially) sorted list $\bar{O}$ (`sorted_ops`), as well as the `db` and `snapshot` snapshots. The functions then perform an in-place reordering of $\bar{O}$.

4.2.7.2 Detecting cycles

To detect cycles we connect the order dependencies found above, see figure 4.4. As is easy see it's impossible to build cycles using only *delete* and *create* operations. Cycles *always* include at least one *move* operation. It is possible to find *multiple* cycles. The length of each cycle (as well as the total number of cycles) is limited by $k = |O|$, see lemma 1 on page 49.

Algorithm 2 shows how a *total* order of operations is computed using the eight `fix_xyz()` functions presented in appendix A.4 on page 205, where each function only establishes a *partial* order.

```
Input: ops (unsorted set of operations, O), db, snapshot
Output: sorted_ops, list of sorted operations
Global variables: has_order_changed, reorderings

sorted_ops = list(ops)
complete_cycles = []
reorderings = []  # type: List[Tuple[Operation, Operation]]
while True:
    has_order_changed = False
    # All functions perform in-place reordering. If the order was changed, they
        set variable has_order_changed to True and add the partial order to '
        reorderings' (unless 'reorderings' already contains it)
    # Note that the call order of the fix methods does NOT matter
    fix_delete_before_move(ops, sorted_ops, db, snapshot)
    fix_move_before_create(ops, sorted_ops, db, snapshot)
    fix_move_before_delete(ops, sorted_ops, db, snapshot)
    fix_create_before_move(ops, sorted_ops, db, snapshot)
    fix_delete_before_create(ops, sorted_ops, db, snapshot)
    fix_move_before_move_occupied(ops, sorted_ops, db, snapshot)
    fix_create_before_create(ops, sorted_ops, db, snapshot)
    fix_move_before_move_hierarchy_flip(ops, sorted_ops, db, snapshot)
    if not has_order_changed:
        break
    complete_cycles = find_complete_cycles(reorderings)
    if len(complete_cycles) > 0:
        break

if len(complete_cycles) > 0:
    resolution_operation = break_cycle(complete_cycles[0])
    return [resolution_operation]
else:
    return sorted_ops
```

Algorithmus 2 : Pseudo-code for `sort_operations()`

The function `find_complete_cycles()` used in algorithm 2 analyzes the `reorderings` collected so far and attempts to find complete (closed) cycles by chaining the partial reorderings. The mundane details of the implementation of this function are omitted. A developer needs to ensure that cycles are found, even if...:

- ... the order in which reorderings are found do not follow the order of the operations in the cycle. For instance, if a cycle consists of the operations (A, B, C, D), that cycle still has to be found even if the partial reorderings were found in the order (B,C), (D,A), (A,B), (C,D).
- ... the cycle is *hidden within a chain*, in other words, a cycle can be formed not by just testing whether the *end* of an just-extended chain coincides with the *beginning* of that chain, but it might also coincide with any of the inner nodes of the chain. As an example, the reorderings might be found and processed in the following order: (B,A), (A,C), (C,A). Simply connecting these orderings produces a chain, but also an inner cycle (A,C).

The termination property of algorithm 2 is proven in the following theorem.

Theorem 1. *The `sort_operations()` algorithm terminates.*

Proof. by contradiction: To not terminate, the algorithm would have to be caught in the `while` loop indefinitely. However, this cannot happen, because after a finite number of iterations, the loop is always left (line 21, 24).

At line 20 there are two cases that can hold: (1) one or more cycles exists (irrespective of whether our algorithm already discovered them), or (2) no cycles exist at all (thus our algorithm cannot possibly detect one).

We start with (1). If there is a cycle, then at line 20 `order_changed` must be *True* in *every* loop iteration, because `sorted_operations` is a *list* with start and end, thus, an operation located at the beginning of the list will always be moved to the end of the list by one of the `fix` functions. In this case, a cycle is either detected in line 22 (because our algorithm discovered all its `reorderings`), in which case we exit the while loop (line 24), or the cycle has not been discovered yet and a new loop iteration is started. Because each cycle has a *finite* length of at most k operations, the cycle must also be discovered within k loop iterations, after which the while loop is exited (line 24).

When considering situation (2), then there must be a total order of operations that is executable. If there are k non-cyclic operations, then after at most k iterations, none of the `fix_op_before_op()` functions will rearrange any operations any more (because they are already in a satisfactory order). Then `order_changed` will be *False* and the loop will be exited (line 21). □

4.2.7.3 Breaking cycles

To break a cycle C we need to find a suitable operation O in C, s.t. *renaming* the object targeted by O to a *temporary* name will break C and convert it into a chain. By closely analyzing the different types of cycles from figure 4.4, we find that for any cycle found in replica X there must always be at least one operation o_X (with $id(o_X) = i$) which frees a location (i.e., a name in a specific directory) that is used by a follow-up operation o'_X. o_X must either be a *move* (dependency rules 6+2) or a delete (dependency rules 1+5) operation. Instead of executing o_X, `break_cycle()` generates a different *move* operation r_Y that breaks the cycle. r_Y renames i by appending a unique suffix to its name. We execute r_Y on Y and the database snapshot and then restart the synchronization. This way, r_Y's effect is not detected after the restart, but o_X and o'_X are still detected because we did not modify X [8]. However, the cycle is now broken, because the order dependency (6, 2, 1, or 5) no longer applies.

An example can be found in figure 4.5. We refer to section 6.3.2 for additional examples and further implementation details.

4.3 Hybrid update detection

A background synchronizer needs to *continuously* synchronize two replicas. Using only the state-based approach would require to *regularly* sample each replica's current state. On large file systems (with a high number of files and directories), computing and transferring the state between replicas is expensive. To avoid high system load and large delays, we propose a *hybrid* approach for those file systems that offer log-based update detection.

The synchronizer determines the replica's state information *once*, when creating its snapshot for the first time. Subsequently, operations are retrieved from the log (via the log-based update detection) which are applied to the cached state, producing the current state. The reconciliation then uses operations computed from the state difference between the *database* snapshot (that contains the state of the file system at the last successful synchronization) and the *cached*, current snapshot. A similar approach was proposed by [Li+12a].

4.3.1 Advantages

This hybrid approach has several advantages:

[8] If o_X is a *move* operation, changing the name of i in the database snapshot to a *unique* name will still find i as moved in X. If o_X is a *delete* operation then it will still be deleted in X

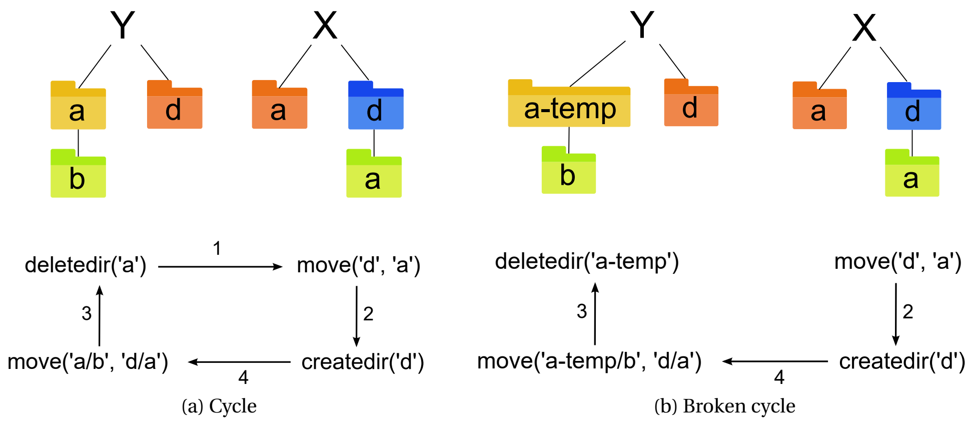


Figure 4.5: Example for breaking operation cycle

Subfigure *a* shows the directory structure on the unchanged replica Y, the changed state on replica X and the four operations computed by *compute_ops()* on replica X. The *actual* operation sequence on X might have been something like *move('a', 'a-temp'), move('d', 'a'), createdir('d'), move('a-temp/b', 'd/a'), deletedir('a-temp')*. The computed operations cannot be applied to replica Y because they form a cycle. Let $o_X = deletedir('a')$ then we can instead apply $r_Y = move('d','a-temp')$ on Y and update the db snapshot. The cycle is converted to a chain as shown in subfigure *b*.

- In case a file system supports any online or offline log-based update detection APIs, these can be leveraged to reduce its load. Since the operation log is applied to a cached snapshot to produce the current snapshot, this effectively converts operation logs to state. In case the log-based update detection becomes (temporarily) unavailable, our synchronizer can always fall back to sampling the file system to build the current state.

- Only a state-based variant of the reconciliation algorithm needs to be implemented, because the update detector can always provide the current state. It is not necessary to implement another variant of the reconciliation algorithm that processes operation logs. Implementing the latter would require significant efforts, using a different algorithm altogether.

4.3.2 Implementation algorithm

The simplified algorithm that detects updates for a file system offering operation logs is shown in algorithm 3. Function `set_up_subscription()` uses the log-based, proprietary update detection API of the file system to subscribe to operation logs. From now on, updates are collected in the background and can be retrieved by calling `get_new_operations()`, which blocks the caller until new operations are available. `apply(ops, snapshot)` takes the operation log and applies it to the provided, cached snapshot, updating `snapshot` internally. The details of this function are omitted, because they depend on the exact log data structure. Applying the log may fail if the log is incomplete or faulty. If applying the logs was successful, the listener (e.g. the reconciliation component) is given the operation log (`ops`) as well as the current `snapshot`. Otherwise, operations are computed from state, using `compute_ops()` from section 4.2.3.

```
Input: root_path to observe, listener whose on_new_operations() method is called
    when updates are detected

set_up_subscription(root_path)
snapshot = create_state_snapshot(root_path)
while True:
    ops = get_new_operations()
    success = apply(ops, snapshot)
    if success:
        listener.on_new_operations(ops, snapshot)
    else:
        # fall back to computing operations from state (slow)
        current_snapshot = create_state_snapshot(root_path)
        ops = compute_ops(snapshot, current_snapshot)
        snapshot = current_snapshot
        listener.on_new_operations(ops, snapshot)
```

Algorithmus 3 : Pseudo-code for hybrid update detection

4.3.3 Caveats and workarounds

While implementing the hybrid update detector component we encountered numerous issues. One general issue is that snapshot creation is affected by race conditions. The function `create_state_snapshot()` may take several seconds up to minutes to complete. During this time period the user may execute concurrent operations. The effect of each of these operations may or may not be contained in the resulting snapshot. We implemented logic in `create_state_snapshot()` to internally apply concurrently detected operations to the snapshot after it was taken. If applying the operations should fail, we restart the snapshot generation.

There are also numerous file system specific issues discussed in the following subsections.

4.3.3.1 Windows

On Windows the `ReadDirectoryChangesW` API fills a size-limited *event buffer* with file system events. The information of each event is limited to a bitmask that indicates the operation type (create, delete, ...) and the affected object path. Most notably, information such as the object's ID, type and lastmodified meta-datum are *not* provided. This results in various caveats:

1. Missing events: when many events occur in a short time period and the event buffer limit is exceeded, events are missed. Because the buffer size cannot be increased indefinitely, the workaround is to fall back to the state-based approach in case a buffer overflow is detected.

2. Bloated rename/move events: when an object is renamed, this causes `ReadDirectoryChangesW` to produce *two* file system events, one that indicates the old, one that indicates the new path. When an object is moved (i.e., its parent directory changes) this causes a *delete* followed by a *create* event. Without further knowledge about object IDs it is impossible to discern actual *move* operations from *delete* + *create* operations.

3. Race conditions when retrieving IDs and other meta-data: because events lack IDs and other meta-data our implementation needs to use the file system's query APIs[9] to retrieve this data after the fact. This may fail because of the time that passes between the operation's execution and

[9]We first open the object's handle using `CreateFileW(path)`, followed by retrieving information via `GetFileInformationByHandle()`, see `https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-getfileinformationbyhandle`, retrieved July 21, 2019.

our implementation calling the query API (even though this happens merely a few milliseconds later). For instance, if the first event indicates the creation of an object at path "x" and the second event indicates that "x" was renamed to "y", querying the information for "x" (when processing the first event) will fail, because "x" no longer exists. Even worse, if the real operation sequence was *createfile("x") + move("x", "y") + createfile("x")*, then querying "x" will succeed, but provide wrong values. We handle such cases by performing extensive plausibility checks that make sure that IDs returned from the query operation, in conjunction with the reported operation type and path, logically match with IDs from the cached snapshot. If the checks fail, we fall back to computing operations from state. To avoid that the event buffer overflows due to spending too much time with applying events, we use a multi-threaded approach. Thread #1 retrieves data from the event buffer and uses the query APIs to complete the information as soon as possible. Thread #1 then hands the events (enriched with query API information) over to thread #2 which applies them to the cached snapshot.

4. Missing directory structure information: when the user moves an existing directory with sub-objects from outside the synchronized directory d into d, then only a *create* event of the new path is reported. Even after we retrieve the meta-data for that path itself, and learn that a new *directory* was created, we still needed to implement extra logic to distinguish whether the user truly created an (empty) directory (and possibly also created other objects inside it soon after), or whether the user moved an existing, non-empty directory into d.

4.3.3.2 macOS

On macOS the *FSEvents* API allows to specify a callback function which FSEvents calls with a list of paths and corresponding event type bitmask that indicates which change(s) occurred. The bitmask indicates whether the object is a file or directory, and the type of operation (create, move/rename, modify, delete). Just as for Windows' `ReadDirectoryChangesW` API, the object's ID, type and lastmodified meta-datum are *not* provided. While FSEvents does not drop events (because its buffers are not limited), we still need to handle several issues:

1. Bloated rename/move events: see Windows. Note: when an object's parent directory changes, the bitmask of each event indicates *move* correctly (not *delete* + *create* as done by Windows)
2. Race conditions when retrieving IDs and other meta-data: see Windows
3. Missing directory structure information: see Windows
4. Event aliasing: FSEvents may *coalesce* multiple operations to a single event in case they all affect the same path within a short time period. There is no documentation how exactly this coalescence is done. For instance, the callback might report two events where the first event is for path "x" and has the *create*, *modify*, *delete* and *rename* bits set, and the second event is for file "y", with only the *rename* bit set. An event may even indicate that it affects both a file and a directory at the same time. Due to this aliasing it is impossible to deduce the exact operations or their order, as there are several valid explanations for these two events. As an example, the user could have performed the following operation sequences:
    a) *delete("x") + create("x") + edit("x", <new content>) + move("x", "y")*
    b) *move("x", "y") + create("x") + edit("x", <new content>) + delete("x")*

Our solution for issue (4) is to no longer rely on the event type bitmask and instead execute the query API on every reported path, deducing operations from the resulting data. For the above example, we would query paths "x" and "y" and look up corresponding objects in the cached snapshot, using the IDs delivered by the query API calls. query("x") would result in an error, and query("y") would either yield the ID known in the snapshot at path "x", indicating explanation (b), or would yield an ID unknown in the snapshot, which would indicate explanation (a).

4.3.3.3 BSCW

BSCW 7 offers a proprietary REST API *sync_events* for offline update detection. It delivers operations, including their type (link, unlink, delete, undelete, create, rename, move), affected object and all its meta-data, including its ID. There are still several caveats to consider:

1. Missing events: to save space the server admin may purge the event history. In this case a special purge-event is inserted, which indicates this fact and allows our implementation to fall back to computing operations from state.
2. Irrelevant events: because BSCW allows an object to be linked into *multiple* parent directories, *sync_events* reports events that are irrelevant for our synchronized directory. For instance, if object o is both linked to the root d_s of the synchronized directory, but also to a directory d_o outside of d_s, then *sync_events* will report irrelevant events (which our implementation needs to discard), such as an *unlink* operation of o from d_o, or a *move* operation that moves o to a different parent directory d'_o under d_o, or a *link* operation of o into yet another directory d''_o outside of d_s.
3. Name aliasing: BSCW does not store historic information of an object's *name*, but only the most recent name. If, inbetween two *sync_event* calls, a user performs the operations *create("x") + rename("x", "y")*, *sync_events* would deliver a *create* event for "y" and a *rename* event that indicates that "y" was renamed to "y", which is superfluous and needs to be discarded.
4. Missing directory structure information: see Windows

4.3.4 Summary

We found our hybrid update detection approach to be feasible and efficient, but challenging to implement. Operation log data provided by file systems needs to be treated carefully, as it may lack important information (such as object IDs) or be affected by aliasing. We can generalize that an update detector of *any* ID-based file synchronizer that supports our examined file systems and relies *only on operation logs* would need to implement *hybrid* approach anyway, using a cached snapshot, because the operation log APIs do not provide sufficient detail on their own.

4.4 Conclusion

The goal of this chapter was to find an update detection approach that is both efficient and provides not only the current state, but also a log of operations. The hybrid update detector we presented solves this problem, by combining state-based with log-based methods. We examined the foundation, state-based update detection, in detail for our file system model $\mathscr{F}$, which supports the *move* operation. We identified the concept of *operation consolidation*, which complicates detecting a valid operation order. Our analysis of the side effects of this consolidation yields not only issues with operation *commutativity* addressed by RQ2, but also the possibility of *cycles*. We presented a sorting algorithm that is based on operation ordering rules which we found by analyzing the pre- and postconditions of the operations of our model $\mathscr{F}$. It solves RQ2 and the problem of finding cycles by iteratively applying the ordering rules to a partially sorted list, until no more rules are apply, or the applied rules start to repeat, in which case a cycle was identified.

Chapter 5

Consistency and conflicts for file synchronizers

The goal of a bi-directional file synchronizer is to achieve eventual consistency between two replicas. We first mentioned the concept of consistency on a *technical* level in section 2.1.1. Achieving consistency between replicas means that *divergences* of each replica, which exist due to operations that were applied to replicas in isolation, are eliminated by applying them from one replica to the other one. Once the consistency algorithm has finished, the state of both replicas should converge *and* incorporate the previously detected divergences[1]. To achieve this, the synchronizer has to propagate non-conflicting changes, and detect and resolve conflicting ones. The most challenging aspect of consistency is conflict handling. Conflicts are combinations of operations that are discovered to be incompatible with each other by the synchronizer during synchronization, even though they were *successfully* executed on each replica in isolation. On a technical level an operation o_X detected in replica X is conflicting with operation o_Y detected in replica Y (and thus cannot be applied to Y by the synchronizer) if the preconditions of o_X no longer hold for Y's new state that resulted from applying o_Y to Y. A typical example is when the content of a file is updated in replica X while that file is deleted in replica Y.

This chapter elaborates on consistency for file synchronizers. We start with an analysis of conflict resolution approaches of related work in section 5.1. With a few exceptions we find the presented conflict resolution methods to be *arbitrary*. They are typically not backed up by a *rationale*, nor do they consider prior work or discuss side effects. Only two works, [TSR15; NS16], present a general set of consistency *properties* which are reflected in the concrete conflict resolution approaches (see section 5.1.2). To not make the same mistake of presenting arbitrary conflict resolution approaches, we design a four-step framework shown in figure 5.1. This top-down approach starts with an informal definition of consistency properties and iteratively refines it to a set of formal and detailed steps for detecting and resolving concrete conflicts. The first step is the definition of a high-level *consistency philosophy* in section 5.2, which is a set of consistency properties that define what it means to achieve consistency, based on the consistency properties of [TSR15; NS16] and our own suggestions. Step 2 defines *conflict resolution policies* presented in section 5.3, which are policies that generally apply for resolving conflicting operations, but are still independent of any concrete conflict definitions. Because many concrete conflicts have common traits, they should also be resolved using a similar approach. We group such similar conflicts to *conflict patterns* and present them in section 5.4. In section 5.5 we present individual conflicts in detail, focusing on the *NH-MD* file system. As discussed in section 2.3.1, there are several other file system definitions. We discuss the relationship between the complexity of each file system and the number of conflicts that need to be detected in section 5.6. Since a file system can be affected by more than one conflict at once, which has not been discussed in related academic works, we present how we *iteratively* resolve conflicts in section 5.7. Finally, we proof the termination properties of our conflict resolution approach in section 5.8.

[1]Consequently, a trivial bi-directional synchronizer that always deletes all files and directories in both physical replicas and the database snapshot would only achieve *convergence* but not consistency.

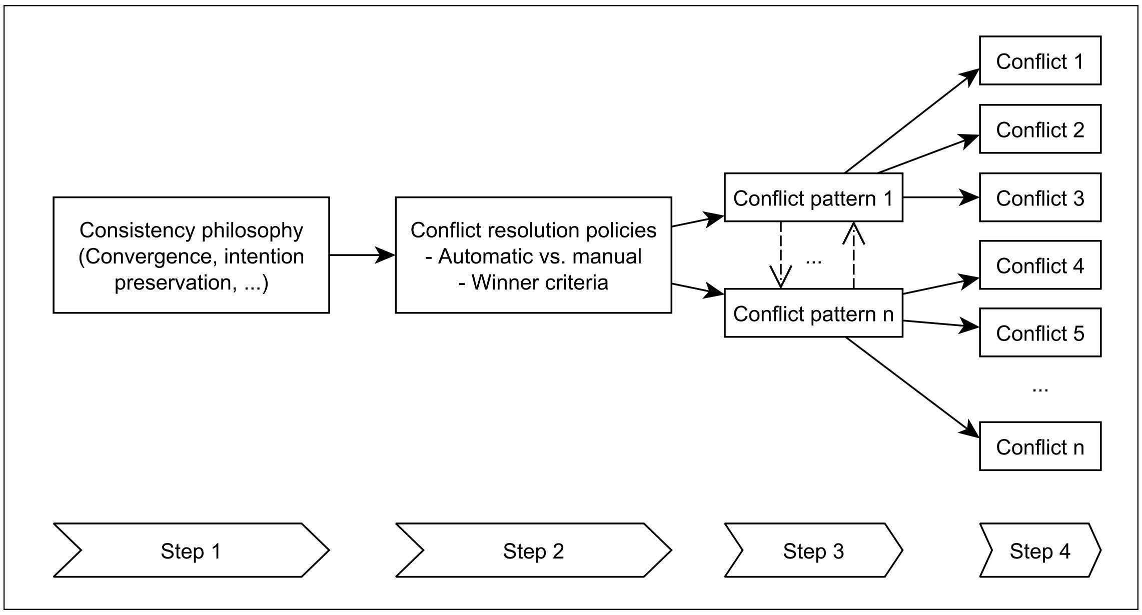


Figure 5.1: Conflict handling framework

5.1 Related work analysis

The goal of the analysis is to determine how academic file synchronizers or Distributed File Systems (DFS) *resolve* file system related conflicts (i.e., *detection* is not our concern, because the mechanisms are very diverse and depend on the file system definition). We look both for concrete approaches for specific conflict types and generic guidelines or policies the concrete approaches are based on. Additional points we examine are the following:

- Is conflict resolution always automatic, always manual, or a mixture of both?
- How clear are conflict resolution approaches described?
- Given a clear description, is there a *rationale* provided for the specific approach?
- Do authors compare their approach with other approaches, to defend their rationale?
- Are any conflict *awareness* mechanisms being discussed?

We present the findings of our analysis in section 5.1.1. We keep the analysis on a high level. The description of the resolution of specific conflicts is deferred to subsections of section 5.5, after we described the conflicts themselves in detail. In a few works we found generic consistency properties, which we refer to as "consistency philosophies" which we describe in section 5.1.2 on page 69.

5.1.1 Conflicts

In this analysis we examined the works from section 2.4 as well as the DFSs *Ficus* and *Coda*. An overview of the results is shown in table 5.1.

Resolution mode

A first glance at the *Resolution mode* column reveals a large degree of diversity. While some systems are fully automatic or fully manual, others are hybrid. *Hybrid* systems get their name either because they don't mention the mode at all [CJ05], allow users to configure which mode they prefer [UFB10], or

	Type	Resolution mode	Clarity of conflict *resolution* description	Remarks (e.g. introduced winner-criteria or philosophy)
[Guy91; Rei+94] (Ficus)	DFS	Some automatic (call user-made resolver), some manual	Clear for automatic, unclear for manual	1) *No lost update* criterion, applies to concurrent file update + delete. Resolve by move to orphan directory 2) Email report provides conflict awareness
[Sat+90] [KS91] (Coda)	DFS	Some automatic (call user-made resolver), some manual	Clear for automatic, unclear for manual	1) Repair tool user invokes to manually resolve conflicts
[BP98; PV04]	Sync.	Manual	None	Unison implementation offers several options to resolve conflict
[Bao+11]	Sync.	Automatic	Only for file content update conflict. Mentions but ignores other conflicts	–
[Bjø07]	Sync.	Automatic	None	–
[CJ05]	Sync.	Automatic or manual	None	–
[Mol+03]	Sync.	Automatic	Clear (formal, via OT transformation functions)	Object unique ID comparison used as criterion to choose a winner
[Naj16]	Sync.	Automatic	Clear	–
[RC01; Csi16]	Sync.	Manual	None	Discusses conflict encoding into file system vs. special GUI
[TSR15]	Sync.	Automatic	Clear	Presents philosophy that conflict resolution approaches (mostly) follow
[UFB10]	Sync.	Automatic or manual	Only for file content update conflict.	–
[Li+12a; Li+12b]	Sync.	Automatic	Clear	–
[LKT05]	Sync.	Some automatic (speculative merging), some manual	None	–
[NS16]	Sync.	Automatic	Clear	Presents philosophy that conflict resolution approaches follow

Table 5.1: Analysis of conflict resolution approaches

because the mode depends on the conflict. That is, some conflicts can be resolved automatically and some require manual user intervention.

Clarity of conflict resolution

The *clarity* column demonstrates that out of the 14 analyzed works five provide a fully clear description, five do not describe conflict resolution at all, and the remaining four only describe it partially. We would have expected works which use *manual* resolution to describe which resolution *alternatives* are presented to users for them to choose from. However, the majority of works does not provide any alternatives. Coda mentions a conflict "repair tool", but it is never illustrated or explained.

Resolution rationale

We found that the majority of works which describe their resolution do *not* present any *rationale* to explain their concrete conflict resolution approaches, nor do they compare their approach with related work. There are, however, a few notable exceptions:

- Ficus mentions the "no lost update rule" which states that if a file was deleted in one replica but modified in the other, the file should be restored to avoid losing the update. However, the resolution of other conflict types is not following any identifyable rules.
- In [NS16] and [TSR15] the authors define a *consistency philosophy*, which they follow in their concrete resolution implementation. We present further details in section 5.1.2 on the facing page. The authors make several specific choices, however, not covered by the philosophy, which seem to be *arbitrary*. For instance, if a specific name is concurrently occupied (via *create* or *move* operations) on both replicas, [NS16] decide to resolve the conflict by renaming *both* objects, while [TSR15] use a different approach for each object type combination (file-file, file-directory, directory-directory) and provide a rationale for only one of these combinations.
- In [NS16] the authors evaluate the convergence behavior of their own work with those of industrial synchronizers Dropbox, Google Drive and Microsoft OneDrive. In this evaluation the authors briefly discuss advantages of their own conflict resolution approaches, *comparing* it to the ones applied by industrial tools. For example, the authors note that if an industrial synchronizer recursively merges the sub-objects of two independently created directories (where the directories are located in the same parent directory and using the same name), unmerging is a painful process for the user in case the automatic merge was inappropriate for that specific directory.
- Sometimes the rationale is dictated by the specifics of the underlying *algorithm*. In [Naj16] the authors use *CRDT sets* and *maps* to model the file system. For those CRDT data types the conflict resolution mechanism for concurrent *add* or *remove* operations is limited to "add wins" or "remove wins". Consequently, update/delete conflicts (e.g. where one user updates a file that the other user deletes) are resolved using one of these choices.

Winner-criterion

While most works define no consistency philosophy or general guidelines, we identified several *winner-criteria* that can be used to identify the winner of a conflict. These include:

- Object type, e.g. *directory* wins over *file*, see [TSR15]
- Replica, e.g. let a *specific* replica's operation take precedence, or choose a *random* replica, see [Li+12a; Li+12b]
- Object ID, e.g. the object with higher ID wins, see [Mol+03]
- Timestamp, e.g. last writer wins (LWW), see [UFB10]
- No lost [file content] update, i.e. let file modifications win over deletions, see [Rei+94]

Conflict awareness

Most works do not discuss how to make the user aware of conflicts or their resolution. Some DFS such as Ficus or Locus [Wal+83] mention *email reports*, but provide no specifics of the email's content, nor whether reports are sent for automatically resolved conflicts.

Final remarks

In [RC01; Csi16] the authors note that when a synchronizer detects a conflict, it can either:

1. Keep knowledge about the conflict at the site of its discovery, making it accessible via a special user interface of the synchronizer. Only one user is aware of the conflict. Files affected by the conflict remain in their old state (diverging from other replicas) until the user chooses a specific resolution.
2. Encode the conflict's *detection* (and optionally a preliminary *resolution*) into the file system, which allows replicas to immediately converge, making *all* users aware of the conflict.

The second alternative demonstrates that if a synchronizer claims to *resolve* a conflict automatically, e.g. by appending a unique suffix to an object's name in case of a name clash, it has in fact only encoded the conflict's *detection* into the file system, using a preliminary resolution. The conflict is only truly resolved once a user performs further actions on the automatically renamed object.

5.1.2 Consistency philosophies

The two works [TSR15] and [NS16] present high-level consistency philosophies which we now review.

[TSR15] define that the common state of a file system is ”meaningful” to the users *“when the users can still see the effect of their own updates”* after the synchronization finished. They define the two properties *element preservation* and *relationship preservation*. Element preservation means that (1) *“if the user updates any file or directory, the updated elements should also be available in the merging outcome”*, and (2) that separate objects are not merged into a single object. Relationship preservation requires that path-related changes made by the user on a replica should be visible after synchronization. The underlying principles of the properties are that not a single update should ever be lost (even in the face of conflicts), and that merging the changes of two replicas should have no *side-effects*, such as anomalous results where objects unexpectedly disappear.

While most of these statements intuitively make sense, it is easy to see that the first underlying principle, “no lost updates”, cannot possibly hold for every conflict[2]. The authors even provide a counter example in their work. They define the “state conflict”, which is concerned with deleting an object in one replica and updating it (e.g. the file's content) in the other one. The authors chose the *update* operation to take precedence over the *delete* operation in the merged file system. The *delete* operation is effectively lost, which contradicts their *relationship preservation* property and the underlying “no lost updates” principle.

A slightly different philosophy is defined as a set of *consistency requirements* in [NS16], where the consistency reflects the limitation that not *all* updates can be preserved:

- Causality preservation: causally related operations must be executed in their causal orders,
- Convergence: after all operations were sent to all replicas, the state of all replicas must be equal,
- Intention-confined effect: operations applied to replicas by the synchronizer must be based on operations generated by the user,

[2]We note that in [TSR15] the definition of “update” goes beyond the operation that changes the content of files (which was the narrow definition used in [Rei+94]), but also includes other operation types.

- Aggressive effect preservation: *"effects of compatible operations should be preserved fully; effects of conflicting operations should be preserved as much as possible"*.

These four properties are a variation of the consistency model from Operational Transformation (OT), see *convergence*, *causality preservation*, and *intention preservation* in [SE98]. These OT-based papers are founded on work in the area of distributed systems with seminal works like [Lam78]. We note that *Causality preservation* applies to *operation*-based systems, but not *state*-based systems such as the synchronizer we build in this work, where the exact operation sequence is not available.

Finally, we note that both [TSR15] and [NS16] defined their consistency philosophies with fully automatic conflict detection and resolution in mind.

5.2 Consistency philosophy

In this section we define the consistency philosophy we use in our synchronizer. We cherry-pick properties from the two philosophies presented above, refine them and add additional properties.

Like for [TSR15] and [NS16] our philosophy is designed for *automatic* conflict resolution mode. We prefer *automatic* resolution for two reasons. First, user involvement to keep replicas synchronized becomes *optional*. Users only need to act and repair the file system if the automatic resolution was inappropriate in that specific case (in the eyes of the user), which allows them to concentrate on their actual work. Second, our algorithm is built such that it requires to first resolve conflicts before non-conflicting operations can be applied (see section 6.1). By automatically resolving conflicts we can immediately propagate the non-conflicting operations, too, which reduces the risk that new conflicts occur for those non-conflicting files and directories. Section 5.3.1 provides further details regarding the automatic vs. manual resolution mode.

Our consistency philosophy has the following properties:

- *Convergence*: after a completed synchronization, the state of all replicas must be equal,
- *Impact*: operations applied to replicas by the synchronizer must be based on operations generated by the user,
- *Intention preservation*: the synchronizer applies non-conflicting operations from one replica to the other one without modification. For conflicting operations, the synchronizer preserves the intention of both operations as much as possible, by modifying *one* of the operations. If preservation is not possible the synchronizer discards the operation that required less effort to generate (for the user). At the same time the synchronizer minimizes undesirable, manual clean-up work for the user in case the automatic resolution was inappropriate. The synchronizer should store sufficient information to allow the user to *retroactively* choose a different resolution option for a conflict, in case she considers the default approach to be inappropriate.
- *Awareness*: if the synchronizer applies a non-conflicting operation from replica X to Y, the user of Y must be able to inspect the operation history *on demand*, to understand which file system changes originate from replica X. If the synchronizer modifies or discards conflicting operations generated by the user, the user must be actively notified (e.g. via a push notification). This improves transparency of the actions done by the synchronizer.

In comparison to previously presented philosophies such as [NS16] we added two major contributions.

First, we refined the *intention preservation* property by (1) specifying that each conflict has a winner and a loser, (2) the synchronizer should prefer to *modify* the losing operation - and only if that fails, *discard* the losing operation, (3) the modified/discarded operation is chosen according to the amount of effort put into it by the user (if possible to estimate), and (4) the resolution approach considers the ease of undoing the resolution if the user found that specific resolution to be inappropriate for the specific conflict.

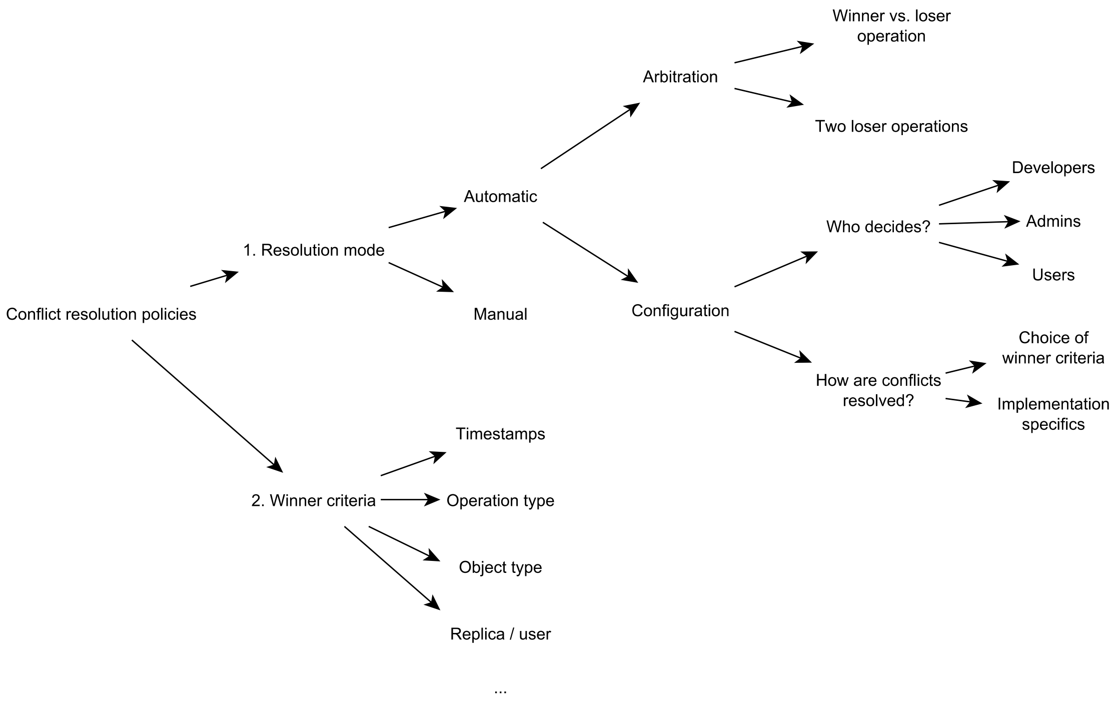


Figure 5.2: Overview of conflict resolution policies

Second, we added the *awareness* property. Awareness, which [DB92] define as *"an understanding of the activities of others, which provides a context for your own activities"*, is a well-researched topic in CSCW literature. Various classifications (e.g. *workspace* awareness [GG02]), awareness types (e.g. activity or process awareness [CSS09]), models (e.g. event-based [Pri93] or spatial models [BF93]) have been identified. According to [FPP95] our awareness property asks for *asynchronous* awareness, which is either tightly coupled (for conflicting operations) or loosely coupled (for non-conflicting operations affecting objects the user of replica X did not work on).

5.3 Conflict resolution policies

Conflict resolution policies are a set of rules that refine the conflict-related parts of the above consistency philosophy, specifically the *intention preservation* property. An overview is shown in figure 5.2.

5.3.1 Resolution mode: automatic vs. manual

5.3.1.1 Choosing an approach

The first, most important decision to make is whether to resolve conflicts automatically or manually[3]. As stated in [RC01], choosing between *automatic* and *manual* conflict resolution is still an open problem. The advantage of *manual* resolution is that the user has full control, which may increase her trust in the synchronizer, because resolution actions have to be confirmed. Users also avoid subsequent, unnecessary clean-up work that would have been required if the synchronizer had automatically resolved a conflict inappropriately. The downside is that users may feel annoyed by being requested to act. This particularly affects users who often work offline for extended periods of time, which increases

[3]We note that a synchronizer can make *different* choices (automatic vs. manual) regarding the (1) general synchronization, and (2) conflict resolution. For instance, a synchronizer may be configured to synchronize automatically whenever changes are detected, but still enforce manual conflict resolution. This section focuses solely on aspect (2).

the chance for conflicts. Additionally, this would increase the risk that new conflicts occur for *non-conflicting* files in case a synchronizer's algorithm requires that all conflicts have to be resolved first before non-conflicting operations can be applied.

The advantages and disadvantages of *automatic* resolution can be derived from inverting those of *manual* resolution. Users benefit because they save time and effort in case the resolution was appropriate. However, in cases where the user considers a specific resolution to be inappropriate, the costs incurred for subsequent, manual repair must be manageable. We refer to section 5.4.3 for a concrete example regarding two conflicting *createdir* operations.

5.3.1.2 Challenges of automatic resolution

The challenge for automatic resolution is to find default settings that are appropriate for the variety of workflows and scenarios in which the synchronizer is used. The following points outline different choices a synchronizer developer must make.

First, the synchronizer needs to *arbitrate*, i.e., choose winners and losers. When two operations conflict, there are two basic ways of arbitration:

1. The synchronizer picks a *winner* and a *loser* operation, based on some criterion (see next section). The winner operation is applied as is to the other replica, the loser operation is first modified in some way, depending on the conflict. In the most extreme case, the loser operation is discarded completely. The advantage is that only the loser has to be notified about the conflict, because only her operation was not applied as originally intended.
2. The synchronizer considers *both* operations as *losers*. Both operations require modification. This approach avoids having to choose a winner-criterion, at the expense of preserving neither user's original intention of the operation.

For our own implementation we decided to choose a winner and loser operation to maximize intention preservation.

The second choice is the *configuration* of automatic resolution, which can be split into *who* decides how to resolve conflicts, and *how* is each conflict resolved.

***Who* decides?** In the majority of file synchronizers we found the *developer* of the synchronizer makes this choice and hard-codes *one* specific resolution mechanism into the algorithm. We find this approach problematic due to its lack of flexibility and instead suggest that the synchronizer offers *several* resolution approaches. However, who should be allowed to pick from those options?

1. Give *administrators* this choice. Compared to end-users they have a higher technical expertise required to choose resolution approaches on an abstract level without concrete examples. However, they may not be familiar with exact workflows of the users. There might also be *several* groups of users with different workflows, which further complicates the selection for administrators.
2. Let *users* decide. This improves control over the program, but if implemented incorrectly (e.g. showing an overloaded technical dialog such as shown in figure 7.1 on page 130) the choice may overburden non-technical users and allow them to pick options that are often inappropriate.

We suggest to let *users* decide. To avoid overburdening them, users should be able to tweak the conflict resolution on a case-by-case basis. Whenever a conflict occurs, the synchronizer resolves it automatically using some pre-configured option and notifies the user. This opens a graphical dialog with several functionalities:

- It explains the conflict to the user. This includes providing context the user needs to understand and resolve the conflict. For instance, if a file was updated by two users, the synchronizer can open both conflicting files for the user with a single click (or even open a comparison view if the corresponding application offers one).
- If the user considers the applied resolution as appropriate, she can just confirm that she has seen the conflict.
- Otherwise she can ask the synchronizer to retroactively apply a different approach, from a set of alternative resolution options. This way the user is not burdened with configuring resolution options for *other* conflict types she has never seen (but just the one at hand), and is given a concrete example she can understand. The synchronizer can learn from the user's choices over time, e.g. applying the user's previous choice the next time a similar conflict occurs. A challenge left as future work is how exactly the learning algorithm works, including how the user can influence the learning process.

***How* are conflicts resolved?** Any resolution approach requires two decision steps. First, the choice of a *winner-criterion* (see next section) that defines the winner operation (in the presence of a multitude of criteria, which may also be ranked). Second, the specifics of how the synchronizer manipulates/discards the loser operation and executes the winner operation. Because the concrete choices depend on the conflict type we discuss our suggestions in the corresponding conflict sections.

5.3.2 Criteria for choosing a winner in automatic resolution

We collected numerous criteria from our related work analysis and user suggestions, and added our own ideas. The following list explains and criticizes each criterion.

- *Timestamps*: the "last writer wins" (LWW, [JT75]) approach is well known in research [SS05] and practice and has also been adopted by [UFB10]. The general idea is that *newer is better*. We frequently observed that this is not necessarily true in practice and heavily depends on the concrete scenario. Consider an example where a user revises a document on a laptop computer for 3 hours on a train ride, returns home, turns on the desktop PC and fixes a typo in the same document which has not been synchronized yet and thus contains the outdated version. It would be very inappropriate if the synchronizer overwrote the document on all machines with the typo-fixed version, only because it is newer. We advise against using this criterion, because a single timestamp bears no information w.r.t. the work put into the affected operation. There are also many other issues with timestamps. Some file systems may not provide them for all operations[4]. Several types of timestamps may exist (timestamp of when a change was registered on a *client* vs. *server*). Also, wall-clock-based timestamps can be tampered with or may accidentally have lost synchronicity, making it difficult to even decide who is the last writer.
- *Operation type*: suppose two conflicting operations have a *different* type (e.g. *move* vs. *createfile*, or *edit* vs. *delete*) then we can choose one to take precedence. The "no lost update" rule of [Rei+94] is an example for this criterion. It is not conclusive if both operation types are equal. As a general framework we propose that operations are ranked by the *amount of work* that presumably went into it. We propose the following simple order:
  - *Delete* operations are ranked lowest - the user only needs to choose a target object and press a button to delete it.
  - *Move* operations require selection of source and target object, which may involve typing characters if the object is renamed. Similarly, *createdir* operations require the selection of the parent directory and choosing the directory's name.

[4]For instance, the time when a file was renamed or moved is not available on many file system implementations.

  - *Edit* and *createfile* operations come last (and are ranked equally) because they involve changing content, which is presumably a lot more work.

- *Replica/user*: the synchronizer selects the operation of a specific replica or user as winner. This is a suitable fallback criterion if other criteria are inconclusive or make no sense. We found that a good choice is to prefer the operation performed on the *server* replica for synchronizers deployed in a client-server star-topology setup. The server's replica reflects the current state affected by (potentially) *many* users, whereas the client's local replica is typically only affected by a *single* user. We think that a conflict caused by one user should not negatively affect the work done by other users, especially since these users already have agreed on the common server replica state.
- *Object ID*: this approach is presented in [Mol+03]. The (unique) object IDs of the two objects affected by the conflicting operations are compared to determine the winner operation. We do not recommend this approach because users are typically not aware of object IDs and thus the resolution result will appear *arbitrary* to the user.
- *Object type*: in [TSR15] the authors use this criterion for conflicts related to name clashes. If one operation affects a directory and the other one affects a file, then one type can be chosen to take precedence over the other. This criterion is inconclusive in case both objects have the same type.

We note that a user-centered design approach makes sense where users are involved in adding new domain-specific winner-criteria. For instance, the following criteria were proposed by members of the author's organization:

- Estimate the *amount of work* for two conflicting edited files by counting the number of modification file system events of each replica since the last synchronization. The user who updated a file more often is assumed to have done more work and thus her file should win the conflict. More elaborate versions of this criterion can also be built, e.g. including the delta of file's size after each modification. However, this criterion may only make sense for specific file types (where the amount of work is reflected in the byte size delta) and user behaviors (where users or the application actually updates a file regularly).
- *Time of day*: for example, in certain workflows the changes that occur at late hours may be more important than changes registered in the afternoon. However, the same disadvantages as mentioned for *Timestamps* above do apply.
- *File type* (extension): is not actually a criterion, because by itself it is always inconclusive. However, it can be used as a precursor in combination with another criterion. For instance, the configuration could define that for concurrent file edit conflicts, the *replica* criterion is fixed to "server file wins" for *document* files, whereas it is fixed to "client file wins" for *spreadsheet* files.
- *Namespace location*: similar to the *file type* presented above the location in the namespace may be a precursor for another criterion, s.t. conflict that affect objects in one part of the hierarchy are resolved differently than those situated in another part of the hierarchy.
- *User*: operations made by specific user accounts (or groups) may take precedence over the operation made by another user/group.

The set of criteria may also be combined to a *cascaded* set of criteria with different priorities. For instance, the concurrent file edit conflict might be configured as "Server Replica wins" (lowest priority used as fall back) + "Change made by Prof. Pepper wins" (higher priority) + "Client changes on *document* files win" (highest priority). Due to the complexity we left implementing user-proposed winner-criteria and cascaded winner-criteria as future work.

Pattern	Related concrete conflicts	Affects name	Affects content	Intentions preservable
Pseudo conflict	Create-Create, Link-Link, Move-Move (Source), Move-Move (Dest), Edit-Edit	✓	✓	✓
Name clash conflict	Create-Create, Link-Link, Create-Link, Move-Create, Move-Link, Move-Move (Dest)	✓	X (✓ for Create-Create)	✓
Edit conflict	Edit-Edit	X	✓	✓
Delete conflict	Edit-Delete, Move-Delete, Link-Unlink	✓	✓	X
Move conflict	Move-Move (Source)	✓	X	X
Indirect conflict	Move-ParentDelete	✓	X	X
	Create-ParentDelete	✓	✓	✓
	Move-Move (Cycle)	✓	X	X
	Node-typing	✓	X	✓

Table 5.2: Overview of conflict patterns

5.4 Conflicts patterns

Our consistency philosophy and conflict resolution policies are high-level concepts which are consistently applied to the concrete conflicts presented in section 5.5. Because many conflicts share some characteristics, they should also be resolved in similar approach. The *conflict patterns* presented in this section group these conflicts to a set of conflict patterns.

Conflict patterns borrow ideas from Alexander's *Pattern Language* [Ale+13]. At its core are *design patterns*, which refer to each other, forming a pattern language. Design patterns describe problems and possible solutions in a structured way. They are written from a high-level perspective such that their solutions typically require adaptation to solve the concrete problem at hand. Patterns aren't considered perfect, but rather being hypotheses of what is most likely an appropriate solution for a given problem. To quote Alexander,

> [...] each pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented.

Although the origin of the pattern language is in architecture, it has been applied to many other domains, including software engineering [Gam+77]. We apply the pattern language to file system conflicts. We note that even though this domain is very specific, it is straightforward to abstract our patterns and their foundation (the file system) s.t. they apply to generic graph structures whose operations follow similar rules.

We start with an overview of patterns in section 5.4.1 and then present each pattern in sections 5.4.2-5.4.7. From these patterns we derive the list of concrete conflicts presented in section 5.5.

5.4.1 Overview

An overview of conflict patterns is shown in table 5.2. The *related concrete conflicts* column shows the relationship between *conflict patterns* and *conflicts*. The *Intentions preservable* column addresses whether the intention of *both* conflicting operations can be preserved - either completely or at least to a large extent. Where this is not the case, one of the operations typically has to be discarded by the synchronizer.

The general naming scheme for a conflict is <*operation on replica X*> **-** <*operation on replica Y*>.

For instance, *Move-Create* means that a *move* operation of one replica conflicts with a *create* operation on the other replica. Some conflicts such as "Move-Move (Dest)" deviate from this scheme slightly. The corresponding conflict sections explain the deviation in more detail.

5.4.2 Pseudo conflict

Description: At each replica an operation with the *exact same outcome* was applied. A pseudo conflict may also be referred to as *false conflict* in other works.

Solution: Such kinds of conflicts can be solved without data loss and don't require user notification. The synchronizer only needs to avoid that the same operations are detected again in the future, by updating its internal database to reflect the change. The replicas do not have to be manipulated, because they both already have the same operation applied to them. A synchronizer that correctly differentiates real vs. pseudo conflicts enable maximum conflict avoidance.

Conflicts:

- Create-Create: creating a *file* with the same name under the same parent where the content is the same on both replicas
  - Note: the concurrent creation of a *directory* is not listed here. We argue that if it were a pseudo conflict, the effect would be that the synchronizer recursively merges sub-objects, which may not be the intention by the user.
- Link-Link: creating a link to the same file in the same parent and name
- Move-Move (Source+Dest): moving the same object to the same parent and name
- Edit-Edit: editing the content of a file such that the content is equal on both replicas

To verify that two files have the same content (Create-Create, Edit-Edit), *checksums* are used. Various algorithms exist, such as MD5 or SHA-1. These algorithms take binary data of *any* length (the file's content in our case) and produce a short, *fixed*-length output, the checksum, which can be compared with negligible costs regarding network bandwidth and CPU time. There are two disadvantages. First, computing them takes at least as much time as reading the entire data. Second, two files with *different* contents can theoretically produce the *same* checksum, however, with a very low probability. We use checksums because of the following advantages:

- Transmitting large files, which is extremely costly over slow networks, can be omitted in case of an equal checksum.
- Checksums avoid that the synchronizer causes Create-Create or Edit-Edit conflicts when the synchronization process was interrupted by the user while the synchronizer transmits a file to the remote server. This is an implementation-specific detail due to the lack of transactions in some protocols such as HTTP, and is further explained in figure 5.3.
- Support for *migrating* all objects of a replica from one location to another one without the need of retransmitting sub-files: The user may desire to change the physical location of either replica (i.e., on the local disk or on the remote server), e.g. because of quota restrictions or because she wants to reorganize the file system namespace for any reason, then the use of checksums enables the following workflow:
  1. User deletes the configured pairing of the two root directories in the synchronizer user interface. This deletes all synchronization-related meta-data, including the local database.

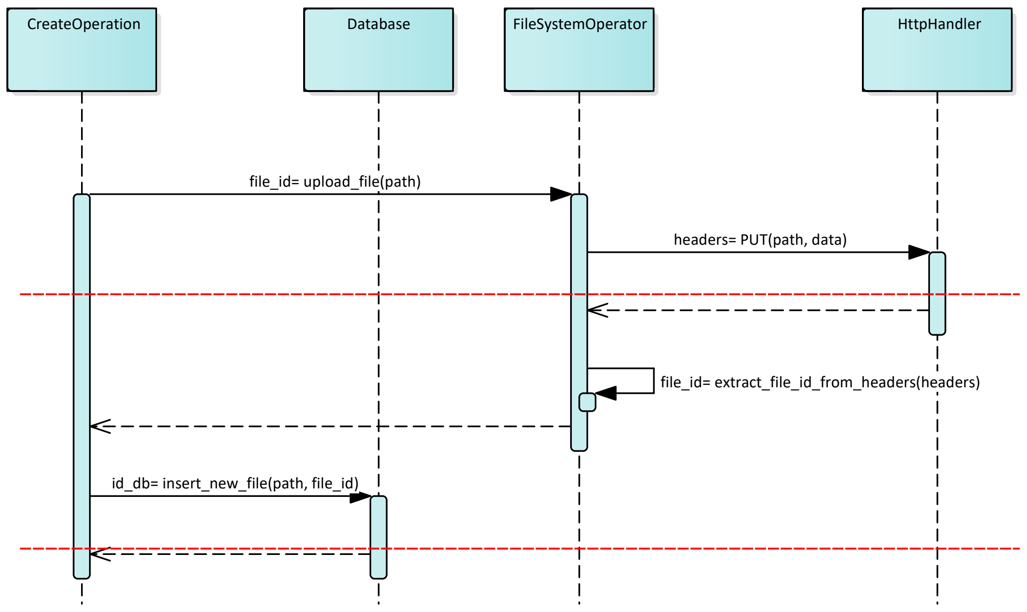

CreateOperation
Database
FileSystemOperator
HttpHandler
file_id= upload_file(path)
headers= PUT(path, data)
file_id= extract_file_id_from_headers(headers)
id_db= insert_new_file(path, file_id)

Figure 5.3: Uploading a file to a server

This figure is an UML sequence diagram, illustrating the upload process of a file. Time flows from top to bottom. The synchronizer executes a *CreateOperation*, whose first task is to upload the file, expecting the file's server-replica-specific ID, followed by inserting the name, file ID (and other meta-data such as lastmodified meta-data) into the database. Assume that at the first red dashed line the file has been fully received by the server and was stored in its persistent storage. The *CreateOperation* can be aborted at any time, either by the user, or, say, by a power failure or program crash. Should this happen anywhere in-between the two red dashed lines, then the upload is complete, but the change is not reflected in the synchronizer's database yet. When the synchronization is restarted, the file would be detected as a Create-Create conflict when not using checksums.

2. User physically moves the root directories to a new location on either replica, using the tools they are used to (such as a file manager).
3. User sets up a new pairing in the synchronizer user interface. The initial synchronization will detect Create-Create pseudo conflicts for all files and directories[5]. Effectively, no files are transmitted, only the local database is repopulated.

5.4.3 Name clash conflict

Description: The *create*, *link* or *move* operation o_X detected in replica X and *create*, *link* or *move* operation o_Y detected in replica Y both target the same name within a specific parent dir. However, a name may only be used once within a parent dir.

Exception: See *Pseudo conflict* pattern.

Solution: In general there are two approaches which preserve both user's intentions:

[5]In this case, concurrent creation of directories would need to be considered as pseudo-conflicting.

1. *Merge* the content of the two objects affected by o_X and o_Y,
2. *Rename* one or both objects. In the latter case finding a more suitable name for the objects is left to the user after being notified about the conflict.

We note that there are other options for resolving name clash conflicts, e.g. discarding one operation by deleting the affected object. However, such options would completely discard the intention of one of the users and thus we do not follow up on them any further.

Which solution is better depends on the operation and the type of the object. We generally recommend the *rename* approach, because when the users generated the operations, their intention was for o_X and o_Y to modify *different* objects. We agree with [TSR15] in their assumption that the user wants the synchronizer to keep both objects separate, instead of merging them. Merging has several caveats (see section 5.4.4 for issues when merging *files*), most notably that unmerging, in case of a bad merge, is more tedious than keeping objects separate in the first place, deferring the merge to the user. The authors of [Naj16] disagree and propose to always merge, but do not explain their decision.

One exception where *renaming* may not be appropriate is the case where $o_X = o_Y = createdir$. All tested industrial synchronizers (see section 8.2.5.4) and [TSR15; Naj16] prefer to *merge* the contents of the directories recursively. While [TSR15; Naj16] don't provide a rationale for this decision, we understand that merging directories avoids conflicts. This is important when establishing a new synchronization between two replicas (for which no synchronization was set up before), but whose file system are already equal (as was explained above in section 5.4.2). Other works [Mol+03; NS16] prefer *renaming* over merging for two *createdir* operations. In [NS16] the authors argue that *"the merging is not the intention of any operation involved"*, merging directories recursively may involve finding and solving conflicts for sub-objects which is *"difficult for users to understand and complex to implement by the underlying system"* and merging *"may create a nontrivial burden to users who need to unmerge the merged subtrees"*. We consider the best solution to be a mixture of both approaches: generally prefer *renaming*, but initially switch to *merge* mode temporarily until the very first synchronization has completed.

Whenever name clash conflicts are solved by *renaming*, the author of the synchronizer needs to choose whether to rename *one* or *both* objects. In either case we recommend that objects are renamed by appending a globally unique suffix to their name. Other helpful information to include for the new name are the date and time of conflict discovery, or the name of the replica or user. When renaming *both* objects, the synchronizer implicitly considers both replicas to be negatively affected and should consequently notify *all* affected users. When renaming just *one* object, one replica is implicitly chosen as the winner and only the loser needs to be notified.

As for the winner-criterion (section 5.3.2) different criteria can be used for chosing the winner, such as *object type*, *operation type* or *replica/user*. In our implementation we use only a *single* criterion, *replica/user*, to make conflict resolution predictable and comprehensible for the user. The first two criteria (*object type*, *operation type*) are inconclusive in case o_X and o_Y use the same object or operation type. As discussed in section 5.3.2, we suggest to prefer the operation performed on the *server* replica for synchronizers in a client-server star-topology setup.

5.4.4 Edit conflict

Description: The content of a file was changed on both replicas.

Exception: See *Pseudo conflict* pattern.

Solution: The goal is to preserve both replica's changes. There are several approaches:

1. *Merging*: the synchronizer merges the content of both files. It needs to support specific file formats (in [Mol+03] the authors support text and XML files). If the synchronizer is not integrated into the application used to manipulate the file, this is done by a state-based 3-way-merge approach. Merging has several disadvantages, which is why we do not recommend this approach:

- The 3-way-merge procedure needs to be implemented for each file format separately. This is infeasible in practice, due to the large number of proprietary file formats.
- Either the complete history (for log-based approaches such as OT) or a base version (for 3-way-merge) is required to merge both replica's changes. The latter may not be available. In particular, basic file system implementations such as those of macOS, UNIX and Windows do not store older versions of a file by default.
- The merge itself may fail due to conflicts within the data, requiring user intervention.
- If the merge is successful, the result is a *syntactic* merge [SS05] which may be *semantically* incorrect.[6]

2. *Renaming both files:* like for *name clash* conflicts, each file is renamed to a unique name. This approach is applied in [TSR15]. All affected users should be notified.

3. *Renaming one file:* similar to *name clash* conflicts this requires to choose a winner and rename the file of the loser (notifying the corresponding user). Renaming the loser can be implemented by a simple rename operation, or moving the file to a different location, possibly outside of the synchronized namespace, to preserve the loser's changes. To select the winner there are several criteria available, such as:

   - *Last-modified timestamps:* the file with the higher timestamp is assumed to be winning copy [UFB10]
   - *Replica or user:* (see *name clash* conflicts) [Bao+11; Li+12a; Li+12b]

For the typical setup where synchronization happens in a client-server star-topology, we recommend approach 3, with the *server* replica's file taking precedence. We advice to *not* synchronize the loser's renamed file to the server (but keep it only on the local user's replica). This avoids unnecessary bandwidth use and makes sure that other users not involved in the conflict are oblivious of it.

5.4.5 Delete conflict

Description: On one replica an object is moved or edited, on the other replica the corresponding object is deleted (either directly or as a consequence of deleting a parent directory).

Solution: Whether deletion or the edit/move operation should take precedence depends on the environment in which the synchronizer is used, hence we don't give a recommendation for the preference itself. While using a criterion such as *replica/user* would be possible, the majority of works from research and industry use the *operation type* criterion, with the configuration to prefer the move/edit operation [NS16; Li+12a; Li+12b; TSR15; Rei+94; Naj16]. This is in line with our philosophy to discard the operation that was less work for the user. Given a selected preference, there are different ways to implement the manipulation of the loser operation:

- If deletion is preferred:
  - If the object is a file:
    - If it was only moved, delete the moved file
    - If it was (also) edited, the edited copy should be backed up prior to its deletion to preserve the intention by the user who edited the file
  - If the object is a moved directory, it should be deleted, but the implementation first needs to check for sub-node conflicts and resolve these. This can be achieved by undoing the move operation on the corresponding replica.

[6]For instance, if the base version is a text file that contains "The dogs jumps", which is modified in replica X to "The dog jumps" and in replica Y to "The dogs jump", the merged result will be "The dog jump" which is semantically incorrect.

- If the edit or move operation is preferred, this should lead to a reconstruction of the object on the replica that deleted it (in case the object was a dir: including the reconstruction of its sub-objects), at the new location (the move operation's destination).

5.4.6 Move conflict

Description: On both replicas the *same* file or directory was moved to a different location. That is, on each replica either the name or parent directory (or both) differs. Note: if the operation affects a *file* on a file system with *hardlink* support, this is *not* necessarily a conflict because the synchronizer can create *two* links for the file in the merged result.

Exception: See *Pseudo conflict* pattern.

Solution: For file systems where a file or dir may only exist once, it is difficult to preserve the move-intention of both replicas. One possibility is to create a *copy* of the object, as done in [TSR15] for directories. However, this approach no longer resembles the original intention of the user, because a *second*, different object has come into existence, cluttering the file system and confusing the user. We suggest that the synchronizer discards the move operation of one replica and prefers the one of the other replica, which was also implemented in [NS16; Mol+03]. We use the *replica/user* criterion in our implementation, configured to let the server replica's move operation take precedence.

5.4.7 Indirect conflicts

Two operations indirectly conflict with each other if they don't target the same object or name, but different objects. There is always a hierarchical parent-child relationship between the objects affected by the operations. Refer to sections 5.5.6+5.5.7+5.5.10 for the description and solutions of the conflicts *Move-ParentDelete*, *Create-ParentDelete* and *Move-Move (Cycle)*. The *Node-typing* conflict is described in appendix A.2.4 on page 201.

5.5 Conflicts

This section presents each individual conflict and its resolution in full detail. Each subsection is dedicated to a specific conflict, providing a description, a formal definition using the operations from table 3.6 on page 44 and the resolution approach. Our implementation actually finds conflicts using the update trees introduced in section 4.2.4 on page 50. Detailed pseudo code for finding conflicts in these trees is presented in appendix A.6 on page 211. To limit the scope of this work the following subsections only present details of the conflicts and their resolution for the *NH-MD* file system introduced in section 2.3.1 on page 15. For the file systems *H-All* and *NED-All* we provide conflict descriptions and definitions in appendix A.2 on page 197 for the interested reader, but omit details for their *resolution* because our implementation focuses on the NH-MD file system.

Because it is possible that *several* conflicts are found, we find that a special property that must hold for resolving any conflict is that the resolution must work independently of the existence of other conflicts, and have no negative effect on other conflicts. We will provide further details in section 5.7, but already mention here that our high-level conflict resolution approach is *iterative*. We find all conflicts, sort them, resolve the conflict with highest priority and the restart the synchronization.

5.5.1 Create-Create

Description

On both replicas a new file or directory is created with the same name under the same parent dir. Is a pseudo conflict if both operations are *createfile* operations where the content/checksum of the files is equal, or if both operations are *createdir* operations and this synchronization iteration is the very first one.

The associated pattern is the *name clash* conflict.

Definition

Let $firstsync$ be a property that holds if the following two conditions hold: (1) synchronization has not yet fully completed since the user configured the synchronization, (2) all nodes in the update trees either have no change-event or only the *create* change-event.

Let

$$\begin{aligned} o_X &= create_X(i_X, p_X, n_X) \\ o_Y &= create_Y(i_Y, p_Y, p_Y) \end{aligned}$$

with $create := createdir \lor createfile$.

Then

$$\begin{aligned} pseudo(o_X, o_Y) =& [firstsync \land type(o_X) = createdir \land type(o_Y) = createdir \\ & \land corresponding\_object\_id(i_X, X) = i_Y] \lor [type(o_X) = createfile \\ & \land type(o_Y) = createfile \land corresponding\_object\_id(i_X, X) = i_Y \\ & \land checksum(snapshot_X, i_X) \\ & = checksum(snapshot_Y, corresponding\_object\_id(i_X, X)] \end{aligned}$$

is a function that indicates whether two *createfile* operations are pseudo-conflicting. Function *corresponding_object_id()* is defined in appendix A.5.1.

We can now define the Create-Create conflict, using the ⊗ symbol to indicate that the two operations are conflicting:

$$\begin{aligned} create_X(i_X, p_X, n_X) \otimes create_Y(i_Y, p_Y, n_Y) =& \neg pseudo(o_X, o_Y) \\ & \land corresponding\_object\_id(i_X, X) = i_Y \\ & \land \neg isedit(o_X) \land \neg isedit(o_Y) \end{aligned}$$

where

$$isedit(o_X) = type(o_X) = createfile \land \exists deletefile_X(j, v) : v = p_X \land name(dbsnapshot_X, j) = n_X$$

ensures that the *createfile* operation o_X is not replaced by an *edit* operation, as explained in section 4.2.4.2.

The violated precondition is $id(p, n) = error$ of the *createfile* or *createdir* operation. A file or dir cannot be created at a location that is already occupied.

Resolution

Following the recommendations from section 5.4.3, we rename the object of the loser operation using the following form, which is also used for other conflict types:

```
<name without file extension>-conflict-<datetime (Y-M-D-H-M-S)>-<random string>[.<extension>]
```

The random string forces the new name to be unique, making sure that the rename operation will not fail in practice (e.g. because the new name might be occupied by another object). After the rename operation finished and the synchronization is restarted, the two affected objects no longer have the same name and the conflict is resolved.

We note that our implementation offers flexibility by providing two options for the *replica* criterion, usually chosen at compile-time of the program. We offer options for Create-Create as well as most other conflict types. We implemented the following options for Create-Create conflicts:

- **Option 1**: Local replica always wins.
- **Option 2**: Remote replica always wins.

5.5.2 Edit-Edit

Description

The content of an already synchronized file was changed on both replicas. Is a pseudo conflict if the new content of both files is equal.

The associated pattern is the *Edit* conflict.

Definition

$$\begin{aligned} edit_X(i_X, op_X) \otimes edit_Y(i_Y, op_Y) = & [i_X = cid(i_Y, Y)] \\ & \wedge [checksum(snapshot_X, i_X) \neq checksum(snapshot_Y, i_Y)] \end{aligned}$$

Function *cid()* is defined in appendix A.5.2.

Note that a $deletefile$ and $createfile$ can count as $edit$ due to the consolidation of these two operations described in section 4.2.4.2.

Syntactically, no precondition is violated, but overwriting the file content on replica X with the one from replica Y would cause X's changes to be lost. This *semantic* violation instead takes place on the *update tree* levels, which are aware of the before-state (particularly its *lastmodified* meta-datum) of the file and therefore know that both replicas independently made incompatible modifications.

Resolution

We resolve this conflict as described in the Edit conflict pattern, see section 5.4.4. We first back up the file on the loser replica and then *replace* it with the content of the winner replica, updating the database snapshot with updated replica-specific IDs, lastmodified meta-datum and checksums. Some replicas may store older file versions automatically, in this case creating a backup is not necessary.

Like for Create-Create conflicts, we implemented the following options for the *replica* criterion:

- **Option 1**: Local replica always wins.
- **Option 2**: Remote replica always wins.

5.5.3 Move-Create

Description

On one replica the user moved an object into a specific parent dir v, giving it the name n, on the other replica the user created a new object named n in the parent directory corresponding to v.

The associated pattern is the *name clash* conflict.

Definition

$$create_X(i_X, p_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = [p_X = cid(v_Y, Y)] \wedge [n_X = n_Y] \\ \wedge \neg isedit(create_X)$$

with $create := createdir \vee createfile$

Violated preconditions are $id(p.n) = error$ of the *createfile*/ *createdir* operation, and $id(v, n) = error$ of the *move* operation. A file or dir cannot be created at (or moved to) a location that is already occupied.

Resolution

Conflict resolution happens the same way as for Create-Create conflicts. Our implementation offers four options, two for the *replica* criterion, two for the *operation type* criterion, using option 2 by default:

- **Option 1**: Local replica always wins.
- **Option 2**: Remote replica always wins.
- **Option 3**: Move operation always wins.
- **Option 4**: Create operation always wins.

5.5.4 Edit-Delete

Description

On one replica the content of an already synchronized file was changed, on the other replica that file was deleted.

The associated pattern is the *delete* conflict.

Definition

$$edit_X(i_X, op_X) \otimes deletefile_Y(i_Y, p_Y) = (i_X = cid(i_Y, Y))$$

Note that a $deletefile$ and $createfile$ can count as $edit$ due to the consolidation of these two operations described in section 4.2.4.2.

On replica Y, $ancestor(i_{root}, i_Y)$ is violated when trying to apply the *edit* operation. On replica X there is no violation on a syntactic level, but on the semantic level: the changes of the *edit* operation would be lost. The user who deleted the file would have done so without knowing that it was recently edited by another user on the other replica.

Resolution

Our implementation offers four options, using the *replica* and *operation type* criterion, to determine whether the *edit* or *deletefile* operation takes precedence, choosing option 1 as default:

- **Option 1**: *Edit* operation always wins.
- **Option 2**: *deletefile* operation always wins.
- **Option 3**: Operation of the local replica always wins.
- **Option 4**: Operation of the remote replica always wins.

If the configured option favors the *edit* operation, we perform the following checks:

- If the delete node's parent has no *Delete* change-event, a delete operation is performed for the edit node which only deletes the corresponding local and remote replica's entries from the *database* (not the file system). This will cause the file to be detected as new in the next sync iteration, thus it will be restored.
- If the delete node's parent has a *Delete* change-event, the delete operation mentioned above would only turn the Edit-Delete conflict into a Create-ParentDelete conflict in the next sync iteration. To avoid that the synchronizer presents the user two conflicts, we first execute a move operation that moves the file to the root physically, with a conflict-suffix, followed by executing the delete operation that removes the file's entry from the database.

If instead the *delete* operation takes precedence, we first perform a backup operation of that file. Next, we perform the delete operation that removes the file physically from the losing replica and from the database.

5.5.5 Move-Delete

Description

On one replica an object was moved, on the other replica the same object was deleted.

The associated pattern is the *delete* conflict.

Definition

$$move_X(i_X, u_X, v_X, n_X) \otimes delete_Y(i_Y, p_Y) = (i_X = cid(i_Y, Y)) \land \neg isedit(delete_Y)$$

with $delete := deletefile \lor deletedir$

and

$$\begin{aligned} isedit(delete_Y) = & [type(delete_Y) = deletefile] \\ & \land \exists createfile_Y(j_Y, v_Y, n_Y) : v_Y = p_Y \land n_Y = name(dbsnapshot_Y, i_Y) \end{aligned}$$

On replica Y, $i \in list(u)$ is violated when trying to apply the *move* operation. You cannot move a file or dir that is already deleted. On replica X there is no violation on the syntactic level, but on the semantic level: the structural change of the *move* operation would be lost. The user who deleted the object would have done so without knowing that it was recently moved by another user on the other replica.

Resolution

Like for Edit-Delete conflicts, the conflict resolution first needs to determine whether the *move* or *delete* operation takes precedence. Because both *Edit-Delete* and *Move-Delete* conflicts belong to the *Delete* conflict pattern, it makes sense that these two conflict types are resolved similarly. Thus, the options described for Edit-Delete also apply here for Move-Delete.

When the move or delete operation applies to *directories*, conflict resolution becomes more involved, because dirs can have children on which the user could have performed additional operations, on either (or both) replicas. Specifically, new files or dirs could have been created, existing ones could have been deleted or edited, and different kinds of move operations could have happened. Consider dir node *A*, then these three types of moves on its child nodes are possible:

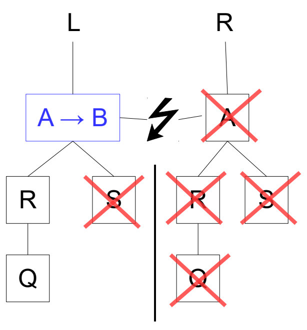
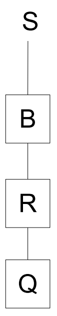
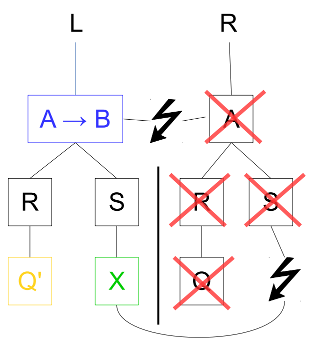
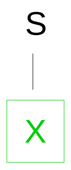
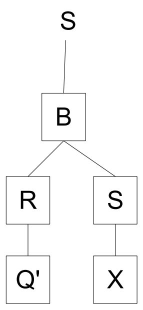
- Moves within the hierarchy of *A* (in other words: *A*'s path is a prefix of both the *source* and *dest* path of such moves) - can only happen on the Move-replica of a Move-Delete conflict.
- Some node *N* outside of *A* is moved so that *N* is now a child of *A* (referred to as *Move-ParentDelete* conflict, see next section) - can only happen on the Move-replica of a Move-Delete conflict.

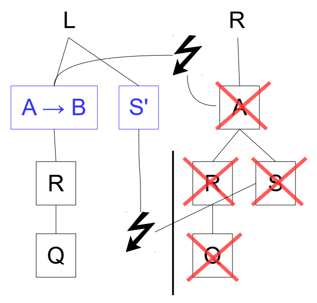
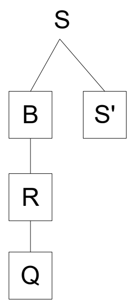
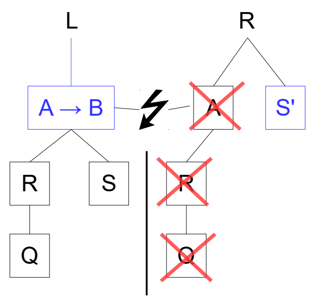
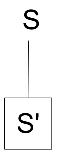
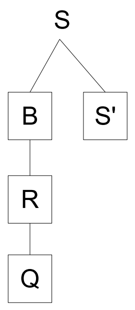
- Some child node of *A* is moved to a destination that is outside of *A* - can happen on both the Move- or Delete-replica of a Move-Delete conflict (on the Delete-replica, the user must have moved the child node out of *A* prior to deleting *A*).

For each of these cases we have to make sure that the synchronizer's behavior is consistent w.r.t. the selected resolution option and that it doesn't produce incorrect file system or database operations.

The outline of the behavior related to child nodes is shown in the following table. It shows desired outcomes for a Move-Delete situation. All situations have in common that on the left replica directory "/A" was moved to "/B", while "/A" was deleted on the right replica. However, in each situation (table row) one or both replicas performed additional operations. The *Delete wins* and *Move wins* columns illustrate the desired outcome for the corresponding chosen Move-Delete resolution option. The color legend is as follows:

- Green: creation of a new object
- Yellow: a file's content was edited
- Blue: node was moved

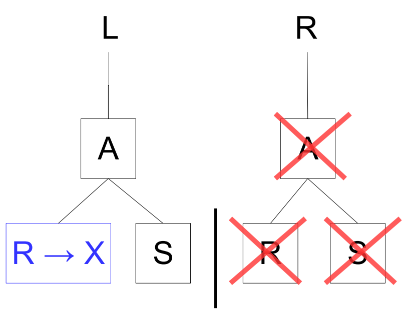
Situation	Delete wins	Move wins
 Move-replica also deleted some of the child nodes	All files are deleted (deletions on move-replica have no effect)	 Child nodes also deleted by the move-replica were deleted from both replicas and thus cannot be restored
 Move-replica creates new child nodes or edits existing ones, inside the moved directory	 After undoing the move locally ("/B" to "/A"), an Edit-Delete conflict for "'/A/R/Q'" and a Create-ParentDelete for "/A/S/X" is detected (see subsection 5.5.7) and resolved.	 "/B" (and *all* of its sub-elements) are detected as new and are synchronized to the delete-replica

Situation	Delete wins	Move wins
 Move-replica moves the child node "/B/S" out of "/B", to "/S'"	Both "/B" and "/S'" are deleted on the move-replica, because a Move-Delete conflict is detected for both of them and the moves are undone	 Both "/B" and "/S'" are detected as *new* and are synchronized to the delete-replica, at the new respective locations
 The delete-replica moves the child node "/A/S" out of "/A" to "/S'" prior to deleting "/A"	 "/B" is deleted on the move-replica, but "/B/S" is first moved to "/S'" on the move-replica	 The move "/A/S" to "/S'" should be executed on the local replica. "/B" (with all its children, except "/B/S") is detected as new on the move-replica and therefore synchronized to the delete-replica

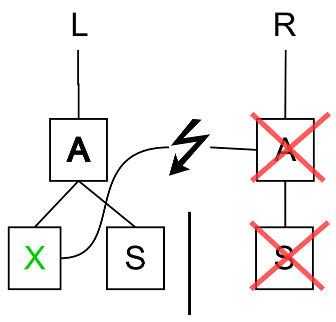
Another specialty in our desired Move-Delete resolution is explained in figure 5.4.

The general behavior of our Move-Delete *resolution* (which realizes the desired behavior described above) is similar to the one defined for Edit-Delete conflicts:

- When the *move* operation takes precedence, that node and all its child nodes are deleted from the database, so that the files are detected as *new* on the subsequent sync iteration and are therefore synchronized to the replica that deleted the node, so that they restored at the *new* location (i.e. the one corresponding to the Move-replica). However, for directories, the user who deleted the dir could have moved one or more child objects outside of the deleted directory. Such objects potentially still exist on *both* replicas and should therefore not be deleted from the database. More details are provided in appendix A.7.2.
- When the *delete* operation takes precedence, we should *not* delete the other replica's object in the file system or database (as done in the Edit-Delete resolution), because this would have undesired



(a) Initial situation



(b) Introduced Create-ParentDelete conflict

Figure 5.4: Avoiding Move-Delete detection for within-dir moves

Subfigure (a) shows the conflict situation. While a dir is deleted on the remote replica, the user of the local replica moves/renames child nodes *within* that dir. According to our conflict definitions, this causes the detection of both a *Move-ParentDelete* conflict (explained in the next section) and a *Move-Delete* conflict for that node. There are two options to resolve this. First, we could treat the situation as *Move-Delete* conflict. When that conflict's resolution option is set to "move wins", this would cause the "/A/R" entry to be removed from the database, causing a Create-ParentDelete conflict as shown in subfigure (b), which is resolved by moving the object to the root (see section 5.5.7 for more details). In other words, treating the situation as *Move-Delete* conflict would save the user's within-dir move operation. The second option is to treat the situation as *Move-ParentDelete* conflict, which is resolved by undoing the move. Effectively this means that within-dir move operations are lost. We prefer and implemented the latter approach, for two reasons: (1) Semantically we see no advantage in saving namespace reorganizations *within* a dir that is deleted by another replica. Moving such objects to the root (as done by the *Create-ParentDelete* resolution) only fragments the synchronized namespace. (2) When interpreting the situation as *Move-Delete* conflict, the user would be confused by *two* conflict messages shown consecutively, one for the *Move-Delete*, one for the *Create-ParentDelete* conflict.

effects on its child nodes in case the conflict affects a *directory*. Instead, we *undo* the move, so that a subsequent sync iteration no longer recognizes the move, but just the delete operation, as a conflict-free situation. The delete operation will eventually be executed, but special situations (such as other conflicts caused by child nodes) can be detected and handled first. Note that undoing the move is an involved operation and is further elaborated on in appendix A.7.1.

5.5.6 Move-ParentDelete

Description

On one replica a directory node A was deleted, on the other replica another object was moved such that it is now an immediate child of A.

The associated pattern is the *indirect* conflict.

Definition

$$move_X(i_X, u_X, v_X, n_X) \otimes deletedir_Y(i_Y, p_Y) = name(snapshot_Y, cid(i_X, X)) \neq error \wedge (cid(v_X, X) = i_Y)$$

The precondition $type(v) = dir$ of the *move* operation is violated. You cannot move a file or directory into a directory that is already deleted.

Resolution

Unlike the conflicts presented so far, we decided *not* to offer any configurable options for resolving this conflict. Instead we always resolve it by undoing the move operation, as described in appendix A.7.1. In the subsequent sync iteration, only the delete operation remains, which is then executed. This approach favors the user's intention to delete the directory over the move operation. Other alternatives have considerable disadvantages:

- Preferring the *deletion*, and *not* undoing the move operation, would destroy the object that was moved inside it, which was neither intended by the user who moved the object nor the one who deleted the directory.
- Preferring the *move* operation would require the synchronizer to recreate the directory (and all ancestor directories leading to it), producing an *inconsistent* preservation of the delete-intention, fragmenting the namespace. To the deleting user the directory would inexplicably re-appear. This merged result is undesirable, also for the user who moved the object. She would not have moved an object into the directory, had she known before that the other user already deleted it. Cleaning up the workspace may cause a considerable amount of time.

Our suggestion to favor the deletion but to first undo the move causes only little loss of data. Once the user is notified about the undone move operation, she can choose to restore the deleted directory, e.g. from a server backup, and repeat the move operation.

5.5.7 Create-ParentDelete

Description

On one replica a directory node *A* was deleted, on the other replica a new object was created as an immediate child of *A*.

The associated pattern is the *indirect* conflict.

Definition

$$create_X(i_X, p_X, n_X) \otimes deletedir_Y(i_Y, p_Y) = (p_X = cid(i_Y, Y)) \wedge \neg isedit(create_X)$$

with $create := createdir \vee createfile$

The violated precondition is $type(p) = dir$ of the *createfile* or *createdir* operation. You cannot create a file in a directory that is already deleted.

Resolution

Like *Move-ParentDelete*, we offer no configurable options for resolving *Create-ParentDelete* conflicts. We resolve it by physically moving created objects to the root, appending the conflict suffix introduced for Create-Create conflicts. On the next sync iteration, the conflict is no longer detected and the directory is deleted. This approach favors the user's intention to delete the directory and we favor this solution over recreating the deleted parent directory(ies), for the same reasons as given for Move-ParentDelete conflicts.

5.5.8 Move-Move (Source)

Description

On both replicas the *same* file or directory was moved to a different location. We denote this Move-Move conflict "Move-Move (*Source*)" because it affects the *same source* object. Is a pseudo conflict if both the destination parent dir and new name of the file or dir are equal.

The associated pattern is the *move* conflict.

Definition

$$move_X(i_X, u_X, v_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = (i_X = cid(i_Y, Y)) \wedge [(v_X \neq cid(v_Y, Y)) \vee (n_X \neq n_Y)]$$

The violated preconditions is $i \in list(u)$. The object is no longer in the expected source location of the *move* operation.

Resolution

Following the recommendation of the Move conflict pattern (section 5.4.6) we resolve this conflict by undoing the move of the loser replica. Like for Create-Create conflicts, we implemented the following options:

- **Option 1**: Local replica always wins.
- **Option 2**: Remote replica always wins.

We decided to offer only a single resolution option that is equal for Move-Move (Source), Move-Move (Dest) and Move-Move (Cycle) conflicts, because the *move* operation is the central operation in all of these conflicts. Hence, resolving these conflicts the same way makes conflict resolution more comprehensible to the user.

As discussed in subsection A.7.2, the *Move-Delete* conflict resolution code may manipulate the locations of *orphaned* nodes in the database. Because this causes *Move-Move (Source)* conflicts, the resolution code adds the i_{db} of the orphaned nodes together with the `winner_replica` to an in-memory registry. Consequently, to determine the winning replica, the Move-Move (Source) resolution code first checks whether the provided node's i_{db} is known in the registry:

- If not, `loser_node` is chosen according to the preconfigured resolution option for Move-Move (Source) conflicts.
- If yes, `loser_node = conflict.local_node if winner_replica == Remote else conflict.remote_node`

Then, `undo_move(loser_node)` is called to resolve the conflict.

5.5.9 Move-Move (Dest)

Description

The users of both replicas each move a *different* object into the same parent directory with the same name. The name of this conflict is Move-Move (*Dest*) because both move operations affect the *same destination.*

The associated pattern is the *name clash* conflict.

Definition

$$move_X(i_X, u_X, v_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = [i_X \neq cid(i_Y, Y)] \wedge [v_X = cid(v_Y, Y)] \wedge (n_X = n_Y)$$

The violated precondition is $id(v, n) = error$. A file or dir cannot be moved to a location that is already occupied.

Resolution

Following the recommendations of the *name clash* conflict pattern (section 5.4.3), we *rename* the file on the losing replica by appending the *conflict suffix*. There are two options for the *replica* criterion:

- **Option 1**: Local replica always wins.
- **Option 2**: Remote replica always wins.

5.5.10 Move-Move (Cycle)

Description

Given two distinct directories A and B, the user moves A into B's namespace on one replica while B is moved into A's namespace on the other replica. This would create a cyclic parent-child relationship in the merged result, which is not allowed by $\mathscr{F}$'s invariants.

Definition

$$\begin{aligned} move_X(i_X, u_X, v_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = & (type(snapshot_{X,}, i_X) = type(snapshot_Y, i_Y) = dir) \\ & \wedge i_X \neq cid(i_Y, Y) \\ & \wedge ancestor(snapshot_X, cid(i_Y, Y), i_X) \\ & \wedge ancestor(snapshot_Y, cid(i_X, X), i_Y) \end{aligned}$$

The violated precondition is $\neg ancestor(i, v)$. A file or dir cannot be moved to a location that is below itself.

Resolution

Since the attempt of applying the move operation of one replica to the other one will fail, the synchronizer needs to undo one of the move operations.

5.5.11 Conflict type completeness

The ten conflicts presented above cover *all* possible conflict types. To show this, consider *OT={createfile, createdir, move, edit, deletefile, deletedir}* which is the list of all operation types of file system $\mathscr{F}$, see table 3.6 on page 44. We start from an initially equal state for replicas X and Y. For any two types t_A, t_B from *OT* we instantiate operations o_X (of type t_A) and o_Y (of type t_B), apply o_X to X (which yields X') and o_Y to Y (yields Y'). We choose the operation parameters (i, p, u, v, n for $\mathscr{F}$) such that either applying o_Y to X', or o_X to Y' fails, due to violated preconditions.

Finding conflicts can be done manually or in an automated approach. We applied the manual, pragmatic approach, examining each individual precondition of each operation type t_A and finding a t_B, o_Y and o_X that produces a conflict. The following list illustrates the relationships between the conflicts defined above and the preconditions:

- createdir, createfile:

- $ancestor(i_{root}, p) \land type(p) = dir$: see Create-ParentDelete conflict
- $id(p, n) = error$: see Create-Create and Move-Create conflict
- Note: precondition $\neg ancestor(i_{root}, i)$ is never violated, because the file system implementation makes sure to choose a unique i

- move:
  - $type(u) = dir \land i \in list(u)$: see Move-Delete and Move-Move (Source) conflict
  - $type(v) = dir$: see Move-ParentDelete conflict
  - $id(v, n) = error$: see Move-Create, Move-Move (Dest) conflict
  - $\neg ancestor(i, v)$: see Move-Move (Cycle) conflict
- deletefile, deletedir:
  - $ancestor(i_{root}, i) \land i \in list(p)$: see Move-Delete, or pseudo Delete-Delete conflict
- edit:
  - $ancestor(i_{root}, i)$: see Edit-Delete conflict.

The above list covers all conflict types except for Edit-Edit. However, this conflict type is not a *syntactic* but a *semantic* conflict, thus no formally defined precondition of the *edit* operation is violated.

5.6 Applicability of conflicts

Table 5.4 illustrates which conflicts apply for each file system model from table 2.1 on page 16. The last row makes apparent that the number of conflicts increases with the complexity of the underlying file system definition. The more conflict types, the more elaborate the implementation of the file synchronizer needs to be, because it requires detection and resolution logic for each conflict. However, this insight should *not* tempt a synchronizer developer to deliberately choose a simplified, internal file system model (such as *NH-D [BP98]* or *NH-M* [Mol+03]), when the underlying file system's definitions is actually more complex[7]. Doing so degrades performance and usability, as explained in section 3.2 on page 41. While a heterogeneous synchronizer, such as the one we develop in this work, also chooses a simpler model (*NH-MD* vs. *H-All*), it does so for compatibility reasons rather than ease of implementation.

5.7 Iterative conflict resolution

5.7.1 Introduction

When resolving conflicts there are two challenges:

1. It is possible that a set of specific files is affected by *multiple* conflicts at the same time. Figure 5.5 illustrates a few examples. However, implementing resolution methods that consider multiple conflicts is futile, as the number of conflict combinations is (countably) infinite.
2. Resolving a conflict may have bad side effects on other (still unresolved) conflicts or may require special logic in case executing the winner operation on the loser replica first requires propagation of other operations.

All related academic works we presented in chapter 2 ignore these challenges. We approach them as follows:

[7]The available implementations of the two cited works operate on top of file systems such as NTFS (Windows) or APFS (macOS), which are similar to the more powerful *H-All* definition.

	NH-MD	NH-M	NH-D	NH-RD	H-All	NED-All
Create-Create	✓	✓	✓	✓	✓	✓
Link-Link	X	X	X	X	✓	X
Link-Unlink	X	X	X	X	✓	X
Create-Link	X	X	X	X	✓	X
Move-Link	X	X	X	X	✓	X
Edit-Edit	✓	✓	✓	✓	✓	✓
Move-Create	✓	✓	X[A)]	✓[B)]	✓[C)]	✓
Edit-Delete	✓	✓	✓	✓	✓	✓
Move-Delete	✓	X[D)]	X	✓[B)]	✓[C)]	✓
Move-Move (Source)	✓	✓	X	✓[B)]	✓[C)]	✓
Move-Move (Dest)	✓	✓	X	✓[B)]	✓[E)]	✓
Move-ParentDelete	✓	X	X[F)]	X	✓	X
Content-ParentDelete	✓	X	✓	✓	✓	X
Move-Move (Cycle)	✓	✓	X	X	✓	X
Node-typing	X	X	X	X	X	✓
No. of conflicts	10	7	4	8	14	8

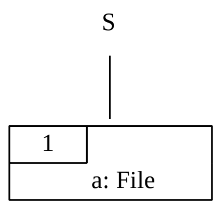
Table 5.4: Conflict applicability per file system

Footnotes:

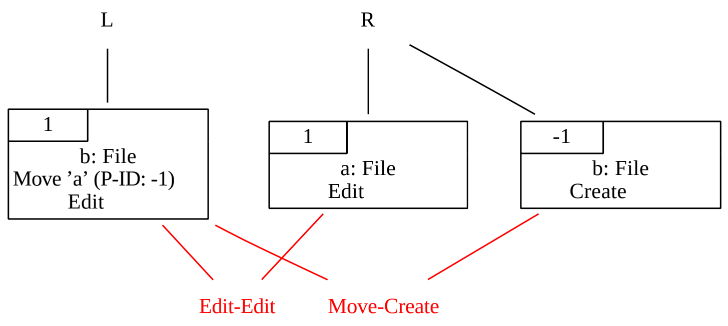
- A) Becomes *Create-Create* conflict.
- B) ✓ if detected *move* is actually a *rename* operation. X otherwise.
- C) ✓ only for directories. X otherwise.
- D) Becomes *Move-Move (Source)* conflict.
- E) Applies to directories *and* files.
- F) Becomes *Create-ParentDelete* conflict.

File system model acronyms, as introduced in table 2.1 on page 16:

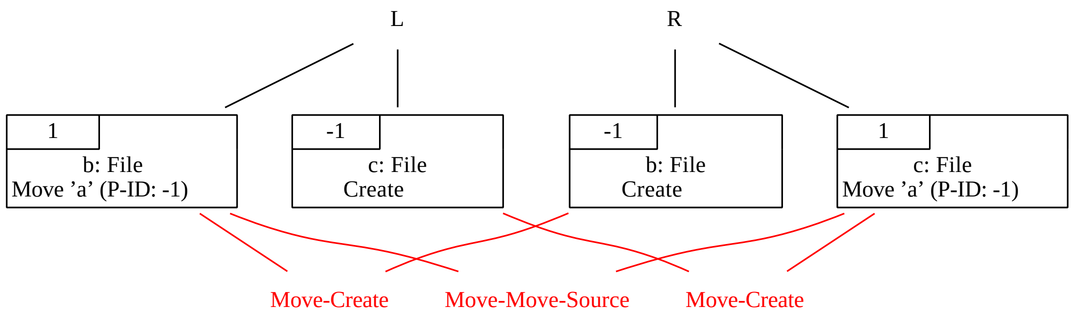
- NH-MD: **N**o **h**ardlink support, support for **m**ove and **d**elete operations
- NH-M: **N**o **h**ardlink support, support for **m**ove but not the delete operation
- NH-D: **N**o **h**ardlink support, support for **d**elete but not move operation
- NH-RD: **N**o **h**ardlink support, support for **r**ename and **d**elete operations
- H-All: **H**ardlink support, support for *all* operations
- NED-All: **N**o (**e**mpty) **d**irectory support, support for *all* operations



(a) Start situation for (b) and (c)



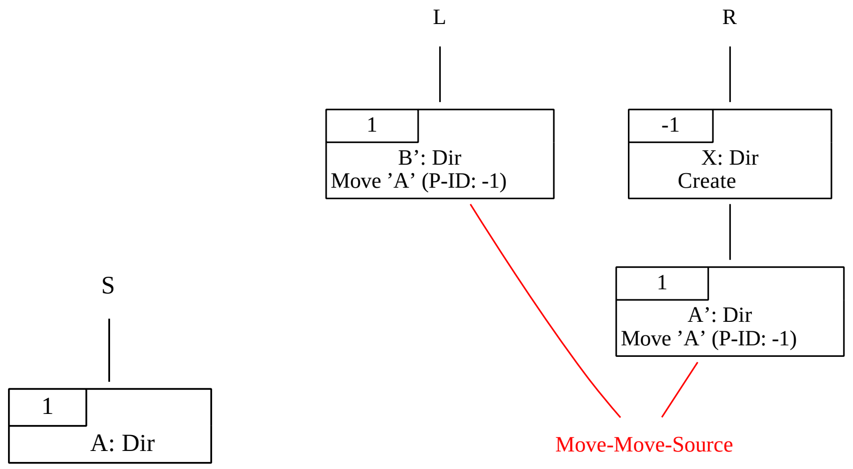
(b) Edit-Edit + Move-Create



(c) Move-Move (Source) + Move-Create + Move-Create

Figure 5.5: Conflict combination examples

Given the start situation shown in subfigure (a), (b) illustrates how a specific file can be affected by both an Edit-Edit and a Move-Create conflict, because on replica L the file was moved from 'a' to 'b', on R another 'b' was created, and the content of the file was edited on both replicas. In (c) we see how a Move-Move (Source) conflict can be combined with two Move-Create conflicts, by creating a new file on each replica, at the same location that is occupied by the respective other replica's move operation.



(a) Initial situation (b) Conflict situation

Figure 5.6: Move-Move (Source) conflict resolution with new directory

1. We find all conflicts, sort them in a specific order (explained in section 5.7.2) and only solve *one* conflict at a time, the one with the highest priority, followed by restarting the symchronization.
2. When resolving a conflict of type *A*-*B*, where operation *A* should win, rather than executing operations that merge the effects of *A* and *B* to yield a file system that looks as if operation *A* had won, we instead only *undo* (or modify) operation *B* and then restart the synchronization. The consecutive synchronization iteration recognizes operation *A* again, which no longer conflicts with *B*, and eventually executes *A* once all conflicts have been resolved. The concrete resolution approaches presented in section 5.5 use this approach.

This iterative approach keeps our implementation logic simple and thus less prone to errors. The following examples illustrate the advantages of our approach:

- For Move-Delete conflicts affecting a directory where the option is set to "delete wins", undoing the move is better than immediately executing the delete operation, because the latter would ignore possibly existing Move-ParentDelete and Create-ParentDelete conflicts on child nodes of the affected dir. See subfigures (a)+(b) of figure 5.11 on page 100.
- For Move-ParentDelete conflicts, just undoing the moves (rather than executing the Delete operation immediately) allows to correctly handle potential Create-ParentDelete conflicts for other child nodes of the deleted directory.
- For Move-Move (Source) or Move-Move (Cycle) conflicts, undoing the move of the losing replica is better than trying to immediately execute the winning replica's move operation, because the winning replica may have moved the object to a new directory which doesn't exist on the losing replica yet. Extra logic would be required to catch such situations and compute the synchronization of such directories. An example can be seen in figure 5.6.

5.7.2 Conflict type sort order

Our implementation sorts all found conflicts in the following order:

1. Move-ParentDelete

2. Move-Delete
3. Create-ParentDelete
4. Move-Move (Source)
5. Move-Move (Dest)
6. Move-Create
7. Edit-Delete
8. Create-Create
9. Edit-Edit
10. Move-Move (Cycle)

If, for any particular conflict type, there are more than one conflicts, these are solved by path as follows:

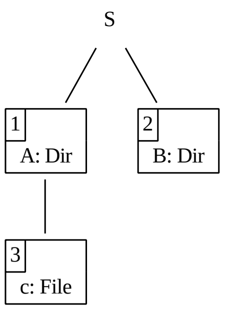
- Move-ParentDelete, Create-ParentDelete: path of deleted directory node
- Move-Delete, Edit-Delete: path of deleted node
- Move-Move (Source): move origin path of the local node
- Move-Move (Dest), Move-Move (Cycle): path of local move node
- Move-Create: path of the create node
- Create-Create, Edit-Edit: path of the local create/edit node

Path sorting first compares the tree *depth level* (e.g. 'x/y' < 'a/b/c' because the former is on level 2, the latter on level 3) . If two paths have equal depth, we use lexicographical order.

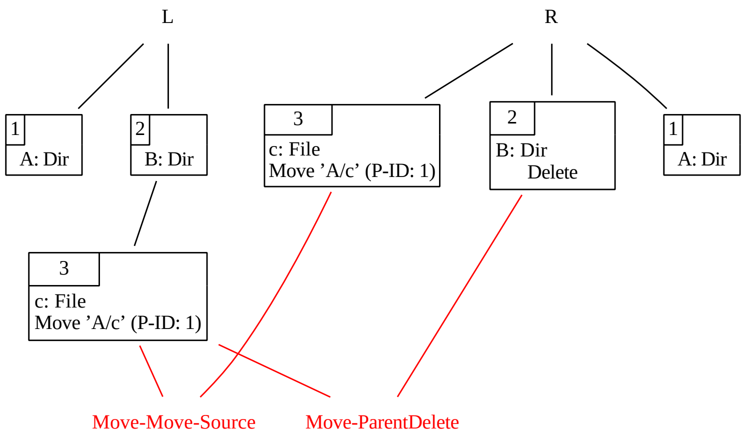
The following analysis explains the rationale for using this particular sort order. While the presented conflict resolution approaches would work in *any* order, a closer look reveals that resolving one conflict can have a large effect on other existing conflicts, such as eliminating them or turning one conflict type into another one. We examined a large set of conflict combination *pairs* and tested different orders (including all variations for the resolution options, where available). The conflict type sort order introduced above is the result of building a *totally* ordered list from the *partial* priorities we determined between two conflict types. We write:

- *type1* > *type2* if conflicts of type1 have a higher priority and should be resolved *before* type2, for reasons as detailed below,
- *type1* | *type2* if the order is irrelevant (as in: "it doesn't matter which conflict type is solved first"), usually due to an independence between the affected nodes, parent nodes or names.

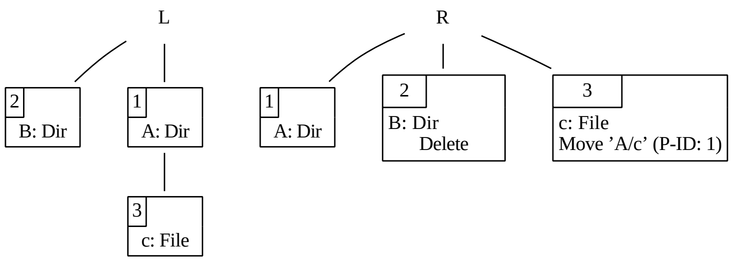
The general metric for deciding whether *type1* > *type2* holds is that the *intention* of both replica's operations and the respective conflict resolution option should be *maximized*. Figure 5.7 illustrates the maximization. The following table provides an overview. Each conflict combination is briefly explained in subsections 5.7.2.1 and 5.7.2.2. From the list of partial priorities it is trivial to construct the totally ordered priority list presented above.



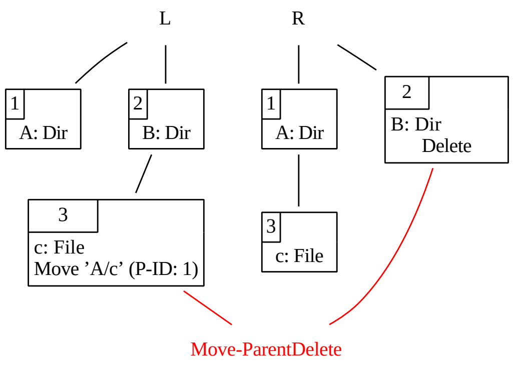
(a) Start situation



(b) Conflict situation



(c) Solving Move-ParentDelete first



(d) Solving Move-Move (Source) first

Figure 5.7: Move-ParentDelete > Move-Move (Source)
The start situation shown in (a) is changed into a Move-Move (Source) + Move-ParentDelete situation in (b). (c) illustrates that when first solving the Move-ParentDelete conflict, the Move-Move (Source) conflict disappears. The intention of the operation of the local replica, moving 'A/c' to 'B/c', was denied because of the Move-ParentDelete resolution, but the intention of the remote replica, moving 'A/c' to 'c' prevails. In (d), we show what happens when instead solving the Move-Move (Source) conflict first, with the configuration option that the *local* replica wins such types of conflicts. In this case, the Move-ParentDelete conflict remains, and once it is solved, *all* moves shown in (b) will have been undone. We end up in a situation like the start situation, with the only difference that 'B' was deleted. None of the move operations prevail. Consequently, we favor the approach of (c) over (d).

Type1 | Type2	Type1 > Type2
Move-Move (Source) | Move-Move (Dest)	Move-Delete > Edit-Delete
Move-ParentDelete | Edit-Edit	Move-Delete > Move-Create
Move-Move (Source) | Edit-Edit	Move-ParentDelete > Move-Delete
Move-Create | Edit-Edit	Move-Delete > Move-Move (Dest)
Move-Move (Cycle) | Move-Move (Dest)	Move-Move (Source) > Move-Create
Move-ParentDelete | Create-ParentDelete	Move-Move (Source) > Create-Create
Move-Move (Cycle) | Move-Create	Move-ParentDelete > Move-Move (Source)
	Move-Delete > Create-ParentDelete
	Move-Delete > Create-Create

5.7.2.1 Same priority conflict combinations

- Move-Move (Source) | Move-Move (Dest): solving the conflicts in either order results in no difference for the final outcome.
- Move-ParentDelete | Edit-Edit, Move-Move (Source) | Edit-Edit, Move-Create | Edit-Edit: for the correctness of synchronization, first moving a file and then replacing it is equivalent to first replacing and then moving it.
- Move-Move (Cycle) | Move-Move (Dest): solving the conflicts in either order doesn't result in any difference. Even though undoing the move as part of solving a Move-Move (Cycle) conflict dissolves the Move-Move (Dest) conflict, the end result is equal, because of the same resolution option configured for all Move-Move (Source/Dest/Cycle) conflicts.
- Move-ParentDelete | Create-ParentDelete: solving the conflicts in either order results in no difference for the final outcome.
- Move-Move (Cycle) | Move-Create: solving the conflicts in either order result in negligible difference.

5.7.2.2 Different priority conflict combinations

- Move-Delete > Edit-Delete: while the resolution order doesn't matter in case both conflicts affect the same *file*, an example scenario exists where the Move-Delete conflict affects a parent directory, and the Edit-Delete conflict affects a file inside that parent directory (i.e. the user who edited the file also moved a parent dir). In this case resolving the Move-Delete conflict first allows the file to stay in the directory. The opposite order would first move it to the root, which is unnecessary.
- Move-Delete > Move-Create: An example is shown in figure 5.8. In case *delete* is preferred over *move* for Move-Delete conflicts, undoing the move also solves the Move-Create conflict. This produces fewer files with conflict-suffix. When choosing other configuration options, the conflict resolution order is irrelevant and produces the same outcome.
- Move-ParentDelete > Move-Delete: as discussed in figure 5.4 this choice was made such that *move* operations that take place within a directory that was deleted on the other replica will be discarded.
- Move-Delete > Move-Move (Dest): An example is shown in figure 5.9. When resolving Move-Move (Dest) first, the Move-Delete conflict always remains, irrespective of the configured option. The end result always includes one file with a conflict suffix. If resolving the Move-Delete conflict first and the configured option prefers *delete* to win, the *move* operation is undone, the Move-Move (Dest) conflict disappears, an no file has the conflict suffix. If *move* wins, the entry for one object is removed from the database and the Move-Move (Dest) becomes a Move-Create conflict., where one file with conflict suffix remains.

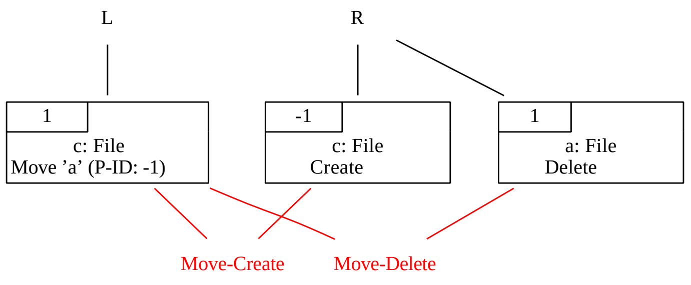
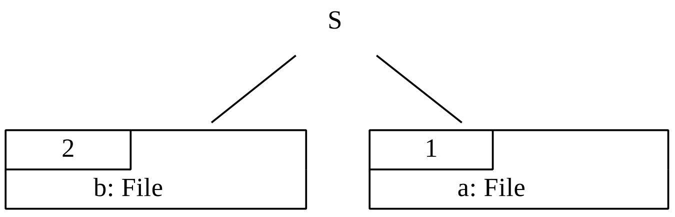
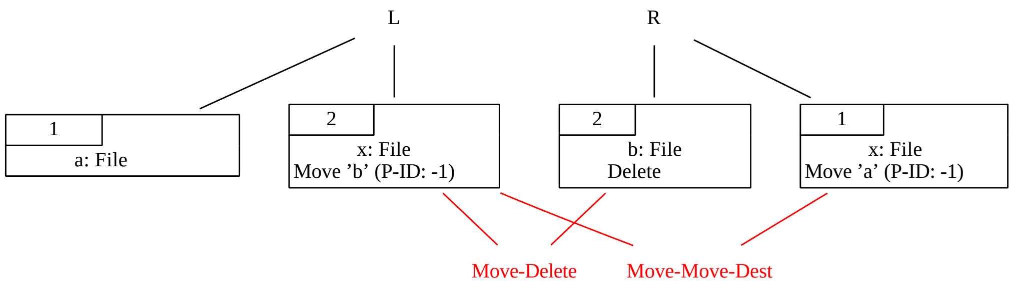


Figure 5.8: Move-Delete > Move-Create

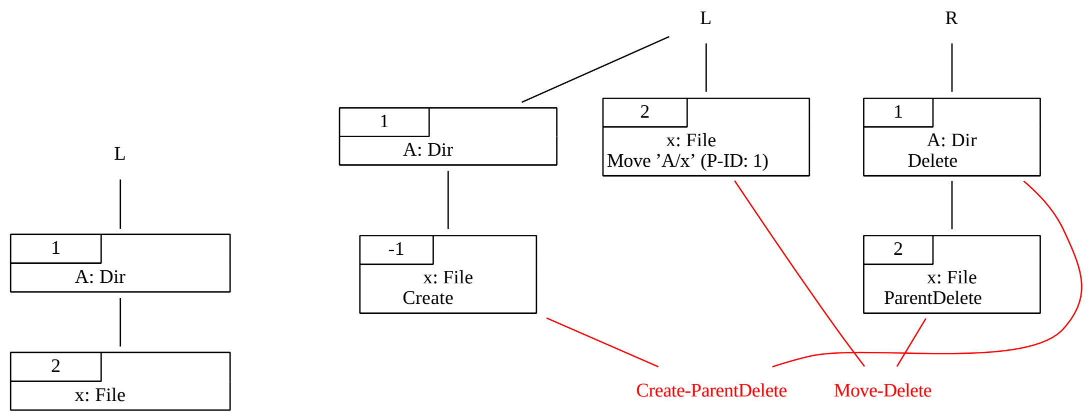


(a) Start situation



(b) Conflict situation

Figure 5.9: Move-Delete > Move-Move (Dest)



(a) Start situation (b) Conflict situation

Figure 5.10: Move-Delete | Create-ParentDelete

- Move-Move (Source) > Create-Create and Move-Move (Source) > Move-Create: Undoing the move (as part of the Move-Move (Source) resolution) solves the respective other conflict, while the opposite order, solving the other conflict first, doesn't. After the synchronization finished, this produces fewer files with a conflict suffix in their name, compared to solving the conflicts in the opposite order.

- Move-ParentDelete > Move-Move (Source): When solving Move-ParentDelete first, the other conflict disappears, since the resolution undoes the move operation of one replica. See figure 5.7.

- Move-Delete > Create-ParentDelete: These two conflict types can be intertangled in different ways. Figure 5.10 demonstrates an example where different resolution orders yield no difference. However, when the shared node of both conflicts is the deleted parent dir node, as shown in figure 5.11, resolving the Create-ParentDelete conflict first might produce a suboptimal result, because the created node would definitely be moved to the root directory, even if the configuration for the Move-Delete is "move wins", which would have produced a result where the created node stays in the user-specified location.

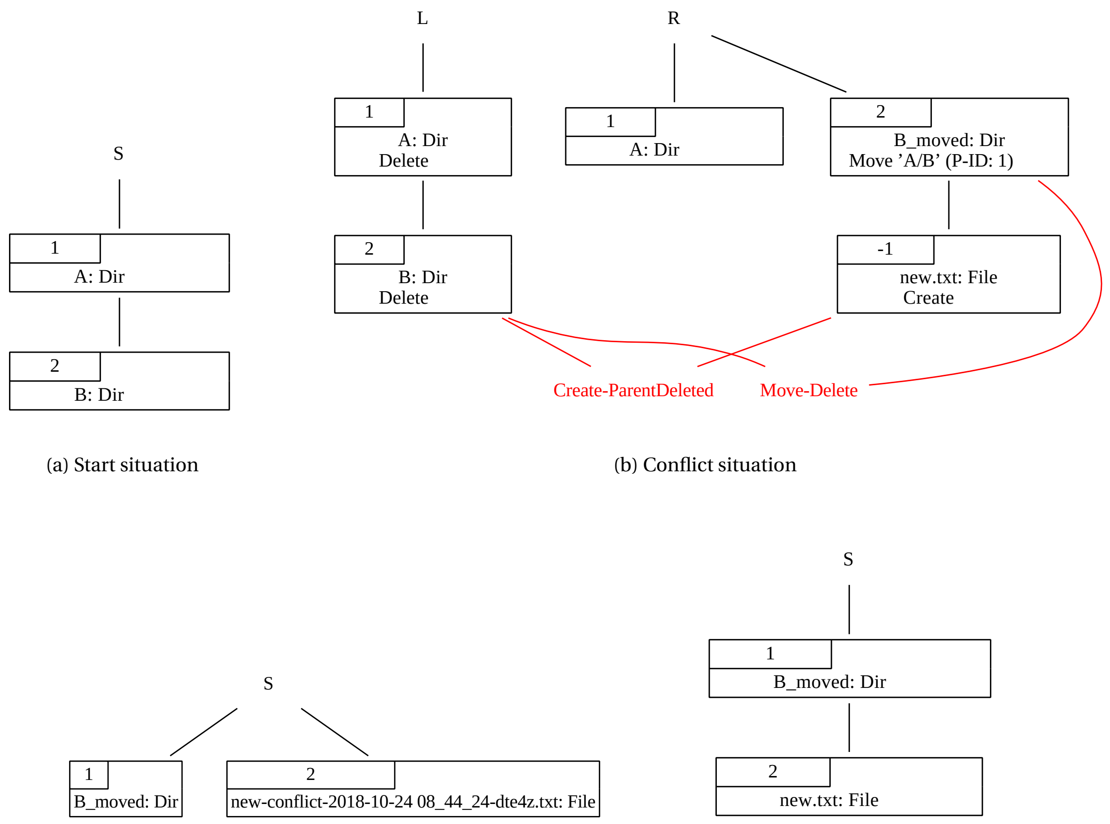
- Move-Delete > Create-Create: Regardless which resolution option is used for resolving the Move-Delete conflict, the final outcome will always contain just one file with a conflict-suffix. When resolving the Create-Create conflict first, two files may have the suffix.

5.8 Proof of termination

5.8.1 Overview

As explained in section 5.7 our conflict resolution algorithm iteratively solves one conflict at a time, until all conflicts are resolved. Here we demonstrate that our algorithm cannot get caught in an infinite loop, by closely examining the effects of resolving conflicts. We find that sometimes resolving a conflict introduces another conflict, or changes the conflict type of another existing one. Figure 5.12 provides an overview of these kinds of dependencies. An arrow that connects conflict c with another conflict c' indicates that resolving c introduces c'. If it were possible to find cycles[8], our algorithm would never terminate. However, below we show that each cyclic dependency (see red arrows in figure 5.12) does *not*

[8]For instance, a cycle would exist if resolving conflict c_1 introduces c_2, whose resolution introduces c_3, whose resolution introduces c_4, etc.



(a) Start situation (b) Conflict situation

(c) Result (resolved Create-ParentDelete first) (d) Result (resolved Move-Delete first)

Figure 5.11: Move-Delete > Create-ParentDelete

cause a cycle. That is, the resolution of a conflict c of type t may cause another conflict c' of type t, but resolving c' will not cause another c'' of type t.

We now present arguments for each conflict type in a separate section, where the following common terminologies hold:

- The conflict indicated by the section title is denoted c.
- For conflicts Move-Move (Source), Edit-Edit, Move-Delete and Edit-Delete c affects nodes n_X and n_Y.
- For all other conflicts c affects nodes n_X and m_Y.
- By default, n_X is the loser node, unless stated otherwise.
- si_t is the synchronization iteration where c is detected. si_{t+1} is the consecutive sync iteration where c is resolved.

For each conflict we explain whether the resolution of c itself introduces a new conflict c' in si_{t+1}. This is typically not the case, but if it is, we clarify that it's impossible that the resolution of c' would cause a conflict c'' in si_{t+2}, etc., leading to a never-terminating loop. For conciseness of our proofs we define that the second and third parameter of $ancestor(snapshot, n, m)$ may also be update tree *nodes* rather than IDs. In other words $ancestor(dbsnapshot, n, m) := ancestor(dbsnapshot, n.i_{db}, m.i_{db})$ and $ancestor(snapshot_X, n, m) := ancestor(snapshot_X, n_X.ID, m_X.ID)$.

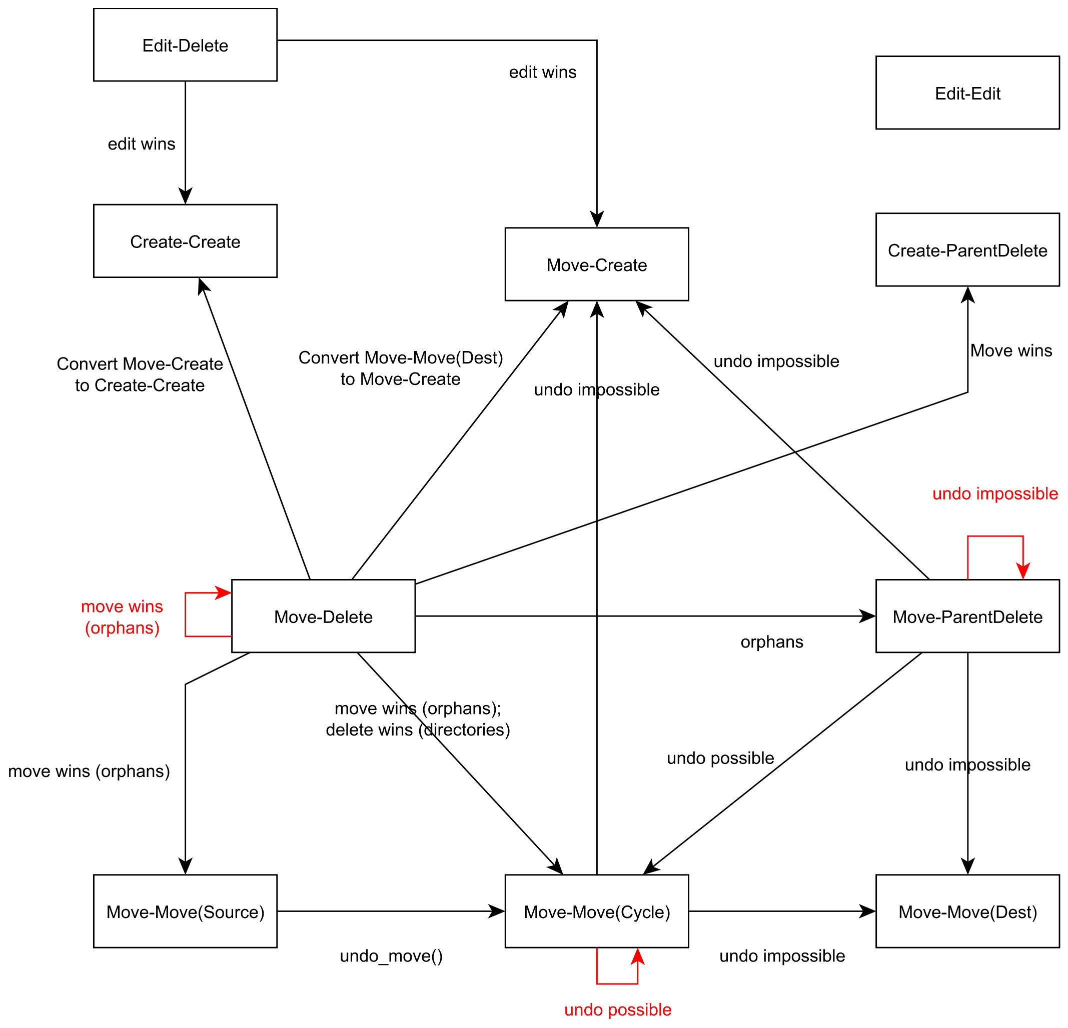


Figure 5.12: Conflict dependency graph

5.8.2 Move-Move (Source)

We apply *undo_move()* to the loser node. Physically undoing the move of n_X may be possible or impossible, as explained in appendix A.7.1.

5.8.2.1 Undo move is possible

The object is moved only physically, the database is untouched. In si_{t+1} the *Move* change-event for n_X is no longer detected. This solves c. A close examination of every other conflict type shows that *undo_move()* can only cause a Move-Move (Cycle) conflict c' to be detected in si_{t+1} in certain situations:

1. Create-Create, Edit-Edit, Edit-Delete, Create-ParentDelete, Move-ParentDelete, Move-Move (Source): resolving c does not cause new *Move*, *Create*, *Edit* or *Delete* change-events in si_{t+1}. Thus, c' must have existed in si_t already.

2. Move-Create, Move-Move (Dest): see point 1 - if another Move-Create / Move-Move (Dest) conflict c'' that already existed in si_t affects n_X with a *Move* change-event, then resolving c also resolves c''.

3. Move-Delete: see point 1 - also note that n_Y cannot have the *Move* and *Delete* change-event at the same time.
4. Move-Move (Cycle): it's possible that the existence of c for node n introduces a Move-Move (Cycle) conflict c' in si_{t+1}. For instance, c' might exist for two other distinct nodes q and r, as illustrated in figure 5.13.

5.8.2.2 Undo move is impossible

The object is moved to the root (with a unique conflict suffix) physically, and the DB entry is updated. In si_{t+1} the *Move* change-event of n_X is no longer detected, which solves c. The *Move* change-event node for n_Y is *still* detected in si_{t+1}. As a side effect this solves other conflicts such as Move-Create, Move-Move (Dest) in some situations.

It is not possible that $undo\_move(n_X)$ applied in si_t introduces a Move-Move (Cycle) conflict c' in si_{t+1}, given that undoing the move is impossible. Theorem 2 shows that no c' can exist for node n and some other node q. Theorem 3 shows that no c' can exist for two other distinct nodes q and r.

Lemma 2. *Consider nodes n and q situated in an arbitrary relationship in si_t, see figure 5.14 for possible configurations. After moving n to the root level, there cannot be a parent-child relationship between n and q in si_{t+1}, unless there already was one in si_t. The proof is a trivial proof by exhaustion, by examining the effect of moving n to the root level for each configuration shown in figure 5.14.*

Theorem 2. *When $undo\_move(n_X)$ is used to resolve a conflict c in si_t (conflict type of c can be Move-Move (Cycle), Move-Move (Source), Move-Delete, Move-ParentDelete) and undoing the move operation of n_X is impossible, $undo\_move(n_X)$ cannot introduce new a Move-Move (Cycle) conflict c′ between n and some other node q in si_{t+1}.*

Proof. by contradiction. For c' to be found only in si_{t+1} (see section 5.5.10 for details) $undo\_move(n_X)$ would need to cause oppositional parent-child relationships between q and n in the replicas in si_{t+1} (where these relationships did not exist in si_t yet, because if they had already existed in si_t, c' would have been detected in si_t already). However, lemma 2 demonstrates that moving n_X to the root cannot *introduce* a new parent-child relationships between n_X and q_X in si_{t+1}.

□

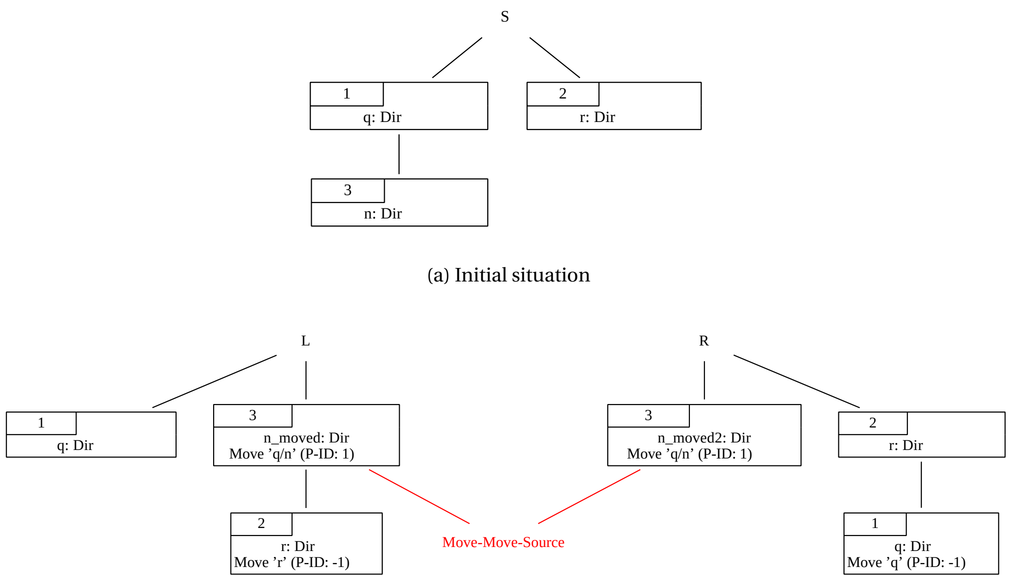
Lemma 3. *Consider nodes n, q and r situated in an arbitrary relationship in si_t, see figure 5.15 for possible configurations. After moving n to the root level, there cannot be a parent-child relationship between q and r in si_{t+1}, unless there already was one in si_t. The proof is a trivial proof by exhaustion, by examining the effect of moving n to the root level for each configuration shown in figure 5.15.*

Theorem 3. *When $undo\_move(n_X)$ is used to resolve a conflict c in si_t (conflict type of c can be Move-Move (Cycle), Move-Move (Source), Move-Delete, Move-ParentDelete) and undoing the move operation of n_X is impossible, $undo\_move(n_X)$ cannot introduce new a Move-Move (Cycle) conflict c′ between two distinct nodes q and node r in si_{t+1}.*

Proof. by contradiction: see proof for theorem 2. Now lemma 3 demonstrates that moving n_X to the root cannot *introduce* a new parent-child relationships between q_X and r_X in si_{t+1}. □

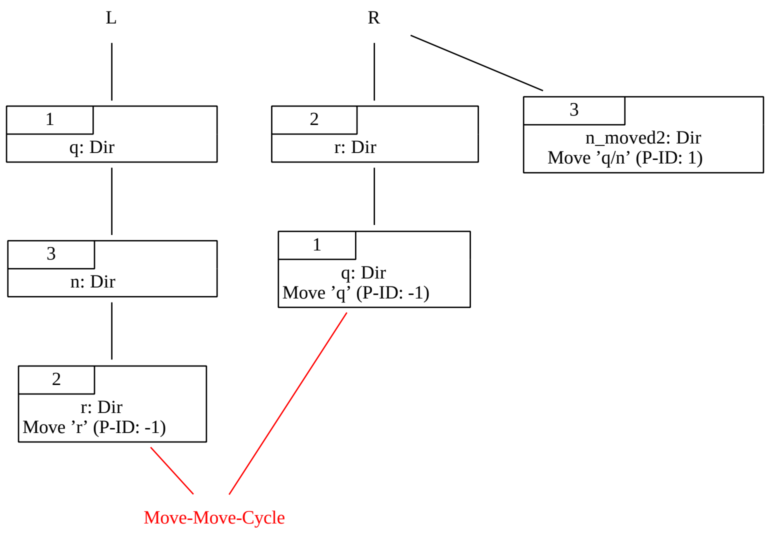
5.8.3 Create-Create

The resolution physically renames n_X by appending a random conflict suffix. In si_{t+1} conflict c is solved because $n_X.name \neq m_Y.name$. It is not possible that the resolution introduces another conflict c' of type Create-Create, Move-Create or Create-ParentDelete, because:



(a) Initial situation

(b) Detection of Move-Move (Source) conflict in si_t



(c) Detection of Move-Move (Cycle) conflict in si_{t+1}

Figure 5.13: Move-Move (Source) turns to Move-Move (Cycle) (undo possible)

Subfigure (a) shows the database state. We construct a situation where we first build a Move-Move (Cycle) conflict c' between nodes q and r. Because $undo\_move(n_X)$ only modifies replica X, q and r must have been in a parent-child relationship in replica Y. We locally move r into n (which also means that r is below q, since n is a child of q) and remotely move q into r. To disguise c' we move the intermediate node, n, to the root on both replicas, but with slightly different names. This causes the Move-Move (Source) conflict c for n, hiding the parent-child relationship between q and r. When synchronizing the current situation, we detect only c as shown in subfigure (b). After resolving c in favor of replica R we detect c' in si_{t+1} as shown in subfigure (c).

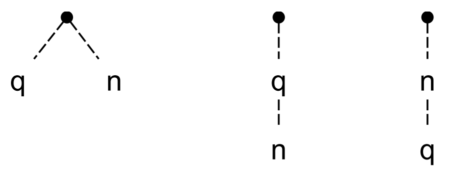


Figure 5.14: Configurations of two nodes q and n

We note that *dashed* lines indicate that other nodes (not shown) may exist between the beginning and end of a dashed line.

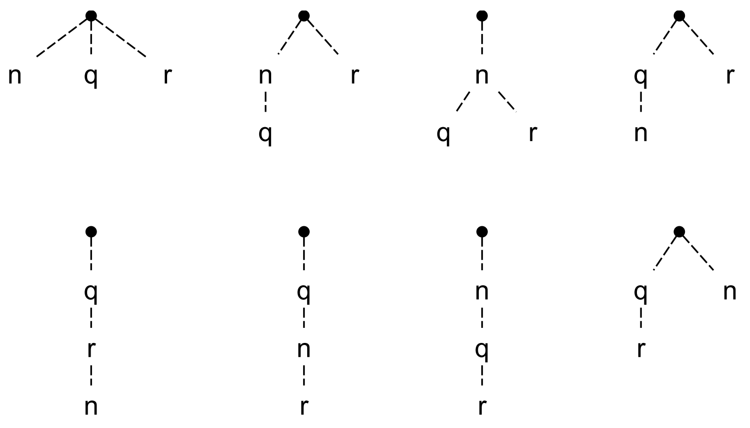


Figure 5.15: Configurations of nodes q, r and n

We note that q and r are interchangable.

- Create-Create, Move-Create: $n_X.name$ was chosen at random. A conflict c' would require that another node m'_Y exists (with *Create* or *Move* change-event), where $m'_Y.parent = m_Y.parent \wedge m'_Y.name = n_X.name$, which is extremely unlikely due to the *randomized* name suffix.
- Create-ParentDelete: The existence of m_Y with a *Create* change-event implies that $m_Y.parent$ must not have a *Delete* change-event (a created object cannot exist in a deleted directory). Because our conflict resolution did not delete the physical path of $m_y.parent$ or $n_X.parent$, their change-events must have stayed the same between si_t and si_{t+1} which means that it is impossible that a c' Create-ParentDelete conflict was introduced by our resolution algorithm.

5.8.4 Move-Create

Appending the conflict suffix to n_X's name has exactly the same effect as described in section 5.8.3, irrespective of whether n_X's change-event is *Move* or *Create*.

5.8.5 Move-Move (Dest)

Appending the conflict suffix to n_X's name has exactly the same effect as described in section 5.8.3.

5.8.6 Move-Move (Cycle)

Uses undo_move() of the loser node.

5.8.6.1 Undo move is possible

Similar to the argumentation of section 5.8.2.1, resolving a conflict c via $undo\_move(n_X)$ may introduce another Move-Move (Cycle) conflict c' in si_{t+1}. An example is shown in figure 5.16.

5.8.6.2 Undo move is impossible

The file is moved to the root (with conflict suffix) physically, and the DB entry is updated. In si_{t+1} this introduces a *Move* change-event for node n_Y. This can cause Move-Create, or Move-Move (Dest) follow-up conflicts, but no Move-Delete, Move-ParentDelete, Move-Move (Source) or Move-Move (Cycle) follow-up conflicts:

- Move-Delete, Move-ParentDelete: the resolution of c just performed a physical *move* operation (and changed the parent and name of the object in the database). Nothing was deleted, thus no *Delete* change-event could have been introduced in si_{t+1}.
- Move-Move (Source): because the resolution of c moved n_X to the root on replica X, no *Move* change-event is detected in si_{t+1} for n_X, thus the introduction a Move-Move (Source) conflict for n is impossible. Since the resolution does not affect m_X at all, the introduction of a Move-Move (Source) conflict for m_Y is also impossible.
- Move-Move (Cycle): see theorems 2+3.

Note that it is generally impossible that our resolution causes an infinite loop, see theorem 4.

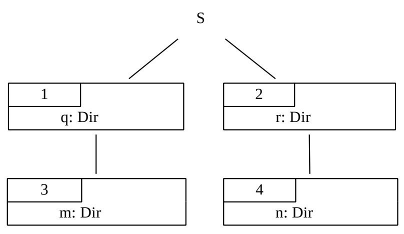
Theorem 4. *Resolving Move-Move (Cycle) conflicts cannot cause an infinite loop due to the introduction of new Move-Move (Cycle) conflicts.*

Proof. by contradiction: A necessary condition for finding a Move-Move (Cycle) conflict c is that at least one directory node with a *Move* change-event exists on *each* replica. To cause an infinite loop, resolving c would have to keep the number of *Move* change-events for directories stable (or even increase it) on *both* replicas. This is impossible, for two reasons: (1) the configured option of which replica should win conflict c is always the same for each resolution process (e.g. "remote replica wins"); (2) the only effect of *undo_move()* on the loser replica is the reduction of *Move* change-events by 1. Consequently, once only Move-Move (Cycle) conflicts remain, the number of directory nodes on the losing replica with a *Move* change-event involved in a Move-Move (Cycle) conflict must reach 0 eventually. □

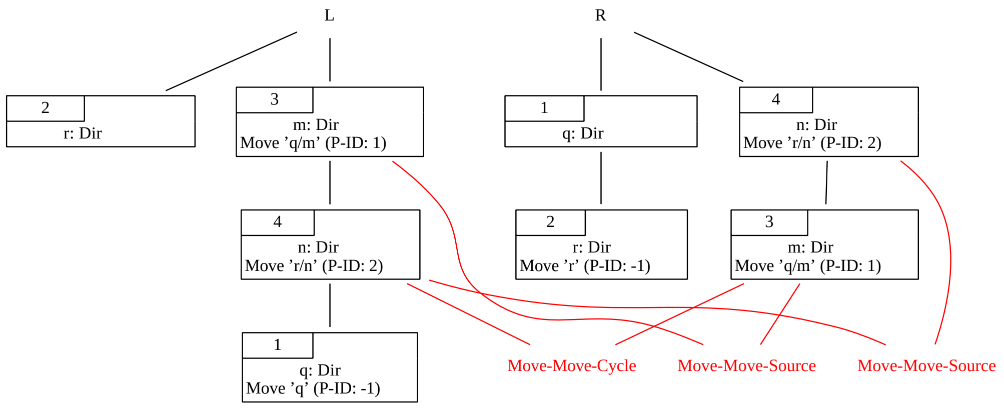
5.8.7 Edit-Delete

If the configured option is *delete wins*: because our resolution of c removes the affected file both physically and from the database, n will no longer exist in si_{t+1}. Because file nodes cannot have children, there cannot be any other side-effects on other nodes.

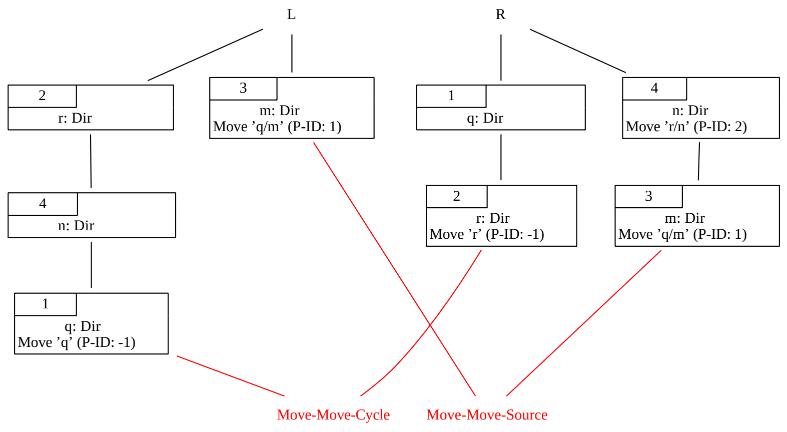
If *edit wins*: resolving c removes the database entry of n. Thus, the only consequence is that the physical file is detected with a *Create* change-event in si_{t+1}. This can cause a Create-Create or Move-Create conflict c' in si_{t+1}, when a corresponding child node of $n_Y.parent$ exists, with a *Create* or *Move* change-event. It cannot cause a Create-ParentDelete conflict, because when we resolve c we explicitly check for a Delete change-event in $n_Y.parent$ and then move n_X physically to the root. For that move operation of n_X to cause a Create-ParentDelete conflict c' in si_{t+1} this would require that replica Y's root was deleted, which is an exceptional situation in which no synchronization is executed anyway.



(a) Initial situation



(b) Detection of Move-Move (Cycle) conflict c in si_t



(c) Detection of Move-Move (Cycle) conflict c' in si_{t+1}

Figure 5.16: Move-Move (Cycle) introduces new Move-Move (Cycle) (undo possible)

Subfigure (a) shows the database state. We construct a situation where we first build a Move-Move (Cycle) conflict c' between nodes q and r which is then disguised by moving n_X into m_X, which causes a Move-Move (Cycle) conflict c between m and n. Subfigure (b) illustrates c. After $undo\_move(n_X)$ was applied, conflict c' is discovered in si_{t+1}, as illustrated in subfigure (c).

5.8.8 Edit-Edit

The only effect of resolving c is that neither n_X nor n_Y will have a *Edit* change-event in iteration si_{t+1}. All other nodes and their change-events are exactly as they were in si_t. Consequently, resolving c cannot cause follow-up conflicts c'.

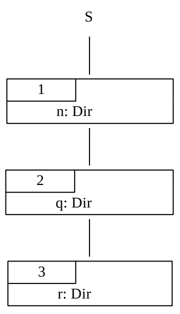
5.8.9 Move-Delete

If the configured option is *delete wins*:

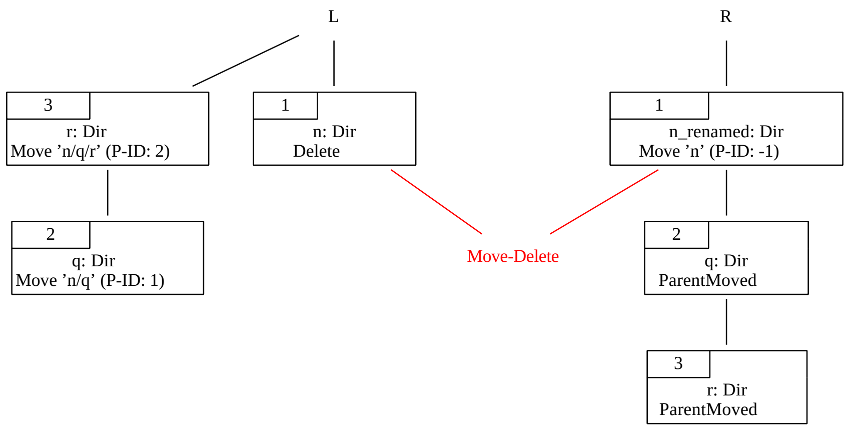
- If node is a *file*: See section 5.8.7 “delete wins”.
- If node is a *dir*:
  - If undo_move() is possible: as described in section 5.8.2.1 a Move-Move (Cycle) follow-up conflict c' is possible.
  - Otherwise, no other conflicts are caused:
    - Move-Move (Cycle): see theorems 2+3.
    - Move-Move (Source) does not make sense, because n_Y is still deleted in si_{t+1}.
    - Move-Move (Dest) and Move-Create are impossible due to the randomness of the conflict suffix.
    - Move-Delete and Move-ParentDelete are impossible, the resolution of c just performed a physical *move* operation (and changed the parent and name of the object in the database). Nothing was deleted, thus no *Delete* change-event could have been introduced in si_{t+1}.

If the configured option is *move wins*: the resolution described in appendix A.7.2 may cause the following follow-up conflicts c':

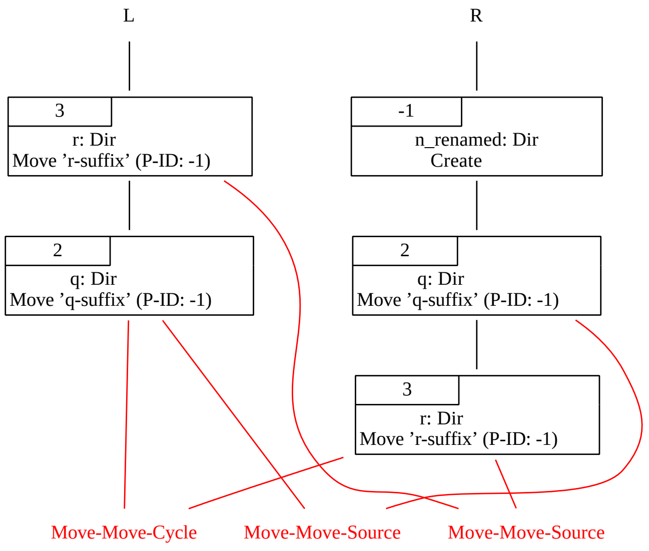
- Move-Move (Source): if orphans are found their paths are bent to the root in the database. This causes Move-Move (Source) conflicts which are automatically resolved in favor of the delete-replica.
- Move-Move (Cycle): Let q and r both be orphan nodes below n for which $ancestor(dbsnapshot, q, r) \wedge ancestor(snapshot_X, q, r) \wedge ancestor(snapshot_Y, r, q)$ holds in si_t. Resolving c' makes q and r independent in the database in si_{t+1}, while not affecting the physical oppositional parent-child relationships. This causes a Move-Move (Cycle) conflict c' in si_{t+1}. An example is shown in figure 5.17.
- Create-Create: given that a Move-Create conflict c'' existed in si_t between n and some other node q (s.t. we detect c and c'' in si_t), then resolving c transforms c'' to a Create-Create conflict c' in si_{t+1}.
- Move-Create: given that a Move-Move (Dest) conflict c'' existed in si_t between n and some other node q (s.t. we detect c and c'' in si_t), then resolving c transforms c'' to a Move-Create conflict c' in si_{t+1}.
- Move-Delete: let q be an orphan node of n. Let node r with $ancestor(dbsnapshot, q, r)$ be deleted on replica X - but r is not modified, moved or deleted on replica Y. Then we detect no Move-Delete conflict c' for r in si_t. However, resolving c will cause r to be detected with a Move change-event in replica Y in si_{t+1}, which introduces c'. An example is shown in figure 5.18. However, as theorem 5 shows, this cannot cause an infinite cycle.



(a) Initial situation

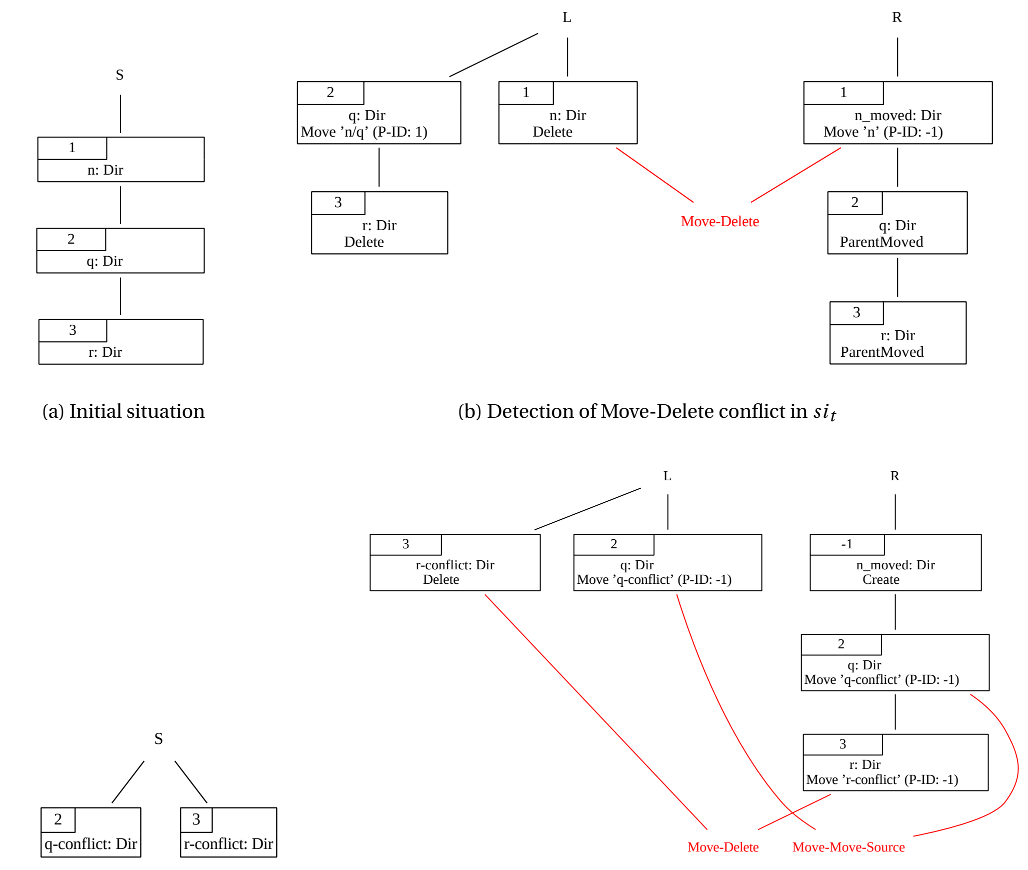


(b) Detection of Move-Delete conflict c in si_t


L
R
3
r: Dir
Move 'r-suffix' (P-ID: -1)
2
q: Dir
Move 'q-suffix' (P-ID: -1)
-1
n_renamed: Dir
Create
2
q: Dir
Move 'q-suffix' (P-ID: -1)
3
r: Dir
Move 'r-suffix' (P-ID: -1)
Move-Move-Cycle
Move-Move-Source
Move-Move-Source

(c) Detection of Move-Move (Cycle) conflict c' in si_{t+1}

Figure 5.17: Move-Delete introduces new Move-Move (Cycle)



(a) Initial situation

(b) Detection of Move-Delete conflict in si_t

(c) Database state after resolving c

(d) Detection of new Move-Delete conflict in si_{t+1}

Figure 5.18: Move-Delete resolution yields a new Move-Delete conflict

- Move-ParentDelete: let q be an orphan node. Let nodes r and s with $ancestor(dbsnapshot, q, r) \wedge ancestor(dbsnapshot, r, s)$ both be deleted on replica X but r and s are not modified, moved or deleted on replica Y. Then we detect no Move-Delete or Move-ParentDelete conflicts c' for r or s in si_t. However, resolving c will cause r and s to be detected with a Move change-event in replica Y in si_{t+1}, which introduces a Move-ParentDelete conflict c' for r and s and one Move-Delete conflict for r and s each.

- Create-ParentDelete: as explained in figure 5.4 on page 87, if n_X is affected by a Move-Delete c and a Move-ParentDelete conflict c' in si_t, resolving c will transform the Move-ParentDelete conflict to a Create-ParentDelete conflict.

- All other conflict types (Move-Move (Dest), Edit-Edit, Edit-Delete) are impossible, because the resolving c does not introduce the corresponding events in si_{t+1}.

Theorem 5. *Resolving Move-Delete conflicts cannot cause an infinite loop due to the introduction of new Move-Delete conflicts.*

Proof. by contradiction: As figure 5.18 demonstrates, the reason why resolving c causes c' is because $ancestor(dbsnapshot, n, q) \wedge ancestor(snapshot, q, r)$ holds, q and r are orphan nodes, and r_X is deleted. c' then applies to r. For c' to cause a new Move-Delete conflict c'', such relationships would still need to exist, i.e., there would have to be child nodes of r that can become orphaned. However, the resolution of c changes the database in such a way that all sub-nodes of r are moved to the root level, i.e., they are in a *flat* hierarchy, where no node below r exists. For any c' caused by resolving c, resolving c' is trivial. The configuration must still be "move wins", thus, the entry for r will be deleted from the database (to reconstruct r in si_{t+2}), without finding any orphan nodes below r. □

5.8.10 Move-ParentDelete

Conflict c affects nodes n and m where n_X is moved into m_X and m_Y is deleted.

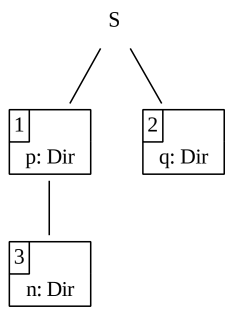
- If undo_move() is possible: Similar to the argumentation of section 5.8.2.1, resolving a conflict c via $undo\_move(n_X)$ may introduce another Move-Move (Cycle) conflict c' in si_{t+1}.
- If undo_move() is impossible: The object is moved to the root (with conflict suffix) physically, and the DB entry is updated. In si_{t+1} this introduces a *Move* change-event for node n_Y. This can cause Move-Create, Move-Move (Dest), or Move-ParentDelete follow-up conflicts, but no Move-Delete, Move-Move (Source) or Move-Move (Cycle) follow-up conflicts:
  - Move-Delete: the resolution of c just performed a physical *move* operation (and changed the parent and name of the object in the database). If n_Y did not have a Move change-event in si_t this introduces one in si_{t+1}. However, n_X was not deleted when resolving c, thus no Move-Delete conflict c' can be introduced for n. Because resolving c did not delete anything, no c' for another node could have been introduced in si_{t+1}.
  - Move-Move (Source): because the resolution of c moved n_X to the root on replica X, no *Move* change-event is detected in si_{t+1} for n_X, thus the introduction a Move-Move (Source) conflict for n is impossible. Since the resolution does not affect m_X at all, the introduction of a Move-Move (Source) conflict for m_Y is also impossible.
  - Move-ParentDelete: resolving a conflict c via $undo\_move(n_X)$ may introduce another Move-ParentDelete conflict c' in si_{t+1}, where c' affects n_Y and q_X, given that $ancestor(dbsnapshot, q, n) \wedge ancestor(snapshot_Y, q, n)$ holds. An example is shown in figure 5.19. However, as theorem 6 shows, this cannot cause an infinite cycle.
  - Move-Move (Cycle): is impossible, see theorems 2+3.

Theorem 6. *Resolving Move-ParentDelete conflicts cannot cause an infinite loop due to the introduction of new Move-ParentDelete conflicts.*

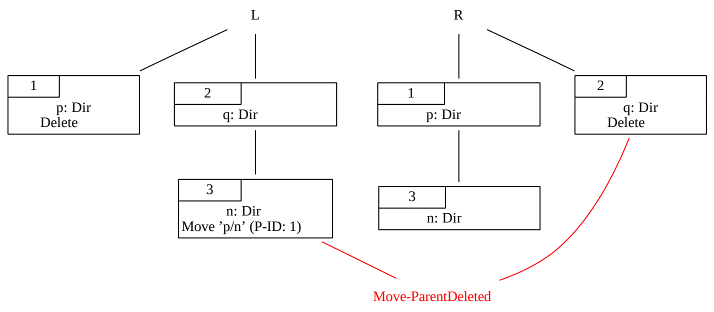
Proof. by contradiction: As figure 5.19 demonstrates, resolving c may introduce c' in case the following two conditions hold: (1) $undo\_move(n_X)$ is impossible, which causes a new *Move* change-event in n_Y in si_{t+1}, and (2) the corresponding node of $n_Y.parent$ already has a *Delete* change-event in si_t on replica X. For the resolution of Move-ParentDelete conflicts to cause an infinite loop, it would be necessary that resolving c' (which exists due to the new *Move* change-event in n_Y) would be able to cause another Move-ParentDelete conflict c'' in si_{t+2}. However, this is impossible, because when resolving c' $undo\_move(n_Y)$ is always possible: $undo\_move(n_Y)$ means to move n_Y to the root level, with the unique conflict suffix (unique part of suffix not shown in figure 5.19 for brevity), which cannot fail, as the path must still be free in replica Y. Consequently, none of the reasons for *undo_move()* to be impossible do apply (see appendix A.7.1). □

5.8.11 Create-ParentDelete

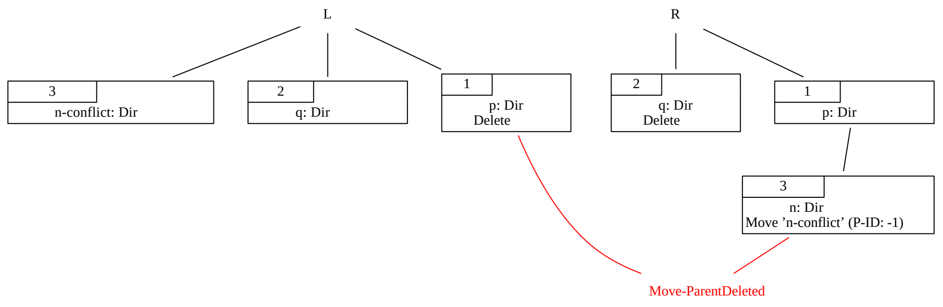
A Create-ParentDelete conflict c affects nodes n_X and m where n_X is a created object under m_X, while m_Y is deleted. Because resolving c simply moves created nodes to the root physically, with a random conflict-suffix (and without modifying the database), it is not possible to cause any follow-up conflict c'. The resolution of c does not introduce any new change-events. The node n_X is detected with a *Create* change-event in si_{t+1}, just like in si_t.



(a) Initial situation



(b) Detection of Move-ParentDelete conflict in si_t



(c) Detection of new Move-ParentDelete conflict in si_{t+1}

Figure 5.19: Move-ParentDelete resolution yields a new Move-ParentDelete conflict

5.9 Conclusion

This chapter answers RQ3 which focuses on conflicts. We solve the problem of *identifying* all conflicts by analyzing the preconditions of operation pairs, looking for cases where the effect of one operation violates the precondition of the other one. For our model $\mathscr{F}$ we found a total of ten conflicts. By repeating the analysis for other file system models we found a dependency between conflicts and model, i.e., the number of conflicts increases with the number and complexity of file system operations. By combining all possible conflict pairs we found that *several* conflicts may affect a specific object. To address this we contributed an *iterative* conflict resolution approach that solves one conflict at a time, automatically without involving the user. It is configurable, as it offers several criteria to decide which operation wins or loses. It also avoids negative side effects for other files or directories that are not part of the conflict, by modifying the loser operation instead of the winner operation. To decide how conflicts are resolved, each part of our four-step framework improves contributions made by related academic works. For instance, we make conflict *awareness* an integral part of our consistency philosophy, and our resolution is not arbitrary but based on a discussions and comparisons of different options. The resulting algorithm is simple to implement and terminates provably.

Chapter 6

Static synchronization

This chapter presents our synchronization algorithm called *Syncpal*, which combines conflict handling (presented in chapter 5) and operation sorting (see section 4.2.7) into a high-level algorithm that solves state-based synchronization of two file system replicas. Parts of this chapter (as well as conflict related aspects from chapter 5) are also found in [She19]. To facilitate the explanation, we discuss the simplified case of *static* synchronization where we assume that the file system is not changed by the user during synchronization. This assumption is reasonable for file synchronizers explicitly invoked by the user, because she can refrain from changing the file system during synchronization (and close all other applications that might cause changes). In practice, however, many file systems cannot be locked by a synchronizer. Thus, concurrent changes may invalidate an ongoing synchronization process. We refer to section 7.3 which introduces the *dynamic* synchronization architecture of our implementation which can handle such concurrent changes. We start with an overview of the algorithm in section 6.1. The remaining sections explain the individual components introduced in section 6.1 and provide a conclusion.

6.1 Algorithm overview

On a high level our algorithm is similar to the one presented in [BP98] where file system synchronization is broken down into a three-stage process - *update detection*, *reconciliation* and *propagation*, which we briefly introduce in the following subsections. In figure 6.1, a flow chart illustrates how our static synchronization algorithm works in detail. The process starts at the top and follows the solid, bold arrows. Computations happen in blue, hexagonal shapes. Outputs of computations are colored green. Bold, dashed arrows associate computations with their output. Inputs are associated to computations using thin, dashed arrows. Decisions are yellow. To reduce clutter, computations that require the local/remote tree as input do not use incoming arrows, but instead have the tree icon as part of their shape.

6.1.1 Update detection

In the *static* synchronization approach, update detection is triggered in regular intervals (whereas the *dynamic* version triggers update detection based on file system activity). This involves generating *four* different snapshots. File system (FS) snapshots are generated using APIs provided by the file system, see section 4.2.1, yielding $snapshot_{local}$ and $snapshot_{remote}$. They represent the current state of the local and the remote replica respectively. To find changes since the last synchronization, *compute_ops*() (see section 4.2.3) does not compare those two snapshots, but compares the file system snapshot of each replica with the corresponding *database* (DB) snapshots $dbsnapshot_{local}$ and $dbsnapshot_{remote}$ introduced in section 4.2.2, which contain the historic state of each replica at the point of the last synchronization. After computing

$$O_{local} = compute\_ops(dbsnapshot_{local}, snapshot_{local})$$

and

$$O_{remote} = compute\_ops(dbsnapshot_{remote}, snapshot_{remote})$$

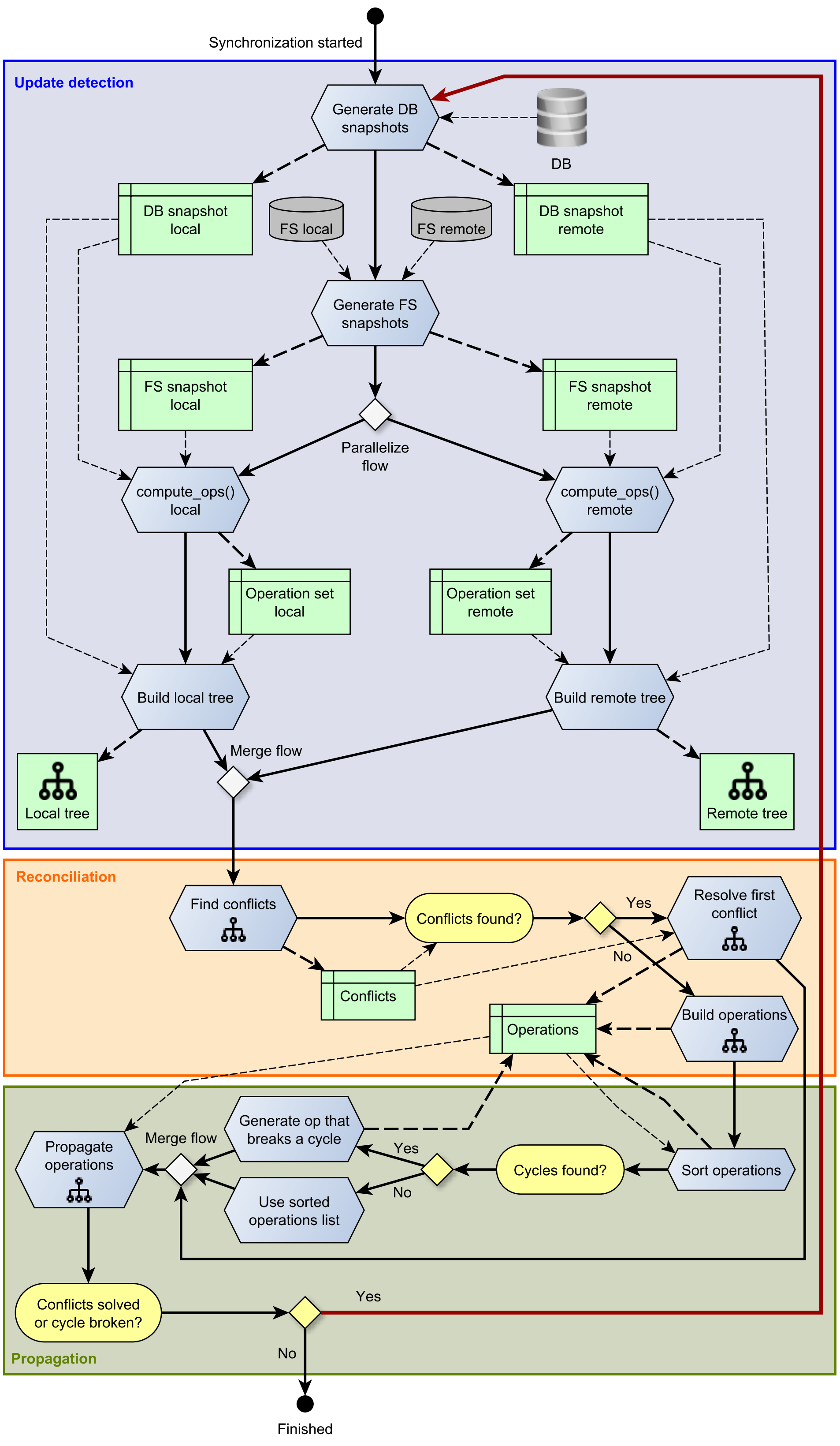


Figure 6.1: Static synchronization algorithm

the operation sets are transformed into the corresponding in-memory *update trees* for further processing, as described in section 4.2.4. These trees are given to the reconciliation phase.

6.1.2 Reconciliation

The goal of reconciliation, which is detailed further in section 6.2, is to compute a list of operations from the local and remote tree that will equalize both replicas once all operations have been executed. In some cases achieving this goal in just *one* iteration (top to bottom in figure 6.1) is impossible. If conflicts are detected our algorithm instead first sorts the conflicts and solves the first one by generating a new list of one or more *already sorted* resolution operations which it sends to the propagation phase. It then restarts automatically, as shown by the dark-red arrow in figure 6.1.

6.1.3 Propagation

In the propagation phase, see section 6.3 for more details, *non-conflicting* operations are first sorted to avoid that operations scheduled earlier violate preconditions of later operations. If the sorting algorithm finds that operations depend on each other in a *cyclic* relationship, the operation list is replaced by a single operation that breaks the cycle. The operation(s) are then executed, updating the physical replicas, the corresponding in-memory trees and the local state database.

6.2 Reconciliation

The reconciliation phase computes the list of *Syncpal operations* that need to be executed. It requires $updatetree_{local}$ and $updatetree_{remote}$ to build them. Reconciliation takes place as follows:

1. Iterate over all nodes in both trees and determine `conflicts`, which is a *sorted* list of conflicts found in the tree (see section 5.5 and appendix A.6.1 for more information).
2. If `conflicts` is non-empty, generate a list of operations that solve the *first* conflict in `conflicts`. See section 5.7 for more information. Provide the list to the propagation phase.
3. Otherwise, if no conflicts are found, iterate over all *unprocessed* nodes[1] in a breadth-first approach and generate *Syncpal operations* according to each node's *change-events* as follows:
   - *Create*: generate a create-operation. As with all other operations presented next, the *operation* data structure contains the corresponding node from the respective update tree, allowing the propagation phase to query the node for the operation-specific information. In case of *Create*, this information includes whether a file up-/download or a remote/local directory creation is requested, as well as the corresponding paths at the time of execution. If the node is affected by a *pseudo* Create-Create conflict, the operation is provided with the *omit* flag (see next section for clarification). Mark the node processed. If the node is affected by a pseudo Create-Create conflict, also mark the corresponding create node processed.
   - *Move*: generate a move-operation. If the node is affected by a *pseudo* Move-Move (Source) conflict, the operation is provided with the *omit* flag and the corresponding node is marked processed. Mark the node processed, unless the node also has an *Edit* change-event, in which case mark the node as partially processed.
   - *Edit*: generate an edit-operation. If the node is affected by a *pseudo* Edit-Edit conflict, the operation is provided with the *omit* flag (and the corresponding node on the other replica is also marked processed). Mark node processed if the node has no move change-event (or has one but is already marked partially processed), otherwise mark it partially processed.

[1]Before starting the iteration in step 3, we mark all nodes *unprocessed*. The iteration algorithm returns the next unprocessed node using a breadth-first approach, considering the union of $updatetree_{local}$ and $updatetree_{remote}$.

- *Delete*: generate a delete-operation, which is given the update tree node and all its child-nodes in case of a directory node (if any), because the propagation component needs to delete the paths of the dir and all its children from the database and file system. If the corresponding node of the other replica also has a delete flag, the operation is provided with the *omit* flag. Mark the node and all its child-nodes processed. Also find the corresponding node in the update tree of the other side and mark it (and all its child-nodes) processed.
- No change-events: don't generate an operation. Mark the node processed.

Step 2 illustrates our *iterative* approach. This keeps the implementation complexity simple, at the expense of execution time. In a scenario where there are 2 files affected by different conflicts, and 10 other files not affected by any conflicts, the first iteration would generate (and execute) operations that solve the first conflict. Then the synchronization would restart and would detect the one remaining conflict and, say, 11 files not affected by any conflict[2]. The second iteration therefore generates and executes operations to mitigate the second conflict. Then the synchronization would restart again, no conflicts would be found and operations are generated for, say, 12 conflict-free files or dirs.

6.3 Propagation

The goal of the propagation phase is to sort and execute the Syncpal operations provided by the reconciliation phase. The *Syncpal operation* data structure contains all necessary information required to do so. Depending on the specific type of the operation, it will modify the relational database, the file system of the replicas, and the $updatetree_{local}$ or $updatetree_{remote}$ objects. A *Syncpal operation* structure is given the following information:

- Both $updatetree_{local}$ and $updatetree_{remote}$.
- Affected node (of the replica where the operation was *detected*).
- Omit flag: boolean flag, if True, the operation only needs to change the database and omits changing the physical replicas. This is the case when the *same* change was physically detected in both replicas (pseudo conflict) and only the database needs to be updated to avoid detecting this change again.
- Corresponding node (optional, only if *omit* flag is True) on the other replica.
- Conflict information (optional), such as name of the conflict or affected paths, to be displayed to the user at the time of execution.

As illustrated in figure 6.1, the operations are sorted in some cases, or remain unsorted in other cases. If the operations are *conflict resolution* operations, they don't need to be sorted, because the resolution algorithm already makes sure the operations are provided in an order which doesn't violate the file system limitations. Otherwise, if no conflicts were detected, the provided list of operations needs to be sorted before execution, because operations are scheduled by the reconciliation phase in an order determined by the *breadth-first* iteration of the trees. The propagation would otherwise fail to execute these operations due to the limitations of the file system APIs, see section 4.2.7.1 for more details. In case a *cycle* is found while sorting, our algorithm generates a single operation that breaks this cycle and executes it. Otherwise the sorted list of operations must be in a valid, non-cyclic order, ready for propagation.

For the remainder of this section, we first discuss the postconditions of each operation in more detail in section 6.3.1. In section 6.3.2 we describe how operations are sorted and cycles are resolved. In section 6.3.3 we describe and solve several challenges that arise during propagation, which exist mainly due to the lack of transactions on desktop file systems.

[2]Assume that solving the first conflict was done in a way that it produces a single new file/dir that is no longer in conflict with the other replica

6.3.1 Operation details

The following subsections detail how each respective Syncpal operation changes the file system, database snapshot and update trees. The first subsection is more verbose than the others, which omit some details already explained in the first one.

6.3.1.1 Create operation

A *create* operation is based on a file system operation $create_X(i_X, p_X, name_X)$ on replica X (where X is a placeholder for $local$ or $remote$) with $create := createdir \vee createfile$, which was detected by *compute_ops()* and caused the corresponding node n_X to be created in $updatetree_X$, with a *create* change-event. If the *omit* flag is set for the *create* operation, it is also given the corresponding create node n_Y in $updatetree_Y$.

The execution of the create operation consists of three steps:

1. If omit-flag is False, propagate the file or directory to replica Y, because the object is missing there.
2. Insert a new entry into the database (and database snapshot), to avoid that the object is detected again by *compute_ops()* on the next sync iteration.
3. Update the *update tree* structures to ensure that follow-up operations can execute correctly, as they are based on the information in these structures.

The implementation of each step requires the explanation of a few details:

1. The path on replica Y is computed as $path_Y = path(n_{p_Y})/n_X.name$. This requires finding the corresponding parent node n_{p_Y} in $updatetree_Y$ via the id_{db} of $n_X.parent$, s.t. $n_X.parent.id_{db} = n_{p_Y}.id_{db}$. The sorting algorithm described in section 4.2.7.1 guarantees that the *create* operation of a parent directory is executed before the *create* operation of any of its children. Therefore, n_{p_Y} must exist in $updatetree_Y$. Once $path_Y$ is computed, the corresponding $create$ operation can be executed. During execution, i_Y and $lastmodified_Y$ are returned by the file system APIs of replica Y. Should the *omit* flag be set, i_Y and $lastmodified_Y$ are already known from the corresponding node, i.e., $i_Y = n_Y.ID$ and $lastmodified_Y = n_Y.lastmodified$.
2. A new tuple $\langle id_{db}, name, id_{local}, id_{remote}, lastmodified_{local}, lastmodified_{remote}, type\rangle$ is inserted into the database, where X and Y take the $local$ or $remote$ value respectively, with $name = n_X.name$. The database component generates and returns id_{db}.
3. Set $n_X.id_{db} = id_{db}$ (with id_{db} from step 2). If n_Y was given (omit-Flag = True), also set $n_Y.id_{db} = id_{db}$. Otherwise create and insert a new node n_Y below n_{p_Y} with the corresponding values.

6.3.1.2 Edit operation

An *Edit* operation is either based on a file system operation $edit_X(i_X, op)$, or by operations

$$deletefile_X(i_X, p_X) + createfile_X(j_X, p_X, n_X)\,[+edit_X(j_X, op)]$$

where the last operation fills the new file with some content, and $n_X = name(dbsnapshot_X, i_X)$. Either case causes the generation of a corresponding node n_X in $updatetree_X$ with an *edit* change-event. If the *omit* flag is set to True for the *edit* operation, it is also given the corresponding edit node n_Y in $updatetree_Y$.

The execution of the edit operation consists of three steps:

1. If omit-flag is False, propagate the file to replica Y, replacing the existing one.
2. Update the database entry (and database snapshot), to avoid detecting the edit operation again.

3. If the omit flag is False, update the $updatetree_Y$ structure to ensure that follow-up operations can execute correctly, as they are based on the information in this structure.

Further details:

1. $path_Y$ is computed as $path(n_Y)$ where n_Y is found in $updatetree_Y$ using $n_X.id_{db}$.
2. The values $lastmodified_X$ and $lastmodified_Y$ are always updated. If id_{local} or id_{remote} no longer match $n_X.ID$ or $n_Y.ID$ because the file was actually deleted+created, id_{local} and id_{remote} are updated, too.
3. $n_Y.ID$ and $n_Y.lastmodified$ are updated with the values determined during propagation.

6.3.1.3 Move operation

A *move* operation is based on a file system operation $move_X(i_X, u_X, v_X, name_X)$ which was detected by *compute_ops()* and caused the corresponding node n_X to be created in $updatetree_X$, with a *move* change-event. The omit flag is set for the *move* operation in case the same move was also detected in $updatetree_Y$ (pseudo Move-Move (Source) conflict).

The three execution steps are as follows:

1. If omit-flag is False, move the object on replica Y (where it still needs to be moved) from u_Y to v_Y, changing the name to $name_X$.
2. Update the database entry (and database snapshot), to avoid detecting the move operation again.
3. If the omit flag is False, update the $updatetree_Y$ structure to ensure that follow-up operations can execute correctly, as they are based on the information in this structure.

Further details:

1. A $move(source_Y, dest_Y)$ operation is executed with $source_Y = path(n_Y)$, where n_Y is found in $updatetree_Y$ using $n_X.id_{db}$, and $dest_Y = path(n_{v_Y})/n_X.name$, where n_{v_Y} is the node corresponding to $n_X.parent$.
2. The path $p_{source} = path(dbsnapshot_X, i_X)$ and the parts of all sub-paths q which start with $p_{source}/...$ are replaced with path $p_{dest} = path(dbsnapshot_X, n_X.parent.db_{id})/n_X.name$. All other values in the database remain equal.
3. If the omit flag is false, n_Y is moved to be an immediate child of n_{v_Y} in $updatetree_Y$ and is renamed to $n_X.name$.

6.3.1.4 Delete operation

A *delete* operation o is based on one or more file system operations *delete*(i_X, p_X) with *delete* := *deletedir* $\vee$ *deletefile*, which were detected by *compute_ops()* and caused the corresponding node n_X (and child-nodes) to be created in $updatetree_X$, with a *delete* change-event. If the omit flag is set to True, o is also given the corresponding delete node n_Y in $updatetree_Y$. As discussed in section 6.2 delete operations are only generated for the *highest-level* node with a *delete* change-event, not for those who have parent nodes with a *delete* change-event. If n_X is a *directory* node, o is also given the list S of sub-nodes which need to be deleted from the database.

The three execution steps are as follows:

1. If omit-flag is False, delete the file or directory on replica Y, because the object still exists there

2. Remove the entry from the database (and snapshot). If n_X is a directory node, also remove all entries for each node $n \in S$. This avoids that the object(s) are detected again by *compute_ops()* on the next sync iteration
3. Update the *update tree* structures to ensure that follow-up operations can execute correctly, as they are based on the information in these structures

Further details:

1. The delete operation uses $path_Y = path(n_Y)$ where n_Y is found in $updatetree_Y$ using $n_X.id_{db}$. If n_X is a directory, the implementation needs to recursively delete all sub-objects.
2. Let path $p_{source} = path(dbsnapshot_X, i_X)$. Then all entries with path p are removed from the database, where $p = p_{source} \vee \exists r \in \Sigma^+ : p = p_{source}/r$, where $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$.
3. Remove n_X and n_Y from the *update tree* structures.

6.3.2 Breaking cycles in operation sorting

6.3.2.1 Introduction

To reiterate the topic of operation sorting, we extracted order dependency rules between all file system operations in section 4.2.7.1 on page 56. By analyzing the operation preconditions we found a total of *eight* rules. We tested all possible ways how two non-conflicting operations (detected on replica X) can be arranged in an order such that trying to apply them on replica Y fails, because executing the first operation invalidates the precondition of the second one. In section 4.2.7.2 we elaborated which kinds of *cycles* can be built from the ordering rules and presented the `sort_operations()` algorithm in algorithm 2 on page 58. In section 4.2.7.3 we discussed the concept of how to break cycles.

So far, the presented algorithms and discussions focused on the scenario where changes occur only in one replica, being applied to the other, *unchanged* replica. In this section we generalize the scenario to bi-directional synchronization, where operations are detected on both replicas, working with *Syncpal operations* explained in section 6.3.1 instead of computed file system operations. Here we also discuss implementation details of `break_cycle()`.

6.3.2.2 From uni- to bi-directional synchronization

Section 4.2.7.3 on page 59 explains how `break_cycle()` works, in a setting where a set of file system operations detected in one replica are applied to another replica (where no operations are detected). It is straightforward to adapt the `fix_op_before_op()` functions introduced in section 4.2.7.1 (details in appendix A.4 on page 205), the `sort_operations()` algorithm (algorithm 2) and `break_cycle()` to our bi-directional synchronization algorithm introduced in this section, where non-conflicting changes are detected on *both* replicas and operations are not pure file system operations but Syncpal operations. The reconciliation phase provides an unsorted set of operations $O = O_{local} \cup O_{remote} \cup O_p$, which `sort_operations()` turns into a sorted list $\bar{O}$. O_{local} and O_{remote} contain operations affecting *only* the respective replica, while O_p contains *pseudo*-conflicting operations (Create-Create, Delete-Delete, Move-Move (Source), Edit-Edit) which affect *both* replicas. We assume that there is no negative effect of some operation $o_{X_i} \in O_X$ on any $o_{Y_j} \in O_Y$ (and vice versa), i.e., the order of any o_{X_i} relative to o_{Y_j} in $\bar{O}$ is irrelevant. We therefore adapt the comparison logic in the `fix_op_before_op()` functions to compare *all* operation tuples *except* for $\left(o_{X_i}, o_{Y_j}\right)$ and $\left(o_{Y_i}, o_{X_j}\right)$ pairs. Comparing such pairs would be computationally expensive. In section 6.3.2.5 we explain that this assumption is not true in all scenarios and elaborates on how we handle such cases.

As discussed in section 4.2.7.3, every cycle must either contain a *delete* operation o (dependency rules 1 or 5) or a *move* operation o (dependency rules 6 or 2). Our implementation of `break_cycle()` takes the first cycle of Syncpal operations it finds and looks for a matching *delete* operation o in it. If no such o can be found, a matching *move* operation o is found instead.

We then generate a *resolution rename* operation o_r (see next section) and only execute o_r, followed by restarting the synchronization. Let n_X be the node affected by o, then o_r is given n_X's corresponding node n_Y which we find in $updatetree_Y$. o_r is provided with flag $t = db$ if the *omit* flag is set for o (when $o \in O_p$), or $t = both$ otherwise. Flag t is explained below.

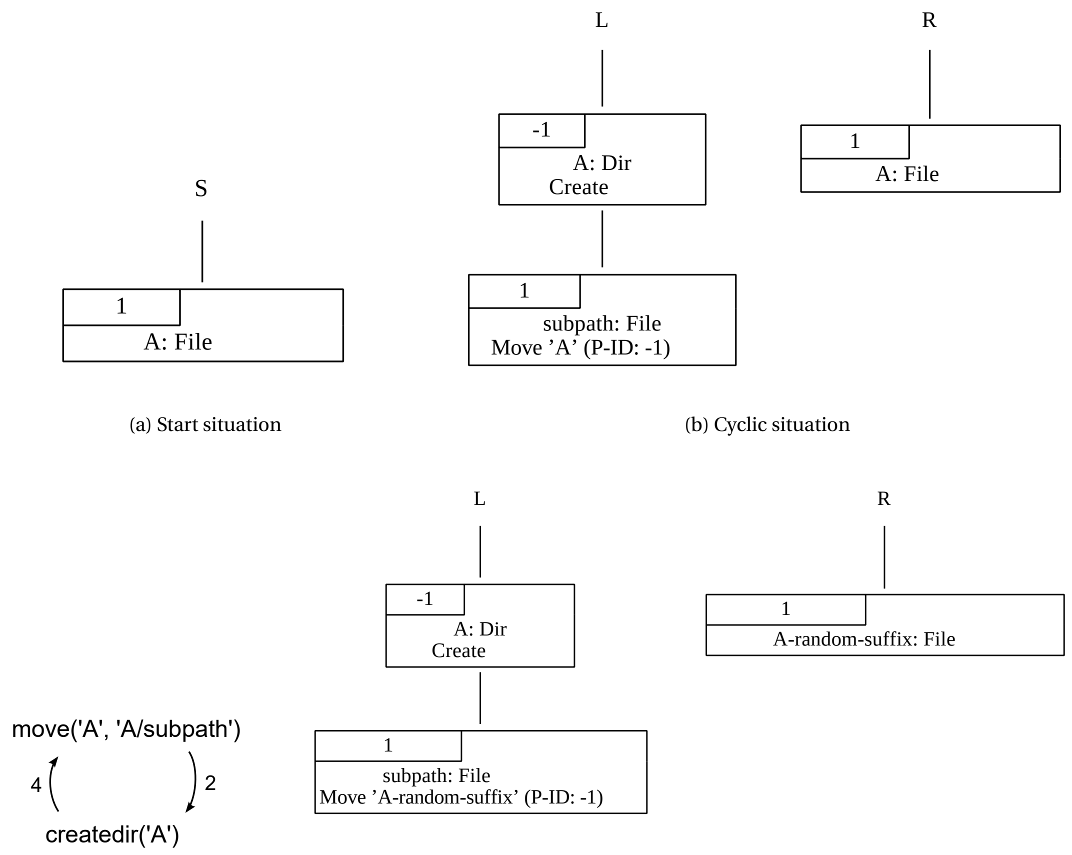
6.3.2.3 Resolution rename operation

A *resolution rename* operation is a new type of Syncpal operation which changes only the *name* of an object by appending a *random* suffix to it. It is provided with a flag t (target) where $t = db$ indicates to rename the object only in the database, whereas $t = both$ means to rename it both physically and in the database. Similar to other operations described in section 6.3.1, the paths on the physical file system are computed using the current state, i.e., a $move(source, dest)$ operation computes $source = path(n_Y)$ and *dest* = *source*+*suffix*.

6.3.2.4 Effect and examples

The key to our approach is that the information of the move/delete operation o is *not* lost when executing o_r, because o_r affects replica Y, while o affects replica X. Sorting (and thus, resolving a cycle) is only done when *no* conflicts are found. After restarting the synchronization, both n_X and n_Y will still have the same change-events they had before executing o_r[3]. Operation o will be detected again, but the sorting algorithm will no longer find the dependency rule 1, 5, 6 or 2 for o that was found before executing o_r. Due to the randomness of the suffix, executing o_r cannot cause *new* order dependencies either, because it's extremely unlikely that the new object name (containing the suffix) would coincide with another existing object. Consequently, the cycle is broken and becomes a chain. Figures 6.2+6.3 provide further examples.

[3]If o is a delete operation, renaming the database entry (and the physical object on replica Y if it still exists, i.e., if the omit flag is False) cannot possibly change the *Delete* change-event in n_X, and won't change the change-events in n_Y either. The same holds if o is a move operation.



(a) Start situation

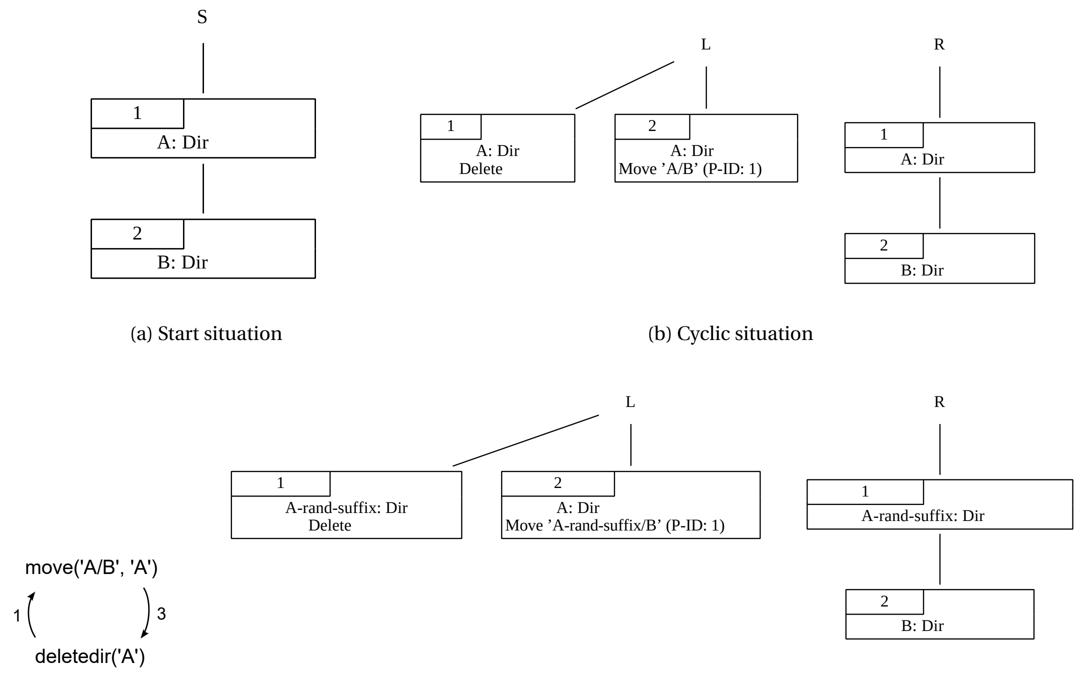
(b) Cyclic situation

(c) Cyclic order dependencies

(d) Broken cycle situation

Figure 6.2: Breaking a cycle, example 1

Subfigure (a) shows the start situation. On the local replica, the operations *move('A',' temp')*, *createdir('A')*, *move('temp', 'A/subpath')* were performed, which are detected as *createdir('A')* and *move('A', 'A/subpath')*, see subfigure (b). Subfigure (c) shows the cyclic order dependencies. Our algorithm breaks the cycle by appending a random suffix to A's name, on the remote replica and in the database, which results in a cycle-free situation shown in subfigure (d), where only order rule 4 holds.



(a) Start situation

(b) Cyclic situation

(c) Cyclic order dependencies

(d) Broken cycle situation

Figure 6.3: Breaking a cycle, example 2

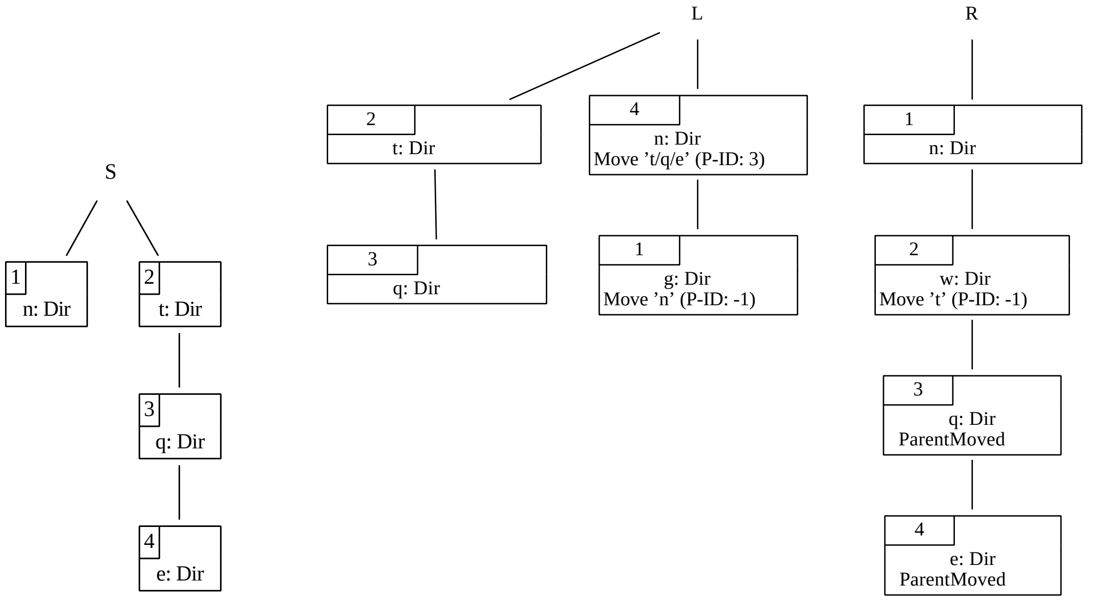
Subfigure (a) shows the start situation. On the local replica, the operations *move('A',' temp')*, *move('temp/B', 'A')*, *deletedir('temp')* were performed, which are detected as *deletedir('A')* and *move('A/B', 'A)*, see subfigure (b). Subfigure (c) shows the cyclic order dependencies. Our algorithm breaks the cycle by appending a random suffix to A's name, on the remote replica and in the database, which results in a cycle-free situation shown in subfigure (d), where only order rule 3 holds.

6.3.2.5 Handling cross-replica operation dependencies

In section 6.3.2.2 we stated the assumption that the order of an operation $o_{X_i} \in O_X$ relative to some operation $o_{Y_j} \in O_Y$ is irrelevant. Dependencies between operations detected in different replicas are typically detected as real *conflicts*, but our algorithm already resolved these conflicts once function `sort_operations()` is called. However, our randomized testing procedure presented in section 8.1.3 on page 146 found several counter examples where the precondition $\neg ancestor(i, v)$ of operation $move_X(i_X, u_X, v_X, name_X)$ (that is, an object cannot be moved to a target below itself) is violated by another $move_Y(i_Y, u_Y, v_Y, name_Y)$ operation. A requirement is that each move operation targets a different object, and both objects are *directories*. An example is shown in figure 6.4, where subfigures (a+b) demonstrate the scenario and subfigure (c) illustrates the detected order dependencies, found for each replica. The final result of `sort_operations()` is

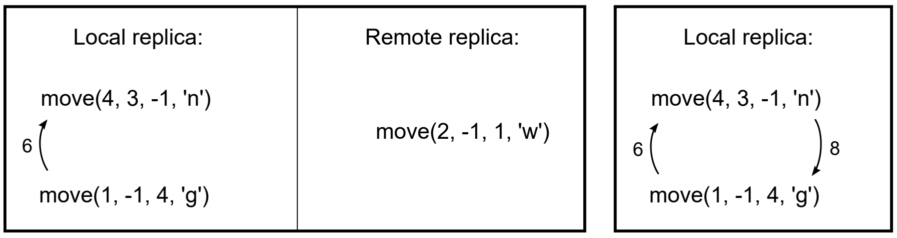
$$\bar{O} = [move_L(1, -1, 4, 'g'), move_L(4, 3, -1, 'n'), move_R(2, -1, 1, 'w')]$$

and the propagation of these operations to the other replica will fail, because the precondition of the first move operation is violated. The remote replica does not allow to move the node with $i_{db} = 1$ into



(a) Start situation

(b) Changes by local and remote replica



(c) Incomplete order dependencies

(d) Cyclic order dependencies

Figure 6.4: Example for cross-replica operation dependencies

the node with $i_{db} = 4$. This situation is *not* a *Move-Move (Cycle)* conflict. It can be solved by first applying the move operation detected in the remote replica to the local one (i.e. executing $move_R(2,-1,1,'w')$ locally) followed by restarting the synchronization. In the next iteration `sort_operations()` will find a *cycle* for the the operations with $i_{db} = 1,4$, as illustrated in subfigure (d). After breaking the cycle, the remaining move operations can be successfully propagated.

The underlying reason for the incorrect behavior of `sort_operations()` is that some functions `fix_op_before_op()` called for some operation pair involving operation o_{X_i} require information from the *database* snapshot, which changes after operations from O_Y are executed. In the example shown in figure 6.4 the function `fix_move_before_move_parent_child_flip()` (for order dependency rule 8) presented in appendix A.4 on page 205 fails to detect the necessary database condition[4], because it can only be detected once the move operation for node with $i_{db} = 2$ has executed. We analyzed similar dependencies in other `fix_op_before_op()` functions by hand but found no related issues.

We now present our approach for solving this issue, referred to as *Operation reshuffling*.

This approach assumes that the only side effect that exists between two operations o_{X_i} and o_{Y_j} is the one we found in figure 6.4, affecting `fix_move_before_move_parent_child_flip()`. It assumes that

[4]Variable `is_y_below_x_in_db` would have to be true, but is detected as false.

our analysis of all other `fix_op_before_op()` functions was flawless and that there are not any other side effects we overlooked.

The *Operation reshuffling* approach gets its name because it reshuffles the order of the scheduled operations, lazily. It defers the execution of those impossible move-directory operations to a later synchronization iteration, executing those first which *are* possible. It performs a simple analysis of `sorted_ops` once it is available. Consider the pseudo-code of `sort_operations()` shown in algorithm 2 on page 58. We replace the last line, `return sorted_ops`, with the following code:

```
reshuffled_ops = fix_impossible_first_move_op(sorted_ops)
if reshuffled_ops:
    return reshuffled_ops
return sorted_ops
```

Function `fix_impossible_first_move_op(sorted_ops)` analyzes whether the first operation in `sorted_ops` is an impossible move-directory operation. If yes, it returns a list of operations to execute instead of `sorted_ops` (more details below). If not, it returns nothing. Of course there might be an operation o_{X_i} in `sorted_ops` which is an impossible move-directory operation, but o_{X_i} is not the *first* operation. Determining whether o_{X_i} is (im)possible is not easily possible, unless all operations in `sorted_ops` were simulated first, which is computationally expensive. Instead our approach deliberately ignores this issue. Instead, it restarts synchronization whenever it detects that a move-directory operationo_{X_i} would fail (at the point when it is executed). After restarting the synchronization o_{X_i} will come *first* in the next synchronization iteration, and `fix_impossible_first_move_op()` will handle it. We now explain the details of `fix_impossible_first_move_op()` and relate its steps to the example from figure 6.4:

1. Check whether the first operation o_1 in `sorted_ops` is a move-directory operation. If not, return.
2. Compute the paths `source` and `dest` necessary to execute o_1 on the other replica, say, Y, and check whether o_1 is impossible, which is the case if `dest` starts with `source + '/'`. If o_1 is possible, return.
    - In our example $o_1 = move_L('n','n/w/q/e')$ is found to be impossible. On replica R the node with $i_{db} = 1$ cannot be moved into the one with $i_{db} = 4$, because 4 is below 1.
3. From our above analysis we know that $o_1 = move(i_1, u_1, v_1, n_1)$ is impossible because there must be at least one other move-directory operation o_{Y_j} that affects a node situated *between* the source node i_1 and target node v_1. Traverse $updatetree_Y$, starting from the corresponding destination parent directory node v_1, up to the corresponding source node i_1. Build a list l of all move-directory nodes o_{Y_j} found along the way.
    - In our example, we traverse the *remote* update tree, from destination parent directory node $v_1 = 4$, up to the source node $i_1 = 1$. We find $l = [move_R(2,-1,1,'w')]$.
4. Iterate over all operations in l and determine o_{first} which is the operation in l which comes earliest in `sorted_ops`.
5. Return a list `reshuffled_ops` that is a filtered version of `sorted_ops`. It contains only those operations in `sorted_ops` that affect replica Y (or *both* replicas for operations whose omit-flag is true), up to (including) o_{first}.
    - In our example, `reshuffled_operations` = l. The first two operations from `sorted_ops`, $move_L(1,-1,4,'g'), move_L(4,3,-1,'n')$, are excluded.

Syncpal then executes only the operations in `reshuffled_ops` and restarts the synchronization. In our example, $move_R(2,-1,1,'w')$ is executed, updating the local replica and the database. The next iteration will find and break the cycle as depicted in subfigure 6.4d. The consecutive iteration will achieve convergence.

Our approach is easy to implement and has minimal impact on run-time performance. Exhaustive testing using *randomly* generated operation sequences (see section 8.1.3 on page 146), which also produced the example in figure 6.4, found no scenarios that exhibit any side effects other than the one we just discussed. This strengthens our degree of certainty that the assumption we made at the beginning of this section is correct.

6.3.2.6 Proofs of termination

Theorem 7. *The `break_cycle()` operation does not cause new cycles.*

Proof. by contradiction: To cause new cycles, executing the resolution rename operation o_r in sync iteration si_t, which breaks the cycle (see section 6.3.2.3), would need to cause at least one new cycle in si_{t+1}. In general, each cycle must contain one of the reordering situations 6, 2, 1 or 5. Operation o_r dissolves one of these situations, by renaming some node n_Y to a *unique*, un-used name ω, turning the cycle into a chain. ω is impossible to cause a new situation 6, 2, 1, 5 in si_{t+1}, because that would require that a *move* or *create* operation exists in si_{t+1} where the affected node's name matches ω, which is impossible. □

Theorem 8. *The execution of the operations in `sorted_ops` returned by `sort_operations()` cannot fail.*

Proof. by contradiction: For the execution to fail, it would require that `sorted_ops=` $[...,o_i,...,o_j,...]$ where executing o_i breaks the execution of o_j. Because the `fix_op_before_op()` methods ensure that order dependencies of operations on the same replica (or pseudo-conflicting operations) are dealt with, o_i must have been detected only in replica X and o_j only in replica Y. For example, if some o_{Y_i} is placed before some o_{X_i} in `sorted_ops`, such that the execution of o_{Y_i} invalidates o_{X_i} , then there would need to be some kind of a dependency between o_{Y_i} and o_{X_i}. However, this is not possible, because to have a dependency between o_{Y_i} and o_{X_i} would require a (real or pseudo) *conflict* that involves the corresponding two nodes, or involve the cross-replica move operation depedency discussed in section 6.3.2.5 on page 122. However, real conflicts are already solved at the point where `sort_operations()` is called, `sort_operations()` does consider pseudo conflicts, and the cross-replica move operation depedency has also been handled as discussed above. □

6.3.3 Operation execution challenges

During the execution of Syncpal operations, our implementation needs to take care of several details to allow for a successful execution of all operations. These are explained in the following subsections.

6.3.3.1 Concurrency handling

As outlined at the beginning of this chapter, our algorithm treats file systems as *static*, assuming that the user does not perform any operations during the phases *update detection*, *reconciliation* and *propagation*. In practice, file systems are *dynamic*, and we cannot ask the user to stop all activities during these phases, especially since the synchronizer is active in the *background*. As a result, the generated operations may become outdated, and attempting to execute them could have disastrous effects.

Our solution to minimize the chance that such problems occur is to query the file system for meta-data right before performing an operation, to ensure that the operation is still valid. The returned meta-data is compared to the data available from the snapshots and update trees that were generated during update detection. The specific queries depend on the operation:

- Create file, Move:
  - *ID* and *lastmodified* meta-datum of the source object
  - *ID* of the destination's parent directory
  - Verify that the name is not occupied in the destination parent directory
- Edit (file):
  - *ID* and *lastmodified* meta-datum of the source object, on *both* replicas
- Create directory
  - *ID* of the destination's parent directory
  - Verify that the name is not occupied in the destination parent directory
- Delete file: *ID* and *lastmodified* meta-datum of the target object
- Delete dir: *ID* of the target object

If any of the checks fail, the operation as well as all other operations are aborted and a new synchronization iteration is triggered. Note that even though our *dynamic* synchronizer implementation (see section 7.3) automatically detects user-made changes to the file system in real time, we still perform most of these checks. The reason is that, in practice, several hundred milliseconds may pass before a user-made change is registered by our dynamic implementation, which may already be a too long time period.

6.3.3.2 File system tricks to emulate atomicity

There are two types of operations that may take long and whose effect may be atomic[5] in some file systems but not in others:

1. File transmissions (upload or download to/from the server), due to limited transmission speed,
2. Deleting non-empty directories, because some file system APIs only offers methods for the deletion of files and *empty* dirs, thus all dirs have to be traversed and all their children have to be deleted first (recursively). In some cases it's even impossible to delete some sub-files or dirs due to permission problems, e.g. caused by UNIX permission bits or prohibitive ACLs (Access Control Lists). On Windows systems, a file that is opened by another program (even just for reading) cannot be deleted either.

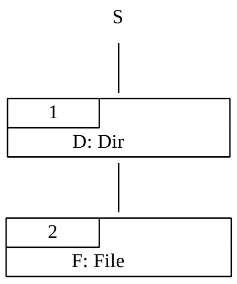
To avoid that the file system, in particular its namespace, is left in an inconsistent state during or after synchronization, or after interrupting an operation, we apply a few tricks to make the effect of these operations to appear to be atomic to the user and the update-detection components.

1. To avoid that the user sees *partially* downloaded files in the synchronized namespace on a local disk, Syncpal downloads files into an invisible, temporary directory (which is on the same disk volume as the file's final destination). Only once the download is complete it is moved to its final destination. Such move operations, as long as they are on the same disk volume, are atomic and take few milliseconds to complete.
2. To avoid that a recursive delete operation of a directory only succeeds partially, we instead *move* the targeted directory to the aforementioned invisible, temporary directory. This move operation succeeds (or fails) atomically and thus it atomically alters the synchronized file system namespace. After the move has completed, the directory and its content can be deleted in parallel to the execution of other operations.

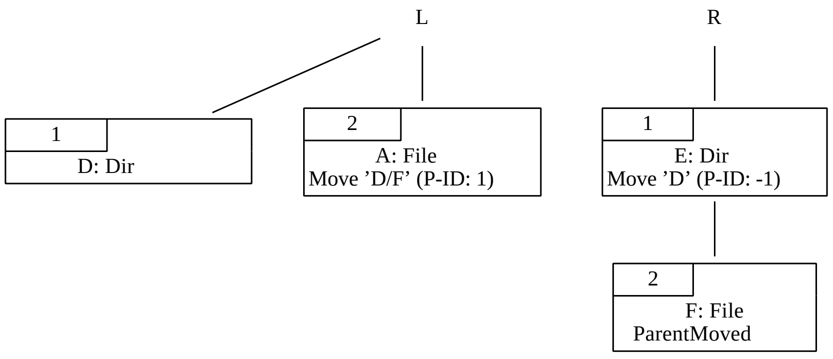
[5]We refer to *atomicity* as known from database systems, see also section 1.3.4 of [EN15].

6.3.3.3 Path computation

Operations need to use the correct, up-to-date paths on the replica to which the operation should be applied. An example is shown in figure 6.5. The *operation* data structure the propagator works with is given the *node* of the respective *update tree* where the operation was detected. We need to compute the path on the *update tree* of the other replica using the `cid()` function from appendix A.5.2.



(a) Start situation



(b) Changes from local and remote replica

Figure 6.5: Path computation during propagation

Given the initial situation shown in subfigure (a), the local replica performed *move(’D/F’, ’A’)* while the remote replica performed *move(’D’, ’E’)*, as shown in subfigure (b). Due to the lexicographical sorting and breadth-first iteration of *update tree* nodes, the operation scheduled first is the move of the *local* replica. To successfully execute it, the corresponding remote source path needs to be changed from *’D/F’* to *’E/F’* (because *’D/F’* no longer exists due to the *move(’D’, ’E’)* operation *already* applied to the remote replica).

6.4 Termination and correctness

It is an important property for our algorithm to be provably correct and to terminate in finite time. In our case, *correctness* is not easily formulated, because of the large amount of possible conflicts (*ten* different types) and the different resolution options we provide for each conflict to make the implementation usable in practice. The combination of different resolution options leads to an exponential explosion of formal notions of correct behavior.

We instead show correctness *indirectly*. We demonstrate that each operation our algorithm generates and executes is both meaningful (because it keeps the *intention* of the user as much as possible) and brings the algorithm one step closer to *termination*. Our algorithm *terminates* if both replicas have *converged*. This is the case once the *local* replica matches the *remote* one. By the law of transitivity this is the case if the *local* and *remote* replica's snapshots each match their corresponding *database* snapshot. We define termination as follows:

Definition 1. Let X, Y be identifiers for two distinct replicas where $snapshot_X$ and $snapshot_Y$ reflect the corresponding current file system state, and $dbsnapshot_X$ and $dbsnapshot_Y$ reflect the file system state known to the synchronizer after the last synchronization (the *database* snapshot). The *compute_ops()* algorithm presented in section 4.2.3 computes *divergences* between these replicas. Our synchronization algorithm $sync(X, Y)$ terminates iff $compute\_ops(dbsnapshot_X, snapshot_X) = compute\_ops(dbsnapshot_Y, snapshot_Y) = \{\}$.

When this situation occurs during update detection, no operations will be generated, and thus there won't be any conflicts, cycles or any non-conflicting operations to propagate.

We provided proofs of termination in different sections of this work. To reiterate, we showed that:

- As long as there are *conflicting* operations, the operations we generate to resolve a conflict cannot cause an infinite loop (i.e., a resolution operation won't generate infinitely many follow-up conflicts). Thus the conflict-detection and resolution phase will terminate, because the number of conflicts is limited by the number of detected operations, which depends on the number of files and directories, which is finite. More details are found in section 5.8 on page 99.
- The part of our algorithm that breaks cycles cannot create infinite loops because the generated breaking operations won't create new cycles. Similarly, the number of cycles is limited by the number of detected operations, which depends on the number files and directories, which is finite. See theorem 7 in section 6.3.2.6.
- An iteration of our algorithm that finds neither conflicts nor cycles and has thus sorted the operations must terminate, because the execution of each operation o cannot fail. See theorem 8 in section 6.3.2.6.

By showing that each step of our algorithm does the "correct step" w.r.t. our consistency philosophy, the consequence is that our overall algorithm is *correct*.

6.5 Conclusion

Our Syncpal algorithm integrates the iterative nature of conflict resolution and operation sorting (presented earlier) into an overall algorithm that builds on state-based update detection, efficient reconciliation using update trees and propagation using Syncpal operations. Our operation sorting approach designed for uni-directional synchronization required some adaptations to work in a bi-directional setup. Each of Syncpal's steps are designed to be minimal (i.e., each step executes as few operations as possible to solve a specific part of the problem) and atomic (by avoiding long-lasting transactions). Therefore the user can interrupt the synchronization at any time without causing side effects. While the theoretical drawback of our iterative algorithm is the increased run-time in those scenarios where *many* iterations are required, we found that this is not problematic in practice. Either the synchronization is triggered often, either manually or because the implementation uses hybrid update-detection that triggers on file system changes. In this case one iteration typically suffices, with only 1-2 operations being propagated. Or the synchronization was triggered rarely, in which case the user expects synchronization to take a considerable amount of time anyway.

While our previous discussions are tailored to the file system model $\mathscr{F}$, our approach is generic. It can be adapted to other models comprised of different operations, preconditions and invariants. While a file synchronizer developer needs to repeat finding all conflicting operations and order dependency rules, she can reuse all other parts of Syncpal without further modifications.

Chapter 7

Implementation

A primary goal of this thesis is not only the design but also the implementation of our Syncpal algorithm. In this chapter we present *BSync*[1], a tool geared towards end-users of Windows and macOS, with *production* level quality. It uses a *dynamic* version of Syncpal presented in section 7.3. BSync runs in the background and detects and propagates file system operations in near real-time. It allows users to configure several different folder pairs, residing on heterogeneous storage systems, and have them synchronized in parallel. BSync provides a graphical user interface (GUI) that offers conflict awareness and inspection. To better understand how (and how often) conflicts affect users, BSync collects statistics about synchronization, which is presented in section 8.3 in detail.

We start by presenting the different iterations of development in section 7.1. Section 7.2 then provides an overview of the software architecture and describes its components and their interaction. The core novelty of BSync is *dynamic* synchronization presented in section 7.3, which automatically triggers synchronization based on user activity and handles concurrent user activity during synchronization. In section 7.4 we describe our userbase as well as how BSync is packaged and deployed to users. We conclude with lessons learned in section 7.5, which describes requirements and problems that surfaced only with the extensive help of BSync's users.

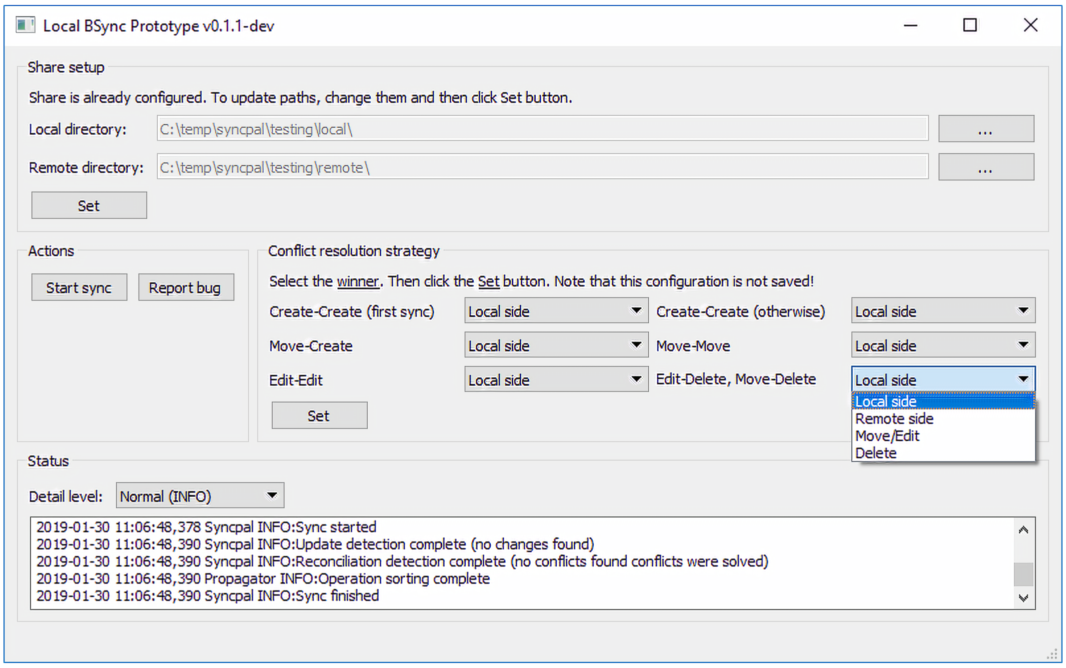
7.1 Iterations of development

7.1.1 Background

The development of BSync and Syncpal did *not* start as an academic PhD thesis. In late 2014 a new work package was created at Fraunhofer FIT for the *EnArgus* project. EnArgus[2] is a platform built on top of BSCW [BHT97] and we were in charge of the work package to create a file synchronizer compatible with BSCW. To avoid building another proprietary solution (tailored to BSCW) we modified the requirements and decided to build a solution with support for heterogeneous file systems. From 2015 to 2016 BSync's development was funded and approached with a software development mindset. In 2017 work on the accompanying PhD thesis began. Since then the design and implementation of Syncpal radically improved, because the academic research mindset and thorough analysis of related work identified and corrected many shortcomings, such as an incomplete analysis of the operation order dependencies.

[1]Abbreviation for Better Sync, as a homage to the application GoodSync, `https://www.goodsync.com/`, retrieved July 21, 2019.

[2]See `https://www.enargus.de/`, retrieved July 21, 2019.


Local BSync Prototype v0.1.1-dev

Share setup

Share is already configured. To update paths, change them and then click Set button.

Local directory: C:\temp\syncpal\testing\local\

Remote directory: C:\temp\syncpal\testing\remote\

Actions

Conflict resolution strategy

Select the winner. Then click the Set button. Note that this configuration is not saved!

Create-Create (first sync): Local side

Create-Create (otherwise): Local side

Move-Create: Local side

Move-Move: Local side

Edit-Edit: Local side

Edit-Delete, Move-Delete: Local side

Status

Detail level: Normal (INFO)

```
2019-01-30 11:06:48,378 Syncpal INFO:Sync started
2019-01-30 11:06:48,390 Syncpal INFO:Update detection complete (no changes found)
2019-01-30 11:06:48,390 Syncpal INFO:Reconciliation detection complete (no conflicts found conflicts were solved)
2019-01-30 11:06:48,390 Propagator INFO:Operation sorting complete
2019-01-30 11:06:48,390 Syncpal INFO:Sync finished
```

Figure 7.1: Local sync development prototype

7.1.2 Development prototypes

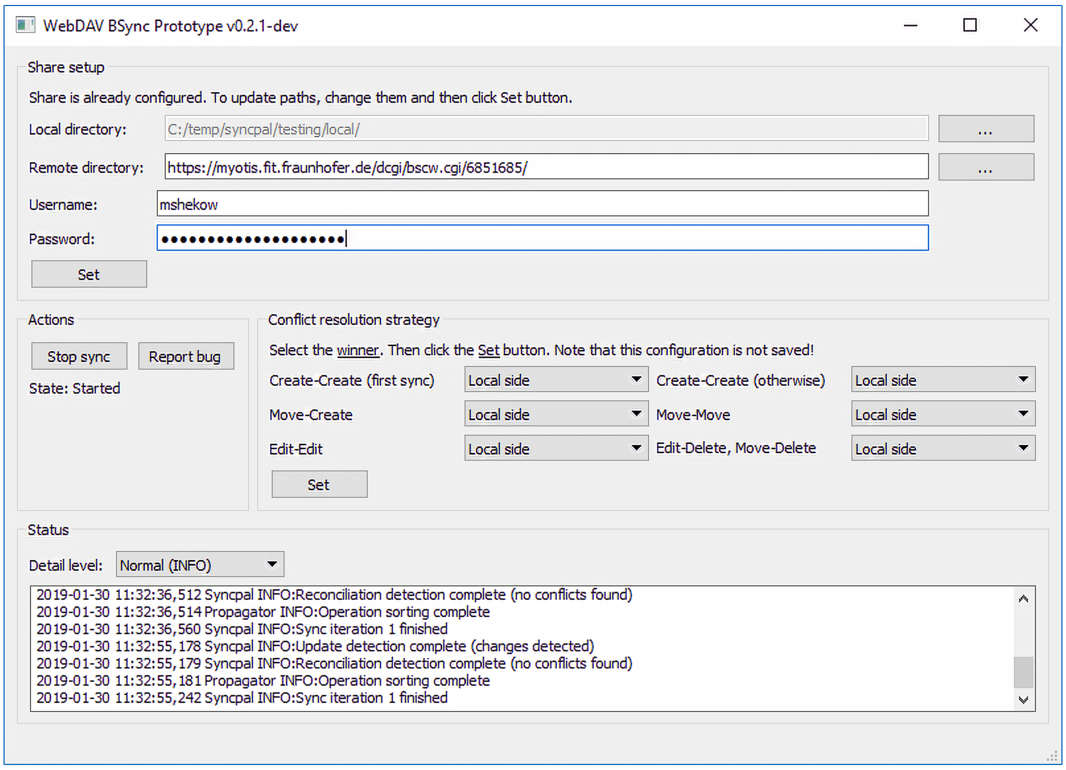
In 2015 and 2016 several internal development prototypes were built which were only distributed among a few close test users. We picked Python 3[3] and Qt5[4] with PyQt5[5] bindings as technology stack, which has not changed since then. Python is a versatile, yet powerful language that allows for rapid prototyping, and Qt5 is used for the GUI and other platform-dependent tasks.

The following prototypes were built:

- v0.1 local synchronization: this very first prototype focused on the synchronization logic itself and only supported the file system APIs of Windows and macOS (which are abstracted by Python's core libraries). It allowed the user to synchronize two local directories, with configurable conflict resolution options. Figure 7.1 shows a screen shot of the application, which consisted of a single window.
- v0.2 WebDAV synchronization: based on version 0.1 we added support for WebDAV-based storage systems, including BSCW. Additionally the synchronization was done in the background, by repeatedly triggering the static Syncpal algorithm every 15 seconds. Figure 7.2 shows the GUI, which is very similar to version 0.1.
- v0.3 Background sync: several alpha versions featuring a GUI comparable to those of industrial synchronizers such as Dropbox, stream-lined for end-users. The user can configure several synchronized folder pairs (referred to as *"shares"* in BSync) which are synchronized in parallel. By default BSync is hidden, and the user accesses it via a tray icon. Figure 7.3 on page 132 has some illustrations. Background synchronization works as in version 0.2.

We also continuously developed automated software tests further presented in section 8.1 along with all versions.

[3]`https://www.python.org/`, retrieved July 21, 2019.
[4]`https://www.qt.io/qt-for-application-development/`, retrieved July 21, 2019.
[5]`https://pypi.org/project/PyQt5/`, retrieved July 21, 2019.


WebDAV BSync Prototype v0.2.1-dev
Share setup
...
Set
Actions
Stop sync
Report bug
Conflict resolution strategy
Status

Share is already configured. To update paths, change them and then click Set button.

Local directory: C:/temp/syncpal/testing/local/

Remote directory: https://myotis.fit.fraunhofer.de/dcgi/bscw.cgi/6851685/

Username: mshekow

Password: ••••••••••••••••••••

State: Started

Select the winner. Then click the Set button. Note that this configuration is not saved!

Create-Create (first sync)	Local side	Create-Create (otherwise)	Local side
Move-Create	Local side	Move-Move	Local side
Edit-Edit	Local side	Edit-Delete, Move-Delete	Local side

Detail level: Normal (INFO)

```
2019-01-30 11:32:36,512 Syncpal INFO:Reconciliation detection complete (no conflicts found)
2019-01-30 11:32:36,514 Propagator INFO:Operation sorting complete
2019-01-30 11:32:36,560 Syncpal INFO:Sync iteration 1 finished
2019-01-30 11:32:55,178 Syncpal INFO:Update detection complete (changes detected)
2019-01-30 11:32:55,179 Syncpal INFO:Reconciliation detection complete (no conflicts found)
2019-01-30 11:32:55,181 Propagator INFO:Operation sorting complete
2019-01-30 11:32:55,242 Syncpal INFO:Sync iteration 1 finished
```

Figure 7.2: WebDAV development prototype

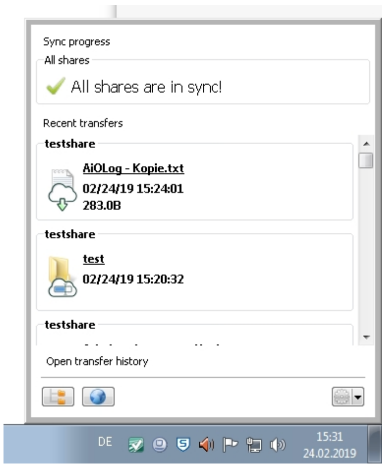
7.1.3 Beta versions

In 2017 and 2018 a total of eight beta versions of BSync were released, distributed to a wider target audience of close to 30 users. The most important changes were the handling of platform inconsistencies, improving the core algorithm in accordance to new findings made in writing this thesis, and the introduction of a truly *dynamic* synchronization approach that is no longer dependent on regularly executing the static Syncpal algorithm. Instead we implemented hybrid, filesystem event-based update detectors for macOS, Windows, BSCW and other file systems, see sections 4.3+7.3. We also added the automatic collection of statistics and further stream-lined the user experience. This included improving the visual appeal of the GUI (see figure 7.4), the option to automatically start BSync after logon, automatic updating (deployment) and automatic retrial of operations that failed temporarily. These beta versions were used exclusively to synchronize with BSCW. Other file system implementations (such as generic WebDAV or ownCloud/Nextcloud) were supported, but were not developed to production-level quality and thus the functionality to select these file system types was not shown in the GUI.

7.2 Software architecture

The simplified software architecture is shown in figure 7.5 . The following list explains the purpose of the individual modules. The red numbered arrows denote the activity flow which is further explained in section 7.3. We use a layered approach where the UI implementation (which could be graphical or text-based) is interchangeable.

- `GUI`: the user interface implemented using the Qt5 framework. It communicates with the application layer via method calls (GUI to application) and Qt signal messages (application to GUI).
- `BSyncCore`: the facade of the application layer. All interaction between UI and application layer take place via this module. It is in charge of managing one or more `Syncpal` instances (one


Sync progress

All shares

All shares are in sync!

Recent transfers

testshare

AiOLog - Kopie.txt
02/24/19 15:24:01
283.0B

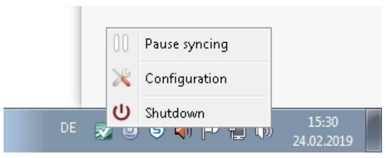
testshare

test
02/24/19 15:20:32

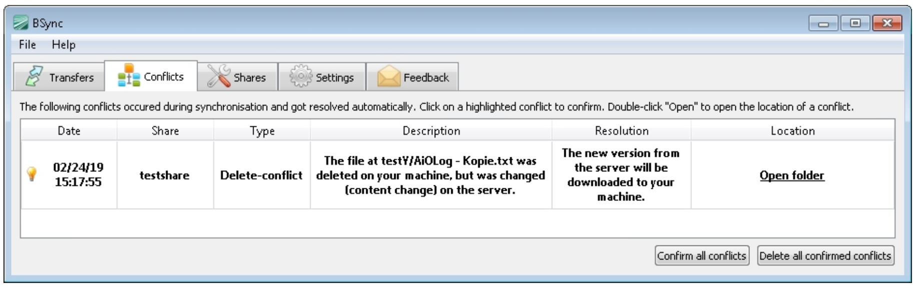
testshare

Open transfer history

(a) Quick view window (single click on tray icon)



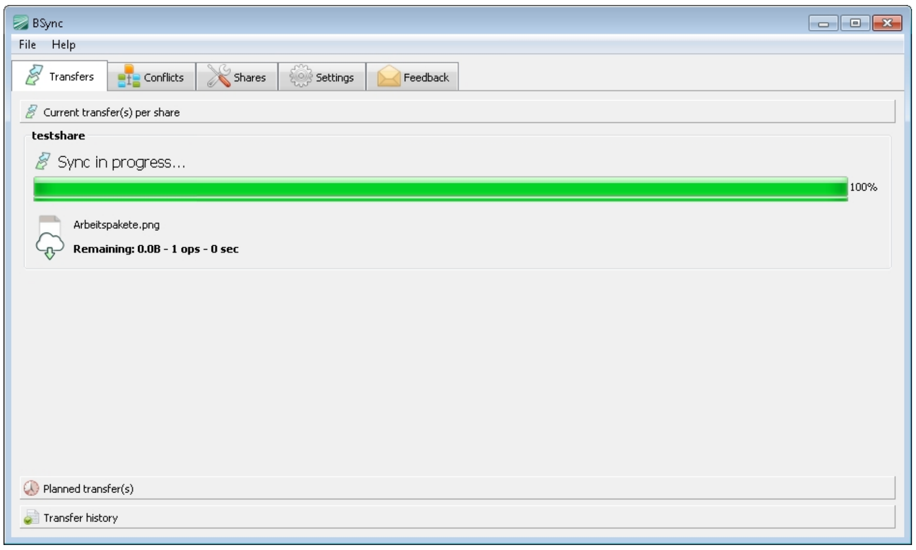
(b) Context menu (right click on tray icon)



The following conflicts occured during synchronisation and got resolved automatically. Click on a highlighted conflict to confirm. Double-click "Open" to open the location of a conflict.

Date	Share	Type	Description	Resolution	Location
02/24/19 15:17:55	testshare	Delete-conflict	The file at testY/AiOLog - Kopie.txt was deleted on your machine, but was changed (content change) on the server.	The new version from the server will be downloaded to your machine.	Open folder

(c) Conflict inspection window



Current transfer(s) per share

testshare

Sync in progress...

100%

Arbeitspakete.png

Remaining: 0.0B - 1 ops - 0 sec

Planned transfer(s)

Transfer history

(d) Synchronization progress inspection

Figure 7.3: Background sync development prototype

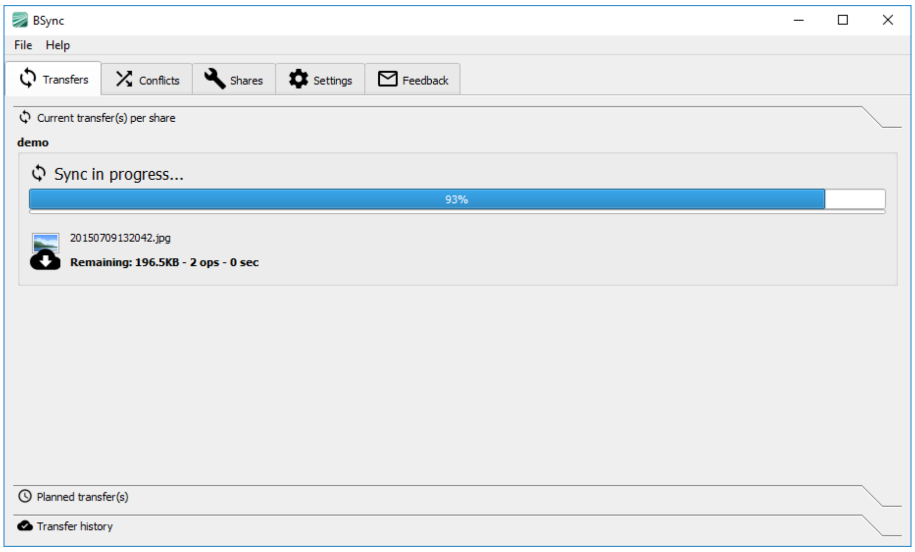


Figure 7.4: BSync beta version

instance per share, i.e., synchronized folder pair). Combines states from different `Syncpal` instances to one common state. Forwards messages from `Syncpal` instances to the GUI.

- `Syncpal`: Is in charge of orchestrating the synchronization process, split into the three phases *update detection*, *reconciliation* and *propagation*. Each `Syncpal` performs its work asynchronously using its own thread.

- `Update Detector`: its task is to discern *expected* from *unexpected* file system events delivered by the `File System Observer` sub-module, to deliver *unexpected* events to the `Syncpal` module, and to generate an *update tree* from the most recent file system state upon request by `Syncpal`. There are *two* `Update Detector` instances per `Syncpal`, one for the remote, one for the local replica.

- `File System Observer`: regularly delivers up-to-date snapshots and file system events of the designated replica to the `Update Detector`. The implementation subscribes to the proprietary event stream and performs *hybrid update detection* where possible, as described in section 4.3. If an unsuitable mapping between namespace and objects is found (see section 3.1.1), that is, if a specific object appears more than once, the `File System Observer` raises an error, causing the synchronization to stop. The user then has to remove all but one link to the object.

- `Reconciliator`: Given two update trees, the `Reconciliator` examines them for cross-platform incompatibility issues and conflicts and returns the list of operations to propagate. The work is divided into several sub-modules described below:

  – `Platform Inconsistency Checker`: examines the names of the objects of each tree for issues they would cause when attempting to propagate these objects to the respective other replica. See section 7.2.1 for a detailed module description.

  – `Conflict Finder`, `Conflict Resolver`: as described in section 5.5 these sub-modules find and sort conflicts and generate resolution operations that resolve the first conflict (if any).

  – `Operation Generator`: Generates the unsorted list of operations for non-conflicting and pseudo-conflicting operations.

- `Propagator`: Given the generated list of operations from the `Reconciliator`, the `Propagator` sorts the operation (if necessary) and the propagates one operation at a time, using the `Executor`.

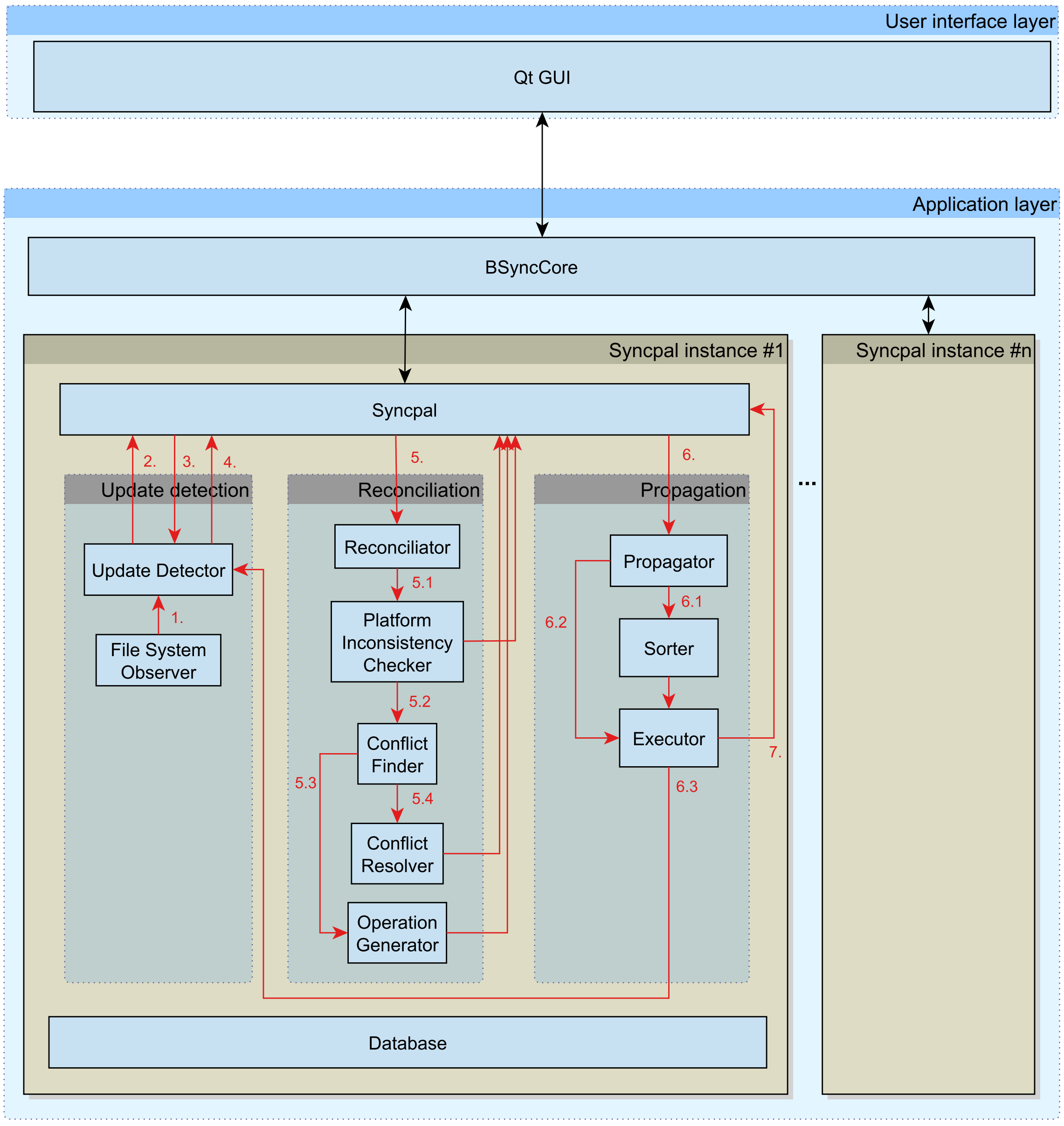


Figure 7.5: Implementation architecture

- `Sorter`: sorts the list of operations as described in section 6.3.2.
- `Executor`: executes each operation and tells the `Update Detector` about the file system event it should expect.
- `Database`: contains the persisted file system state. We implemented the database using the SQLite library.

7.2.1 Platform Inconsistency Checker

In chapter 3 we elaborated on different platform incompatibilities. The `Platform Inconsistency Checker` handles all issues related to the *names* of objects. Our approach is to build a set of rules from the *union* of all limitations of the supported file systems and apply these rules to all replicas. This way, the names of objects can be equal on both replicas and in the database, which simplifies synchronization logic. The rules enforce case- and Unicode normalization *in*sensitivity on sensitive replicas, and ensure that the union of all reserved characters, names and maximum name lengths apply to the names of all objects on all replicas (even on those replicas where no such reservations exist).

Our implementation iterates over the objects of the *update tree* of each replica separately. It examines the names of all objects as well as all sibling nodes. Whenever it finds a name (or sibling names) that violates a rule, the `Platform Inconsistency Checker` generates one or more resolution *rename* operations that change the object's names on the replica. Users are notified about such rename operations, similarly as they are notified about conflicts. The synchronization process is then restarted, as it would be for conflicts. We implemented a two stage process shown in algorithm 4. The split into two stages reduces the complexity of the implementation of each individual stage. The first stage, `normalize_unicode_chars()`, enforces Unicode normalization insensitivity by applying the NFC normalization to object names. It detects clashes and renames one of the objects to enforce insensitivity. There are many intricate details our code deals with which are beyond the scope of this work, such as the HFS+ file system on macOS which is neither Unicode normalization preserving nor correctly implementing Unicode normalization insensitivity.

```
def handle_cross_platform_inconsistencies(update_tree_root_node):
    resolution_operation_list = normalize_unicode_chars(update_tree_root_node)
    if len(resolution_operation_list) > 0:
        return resolution_operation_list
    resolution_operation_list = handle_reserved_chars_and_case_sensitivity(
        update_tree_root_node)
    return resolution_operation_list
```

Algorithmus 4 : Platform Inconsistency Checker pseudo code

Stage two is reached only once Unicode normalization insensitivity has been established. `handle_reserved_chars_and_case_sensitivity()` then checks each object's name for reserved characters or names and generates rename operations that replace, strip or prefix these characters/-names to resolve the issue. It also limits the names of objects to 255 characters and detects clashes.

7.3 Dynamic synchronization

Instead of asking the user to explicitly invoke the synchronization, BSync runs transparently in the background. The development prototypes described in section 7.1.2 simply applied the *static* Syncpal algorithm in regular intervals. This is very inefficient, because sampling a file system's state is computationally expensive. Doing so takes a long time and puts strong load on the system (i.e., does not scale). In general, file system load caused by a user comes in bursts, thus most state samplings are superfluous. If the user did perform changes, they are not picked up until the next sampling, causing a considerable delay until BSync processes those changes.

7.3.1 Requirements

In BSync we require a new, efficient approach that performs synchronization shortly after the detection of user activity, without putting much load on the system. Since most file systems cannot be locked for exclusive access by BSync (and doing so would not be a good idea), our new dynamic algorithm needs to invalidate an ongoing synchronization against concurrent user activity. For instance, let o be a *createfile* operation that uploads file f to the remote replica. If o is scheduled as 5th operation in the queue, by the time o is executed, f might already have been deleted, edited, or moved to a different location by the user. The hybrid update detection approach used by BSync's new approach (see section 4.3) does *not* indicate the origin of the operation (i.e. which program caused it). Thus, some module of BSync must disambiguate those detected file system operations caused by BSync's propagation stage from those caused by the user. That is, the stream of detected operations must be separated into *expected* and *unexpected* ones. BSync then needs an approach to use *unexpected* operations to invalidate the ongoing synchronization.

7.3.2 Implementation

The modules of our implementation are shown in figure 7.5 on page 134. The dynamic version of Syncpal is explained via the activity flow, depicted by the red arrows. We now elaborate on the meaning of each arrow:

- **1.** The `File System Observer` modules report detected operations together with the most recent snapshot to the `Update Detector`, whenever file system activity is detected.
- **2.** The `Update Detector` compares the detected operations against a list of *expected* operations, which it received from the `Propagator` (see step 6.3). Those operations that do not match are, consequently, *unexpected* events. If all received events are *expected*, the activity ends here. Otherwise the *unexpected* events are reported to `Syncpal` which decides how to handle them depending on which stage it is in:
  - If no stage is active (i.e. BSync is *idle*), a new synchronization iteration is started. `Syncpal` switches into *update detection* stage and proceeds with step 3.
  - If some stage is active, `Syncpal` keeps it running and does *not* abort it. `Syncpal` sets a *restart* flag to True, which remembers to start a new iteration once the ongoing iteration has concluded.
- **3.** `Syncpal` requests the two `Update Detector` instances to build *update trees*. During the generation the `Update Detector` implementation is able to handle concurrently detected operations reported by the `File System Observer`, by internally restarting the tree generation.
- **4.** The generated *update trees* are returned to `Syncpal`. Since BSync is implemented in Python whose threads do not use multiple CPU cores due to the Global Interpreter Lock[6], in step (3) `Syncpal` requests the generation of the *local* and *remote* update tree, one at a time. Once both trees are built, `Syncpal` asks both `Update Detector` instances whether any of the trees are out of date again (e.g. the *local* update tree might have become out of date by the time the *remote* update tree was generated). If so, `Syncpal` goes back to step (3), otherwise it proceeds to the next step.
- **5.** `Syncpal` switches to the *reconciliation* stage and requests that the `Reconciliator` builds a list of operations from the two *update trees*. The reconciliation process is divided into several steps and submodules, each with a specific task:
  - **5.1** The `Platform Inconsistency Checker` is invoked first. It checks each *update tree* for problems with object names. If any are found, *reconciliation* ends and the *rename* operations that resolve these problems are provided to `Syncpal`. Otherwise we continue with step 5.2.

[6]`https://wiki.python.org/moin/GlobalInterpreterLock`, retrieved July 21, 2019.

  - **5.2** The `Conflict Finder` examines both trees for real and pseudo conflicts. If no real conflicts are found, continue with step 5.3. Otherwise the found conflicts are given to the `Conflict Resolver`, see step 5.4.
  - **5.3** The `Operation Generator` builds the list of non-conflicting operations and returns them to `Syncpal`.
  - **5.4** The `Conflict Resolver` sorts the conflicts found in step 5.2, and returns a list of resolution operations that resolve the *first* conflict to `Syncpal`.

- **6.** `Syncpal` switches to *propagation* stage and provides the operations to the `Propagator`. If the operations consist only of non-conflicting operations, the `Propagator` gives the operations to the operation `Sorter` (step 6.1). Otherwise the `Propagator` proceeds with executing them, by handing them to the `Executor` (step 6.2).
  - **6.1** The `Sorter` sorts operations and finds cycles, as described in section 6.3.2. It either provides the sorted list of operations (if no cycle was found), or the resolution rename operation that breaks a cycle, to the `Executor`.
  - **6.2** The `Executor` executes one operation o at a time. It provides each o with the current list of *unexpected* detected operations it retrieves from the `Update Detector` modules. Each operation has a tailored implementation that compares every unexpected event against the update tree node(s) it is supposed to propagate. If at least one unexpected event is found to *disturb* o, the `Executor` aborts propagation prematurely and continues at step 7.
    * For instance, let o be a *Create* operation for a directory node n_X such that executing o will create the directory on replica Y (see section 6.3.1 on page 117). The following *unexpected* events would be disturbing:
      - *move(P, any)* where *P* is the path of some parent directory of n_X (check *move* operations on both replicas)
      - $move_X$*(path(n_X), any)*, as the object is no longer at its original location
      - $move_Y$*(any, $path_Y$)* to detect a Move-Create conflict
      - *delete(P)* where *P* is the path of some parent directory of n_X (check on both replicas), which also detects Create-ParentDelete conflicts
      - $delete_X$*(path(n_X))*, as the object no longer exists
      - $create_Y(path_Y)$ to detect Create-Create conflicts
      - $edit_X(path(n_X))$, as the object's meta-data stored in n_X has become out of date
      - $edit_Y(path_Y)$ - only for for pseudo Create-Create conflicts of a file

- **7.** The `Executor` reports to `Syncpal` that it finished. It provides a boolean flag b that indicates whether a new synchronization iteration should be triggered, either because a cycle was broken, or because a disturbing unexpected event was detected.
  - If b or the *restart* flag from step 2 is *True*, `Syncpal` starts a new synchronization iteration (go to step 3).
  - Otherwise the synchronization has concluded and `Syncpal` is *idle* again.

As these detailed steps illustrate, our approach simply lets the `Update Detector` modules detect and collect *unexpected* (user-made) operations. Each executing operation needs to request (*pull*) those unexpected events and compare them against its own data.

We built this approach after several other, unsuccessful designs. For instance, we found a push-based approach to be problematic, where each core module (`File System Observer`, `Update Detector`, `Reconciliator`, `Propagator`) is implemented as asynchronous module with its own thread. Information was *pushed* from module to module, i.e., an `Update Detector` would asynchronously generate an *update tree* and then push it to the `Reconciliator`, the `Reconciliator` would asynchronously build a list of operations and push them to the `Propagator`, etc. However, such a design suffers from many problems, such as high code complexity, difficult-to-catch race conditions or dead-locks, and large delays between detecting unexpected events and aborting an ongoing synchronization.

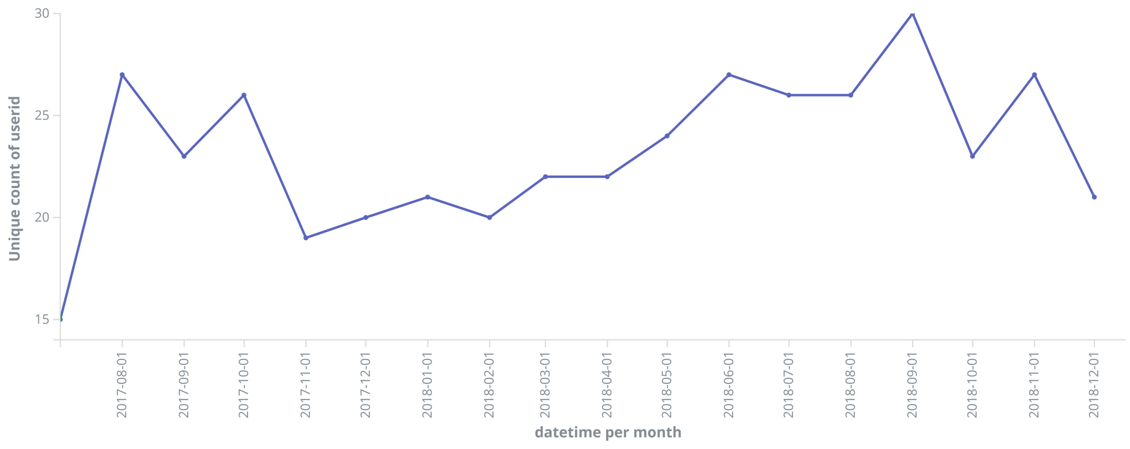


Figure 7.6: BSync's monthly unique users over time

7.4 Software deployment

Between July 2017 and December 2018 close to 30 unique users transmitted usage statistics, see figure 7.6. The majority of users were from our organization, Fraunhofer FIT, as well as associated partners like RWTH Aachen. The author of this thesis has also used BSync throughout this time period to synchronize thesis documents between personal and work computers.

To allow for an easy installation of BSync we developed a one-click build tool that produces an installation program for Windows and macOS. The build tool performs the following fully automated steps:

1. *Freezing* of the Python code, using pyqtdeploy[7], which compiles Python source code into byte code and bundles them in an executable binary (.exe on Windows) with a C++ compiler. This binary also bundles a Python interpreter, such that users are not required to install Python itself,
2. Collection of dependencies, such as dynamically linked libraries like Qt, SQLite or (py)curl, which need to be shipped with the binary for it to work,
3. Building of an installation program, using the WiX toolset[8] on Windows, and DropDMG[9] on macOS. See figure 7.7 for an illustration of the installation program.

The resulting installation programs allow end-users to install or update an existing BSync installation with ease.

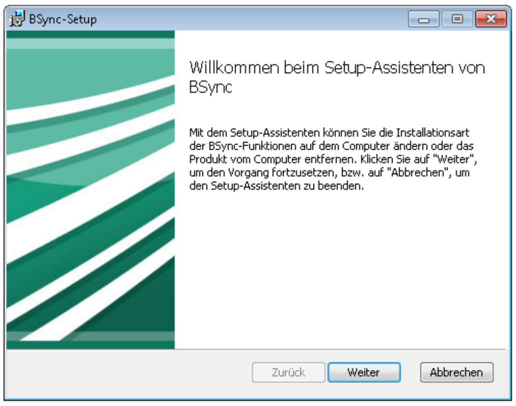
7.5 Lessons learned

Over the years of developing BSync we acquired numerous requirements and insights from our users, presented below.

7.5.1 Partial synchronization

The core idea of BSync has been to synchronize *all* divergences of two file system replicas, in both directions. However, users presented two use-cases where only a subset of objects should be synchronized:

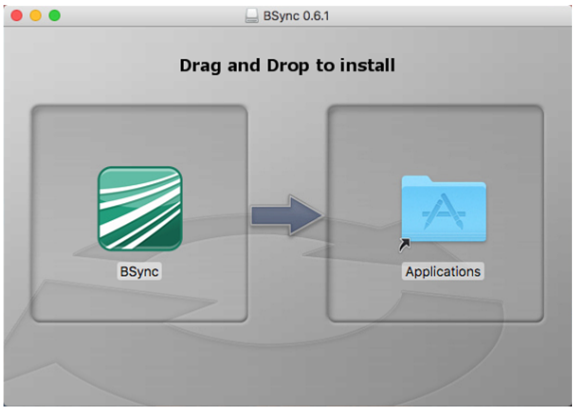
[7] `https://pypi.org/project/pyqtdeploy/`, retrieved July 21, 2019.
[8] `http://wixtoolset.org/`, retrieved July 21, 2019.
[9] `https://c-command.com/dropdmg/`, retrieved July 21, 2019.



Willkommen beim Setup-Assistenten von BSync

Mit dem Setup-Assistenten können Sie die Installationsart der BSync-Funktionen auf dem Computer ändern oder das Produkt vom Computer entfernen. Klicken Sie auf "Weiter", um den Vorgang fortzusetzen, bzw. auf "Abbrechen", um den Setup-Assistenten zu beenden.

(a) Windows



(b) macOS

Figure 7.7: BSync installation programs

1. Exclusion of specific files or directories: a user may want to avoid that a specific *local* sub-directory is uploaded to the server (and thus shared with other users). Conversely, the user may want to exclude a specific *remote* sub-directory from being downloaded, e.g. because she doesn't need it, or because it's child objects would consume too much local disk space.

2. *Uni*-directional synchronization: file systems like BSCW may limit the permissions a user has for specific directories. These permissions themselves are not present on the corresponding other replica, such as the local disk. When the user modifies objects locally, BSync needs to detect that these operations cannot be propagated to the other replica (due to a permanent lack of permission) and should not even try to do so.

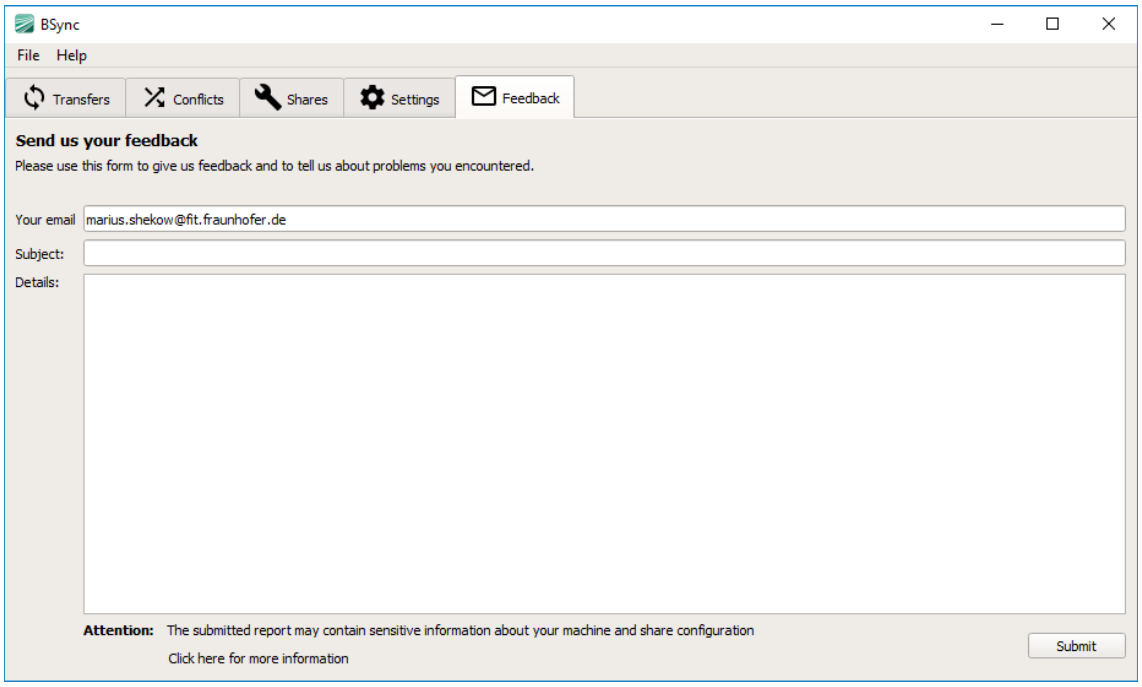
Addressing these use-cases is left as future work, see section 9.3.

7.5.2 Handling of temporary objects

The majority of files BSync propagates are managed by other third-party applications which store their state in a set of files. While these applications are in use, they create temporary objects, such as temporary files containing intermediate results, automatic backups or *lock* files. For instance, Microsoft Word creates a lock file named "~$filename.docx" when the user opens "filename.docx", which is automatically deleted once the user closes the document again. Because synchronizing these files is not helpful and consumes unnecessary bandwidth, earlier beta versions of BSync used a *hard-coded* filter that discards such objects on the lowest level, during update detection. Newer BSync versions allow expert users to add their own filter patterns, to accommodate other applications.

7.5.3 Non-deterministic file system behavior

We have observed different kinds of erratic behavior in the implementations of file systems where a file system operation (that should execute successfully) was unexpectedly refused, with some kind of permission error. These kinds of errors were hard or impossible to synthesize in automated tests. The issue is sometimes *temporary*, where failures only occur due to high concurrent load. For example, Windows might refuse to delete a directory because it was supposedly not empty, even though BSync previously deleted all its children and verified that the directory was empty. The issue may also be *permanently*, e.g. if a Antivirus program, which interferes with file system operations on a kernel level, blocks access to a (supposedly) infected file, which inhibits BSync from reading or uploading it.



Send us your feedback

Please use this form to give us feedback and to tell us about problems you encountered.

Your email	marius.shekow@fit.fraunhofer.de
Subject:	
Details:	

Attention: The submitted report may contain sensitive information about your machine and share configuration

Click here for more information

Figure 7.8: BSync user feedback form

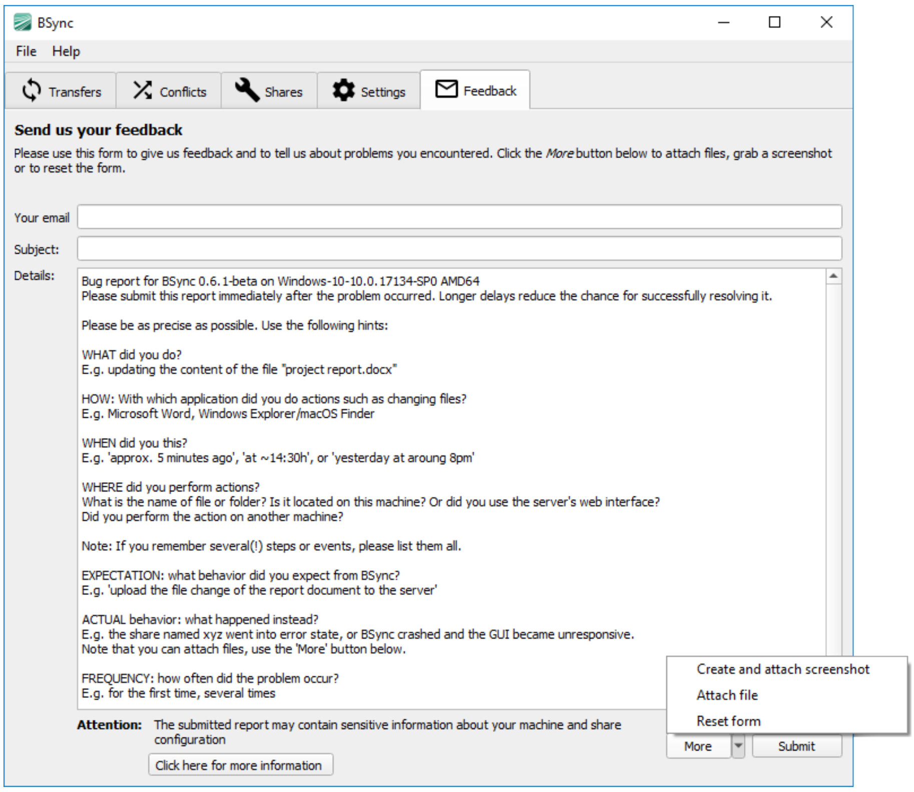
7.5.4 Bug triage

When deploying software to users it is important to detect and resolve bugs that do not occur during development. Such defects occur e.g. due to a different usage pattern of the software or due to a different environment, such as system configuration. In BSync we included a feedback form shown in figure 7.8 that allows users to report errors (or suggest new features). The form's user interface is similar to authoring an email. Users can provide a summary, a longer description, and BSync automatically attaches the most recent replica states, the database state and a log file to the report. The log file contains extensive traces of BSync's modules.

Even with this feedback mechanism in place, we still found bug triage to be very challenging. Data synchronization is a state-based process that involves the state of several systems (file system replicas in this case). Bugs can occur at different places (file system and its observation, update detection, reconciliation, propagation, etc.) and in some instances BSync's implementation is not even to blame. For instance, if the server's file system lists files that are supposedly in a specific directory, but returns "file not found" errors when BSync attempts to download them, the user blames BSync first. The main challenge for a synchronizer developer is to answer the following questions:

1. What did truly happen? E.g. what operations did the user perform, or what was the remote replica's state?
2. What did the synchronizer think was happening?
3. What did the user expect to happen? What should the synchronizer have done?

Logs and state files (attached to the feedback form) only cover question 2, as well as the synchronizer's behavior (including error messages). However, logs do not cover errors in the user interface, which require screenshots to understand the issue. Questions 1 and 3 need to be answered by the user. An analysis of the submitted reports has provided the following two key insights:



Send us your feedback

Please use this form to give us feedback and to tell us about problems you encountered. Click the *More* button below to attach files, grab a screenshot or to reset the form.

Your email

Subject:

Details:

```
Bug report for BSync 0.6.1-beta on Windows-10-10.0.17134-SP0 AMD64
Please submit this report immediately after the problem occurred. Longer delays reduce the chance for successfully resolving it.

Please be as precise as possible. Use the following hints:

WHAT did you do?
E.g. updating the content of the file "project report.docx"

HOW: With which application did you do actions such as changing files?
E.g. Microsoft Word, Windows Explorer/macOS Finder

WHEN did you this?
E.g. 'approx. 5 minutes ago', 'at ~14:30h', or 'yesterday at aroung 8pm'

WHERE did you perform actions?
What is the name of file or folder? Is it located on this machine? Or did you use the server's web interface?
Did you perform the action on another machine?

Note: If you remember several(!) steps or events, please list them all.

EXPECTATION: what behavior did you expect from BSync?
E.g. 'upload the file change of the report document to the server'

ACTUAL behavior: what happened instead?
E.g. the share named xyz went into error state, or BSync crashed and the GUI became unresponsive.
Note that you can attach files, use the 'More' button below.

FREQUENCY: how often did the problem occur?
E.g. for the first time, several times
```

Attention: The submitted report may contain sensitive information about your machine and share configuration

Figure 7.9: BSync user feedback form, revisited

- Without guidance users have a hard time formulating informative reports. Many reports contained barely any information beyond statements such as “The sync doesn’t work” or “The application is just hanging...”.
- Timing is important: the larger the time period between error and extraction of information from the user, the less likely users still remember their last steps (to answer question 1).

Even when interviewing users following a bug report *a few minutes* after it was submitted, users were already fuzzy about the file system operations they performed. Answering question 1 is further complicated when *several* BSync users are involved, whose operations are distributed via the server.

To improve BSync’s quality we used two approaches. First, we provided guidance and improved the feedback form, see figure 7.9. Users are now presented with a number of questions (with sample answers) as guidelines for what information to provide in a report. Users can also create screenshots (for UI problems) and attach arbitrary files. However, this approach is no silver bullet. It is unrealistic to expect that users remember their actions. Our second approach addresses this via in-house testing. Before releasing a new version of BSync with major changes we perform a one-hour test session with multiple users and developers, all simultaneously located in the same room. Users were instructed to work with BSync and their third-party applications as usual, but keep their latest actions in mind, or even note them down. Sometimes users were also asked to design and execute collaborative scenarios with a high degree of concurrency and system load. Whenever problems occur, users report them immediately with the feedback form and the developer additionally examines the problem personally, or assists with filling the report.

7.5.5 Platform incompatibilities

We discovered the majority of platform incompatibility issues early on in section 3.1, via manual testing and examination of the official documentation of file system APIs. Here we present how these issues manifest in practice on the Fraunhofer FIT BSCW server which covers over 20 years of user activity. While recent BSCW versions perform similar reserved object name checks as Windows does, for compatibility reasons, earlier BSCW versions did not. Consequently, a number of work spaces contained legacy objects with names that contain such reserved patterns, e.g.:

- A directory named “Interview, etc.” would require to strip the trailing dot.
- When BSCW auto-generates objects, such as archived emails, the names contain quotation marks which BSync needs to remove.
- Objects with names like “A/B Test 1.8.2001 14:30 Uhr” contain multiple reserved characters, such as forward slash or colons often used in time stamps.
- Some file systems APIs such as the one of macOS allow object names to contain control chars (ASCII range 1-31) not supported by other file systems.

We also investigated whether the varying support of *namespace to object mapping* causes issues in practice. In the largest work spaces we examined we found that only a handful of BSCW objects were linked into more than one parent directory, which is rejected by Syncpal. Manual removal of all but one link could be achieved within a short amount of time to resolve the issue in practice.

7.6 Conclusion

Building a near real-time synchronizer is difficult. In contrast to the *static* variant of Syncpal, we need to detect and handle concurrent operations. This requires an appropriate software architecture with an efficient, yet simple data flow between components. Over time, we tested and implemented various approaches. We found that purely push-based ones do not perform well, which push data from *update detection* to *reconciliation* to *propagation*. It has proven difficult to manage the high, multi-threaded code complexity, race conditions and latency. Our final approach is much simpler. As figure 7.5 shows, the three stages are no longer arranged as a strict chain. We only push file system operations into the *update detection* component, which identifies and caches unexpected user-made operations. These are then *pulled* just in time, by the propagation component. The dynamic version of Syncpal is a single-threaded component that coordinates the overall synchronization process. While this limited use of multi-threading slows down some aspects of synchronization, it reduces the overall CPU usage to a level that users accept and expect of a *background* process. Another side effect of near real-time synchronization is the necessity to translate conflict detection logic. As conflicts could arise during an ongoing propagation, it would be inefficient to run the reconciliation process (including conflict detection) whenever a new, unexpected operation is detected. We instead built simple, purely operation-based heuristics that evaluate whether the history of all detected, unexpected operations disturbs the currently scheduled operation. While this approach requires additional implementation effort, it improves the synchronization efficiency considerably.

The longitudinal evaluation of the implementation by the BSync user group yielded numerous user requirements and a deep understanding of the day-to-day use of our file synchronizer. Many of these learnings have been adopted but also provide insights for future work, most notably the ability to synchronize only *parts* of a replica.

Chapter 8

Evaluation

In this chapter we perform a technical evaluation of Syncpal and its implementation. We first focus on *correctness* of *our* implementation by building hand-crafted as well as automatically generated tests in section 8.1. We then examine the *correctness* of *other* synchronizers in section 8.2, where we develop and apply a test framework to four industrial-grade file synchronizers. Our hypothesis is that their convergence behavior varies strongly, because we found strong divergences in academics works as well, as demonstrated in section 5.1. To understand how our implementation is used in practice we collected statistics which we analyze in section 8.3. We examine the complexity in terms of computation time and memory use in section 8.4 and conclude in section 8.5.

8.1 Automated testing to verify correctness

The theoretical proofs presented in previous chapters are a necessary, but not sufficient, condition for the correctness of the *implementation* of Syncpal. To complement the proofs we created three types of automated tests in the form of black-box tests [Nid12] where implementation details of our algorithm are considered to be unknown. We apply these tests to our implementation of the Syncpal algorithm, BSync, which we presented in chapter 7. They cover aspects not covered by the theoretical proofs, e.g. platform-specifics, and verify that the translation of our algorithm (from English language or pseudo-code to Python code) is free of errors.

An inherent issue in testing is that the problem's state space (file system operation sequences) is infinite, but test execution is required to be a finite process[1]. Consequently, tests cannot verify correctness of an algorithm the same way proofs by contradiction or induction can, which are not example-based. To reduce the chance for errors to a minimum we apply the combination of both approaches, formal proofs and explorative black-box testing. During our development process those tests also turned out to be very valuable because they discovered regression errors introduced when changing parts of the code.

We apply the concept of *falsifiability* [Pop02; HO48] to the statement "the synchronization result of our BSync implementation is correct for all existing file system operation sequences executed concurrently on two replicas", by building tests that demonstrate *counter examples* of the following forms:

- Synchronization does not finish in a certain time period (caught in an infinite loop), or crashes with an error.
- Synchronization finishes without error, but the two replica states are not equal.
- If a test *oracle* (=expected result) is available, the two equal replica states (determined after synchronization finished without errors) do not match the oracle. Such an oracle can either be crafted by hand or can be inferred if an arbitrary list of operations is only applied to replica X by the test (i.e., the test does not manipulate replica Y). In that case, we can automatically build a test oracle which expects that the synchronizer applies an equivalent set of operations to replica Y.

[1] In the release process of a software the test execution is typically expected to finish within a few hours.

The steps performed by each test are as follows (where steps 2, 4 and 8 only apply to tests for which a test oracle is available):

1. Establish a *start scenario*: create a set of files (with random content) and directories on one replica and synchronize them the other (empty) one, s.t. both replicas and the database state match.
2. Create two expected states e_{local}, e_{remote}, by taking a snapshot of each replica, which still resemble the start scenario.
3. Execute a list of file system operations $\bar{O}_{local}, \bar{O}_{remote}$ on the physical local and remote replica.
4. Execute a list of simulated file system operations by manipulating e_{local}, e_{remote} to reflect the expected final replica states after synchronization has completed.
5. Start the BSync implementation, wait for it to complete within a limited time.
6. Take final state snapshots f_{local}, f_{remote} of the local and remote replica.
7. Verify that $f_{local} = f_{remote}$.
8. Verify that $e_{local} = f_{local}$ and $e_{remote} = f_{remote}$.

When a test fails, either the algorithm or its implementation is the cause. To investigate, our test implementation ensures that sufficient information is kept to repeat the same scenario.

We now explain the three types of test we built, each one presented in a separate subsection. The first type is a set of *hand-crafted* test cases presented in section 8.1.1, which focus on implementation- and platform-specific aspects as well as conflicts and their correct resolution. Sections 8.1.2+8.1.3 explain *automatically generated* tests, where the first type generates all possible scenarios, each with a limited number of operations, and the second type *randomly* generates a long list of operations.

8.1.1 Hand-crafted tests

We built over 350 tests (with test and tooling code exceeding 13'000 lines of code). Around 150 tests are *white-box* tests, with the following goals:

- Test individual components, such as the database or file system observers,
- Regression tests of libraries used by BSync (including Python's file system abstraction), to verify that the behavior of these libraries is stable when updating the library or using different operating system versions,
- Synchronization logic tests, where the test needs state introspection to detect when a new synchronization is started. These tests generate non-conflicting, concurrent operations, which modify a replica while the synchronization is in progress. They verify that the affected operations are skipped (if necessary), and that BSync starts a consecutive iteration which synchronizes the remaining operations.

The remaining tests are black-box tests where each test manually defines a list of operations to execute on each replica, and a test oracle. The tests are executed on *static* file systems and cover the following goals:

- Verify that BSync applies non-conflicting operations to the other replica without modification. The tests include both simple sanity checks and more involved scenarios with complex operation interleavings. The tests also consider case-insensitivity, operation *sorting* (as well as breaking cycles) and *pseudo* conflicts.

- Verify that BSync handles namespace limitations correctly, such as reserved names or characters. See section 7.2.1 on page 135 for further details.
- Verify the correct detection and resolution of conflicts. Our tests cover both simple and complex scenarios, where a set of files is affected by *multiple* conflicts. We provide test oracles for all conflict resolution options.

When implementing tests we applied Equivalence Class Partitioning (ECP) and Boundary Value Analysis [SLS14; Nid12] to limit the number of tests.

8.1.2 Generated deterministic tests

Building hand-crafted tests does not scale and misses a lot of incorrect behavior of the system under test [Amj04]. We applied a variant of Model checking [Cla08], which was also done by the file synchronizer work [Bjø07]. Model checking is *"an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model"* [BK08]. A common property to check is the *safety* property. In our case, the synchronizer should not do anything undesirable, such as crash because of a bug or because a precondition of a file system operation was violated, or terminate without crashing, producing two divergent replicas. Another common property is the *liveliness* property, that states the synchronizer performs those actions it is supposed to do, such as propagating all non-conflicting operations to the other replica, producing two convergent replicas.

Applying model checking to our file synchronizer requires reducing the problem space, as the space is otherwise infinitely large when tree depth and/or the set of object names is unlimited[2]. We implemented the following reductions, some of which were also used in [Bjø07]:

- We only apply operations to *one* replica. This eliminates the need to check the correct resolution of *conflicts*, which would require another, independent implementation that finds and resolves conflicts. This also allows us to compute the test oracle, as discussed in section 8.1. The synchronization problem space is now defined as follows: for any synchronized *start state*, with replica states $s_X = s_Y$, executing any list of file system operations l that transforms s_X to s'_X ($s_X \neq s'_X$), run the synchronization and verify that the synchronizer changed s_Y to s'_Y such that $s'_Y = s'_X$.
- We limit the *start state* space to consist of b_f files and b_d directories. We find the distinct set of start states by using a recursive solution to the well-known *integer partition* problem [And03].
- We limit the names of objects to single-character strings, drawn from the lower-case ASCII alphabet, a-z.
- We limit the number of operations to up to c_{cf} create file operations, c_{cd} create directory operations, c_m move operations, c_d delete operations and c_e edit operations.

The generation of test scenarios is a recursive process:

- Build a set OTL of operation type lists by computing all permutations of the operation types. For instance, if $c_{cd} = 1, c_m = 2$ and all other c variables are 0, then
$OTL = permutate([createdir, move, move])$
$= \{[createdir, move, move], [move, createdir, move], [move, move, createdir]\}$.

[2]In practice there are limitations, such as maximum path length and the limited alphabet of names, e.g. based on the Unicode alphabet. However, the space of all file system states is still extremely large, and the problem space of every possible transition between any two states is exponentially larger.

- For every start state s, iterate over every operation type list $otl \in OTL$, and iterate over every operation type $ot \in otl$. Generate all possible operations o for the given operation type ot. For instance, if $ot = delete$, iterate over every possible object in the file system (except for the root directory) and instantiate a *delete* operation o for that object. This requires keeping a shadow copy of the file system state in memory for every possible operation sequence. Add o to operation list l.
- Whenever an operation o was applied, persist a new test scenario to disk.

Because this simple approach produced many similar test scenarios, further reductions were necessary:

- For any two *equal* states s'_{X_1} and s'_{X_2} that result from applying two *different* operation list l_1, l_2 to s_X, discard one of the lists. For instance, if s_X consists of a single file named *'a'* and $l_1 = [move('a','b')], l_2 = [move('a','c'), move('c','b')]$, then we can discard either l_1 or l_2.
- For any two *distinct* states s'_{X_1}, s'_{X_2} whose sets of names are different but would be coincidental if every new, unused name (used by any of the operations) were replaced by a universal variable name, discard one of the states (and the corresponding list). For instance, if s_X consists of a single file named *'a'* and s'_{X_1} is based on $l_1 = [move('a','b']$ and s'_{X_2} is based on $l_2 = [move('a','c')]$, then both lists have in common that object *'a'* was renamed to a previously unused name and we can discard either l_1 or l_2.

For the implementation of our model checker we applied additional implementation-specific techniques to speed up test generation and execution:

- We used parallelization for generating test scenarios, distributing the computation using a divide and conquer approach. Generation was run on over 120 CPU cores (distributed over dozens of machines) for almost one month, followed by merging the results on a single machine (filtering duplicated results) over a period of 5 days.
- We analyzed slow code paths and optimized them by a factor of 4-16 using Cython, which converts Python to C code.
- For generating and executing test scenarios we implemented a simple in-memory file system in C (using Cython) used instead of the real device file system, because real file systems (even RAM disks) turned out to be a performance bottleneck.

Despite these optimizations we had to choose rather small values for b and c defined above, to limit generation and execution time. We chose $b_f = 1, b_d = 3$, focusing on a larger number of *directories* (rather than files) because move operations on directories produce the most challenging scenarios for the *operation sorting* routines. We chose $c_{cf} = 1, c_{cd} = 2, c_m = 3, c_d = 3, c_e = 0$, ignoring *edit* operations because they have no impact on operation ordering. The resulting *5.5 million* test cases (computed in almost one month as described above) helped solving various issues in the Syncpal algorithm and its implementation.

8.1.3 Generated random tests

To overcome the limits resulting from state explosion in the model checking approach described above, we also implemented the generation of *random* test scenarios. As noted in [GHJ07], who also applied randomized testing, the *"difficulty of model checking (and theorem proving) makes randomized testing attractive, when software actually has to be delivered"*.

The randomized testing approach generates a much larger number of operations (e.g. 30 and more) before starting the synchronizer. Each operation type and its parameters is chosen at random, using a *uniform* distribution. Thus, every operation type is equally likely. We implemented two test modes. The first mode applies the randomly generated operations only to replica X. The test oracle expects Syncpal to apply an equivalent set of operations to replica Y. As such, this mode is similar to the generated deterministic tests presented in the previous section. The second test mode applies operations to replicas X and Y separately, deliberately causing conflicts. Here the test oracle can only verify that the state of both replicas has converged, but does not assume that the final states coincide with s'_X or s'_Y, due to possible conflicts.

When generating a large number g of operations, an initially open question is how large g should be, and the effect of g on the tree size. We identified two conditions that must be avoided when running the tests:

1. When g is large, the file system tree size (i.e., the number of objects) might grow beyond a threshold, causing performance bottlenecks. For instance, if the tree grew to, say, 100'000 nodes, performance would degrade, because every aspect of running the test becomes more complex (e.g. executing Syncpal, or generating suitable operations). A work around is to enforce pruning of the tree once its size exceeds a threshold.
2. Trivial situations where $compute\_ops(s_X, s'_X)$ and $compute\_ops(s_Y, s'_Y)$ both yield a set of *delete* and *create* operations, without *move* or *edit* operations. For instance, it is intuitively obvious that applying 1'000 randomly generated operations to a tree of 20 objects is very likely to delete all those 20 objects that originally existed in s_X, and create, move, delete and edit an arbitrary number of objects which then all appear as *new* (*create* operation) in s'_X, due to operation consolidation. However, such scenarios are simple to synchronize and we strive for generating operation sets O_X, O_Y which include *all* types of operations equally, to increase the chance that *all* types of conflicts and operation order dependencies are triggered.

We experimentally collected several statistics, by running tests for uni-directional synchronization (affecting only replica X) as well as bi-directional synchronization, repeating tests thousands of times for each $g \in \{3, 10, 20, 30, 50, 100, 200, 1000\}$. The following statistics were collected:

- Tree size s_t after the synchronization has completed.
- Percentage $s_p = \frac{\#\,of\,create\,operations}{s_t}$, determined at the stage where all conflicts have been resolved. Condition (2) above states that s_p should be as *small* as possible.
- Number of conflicts s_c found in the first reconciliation iteration (only for bi-directional tests), to determine how many conflicts the tests cause, in relation to s_t.

By analyzing histograms we found that s_t does not grow beyond 40 objects for uni-directional or beyond 250 objects for bi-directional synchronization, for any g. Thus, explicitly pruning the tree is not necessary. However, for $g > 30$ the majority of test iterations yield $s_p = 100\%$. The statistic s_c roughly follows a normal distribution with $\mu \approx \frac{g}{3}$, as long as $g \leq 100$. For $g \geq 200$ the value of s_c is typically below 4 again, because the distribution of s_p is very strongly skewed towards 100%.

As a consequence we decided to also randomize g. We chose the intervals $[3, 100]$ for bi-directional tests, and $[3, 30]$ for uni-directional tests, using a uniform distribution. This makes it possible to test all kinds of scenarios. We ran millions of test cases which uncovered many smaller implementation issues. We stopped testing once no anomalies were discovered for several weeks.

8.2 File synchronizer comparative test

The goal of this section is to explore the convergence behavior of industrial-grade file synchronizers. We start with discussing related work in section 8.2.1 and present the selected synchronizers in section 8.2.2. We briefly introduce the test categories we designed in section 8.2.3. The generic test framework and setup is presented in section 8.2.4. Section 8.2.5 then describes the concrete tests and the detailed results. We conclude in section 8.2.6 with a result summary and a discussion.

8.2.1 Related work

When considering papers that compare file synchronizers, the majority of related work is primarily concerned with *performance* benchmarks, not with convergence behavior. In [BDM15] the authors benchmark 11 cloud file synchronizers w.r.t. how client capabilities such as *chunking*, *streaming*, *deduplication* and others affect metrics such as consumed traffic and synchronization completion time. The authors in [CLD16] perform similar benchmarks, but for synchronization apps running on mobile devices with limited computing power and unreliable networks. Further benchmarking works exist, see e.g. [KK17; LZD13; Wan+16]. Although these papers do not deal with convergence, they provide valuable clues for how to build a test framework for synchronizers based on virtual machines.

A few works exist which focus on *convergence*. [Hug+16] apply *property-based* testing using Quviq QuickCheck [Art+06] which *"tests properties—universally quantified boolean formulæ—by generating random values for the quantified variables and checking that the formula evaluates to true"* [Hug+16]. They test the file synchronizers of Dropbox, Google Drive and ownCloud, building a formal specification of the behavior of these synchronizers in the process. To reduce the number of model states the authors limit the operations they randomly generate to read, write and delete operations on a *single* file. Contrary to our work, they omit testing the *move* operation or working on multiple files or directories.

Two academic works presenting novel file synchronization algorithms also compare their convergence behavior with those of industrial synchronizers. The first work, [NS16], formally defines a *path-based* file system with its operations (create, update, delete, rename, but without the *move* operation). The authors formally define their merge algorithm for concurrent operations (conflicting and non-conflicting) as "combined effects", and generate test cases by building all possible *pair-wise* operations (one operation on the local, one on the remote replica). For each pair they iterate over the operation parameters using similar strategies as the *parameters* we introduce in section 8.2.5, e.g. operation types, number of targets and path relationships, which yields a total of 61 distinct tests which they execute in a similar VM-based setup. The second work, [TSR15] also formally defines the file system and merge behavior, but limits the test to five hand-selected scenarios that mostly deal with conflicting operations. Both works test Google Drive, OneDrive and Dropbox and discover some of the same issues as we do in this section. However, the degree of detail of testing the propagation of *conflict-free* operations is limited in both works. This, and the fact that the results are over two years old, justify a repetition.

There are also works that perform black-box testing of other distributed systems with eventual consistency, similar to file synchronizers. For instance, [BMP17] test the convergence behavior of near real-time *text editors* like Google Docs. The authors develop an approach to automatically build test cases, while eliminating redundant ones, which speeds up test execution. However, their approach does not test an accumulation of *multiple* operations, as it always executes a *single* concurrent operation per replica.

8.2.2 Synchronizers under test

There is a tremendous number of industrial-grade synchronizers available on the market[3]. In our selection we focused on implementations that work on macOS and Windows, have a provably wide-spread user base and popularity and offer an English or German user interface and documentation. Testing took place from January to April 2018. We selected the following implementations, with version numbers and remarks shown in table 8.1:

- Dropbox [Dro18], Google *Backup and Sync*[4] [Goo18] and Microsoft OneDrive [Mic19], because they are synchronization clients for commercial cloud services with the three largest documented

[3]Examples include Dropbox, Google Backup and Sync, Microsoft OneDrive, Box, Amazon Drive, OwnCloud/NextCloud, Goodsync, cloudStore, Resilio, Seafile, SugarSync, MagentaCloud, SpiderOakOne, Leitz Cloud, Tonido, TeamDrive, MyDrive, Strato HiDrive, CloudMe, hubiC, pCloud, sync.com, tresorit, iDrive and many more.

[4]The Backup and Sync client synchronizes data with the Google Drive service, see `https://drive.google.com`. The client was formerly known as "Google Drive" [Hac17].

Name	Windows version	macOS version	Remarks
Dropbox	42.4.114	42.4.114	LAN sync was deactivated.
Google Backup and Sync	3.38.7642.3857	3.39.8297.0200	Only the *sync* functionality was used (all *Backup* functions were deactivated).
Microsoft OneDrive	Version 2018, 17.3.7294.0108	Version 2018, 17.3.7131	"On-demand" option was deactivated to ensure that files are fully downloaded.
NextCloud	2.3.3.1	2.3.3 (84)	Synchronization via NextCloud 13.0.1 server.
Unison	2.48.4	2.48.15	Synchronization via Linux server instance also running version 2.48.4.

Table 8.1: Tested synchronizer implementations

number of users[5] we could find.

- ownCloud/NextCloud [Nex18]: an open source industrial-grade client-server software suite used by organizations such as the author's employer (Fraunhofer FIT), to set up an in-house private enterprise cloud. We tested the *NextCloud* client which, at the time of testing, was technically equal to the *ownCloud* client[6].
- Unison [Pie18]: an open-source file synchronizer client for Windows, macOS and Linux from research by the authors of [BP98], which is still under active development!

Apart from Unison we also found the research synchronizers Tra[7] by the authors of [CJ05], and So6[8] from [Mol+03]. We discarded testing the implementations for the following reasons:

- Tra: the implementation was only available for Linux as source package, which has been unmaintained since 2005. Compilation with modern compilers on modern Linux systems was not successful despite considerable efforts. Support from the original author was not available. Cross-compilation for Windows was not possible.
- So6: the available client implementation does not work as described in the paper. The core concept of the command-line based So6 synchronizer is to synchronize multiple *workspace* directories with a central *queue* directory. Similar to Version Control System tools such as Subversion, users modify their own workspace and then issue a *commit* to the queue via So6, or they pull changes from the queue by issuing an *update* command. Another similarity to Subversion is that a *commit* only succeeds if the workplace is already up to date. We identified an issue that So6 does *not* allow concurrent, isolated *move* operations in two different workspaces. When applying move operations directly to the file system via the file manager, So6 detects them as delete + create operations, which is not what we intended. However, So6 provides a *"rename"* subcommand that moves (or renames) a file or directory, making So6 aware of that operation. The flaw of the So6 implementation is that this *rename* subcommand immediately triggers a *commit* command. Since a commit command will fail whenever the workspace is not up to date w.r.t. the queue, concurrent move or rename operations are impossible. For clarification we had a discussion with the authors but were unable to find a solution for this issue.

[5]Dropbox: 500 million (03/2016) [Dro16], Google Backup and Sync: 800 million (01/2016) [Pri17], OneDrive: 115 million *daily users* (08/2017) [Sur17] or 250 million (11/2014) [Gri14].

[6]See `https://github.com/owncloud/client`, retrieved July 21, 2019.

[7]See `https://swtch.com/tra/`, retrieved July 21, 2019.

[8]See `http://dev.libresource.org/home/doc/so6-user-manual.html`, retrieved July 21, 2019.

8.2.3 Test categories

We divided our tests into three categories that target different aspects of synchronization. They were chosen because we put considerable effort into solving the related issues in BSync. The tests determine the amount of effort developers of other synchronizers put into their implementation. We now briefly describe each category. Detailed test descriptions are presented later, in section 8.2.5.

8.2.3.1 Conflict-free operations

These tests aim at exploring how a synchronizer handles both easy and challenging concurrent file system operations which do *not* conflict. The goal is to determine how closely operations performed at one replica are applied to the other replica. In an "easy" test we examine whether a *move* operation applied to a file or directory at one replica is also applied as *move* operation to the other replica. More "challenging" tests perform a series of *move*, *create* and *delete* operations that are challenging to reproduce on the other replica due to operation consolidation. By shutting down the synchronizer we enforce state-based update detection. Some tests only operate on one replica, leaving the other replica untouched. Others modify both replicas concurrently, but avoid conflicts by always targeting *different* objects.

8.2.3.2 Conflict operations

Tests of this category apply operations on both replicas in order to produce conflicts. We investigate how synchronizers resolve conflicts and how they visually present them to the user. As explained in section 5.1 there is a high degree of variance in conflict detection and resolution in academics works, and we assume that industrial synchronizers lack standardization as well.

A core assumption we make in all our comparative tests is that we presume that the tested file synchronizers use a similar file system model $\mathscr{F}$, as defined in section 3.2. Thus, the set of conflicts and propagated operations should match the ones of Syncpal. The tests result presented in section 8.2.5 will show if this assumption does not hold. For instance, a synchronizer that internally models the file system as a set of *paths* without IDs is unable to reliably detect *move* operations, which will be reflected in the test results.

8.2.3.3 Cross-platform issues

As discussed in section 3.1 filesystems on Windows and macOS have many subtle yet important differences. Synchronizers that support both operating systems need to handle undesired side effects and should provide awareness to the user in case files or directories with incompatible characteristics (e.g. *names*) cannot be synchronized. Our tests focus on aspects such as case-insensitivity and reserved names.

8.2.4 Test framework and setup

To run tests introduced in section 8.2.3 we designed a framework and test setup that enables complete reproducibility of the results by eliminating errors resulting from manual test execution. We built a set of fully automated test suites where tests are executed in an environment that is tightly controlled. We note that while test *execution* is fully *automated*, the *analysis* of the results is *manual*. We initially implemented automated result analysis, however, we found that client behaviors varied strongly, making the implementation of test oracles cumbersome and error-prone. By repeating each test three times we minimize the chance of result misinterpretation due to human error.

8.2.4.1 Requirements

Before designing and implementing the framework, we collected two general requirements:

1. Testing new synchronizers must be effortless. These synchronizers typically come with a proprietary *server* file system. Our framework should avoid additional implementation efforts to support

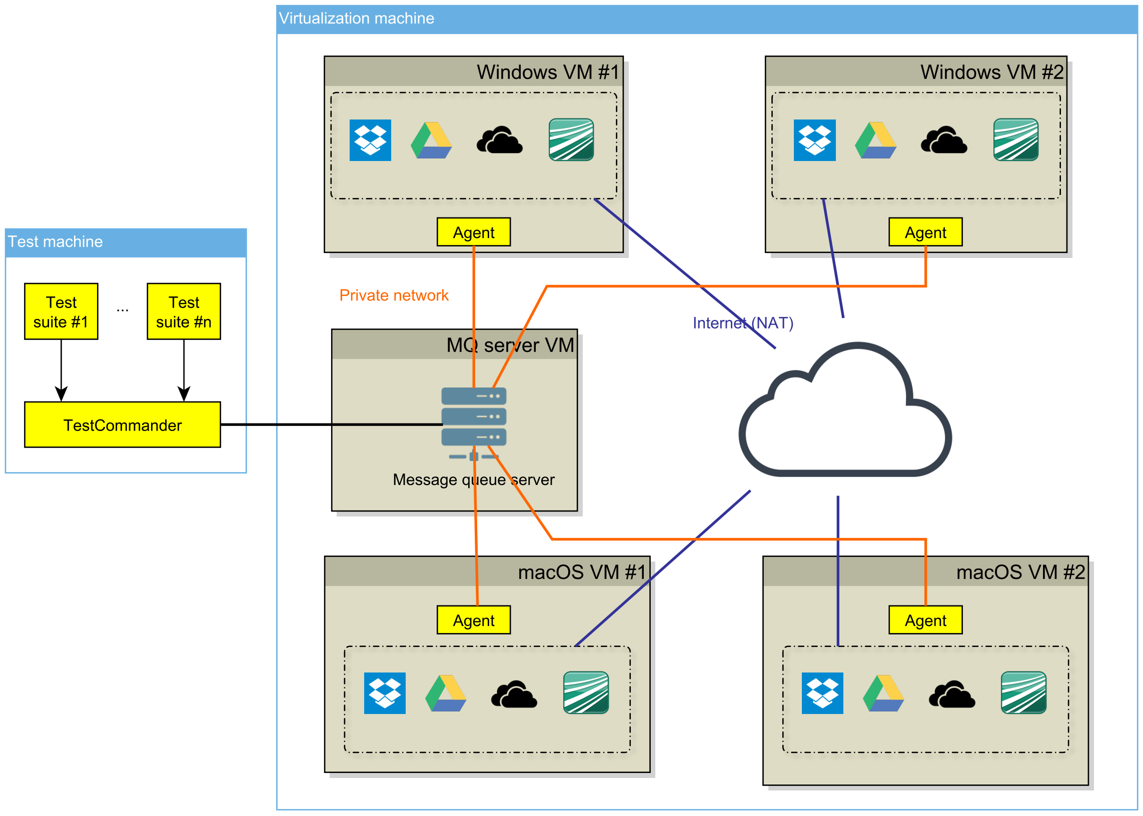


Figure 8.1: Synchronizer evaluation system setup

those server file system APIs. Only some *configuration* effort should be required when adding a new synchronizer, e.g. the specification of the local data directory root path, and the path to the synchronizer client binary file.

2. Generate *reproducible* results not affected by race conditions, by strictly controlling concurrency. Other works such as [BMP17] are affected by race conditions, which complicates the result analysis.

8.2.4.2 Testing method

Because the tested synchronizers provide no introspection into their state, we are limited to *black-box* testing. We only manipulate and monitor file systems and network activity to detect quiescent states. We control concurrency by explicitly starting and stopping synchronizers, solving requirement 2, and collects results for analysis.

8.2.4.3 Architecture and setup

The synchronizers under test typically synchronize a specific local disk directory with a *server* file system. They don't allow to synchronize two local directories with each other. For this reason, and because we want to test the synchronizers on both Windows and macOS, a *multi*-machine setup is required. As noted in [BDM15], virtual machines (VM) are most suitable, because they allow to establish a controlled environment, with fixed hardware configurations, eliminating uncontrolled variables in the experiment.

Our system setup is shown in figure 8.1. It consists of two physical machines. The test suite runs on a *test machine* such as the author's regular work station. On the *virtualization machine* we run two VMs of each operating system (OS). This allows to test for different behavior of the synchronization clients w.r.t. platform combinations (e.g. Windows-to-Windows, Windows-to-macOS, etc.). In each VM we install the synchronizer clients under test and configure them with equal settings, such as account credentials and data directory path on the local disk. To test cross-platform issues such as case-sensitivity, we set up a *case-sensitive* disk volume on macOS instead of using the default volume which is *case-insensitive*.

We built a test framework split into three components: *agents*, *TestCommander* and *test suites*:

- *Agents* are components we developed, installing one in each VM. Their job is to execute commands received by the *TestCommander*, such as *start Dropbox*, *create directory 'test' in the Dropbox data directory*, *await file system activity in the Dropbox data directory to cease* or *await network activity to cease*. To improve configuration flexibility, the commands are not sent from the test machine to an agent directly but are relayed via a message queue server.
- The *TestCommander* offers the same functionality as the *agents*, but augments the agent's API by a *host* parameter that specifies on which VM host(s) the command should be executed. It also implements two helper methods. The *reset synchronization* method provides a clean slate for a specific synchronizer by clearing its data directories on all hosts, making sure that all clients are in an empty, quiescent state. The method *establishBase* establishes a base set of files and directories. It first *resets synchronization*, followed by creating the files (with random content) and directories at one host, waiting that they are synchronized to all other hosts.
- The *test suites* are a set of tests that use the *TestCommander* to establish the base scenario consisting of a predefined set of files and directories, as well as applying operations to the local and server replica.

The multi-machine setup makes it easily possible to account for requirement 1. Instead of directly manipulating the *local* and *server* replica, we use *two* hosts, the *remote* and the *local* host, and install the synchronizer and agent in each of them. The *remote host* runs the *remote client*[9], the *local host* runs the *local client*. The remote client is used to *indirectly* manipulate the *server* replica as follows:

1. Shut down the *local* client, to avoid concurrency effects,
2. Start the *remote* client,
3. Manipulate the file system on the *remote* host,
4. Wait for the network activity to cease on the *remote* host.

The replicas and agents are shown in figure 8.2.

In summary, the execution of a test involves the following steps:

1. Establish a *base scenario*, using the *TestCommander*,
2. Shut down *local* client,
3. (optional) manipulate server replica via remote replica, wait for remote client to synchronize changes (see steps 2-4 described above),
4. (optional) manipulate *local* replica,
5. Start *local* client, wait for synchronization to finish,
6. Collect results from the local and remote agent.

[9]"Remote client" = file synchronizer client application executed on the remote host.

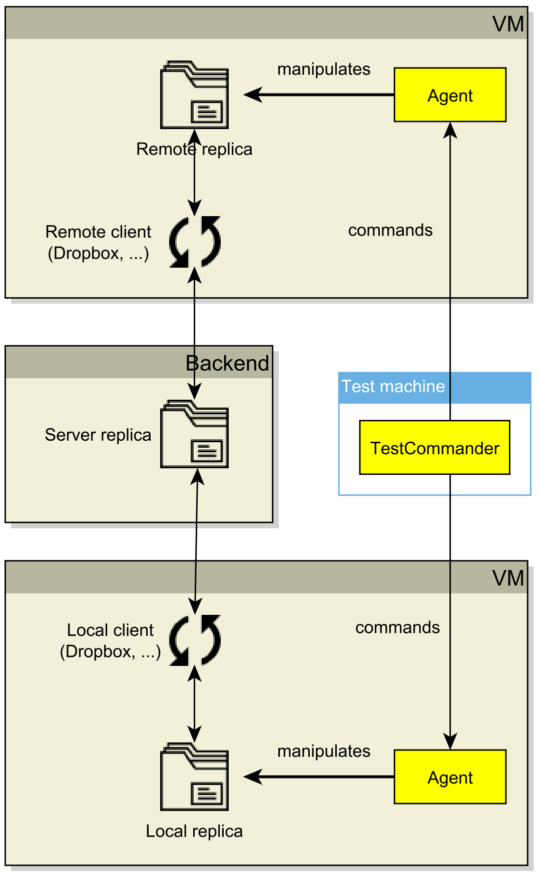


Figure 8.2: Indirect manipulation of the server replica

8.2.4.4 Test result analysis

The analysis of a finished tests either yields that it passed or failed. A passed test should indicate that the synchronization was successful. Informally a user will come to this conclusion if the state of the local and remote replica matches (considering directory structure and file contents, as presented by the file manager) and if the effect of the operations applied by the user to both replicas separately is observable in the final state. We use a similar assessment to decide whether a test has passed or failed. Formally, we consider the distributed system of the local, remote and server replica to be a labeled transition system $(S, O, \rightarrow)$. S is a set of states (each $\sigma \in S = (\sigma_l, \sigma_r, \sigma_s)$ is a triplet of the local, remote and server replica state). O is the finite set of operations that are part of the test. $\rightarrow$ is a set of labeled transition relations ($S \times O \times S$) that transform one state into another. Let σ_b be the state that represents the *base scenario*, i.e. the set of initial files and directories established at the beginning of each test, before applying operations.

Definition 2. Two states $\sigma_1, \sigma_2 \in S$ are *equivalent*, or $\sigma_1 \equiv \sigma_2$, iff the directory structures (set of paths) of both states match, the type of every path (file, directory) matches and checksums (e.g. SHA-1) of *files* match.

We do not require the ID or *lastmodified* meta-datum to match, because each replica uses its own values, and because we found that some synchronizers *copy* objects they are supposed to move, which creates new IDs. The average user is typically not aware of IDs on the local disk, nor are IDs presented by the file manager.

Definition 3. A necessary condition for a test to *pass* is that $\sigma_l \equiv \sigma_r$, where σ_l, σ_r are the quiescent local/remote states obtained after all test operations were applied (and the synchronizer clients have presumably propagated them).

Definition 4. A test that consists exclusively of *conflict-free* operations has *passed* iff def. 3 and $\sigma_l \equiv \sigma_o \wedge \sigma_r \equiv \sigma_o$ holds, where σ_o is the *test oracle* state obtained by applying all test operations (for the local and remote replica) to σ_b in a separate simulation.

For *conflict operations* and *cross-platform issues* tests we do *not* build oracles because we expect each file synchronizer to resolve the situation differently anyway, and we do not claim authority on what the most correct resolution is for each case. Here only definition 3 applies.

8.2.4.5 Implementation

VM provisioning To run all VMs in parallel a powerful host was chosen as virtualization machine. We used a Mac Pro with macOS 10.11.6 (El Capitan), 2x 2,26 GHz Quad-Core Intel Xeon, 24 GB RAM, and a SSD drive. The VMs were created using VirtualBox[10] v5.2.6 and Vagrant[11] v2.0.1, using Boxcutter[12] templates to generate fresh Windows guest operating systems. Each VM was configured with 2 virtual CPUs and 2 GB of memory. We installed Windows 10 Enterprise Fall Creator's update (v1709) and macOS 10.12.6 (Sierra) as guest operating systems. We used Ubuntu server (Xenial) for the message queue server VM, running RabbitMQ[13] v3.5.7 as message queue server implementation.

Machine provisioning was done using Vagrant which allows to execute provisioning commands on each boot, such as copying the current version of the *agent* code into VM's file system and setting up an additional *private network* used for communication between VMs. The agents use this private network to communicate with the message queue server.

Network monitoring and firewall On operating systems such as Windows we found no ready-to-use method to monitor network traffic of a specific process. We instead chose a solution that allows to query the traffic of a network interface card (NIC). To ensure that we only count traffic of synchronizer clients, we executed only one client at a time and used the Binisoft Windows Firewall Control[14] on Windows and the Radiosilence[15] firewall on macOS to limit Internet access to synchronizer clients only. Using a Firewall has additional advantages. For one, synchronizer client versions remain fixed over time, because auto-updaters shipped with the clients cannot detect or download newer versions. Another benefit is that system CPU and network load is kept constantly low, because no other auto-update mechanisms (e.g. Windows Update or macOS App Store updates) can start consuming CPU or network resources. Since we configured an additional private networking interface for each VM for the agent-to-message-queue-server communication, we could easily exclude agent traffic in our network monitoring, by only querying the NIC traffic of the NAT interface used by the synchronizers to communicate with the server replica.

Agent implementation We implemented the *agent* program in Python, to allow re-use of components such as the file system observer used in BSync. The communication between *TestCommander* and *agent* was done using the celery[16] framework, using RabbitMQ as messaging broker. Each agent was assigned to a dedicated message queue so that the *TestCommander* could address them separately.

8.2.5 Test descriptions and results

In this section we present the description and results for each test category. To improve the reading experience, some details are moved to appendix A.8 on page 219. We refer to the specific appendix sections where appropriate.

[10]See `https://www.virtualbox.org/`, retrieved July 21, 2019.
[11]See `https://www.vagrantup.com/`, retrieved July 21, 2019.
[12]See `https://github.com/boxcutter/`, retrieved July 21, 2019.
[13]See `https://www.rabbitmq.com/`, retrieved July 21, 2019.
[14]See `https://www.binisoft.org/wfc`, retrieved July 21, 2019.
[15]See `https://radiosilenceapp.com/`, retrieved July 21, 2019.
[16]See `http://www.celeryproject.org/`, retrieved July 21, 2019.

An unforeseen but very important consideration in test design was to limit the number of tests. While implementing and executing tests we discovered that some file synchronizers are very slow. It may take them up to a minute to transfer even simple changes from one replica to the other. Consequently most of the test execution time is spent waiting for quiescent states. Long waiting periods to detect these states are necessary to avoid invalid test results, causing test execution times of 3 minutes or more, per test. This means that automatic test generation as done in sections 8.1.2+8.1.3 is inappropriate. To have tests complete within finite time and to make the result analysis manageable, we instead carefully selected a small *sample* of tests (a few *hundred*, not thousands or more), using equivalence class partitions (ECP) [SLS14] to reduce the test count.

8.2.5.1 Conflict-free operation tests description

Tests of this category consist of one or more operations that do not conflict with each other. We designed four test *groups* presented in table 8.2. Its "Param." column contains the short-hand of *parameters*, whose values represent different equivalence classes. The tests are run multiple times, s.t. each iteration uses a different parameter value. Table 8.3 introduces the parameters for *conflict-free operations* tests.

Name	Description	Param.	Expected result
Individual single-replica operation	Each test executes a *single* operation on the *remote* replica: *create, delete, edit* and *move*. The operation affects either a *file* or a *directory* (*edit* operation only affects files). These tests are sanity checks.	H, O	The *local client* applies the exact same operation on the *local replica*.
Complex single-replica operations	Multiple *create, move* and *delete* operations with complex order dependencies from section 4.2.7 are executed on the *remote replica*. E.g. swapping the name of two files or directories. See appendix section A.8.1.2 for more details.	H, O	The *remote* replica remains unchanged. The *local* client applies an *equivalent* set of operations s.t. the final *local* replica state matches the *remote* replica state.
Multi-level operations	Two sets of *rename, move, create* and *edit* operations are applied to a set of hierarchically dependent objects. See appendix section A.8.1.3 for more details.	H, O, D	The operations of a set applied to one replica by our test code should be applied to the corresponding other replica in exactly the same way by the synchronizer.
Distributed move and edit of file	One specific synchronized *file* is only moved on the *remote replica* and only edited on the *local replica*.	H, O	On both replicas the file should be located at the destination path of the *move* operation, with the updated content.

Table 8.2: Description of conflict-free operations tests

Parameter name, values	Short-hand	Description
Host pair: Windows-Windows, macOS-macOS, Windows-macOS, macOS-Windows	H	The operation systems of the local and remote host. For instance, “macOS-Windows” means that the *remote* client runs on a macOS VM, the *local* client on a Windows VM. Tests of the categories *conflict-free operations* and *conflicts* are repeated with all four host pair values. Our hypothesis is that the choice of the host pair may influence the test result, because the Windows and macOS code of the synchronizer may differ, or because subtle differences between macOS and Windows file system APIs affect the data collected during update detection and therefore affect the synchronization result.
Operation detection mode: online, offline	O	When “offline”, the *remote* client is first shut down, then our test agent applies operations to the remote replica and finally starts the remote client again. Therefore the remote client is forced to detect operations from state. When “online”, the remote client is kept running during the execution of operations on the remote replica. It can use the log of file system events for synchronization, which may yield different results.
Operation distribution: same replica, distributed	D	Given two sets of non-conflicting operations, when “distributed”, one set is executed on the local, the other on the remote replica. Otherwise (“same replica”) both sets are executed on the *remote* replica.

Table 8.3: Test parameters for conflict-free operations tests

8.2.5.2 Conflict-free operation test results

The test results are shown in table 8.4. A ✓ symbol in a cell indicates that definition 4 on page 154 holds and that the final file system structure corresponds to the expected result from the test description table. The X indicates that the final file system structure is either not equal or does not match the expected result. In that case see the corresponding appendix sections for further details. Please refer to section A.8.1.1 for the remarks.

Test	BSync	Dropbox	Backup and Sync	One-Drive	Next-Cloud	Unison
Individual single-replica operation (create, delete, edit)	✓	✓	✓	✓A)	✓	✓
Individual single-replica operation (move)	✓	✓B)	✓C)	✓	✓	✓D)
Complex single-replica operations	✓	X	X E)	✓	X	✓D)
Multi-level operations	✓	XF)	X	✓	X	X
Distributed move and edit of file	✓	XG)	✓	XG)	XG)	XH)

Table 8.4: Results of conflict-free operations tests

8.2.5.3 Conflict operation tests description

Each of our tests enforce a situation we classify as conflict, equal to those described in section 5.5. To limit the amount of tests, we did not build tests where files are affected by *multiple* conflicts.

During the execution we instruct the *agent* to automatically create screen shots in regular intervals shortly after starting the *local* or *remote* client. This allows us to manually inspect whether the user is visually notified about synchronization issues or automatic decisions made by the client. We created one test per conflict, as illustrated in table 8.5. The parameters are explained in table 8.6.

Conflict name	Description	Param.
Create-Create	A file or directory named “test” is concurrently created in each replica.	H, F, R
Edit-Edit	The content of synchronized file “test” is concurrently changed in each replica.	H, R
Name clash	Forces a *Move-Create* or a *Move-Move (Dest)* conflict. For the former, the *move* operation is always done on the *remote* replica, while the *create* operation is done on the *local* replica. We combine these two different conflict types because all tested synchronizers produce the same result for both types.	H, F
Edit-Delete of file	Concurrently changes the content of a file in one replica and deletes it in the other one.	H, C
Move-Delete	Concurrently moves a file or directory in one replica and deletes it in the other one.	H, F, C
Move-ParentDeleted	Moves a synchronized file “file” into a synchronized directory “test” on one replica. Deletes “test” in the other replica.	H, C
Create-ParentDeleted	Creates a new file “test/new” or edits an existing one at “test/file” in one replica, and deletes directory “test” in the other replica.	H, C
Move-Move (Source)	Moves a synchronized file or directory “test” to “testMoved” in one replica and to “testOtherMoved” in the other replica.	H, F
Move-Move (Cycle)	Given the synchronized directories “A” and “B”, we move “A” to “B/A_moved” on the *remote* replica and move “B” to “A/B_moved” on the *local* replica.	H

Table 8.5: Description of conflict operations tests

Parameter name, values	Short-hand	Description
Object type mix: file-file, file-directory, directory-directory, file, dir	F	When testing conflicts, *different* objects are concurrently manipulated at the local and remote replica. For instance, "file-directory" indicates that the operation applied to one replica affects a file, while the operation applied to the other replica affects a directory. For tests that target a *specific*, already synchronized object, *file* or *dir* means that operations are applied to a file or directory respectively.
File content randomization: random, deterministic	R	When testing *create-create* or *edit-edit* conflicts, "deterministic" means both files are created (or updated) with the same file content in both replicas (which Syncpal counts as *pseudo conflict*), whereas "random" generates different, randomized contents.
Delete on replica choice: remote, local	C	When testing conflicts that involve a *delete* operation (e.g. Move-Delete or Edit-Delete), this parameter decides whether the *delete* operation is applied to the *local* or *remote replica*.

Table 8.6: Test parameters for conflict operations tests

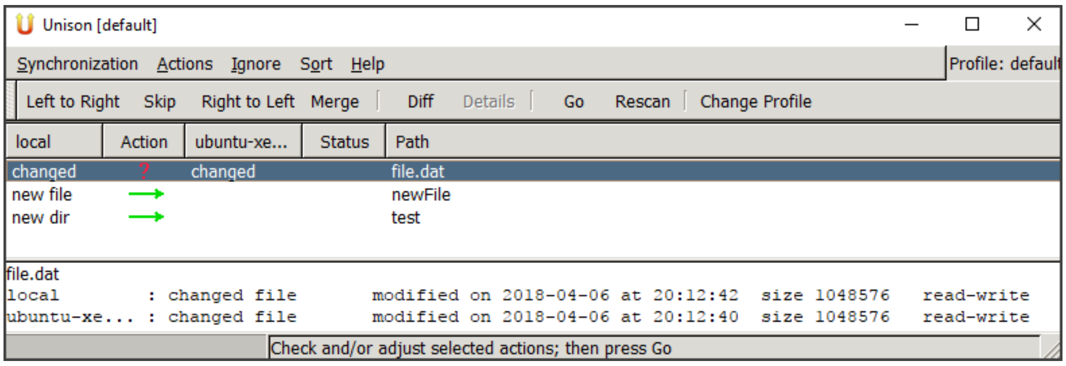
The table in appendix section A.8.2.1 provides a list of alternative expected outcomes.

8.2.5.4 Conflict operation test results

Before discussing test results, table 8.7 presents how each synchronizer client notifies the user about conflicts as they happen and whether users can inspect conflicts after the fact (after the notification has disappeared). We note that the table does not include *Unison* because here the user manually triggers the synchronization process and both conflicting and non-conflicting operations are graphically presented in a table. The "Not synced" tab of NextCloud's client GUI is used to show both *unsynchronizable* files due to *namespace limitations* (such as files using Windows reserved names or characters) and *conflict* files. Figure 8.3 illustrates the GUI users can use to inspect conflicts for our BSync implementation, NextCloud and Unison.

	BSync	Dropbox	Backup and Sync	One-Drive	Next-Cloud
Notification about conflict	✓ Windows, macOS	X	X	✓ Only Windows	✓ Windows, macOS
GUI to inspect conflicts	✓	X	X	X	(✓)

Table 8.7: Conflict notifications

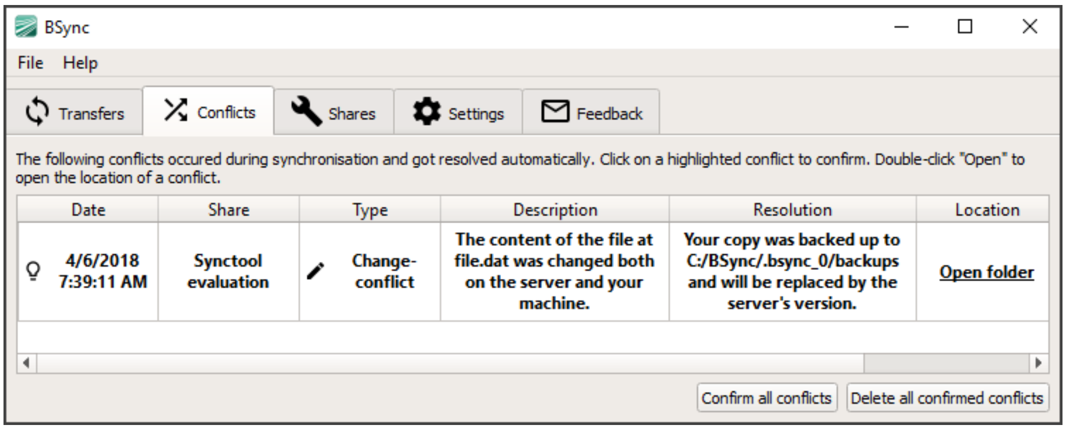

Unison [default]
Synchronization Actions Ignore Sort Help
Profile: default
Left to Right Skip Right to Left Merge Diff Details Go Rescan Change Profile

local	Action	ubuntu-xe...	Status	Path
changed	?	changed		file.dat
new file	→			newFile
new dir	→			test

```
file.dat
local         : changed file       modified on 2018-04-06 at 20:12:42  size 1048576   read-write
ubuntu-xe... : changed file       modified on 2018-04-06 at 20:12:40  size 1048576   read-write
```

Check and/or adjust selected actions; then press Go

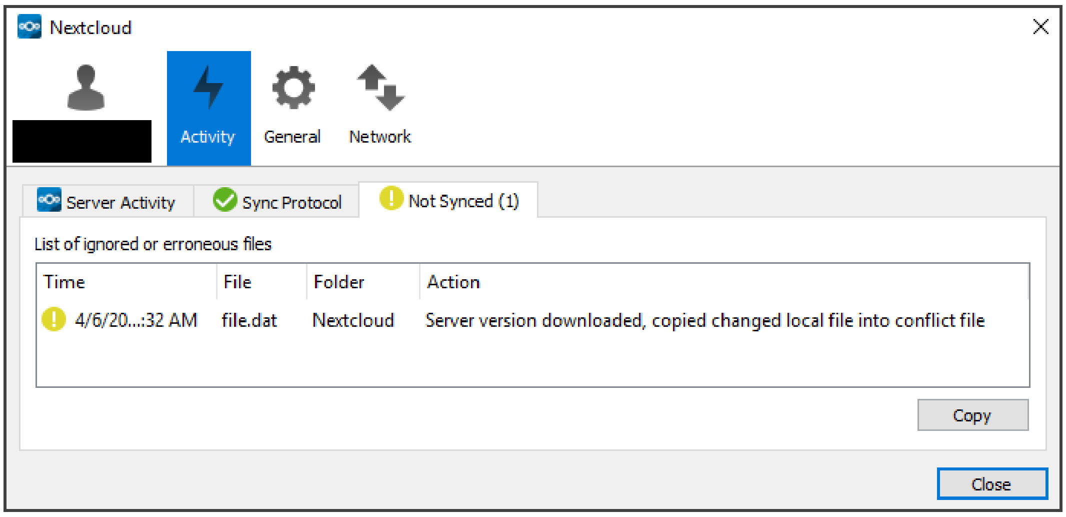
(a) Unison



The following conflicts occured during synchronisation and got resolved automatically. Click on a highlighted conflict to confirm. Double-click "Open" to open the location of a conflict.

Date	Share	Type	Description	Resolution	Location
4/6/2018 7:39:11 AM	Synctool evaluation	Change-conflict	The content of the file at file.dat was changed both on the server and your machine.	Your copy was backed up to C:/BSync/.bsync_0/backups and will be replaced by the server's version.	Open folder

(b) BSync (our implementation)



[redacted]

List of ignored or erroneous files

Time	File	Folder	Action
4/6/20...:32 AM	file.dat	Nextcloud	Server version downloaded, copied changed local file into conflict file

(c) NextCloud

Figure 8.3: User interfaces used to visualize conflicts

Graphical user interfaces provided by Unison, NextCloud and BSync to inspect conflict detections or resolutions (after the fact). Subfigure a (Unison) shows two non-conflicting *create* operations (of a file and directory) as well as the detection of an *Edit-Edit* conflict. The user needs to choose from a set of options for resolving the conflict (left to right, right to left, skip, merge).

Table 8.8 shows the detailed test results. For detailed explanations of the cell labels (Merge, Duplicate, ...) and further remarks see appendix sections A.8.2.1 and A.8.2.2. Unexpected or wrong out-of-sync results are marked in **bold**.

Conflict name	Param.	Our impl.	Dropbox	Backup and Sync	OneDrive	Next-Cloud	Unison
Create-Create	F=file-file, R=random	Rename A)	Rename B)	Rename C)	Rename D)	Rename E)	Detected F)
	F=file-file, R=determ.	Merge	Merge	Merge	Merge	Merge G)	Merge
	F=dir-dir	Merge	Merge	Merge	Merge	Merge	Merge
	F=file-dir	Rename A)	Rename B)	Rename C)	Rename D)	**Out of sync** H)	Detected F)
Edit-Edit	R=random	Overwrite	Duplicate-Sync	Duplicate-Sync	Duplicate-Sync	Duplicate-HostOnly	Detected F)
	R=determ.	Merge	Merge	**Duplicate-Sync**	Merge	Merge G)	Merge
Name clash	F=dir-dir	Rename A)	Merge	Rename, **out of sync** I)	Rename D)	Merge G)	Merge
	F=file-file, file-dir	Rename A)	Rename B)	Rename, **out of sync** I)	Rename D)	Rename, **out of sync** J)	Detected K)
Edit-Delete of file	C=any	Restore	Restore	Delete	Restore	Restore	Detected F)
Move-Delete	F=file, C=any	Restore	Restore	Local N)	Restore	Restore	Restore
	F=dir, C=any	Restore	Restore	Local N)	Local	Restore	Restore
Move-ParentDeleted, Create-ParentDeleted	C=any	Delete	Restore	Local L)	Restore	Restore	Detected M)
Move-Move (Source)	F=file	Remote	**Duplicate**	Remote	Remote / **Duplicate** O)	**Duplicate**	**Duplicate**
	F=dir	Remote	**Duplicate**	Remote	Remote	**Duplicate**	**Duplicate**
Move-Move (Cycle)	–	Remote	**Out of sync** P)	**Out of sync** P)	Remote	**Out of sync** P)	Detected Q)

Table 8.8: Results for conflict operations tests

8.2.5.5 Cross-platform issue tests description

We implemented one or more tests for each of the following five cross-platform issues:

- Case-sensitivity: Windows is always case-insensitive. macOS disk volumes, by default, are case-insensitive as well. However, macOS allows to create case-sensitive disk volumes. We create one and configure the synchronizer to use it, if possible. Our tests explore how synchronizers deal with upper/lower-case file name clashes and whether conflicts are correctly detected if names only differ regarding their upper/lower-case between replicas.

- Unicode normalization: Windows disk volumes are both Unicode normalization preserving and sensitive. On macOS disk volumes formatted with HFS+ are neither sensitive nor preserving as macOS always normalizes Unicode special characters to a NFD-like version. Our tests investigate whether Unicode special characters are automatically normalized to the form that is natively used on the corresponding platform and how synchronizers handle duplicated file names that only differ in their normalization form.
- Reserved characters and names: Windows provides a large set of reserved characters and names that can be created without problems on macOS, including issues such as short (8.3) filenames.
- Maximum name length limit: on both Windows and macOS a file name's maximum length is 255 characters. We test whether synchronizers take this limit into account in situations where they proactively extend a file's name, e.g. when creating conflict copies of files.
- Windows 8.3 file names: 8.3 file names (or *short filenames*, SFN) are a relict from the DOS (Disk Operating System) era. Given an existing file "my longer text.myext", Windows forbids creating (or moving other objects to) "MY_LON~1.MYE". This is counter-intuitive, because "MY_LON~1.MYE" is not part of the parent directory's listing returned by the file system APIs. On macOS, no such limitations exist. Our tests explore how synchronizers deal with situations where corresponding files are created or moved on macOS and then synchronized to Windows.

We refer to section 3.1 on page 29 where we present multiple expected results a client might implement to handle each issue. Like for *conflict operation* tests (see section 8.2.5.3) *cross-platform* tests also instruct the agent to automatically create screen shots.

1) Case-sensitivity:

Name	Description	Parameters
Case-sensitivity clash	On macOS (*remote* replica) we create two files or directories with equal name but varying upper/lower case (one upper-case, one lower-case object). If F=directory, we create a sub-file named "lower" or "upper" in the corresponding directory.	F, H=macOS-macOS, macOS-Windows
Case-sensitivity conflict	On one replica we apply CreateFile(f), on the other we apply CreateFile(F). This needs to cause a conflict on case-insensitive volumes (solved by automatically renaming one file), but not necessarily on case-sensitive ones.	H=macOS-macOS, macOS-Windows

2) Unicode normalization:

Name	Description	Parameters
Unicode normalization - clash	On Windows (*remote* replica) we create two files named “ä” and “ä” (one normalized using the NFC, one using the NFD normalization). This can be synchronized to another Windows host, but needs to be handled on macOS where both normalizations would clash.	H=Windows-macOS, Windows-Windows
Unicode normalization - Windows conversion	On Windows (*remote* replica) we create file “ä” with the NFD-normalization (untypical for Windows) to observe whether the client performs an automatic conversion. We expect the client to either convert the file name to the typical, platform-native NFC normalization on *both* the remote and local Windows host, or to skip conversion and use the NFD-form on *both* Windows hosts.	H=Windows-Windows
Unicode normalization - macOS conversion	On macOS (*remote* replica) we create file “ä” (which macOS always normalizes as NFD) to observe whether the local replica’s Windows client converts the normalization form to the Windows-native NFC form.	H=macOS-Windows

3) Reserved characters and names:

Name	Description	Parameters
Reserved characters and names	On macOS (*remote* replica) we create a file that contains “?”, as a representative for *reserved characters*, assuming that a synchronization client will behave equally for any reserved character. We also create the files "LPT1" and "LPT1.foo.bar" to test *reserved names*. Finally we create files "a ." and "b " to test handling of names that end with “.” or space. We expect a synchronizer to either automatically rename those files, or to skip their synchronization. In any case, they should notify the user about their decision.	H=macOS-macOS, macOS-Windows

4) Name length limit:

Name	Description	Parameters
Name length limit - conflict names	We produce a *Create-Create* conflict for two random-content files whose name length is 252 characters, being very close to the 255 limit. We test whether the *local* client (who discovers the conflict) truncates the original file’s name before appending the “conflict” suffix that is typically used when resolving such conflicts, see section 8.2.5.4.	H
Name length limit - temporary names	This test only applies to BSync and OneDrive. Their local client can correctly apply *complex single-replica operations* (see section 8.2.5.1) which involve having to temporarily rename objects to a randomly generated name to make other necessary *move* operations possible. This test runs the *“Move Occupied 1”* test described in appendix section A.8.1.2. For instance, OneDrive creates a directory “1180107144710-A” which it finally renames to “A” after applying other *move* operations. We test whether the synchronization is still successful if directory “A”’s name is instead 253 characters long.	H=Windows-macOS

5) Windows 8.3 file names:

Name	Description	Parameters
Windows 8.3 file names - distributed creation	On the *remote* replica we create a file “aaaaaa~1” which uses a short file name pattern. The *remote* client transmits the file to the *server* replica. On the *local* replica we create a long file “aaaaaaaaaa” and start the *local* client, to examine whether it detects (and how it handles) the issue that “aaaaaa~1” cannot be created on the *local* replica.	H=Windows-Windows
Windows 8.3 file names - macOS clash	The initial file system consists of two synchronized files named “aaaaaaaaaa” and “b”. On macOS (*remote replica*) we apply the operation *Move(b, aaaaaa~1)* and observe the result on the *local replica*, where this *move* operation is forbidden by Windows.	H=macOS-Windows

8.2.5.6 Cross-platform issue test results

The detailed test result are shown in table 8.9, with shorthands: NN = No Notification, IN = Inconsistent Names.

Unexpected or wrong results are marked in **bold**. For remarks and referenced figures, see appendix section A.8.3.1.

8.2.6 Summary and discussion

Using our test framework we compared the convergence behavior of *BSync*, *Dropbox*, *NextCloud*, Google *Backup and Sync*, Microsoft *OneDrive* and *Unison*. We designed three test categories (conflict-free operations, conflict operations and cross-platform issues), each made of several tests. We built a setup that consists of four virtual machines (2x Windows, 2x macOS) where the corresponding synchronizer clients are installed. A separate test computer runs the automated tests, sending commands to test agents we developed that are installed in the VMs as well. These agents execute these commands, such as operations that modify the file system, or start or stop a specific synchronizer client. We limited the number of tests using Equivalence Class Partitioning. The test execution time was still considerably long. Conflict and cross-platform tests took about 12 hours each, conflict-free tests took about 28 hours, for a *single* run. Because we repeated all tests three times, added run-time amounts to one week. Additional time was required for manual result analysis. In the remainder of this section we summarize our findings.

8.2.6.1 Synchronizer-specific issues

We found that some synchronizers show behavior that affects the result of several (if not all) test categories.

The *Backup and Sync* client appends a numeric suffix such as “ (1)” to the name of objects whenever there is a problem with using the originally intended name. This suffix is only used on *one*, not all, replicas. This causes names to be inconsistent on both replicas. The *Dropbox* client does not seem to use an internal *ID-based* file system model (such as $\mathscr{F}$) but a *path-based* one, which we discovered in our *multi-level* conflict-free operation tests. To avoid performance bottlenecks, Dropbox seems to use heuristics to identify moved *files*, such as comparing checksums, which allows it to send *move* operations to the server, rather than deleting and re-uploading such files. Once the Dropbox client *receives* a *move* operation from the server, it never physically moves the affected object on the local replica, but first *copies* the local file(s) to the destination, followed by either deleting the local source or moving it to an internal cache where it is kept for a varying amount of time before it is finally deleted. This behavior degrades the performance of *move* operations that affect *large* files or directories and consumes additional disk space. The Unison client works similarly in this regard, and it is known that its internal file system model is path-based [BP98].

Test name	Param.	Our impl.	Dropbox	Backup and Sync	One-Drive	Next-Cloud	Unison
Case-sensitivity clash	F=any, H=macOS-macOS	Remote rename	Remote rename, NN	Unchanged	_ A)	Unchanged	Skipped B)
	F=any, H=macOS-Windows	Remote rename	Remote rename, NN	**IN** C)		Skipped D)	Skipped B)
Case-sensitivity conflict	H=macOS-macOS	Autom. rename	Autom. rename	No conflict	Autom. rename	No conflict	Detected G)
	H=macOS-Windows	Autom. rename	Autom. rename	**IN** E)	Autom. rename	**Out of sync** F)	Detected G)
Unicode normalization - clash	H=Windows-Windows	Remote rename	Remote rename, NN	Unchanged	**Out of sync** H)	**Out of sync** H)	**Out of sync** I)
	H=Windows-macOS	Remote rename	Remote rename, NN	**IN** J)	**Out of sync** H)	**Out of sync** H)	**Out of sync** I)
Unicode normalization - Windows conversion	H=Windows-Windows	Autom. conversion K)	Autom. conversion K), NN	No conversion	No conversion	**Normalization mismatch** L)	**Normalization mismatch** L)
Unicode normalization - macOS conversion	H=macOS-Windows	Autom. conversion	Autom. conversion	**No conversion** M)	Autom. conversion	Autom. conversion	Autom. conversion
Reserved characters and names	H=macOS-macOS	Remote rename	Trims whitespace of file "b " N)	Unchanged	Skips chars (syncs names) O)	Unchanged	Unchanged
	H=macOS-Windows	Remote rename	Trims whitespace N), NN P)	**IN, creates undeletable files** Q)	Skips chars and names O)	Skips chars and names on Windows R), NN	Skips chars and names on Windows R)
Name length limit - conflict names	H=any	Truncates file name	**Out of sync** S)	**IN** T)	Truncates file name	**Out of sync** S)	–
Name length limit - temporary names	H=Windows-macOS	Truncates file name	–	–	**Hangs indefinitely** U)	–	–
Win 8.3 file names - distr. creation, see figure A.14	H=Windows-Windows	Remote rename V)	Skips SFN (local), NN	**IN**	Autom. Rename	Skips SFN (local)	Skips SFN (local) W)
Win 8.3 file names - macOS clash, see fig. A.15	H=macOS-Windows	Remote rename V)	Removes file "b" (local)	**IN**	**SFN replaces long file name**	**Out of sync**	**Out of sync**

Table 8.9: Results for cross-platform issues tests

8.2.6.2 Conflict-free operations

We found that all synchronizers were able to synchronize *individual* operations correctly. However, only our implementation and OneDrive managed to correctly apply *complex operation sequences* or *multi-level operations* to the other replica. Other implementations showed various flaws, such as becoming out of sync temporarily or permanently, or even crashing completely. The distributed *edit* and *move* operation applied to a file was only merged by BSync and Google's implementation - all others produced two files: the old file at the new location and the new file at the old location.

8.2.6.3 Conflict operations

We observed several cases of unexpected behavior when clients detect or resolve conflicts.

Our tests show that *NextCloud* could not handle *Create-Create* conflicts in cases where a file and a directory was created on the first and second replica respectively. Some *name clash* conflicts also caused issues with NextCloud, s.t. different replicas became out of sync.

The *OneDrive* client showed inconsistent behavior for *Move-Delete* conflicts. The client always *restored* the object in case it is a *file*, but always favored the locally applied operation in case it is a *directory*. All other synchronizers applied the *same* resolution strategy, irrespective of whether the affected object is a file or directory.

Apart from BSync and OneDrive, all other synchronizers had issues dealing with *Move-Move (Source)* and *Move-Move (Cycle)* conflicts appropriately. They either duplicated directory structures, which clutters the file system, or became out of sync. For *OneDrive*'s client we discovered that the resolution of *Move-Move (Source)* conflicts is done differently on *macOS* vs. *Windows*.

Considering conflict awareness we observed that most synchronizers display little (graphical) information about conflicts and their resolution to the user. Apart from BSync only Unison and NextCloud provided a GUI to inspect conflicts. Unison's GUI showed *detected* conflicts, providing the user with multiple choices for *resolving* them. NextCloud's GUI offered a window that shows entries for *resolved* conflicts, providing the affected file name and an explanation, but only for those types of conflicts the client solved by appending a *conflict-suffix* to the file name. Most notably, conflicts such as *Move-Delete* or *Edit-Delete* (whose resolution does not involve appending this suffix) are *not* found in this GUI. In contrast BSync offers a GUI with detailed conflict information for *all* conflict types. We also found that the only synchronizers that showed real-time *notifications* while conflicts happen are BSync, NextCloud and OneDrive's *Windows* client.

As expected, we found a high degree of variance in the automatic conflict resolution applied by the industrial synchronizers. While all implementations agreed on how to deal with *Create-Create* conflicts, there was no consensus on how other conflict types should be resolved.

8.2.6.4 Cross-platform issues

When one replica is case-sensitive and one is case-insensitive, we found that *NextCloud* and *Backup and Sync* had issues with detecting and resolving case-sensitivity clashes.

Except for BSync, all other synchronizers had issues dealing with *Unicode normalization*, or handling clashes of duplicate names that only differ by their normalization. The clients either went into an unexpected error state, or they did not inform the user about automatically applied rename operations or skipped synchronizations of affected files.

There is no consensus regarding the handling of Windows's reserved characters and names. Some synchronizers transferred *all* those objects to the *server* replica but then omitted their synchronization on the Windows replica, while others (such as OneDrive) skipped sending reserved *characters* to the server replica, but did send reserved *names*. Some implementations applied automatic renaming (that turns reserved characters into allowed ones) already on the macOS replica when encountering those reserved names or characters. We observed that some synchronizers applied *consistent* rename-schemes, while others only renamed some reserved characters but not others.

Apart from our implementation, all synchronizers had issues with some of our tests regarding Windows 8.3 short file names or using names with a length that is close to the limit of 255 characters.

8.2.6.5 Discussion

As our results have shown, other synchronizers often did not correctly synchronize concurrent operations accumulated during longer offline periods. This is particularly relevant in practice for users who frequently work offline. Such bugs cause irritation for the user and lower their trust in the synchronizer[17]. If any error messages were shown at all (for instance, *Dropbox* never did and failed silently throughout all tests), they were as helpful as “Unknown error” or “Upload error” (Backup and Sync, NextCloud) which is not indicative of the cause. These errors and messages indicate that synchronizer developers did not thoroughly analyze the file system model. Our own thorough analysis of file system model $\mathscr{F}$ made it not only possible to avoid these issues, but also helped to design the tests.

We think that further insights may be gained from a dialog between researchers and industry, in particular for conflict handling. It would be interesting to discuss why products such as Dropbox and Google’s Backup and Sync hide conflicts from the user, given that these companies have UX specialists who know how to design usable software. The dialog may also reveal insights for the specific conflict resolution approaches used in each synchronizer.

8.3 Synchronization and conflict statistics

As discussed in section 7.1 on page 129, our implementation, BSync, has been in productive use since July 2017 by close to 30 users. We performed an empirical evaluation by collecting various statistics to answer the following questions:

- Can we confirm the findings of other works [SS05; WRB01; KS92; Rei+94] which state that file system conflicts are rare in practice?
- Which of the ten conflicts discussed in chapter 5 are most common? We assume that Edit-Edit conflicts occur most often, from personal observation and asking other users.
- Considering conflict-*free* operations, how is BSync used by its users? What is the distribution of operation types? Is the number of *create* operations outweighed by the sum of *edit*, *move* and *delete* operations?
- How often do *pseudo* conflicts occur in practice? Was the effort justified to implement pseudo conflict detection?

In the following subsections we discuss related work, how we collected statistics data and present our results.

8.3.1 Related work analysis

There are numerous works [SS05; WRB01; KS92; Rei+94] which re-affirm the use of optimistic replication for file systems. They found that its main disadvantage, *conflicts*, are not actually a problem in practice, because they occur rarely. The cited works can be divided into theoretical analyses and empirical evaluations. The former is done in [WRB01] where the authors built a mathematical model used in a simulation where synchronization and file updates occur regularly. The simulation then manipulates the synchronization interval μ, while files are regularly updated (with a different rate) in each replica. They found that $s = \frac{\textit{\# of conflicts}}{\textit{synchronization}}$ increases with μ. The relationship is not linear, but s asymptotically approaches a limit. Consequently, when $\frac{1}{\mu}$ is the number of reconciliations per time unit, the conflict

[17]We are aware of multiple anecdotes (and have first hand experience) where users stop relying on file synchronizers doing their job correctly after working offline, because of mishaps. Users then copy files they work on out of the synchronized directory and finally copy it back in once they are online again.

rate $r = s * \frac{1}{\mu}$ is small not only for small values of μ (which is intuitive: synchronizing often will cause a very small s, and thus r is also small), but also for large values of μ (synchronizing very rarely also causes r to be small). There is a worst-case for some μ where r is largest.

Other works provide empirical evidence. In [KS92] write update conflicts were only found in less than 0.5% of all write operations. In [Rei+94] the relative conflict frequency was even smaller, stated as ~ 0.0035%. However, this figure only covers *non-directory* updates and ignores name-collision conflicts which were automatically resolved at an earlier stage. The authors also found that *"an environment in which some machines are often disconnected will generate more conflicts"*, which confirms the propositions of [WRB01]. Another discovery in [Rei+94] is that such averaged numbers are misleading. *"Conflicts tended to happen more often to users who worked with the disconnected machines. A few users thus experienced much higher conflict rates, while many users encountered considerably fewer conflicts than the average."* [Rei+94].

8.3.2 Data collection

In BSync we implemented a log that contains all successfully executed operations and resolved conflicts. The log is regularly transmitted to a central server for data analysis. For non-conflicting operations, the log entries include the following information:

- Operation name ("create", "edit", "move", "delete") and affected replica (local, remote),
- Omit flag: if True, then the operation is a *pseudo conflict* operation, False otherwise,
- File extension, e.g. ".pdf", or "dir" in case the affected object is a directory.

For conflicts the log entries include these items:

- Conflict type ("Create-Create", ...),
- Local and remote file extension,
- Loser replica identifier (local, remote).

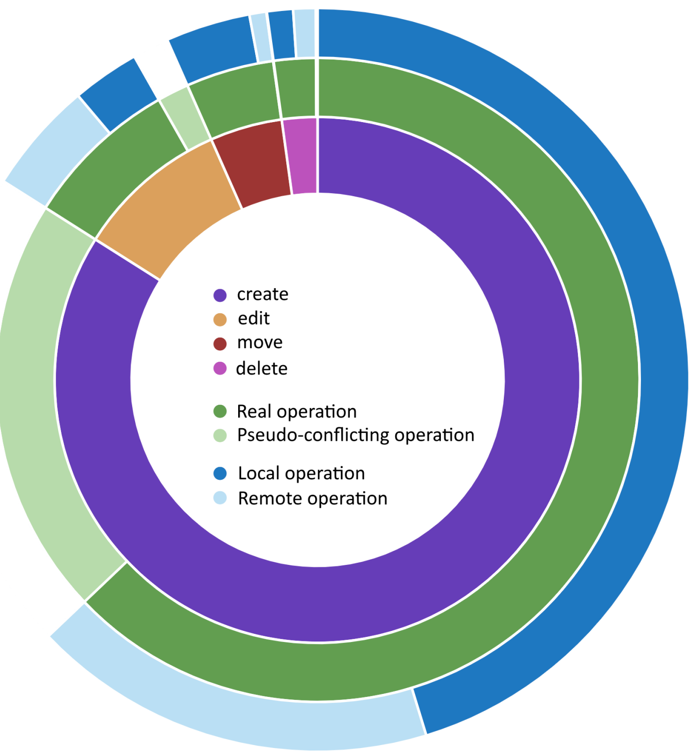
For both conflicts and non-conflicting operations we always transmitted an anonymized user ID. Any information that allows to identify individual users was omitted, such as file or account names. Unfortunately this made it difficult to answer questions like "what user activity causes conflicts?", as this would require user interviews, but we did not know the user's identity. We note that users are able to opt out of the statistics collection in the settings dialog of BSync.

8.3.3 Results

We now present the analysis of the statistics data we collected in the time period of July 2017 to December 2018. We note that the results of every analysis strongly depends on the environment, i.e., the users and the way they work. The discussed user base with close to 30 users consisted predominantly of property administration staff, who work with tools like Autodesk Revit or WSCAD. The files produced by these tools are then synchronized by BSync. The second user group (about one third of the users) were mostly software engineers who use BSync for any kind of files *other than* source code, e.g. documents and images. Source code management is delegated to expert tools, such as Git.

8.3.3.1 Operation analysis

Figure 8.4 breaks down the 793′973 conflict-free operations logged by BSync. The innermost ring shows the operation type distribution. The next ring breaks each operation type down into real vs. pseudo-conflicting operations. The outermost ring breaks the real operations down into local vs. remote operations. *Local operations* refer to operations that were detected on the server and thus propagated to the user's device. The enclosed table provides values rounded to 0.5%.



Name	Operations	Relative	Pseudo conflicting, relative	Local operations (relative to non pseudo conflicting)
Create	666’735	84%	25%	72%
Edit	74’645	9.5%	17%	37.5%
Move	35’276	4.5%	0.5%	83.5%
Delete	17’317	2%	3.5%	54%

Figure 8.4: Break down of operations

Contrary to our assumptions the *create* operation makes up the largest portion of all operations (84%). There are several explanations for this. First, 25% of the *create* operations are *pseudo* conflicting operations, which were created when the user configured a new folder pair, but already possessed the data on the local disk. These operations simply reflect the rebuilding of the persisted state, see section 5.4.2 on page 76 for more details. Second, some users treated BSync as *backup* software for a large number of (smaller) files, such as images, which were never modified again. Third, applications such as Autodesk Revit or WSCAD create a large number of temporary files and folders which they seem to never delete. Because the log entries do not contain directory names we do not know the extent of create operations that can be explained by the last two arguments.

Our mechanism to detect and resolve *pseudo* conflicts has proven beneficial to avoid real conflicts. As the table in figure 8.4 shows, *create* and *edit* operations have a considerable amount of such pseudo conflicts. We note that it is unlikely that two users independently *edited* a specific file resulting in the exact same content. A deeper analysis of these *edit* operations revealed that 84% of them affect files which change very frequently on the local disk, because they are continuously being written to, e.g. lock files or other temporary files. When they change *during synchronization* while BSync is online, the problem discussed in section 5.4.2 on page 76 applies again. The synchronizer detects that the file changed while it was uploaded, aborts the transfer and restarts the synchronization. The next synchronization iteration detects that the file has been edited on both the server and the local disk, because the previous transfer did finish (was not truly aborted).

8.3.3.2 Conflict analysis

An overview of the 951 conflicts is shown in figure 8.5, together with the most common file extensions for Edit-Edit conflicts. The inner ring of figure 8.5a shows the conflict type distribution. The outer ring uses a different color for each unique user experiencing the conflict.

The resulting chance for conflicts per operation is $\frac{951}{793'973} = 0.12\%$, which is in the same range as in [Rei+94; KS92]. We can also confirm the statement of [Rei+94] that conflicts are concentrated on a few users. 79.3% of all conflicts were experienced by *two* users. The chance for conflicts is 1.88% for the user who experienced most of the conflicts (shown in pure green in figure 8.5a), which is 15 times larger than the average. Interestingly, most Edit-Edit conflicts occurred for users who predominantly worked *online*, which is contradictory to previous statements that assumed that conflicts occur the longer the *offline* period is. The three most affected file extensions, shown in figure 8.5b, are extensions used by the WSCAD application. When two distinct users work on a file in parallel in WSCAD, the synchronizer is only notified of the *edit* operation (and can detect conflicts for it) once the file is *saved*, which happens either due to the user's request, or when some condition triggers WSCAD to automatically save it. Since it is possible that a user may work on such a file without saving it for, say, an hour, the chance that some other user also works on that file (and saves it) within that time period is not negligible, causing such conflicts.

8.4 Complexity and performance analysis

To determine the practical viability we evaluate both the complexity and performance of Syncpal. We start with a complexity analysis using the Big-O notation in section 8.4.1, followed by performance measurements of our BSync implementation in section 8.4.2.

8.4.1 Complexity analysis

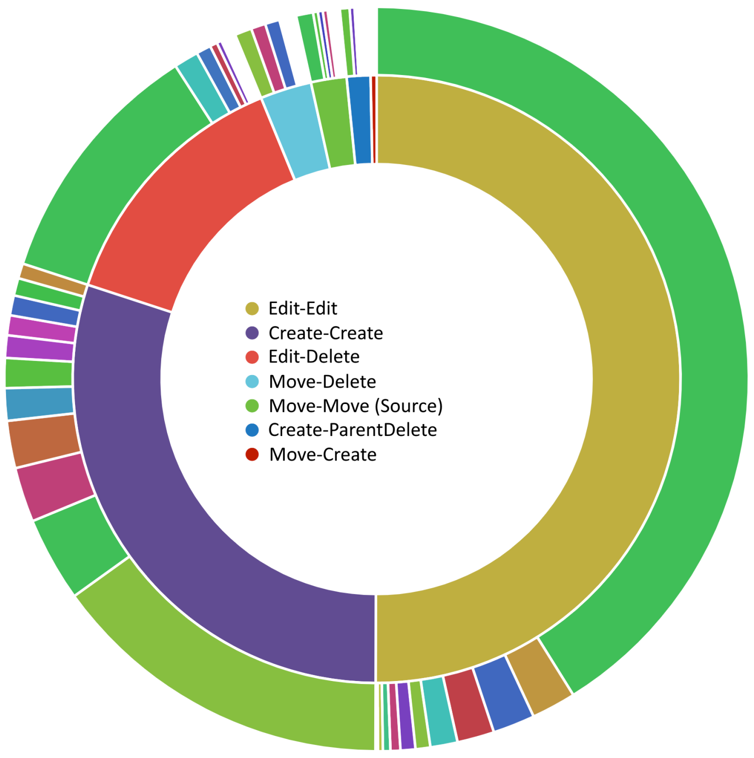
Let $n = count(ids(db)) + count(ids(snapshot))$ be the number of objects. h is the height of the update trees, and k is the number of operations detected by $compute\_ops()$. We start with *memory* complexity analysis, which is straightforward. Syncpal generates *snapshots* and *update tree* structures, each with n objects, as well as k operation objects. Thus, memory complexity is $O(n+k)$. For the *computational* complexity we break down the analysis for each stage separately, followed by analyzing the overall complexity.

We start with *update detection* whose complexities are shown in table 8.10. Building snapshots is $O(n)$

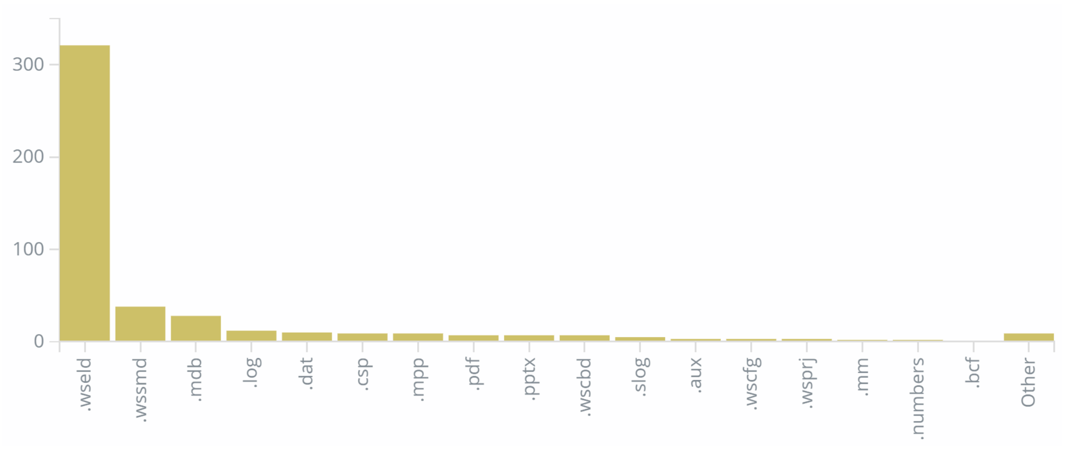
Step	Average case	Worst case
Build snapshot	$O(n)$	$O(n)$
compute_ops(db, snapshot)	$O(n)$	$O(n)$
Generate update trees (step 1-7, step 8)	$O(k\cdot h)$, $O(n\cdot h)$	$O(n^2)$
Generate update trees (total)	$O(n\cdot h)$	$O(n^2)$
Total	$O(n\cdot h)$	$O(n^2)$

Table 8.10: Complexities of update detection stage

because we have to iterate over all objects in the file system (or database), which is $O(n)$, and add them to the snapshot structure, which is $O(1)$. The *compute_ops()* function, see algorithm 1 on page 49, iterates over all objects in the snapshots twice and creates k operations. We can approximate $k = n$, even though in practice this only holds in the *first* synchronization iteration. Typically, $k \ll n$ in consecutive



(a) Break down of conflict types



(b) Most common file extensions for Edit-Edit conflicts

Figure 8.5: Conflict analysis

iterations. When generating update trees, creating and inserting a new node into a tree is $O(h)$. Steps 1-7 depend on k and step 8 depends on n, so in an average case, complexity is $O((k+n)\cdot h))$. In the worst case, where the file system resembles a *degenerate* tree, i.e., $h = n$, the complexity is $O(n^2)$.

Table 8.11 shows the complexities of *finding* conflicts during reconciliation, see algorithms presented in appendix A.6 on page 211. We distinguish between the very first synchronization iteration (`first_sync`

Step	Average case	Worst case
$corresponding\_node\_direct()$, $node.get\_child()$	$O(1)$	$O(1)$
corresponding_object_id()	$O(h)$	$O(n)$
Find Move-Move (Cycle) conflict	$O(k^2)$	$O(n^2)$
Find all other conflicts (`first_sync` is True)	$O(n\cdot h)$	$O(n^2)$
Find all other conflicts (`first_sync` is False)	$O(n)$	$O(n^2)$
Total (`first_sync` is True)	$O(n\cdot h)$	$O(n^2)$
Total (`first_sync` is False)	$O(k^2+n)$	$O(n^2)$

Table 8.11: Complexities of *finding* conflicts

is True) and consecutive iterations. In the first iteration only Create-Create conflicts can exist, and *corresponding_object_id()* is used to find them. The average complexity in this case is $O(n\cdot h)$ rather than $O(k\cdot h)$, because all n objects are detected as created in the first iteration, thus $k = n$. In consecutive iterations, finding conflicts is $O(n)$ for all conflicts except *Move-Move (Cycle)*, because we iterate over all n objects, but finding a conflict for a specific object is $O(1)$. Create-Create conflicts may still exist, but now finding them involves calling *corresponding_node_direct()* rather than *corresponding_object_id()*. Complexities of *resolving* conflicts depend on the conflict, as shown in table 8.12.

Conflict	Average case	Worst case
Create-Create, Move-Create, Edit-Edit, Edit-Delete, Move-Move (Dest),	$O(1)$	$O(1)$
Undo move: Move-Move (Source/Cycle), Move-Delete (when *delete* operation wins)	$O(h)$	$O(n)$
Move-Delete (when *move* operation wins)	$O(n)$	$O(n)$
Create-ParentDelete	$O(k)$	$O(n)$
Move-ParentDelete	$O(k\cdot h)$	$O(n^2)$
Total (take worst of all above)	$O(n)$	$O(n^2)$

Table 8.12: Complexities of *resolving* conflicts

For the *propagation* stage, if *sorting* is not required, complexity is $O(k)$. However, if sorting is required, we need to analyze *sort_operations()*, whose complexities are shown in table 8.13. Considering algo-

Step	Average case	Worst case
Sorting iterations (outer loop in algorithm 2)	$O(1)$	$O(k)$
fix_xyz() (inner loop)	$O(k^2)$	$O(k^2)$
fix_impossible_first_move_op(), section 6.3.2.5	$O(1)$	$O(n)$
find_complete_cycles(), *break_cycle()* (inner loop)	$O(k)$	$O(k)$
Total	$O(k^2)$	$O(n+k^3)$

Table 8.13: Complexities of *sort_operations()* in propagation stage

rithm 2 on page 58 and theorem 1 on page 58, we see that the outer loop is executed at most k times. The *fix_xyz()* functions in the inner loop are $O(k^2)$ each, *find_complete_cycles()* and *break_cycle()* are

$O(k)$, which overall yields $O(k^3)$ for *sort_operations()*. In practice, *cycles* are rare and sorting completes in a single iteration, thus *sort_operations()* is typically $O(k^2)$.

Table 8.14 shows the overall complexity of Syncpal, assuming that there are c synchronization iterations. The complexity of the very first synchronization is lower because no sorting is needed. As long as there

Case	Average case	Worst case
First synchronization (only *create* operations)	$O(c\cdot n\cdot h)$	$O(c\cdot n^2)$
Normal synchronization	$O(c\cdot(n\cdot h + k^2))$	$O(c\cdot(n^2 + k^3))$

Table 8.14: Complexities of Syncpal algorithm

are only create operations (non-conflicting or pseudo-conflicting) only order dependency rule 7 applies. However, the generation of operations described in section 6.2 is done by breadth first iteration, which implicitly satisfies this rule. In practice, when only a small set of changes needs to be synchronized, $c = 1$ and k is so small that even k^2 is irrelevant. Thus, the average case complexity is $O(n\cdot h)$.

8.4.2 Performance analysis

We built an automated test that determines execution times and memory usage while it runs. The goal is to determine the maximum feasible number of objects and operations, and to identify potential optimizations. Our test first creates $n = [100, 1'000, 10'000, 100'000]$ directories on the root level on one replica, followed by synchronizing them to the other one. Then $k = [100, 1'000, 10'000, 100'000]$ operations are executed on one replica and then synchronized to the other one. We evenly divide k into *createdir*, *delete* and *move* operations. We are interested in execution times and memory usage incurred when building snapshots, detecting operations (*compute_ops()*), building update trees, finding conflicts (note that our test scenario does not have any), generating Syncpal operation and sorting them. From the user's perspective, synchronization consists of only two stages, the *planning* stage that builds the list of operations (which entails all steps we just mentioned) and the *execution* stage, which propagates all planned operations. We are only interested in performance of the *planning* stage, because the execution stage's duration depends entirely on the operations themselves[18] and we can provide visual progress feedback to the user. Such feedback is *not* available for the *planning* stage, consequently this stage should be as short as possible. A user would stop using a synchronizer if planning took several minutes for every change, or consumed large amounts of memory.

8.4.2.1 Result analysis

Results for the memory use are shown in table 8.15, execution times are given in table 8.16.

The first bottleneck regarding execution time is *snapshot creation*. While filling a snapshot is efficient ($O(1)$), obtaining meta-data for each object is expensive. On a local disk this requires opening and closing a file handle on Windows, or performing a *stat()* call on macOS, which becomes noticeable ($t > 0.27$ s) for $n > 1'000$. On a remote file system addressed using WebDAV obtaining meta-data is even more

[18]For instance, transferring large files takes much longer than moving objects or creating directories.

Artifact	Memory use (MB)	Growth	Notes
Snapshot	70	Linear with n	Use *per replica*.
Operation list from *compute_ops()*	1	Linear with k	–
Update tree	50	Linear with n	Use *per replica*.

Table 8.15: Measured memory use for $n = 100'000$

Step	Execution time (seconds)	Growth	Notes
Snapshot creation	~ 27	Linear with n	Time *per replica*. Measured on a Solid State Disk.
compute_ops()	~ 1	Linear with n	–
Update tree generation	~ 3	Linear with $n \cdot h$	Time *per replica*.
Reconciliation (find platform inconsistencies, find conflicts, build Syncpal operations)	~ 10	Linear with n, quadratic with k	The 10 seconds evenly distribute among the three sub steps. Measured for $k \leq 10'000$.
Operation sorting	~ 160	Quadratic with k	Measured *after* applying $k = 10'000$ operations (also measured $k = 100, 1'000$ but aborted with $k = 100'000$).

Table 8.16: Measured execution times for $n = 100'000$

expensive, as it requires network calls for *each directory*. We note that our *hybrid update detection* approach presented in section 4.3 on page 59 mitigates this issue in practice, because the expensive snapshot creation is done only during the very first synchronization iteration.

The second execution time bottleneck is *operation sorting*. Sorting 10′000 operations or more is infeasible, as the user would assume that the synchronizer has crashed. However, in practice we observed that reconciliation typically either produces a large number of *create* operations exclusively (which don't require sorting), or a small number (usually below 100) of operations of mixed type, where sorting completes in a fraction of a second.

Considering memory use, the bottlenecks clearly are *snapshots* and *update trees*. For instance, synchronizing a directory with $n = 100'000$ objects would incur $4 * 70 + 1 + 2 * 50 = 381$ MB, because 4 snapshots and 2 update trees are required.

8.4.2.2 Future optimizations

Our first suggestion is to reduce memory use. By moving snapshots from memory to a relational database and implementing the *compute_ops()* algorithm using SQL queries, the memory use can be reduced considerably. However, execution time is expected to increase by one order of magnitude. To reduce the memory footprint of update trees we can omit step 8 of the update tree generation procedure and instead only update missing meta-data of *intermediate* nodes from $dbsnapshot$. However, additional work is required to verify that finding conflicts and sorting still works on such *incomplete* trees, and the *Platform Inconsistency Checker* module needs to be updated to efficiently find inconsistencies using incomplete update trees.

The latter suggestion also has a beneficial impact on *execution time*. As the update tree generation time mostly depends on the number of nodes, omitting step 8 will reduce the complexity from $O(n \cdot h)$ to $O(k \cdot h)$. Omitting step 8 will also speed up reconciliation time, as the majority of time is spent iterating over all n nodes. When instead iterating over the *incomplete* tree, i.e., over k nodes, this reduces the average case complexity from $O(k^2 + n)$ to $O(k^2 + k) = O(k^2)$.

Optimizing sorting is left as future work.

8.5 Conclusion

This chapter provided a technical evaluation of BSync, our implementation of Syncpal, as well as other industrial-grade synchronizers. Except for section 8.4.1 we employed automated testing or data collection combined with a manual result analysis.

We found that the tests from section 8.1, which focus on our own implementation, are a valuable complement to the theoretical proofs. They uncovered several kinds of coding errors. Some affected parts of Syncpal which were formally proven to be correct but were incorrectly translated to code. Others affected modules that were not part of the proofs (e.g. the update tree generation routines). The tests also helped to confirm or refute assumptions, such as the one discussed in section 6.3.2.5. They also uncovered an incorrect definition of the *Move-Move (Cycle)* conflict (see section 5.5.10), which we originally defined with an additional precondition that requires that directories A and B are *hierarchically independent* in $dbsnapshot$. However, our random tests in section 8.1.3 found counter examples where the conflict holds, even when A and B are hierarchically dependent.

Looking outwards, testing other synchronizers revealed a plethora of issues when faced with extended offline periods or platform inconsistencies, which negatively affect their usability. The results thus validate our motivation and confirm all those issues discussed in section 1.1. Our implementation significantly improves usability for the discussed problems.

Finally, our complexity and performance analysis done in section 8.4 revealed numerous future optimizations. These were not implemented yet, as it is good engineering practice to focus on features and correctness first and optimize the code later. Some memory and performance bottlenecks, such as linear growth of complexity with n, affect *state-based* synchronizers in general, not just BSync. Even leading industrial products with large software engineering teams have not been able to overcome them. For instance, the help documents of Dropbox and OneDrive warn users that the synchronizer's performance degrades when synchronizing more than 300'000 or 100'000 objects respectively[19]. In practice BSync did not yet encounter performance bottlenecks, as even large synchronized BSCW workspace directories contained less than 50′000 objects. Future optimizations will still be necessary when rolling out to a wider audience.

[19]See `https://support.microsoft.com/en-us/help/3125202` for OneDrive and `https://help.dropbox.com/space/file-storage-limit` for Dropbox, retrieved July 21, 2019.

Chapter 9

Conclusion and outlook

This work was driven by the motivation to build a file synchronizer for end-users. Compared to state-of-the-art academic works and industrial products it should provide a better handling of heterogeneous file systems and synchronize long offline periods correctly. To show that we have achieved this goal with our Syncpal algorithm we revisit the research questions and discuss the effect of our findings on research and industry in section 9.1. There are still some limitations illustrated in section 9.2. In section 9.3 we present future work, which is derived from these limitations as well as suggestions made by our users who used BSync in practice for several years. We finally conclude this work in section 9.4.

9.1 Discussion of research questions

RQ1 - File systems: What different kinds of file system definitions exist in academia and practice? Which criteria are relevant for file synchronizers? How should a file synchronizer's internal, abstracted file system model look like, which incompatibilities exist and how can they be handled?

The file system models used internally in *academic* file synchronizers have shown to be dissimilar. Some works model files and directories as uniquely identifiable objects, others use anonymous paths. The mechanisms used to link objects in the file system tree may differ. The existence of *directories* as a distinct entity varies between works, and there is a disagreement regarding the set of available operations, including their pre- and postconditions. We extended our analysis to implementations found in *practice*, such as NTFS or WebDAV, which have shown further traits of heterogeneity. These include varying namespace limitations, as well as different mechanisms to manipulate meta-data or lock objects. We proposed a method to build an internal model with maximum compatibility to existing real-world file systems, by integrating compatible traits and suggesting methods to handle incompatible ones, e.g. by adding objects to an ignore list. Our comprehensive heterogeneity analysis and methods for handling incompatibilities improve the *usability* of file synchronizers by providing *awareness* to the user. In contrast, several existing industrial systems ignore some of these heterogeneous traits, causing permanent replica discrepancies which degrades usability. To the best of our knowledge we are the first to discuss heterogeneity for *file systems* and file synchronizers in academia, thus providing a foundation for prospective works.

RQ2 - Operation order: As operations detected during state-based update detection lack order, but not all operations are commutative, how can a valid order be detected and propagated by a synchronizer? This question was already answered for file system models which do not support *move* operations, but remains open for file systems which do support them.

We first solved the problem of finding a valid operation order for the simplified case of *uni*-directional synchronization, where only one replica is manipulated. Using the formally defined pre- and postconditions of the operations of our file system model we found a number of operation order dependency rules[1], by analyzing all possible operation combinations. As our file system model includes the *move* operation, these rules deviate from those of related works that omit this operation. We found that our rules can form *cycles* and proposed a mechanism that breaks them by injecting a temporary rename operation. Our approach also works for *bi*-directional synchronization, but requires minor adaptations to compensate for cross-replica side effects of the *move* operation. Our evaluation of leading industrial synchronizers confirmed that our approach substantially improves the efficiency and correctness of synchronization in practice, when several operations accumulate due to long offline periods. Detected *move* operations are retained, and their synchronization is more efficient than re-transmitting files. Syncpal does not exhibit anomalous behavior found in other academic and industrial solutions, such as temporary or permanent synchronization loops which cause divergent file system replicas.

RQ3 - Conflicts: What sets of operations applied to two disconnected file systems are conflicting? How do conflicts depend on the file system model? Can multiple conflicts be combined? What are possible solutions for resolving individual conflicts and conflict combinations? Which conflicts are relevant in practice? How can conflicts be explained to the user?

Conflicts between operations depend on the specific operations offered by the file system model. Using operation precondition analysis we found ten different conflicts types for our internal file system model. We discovered that the lower a file system model's complexity, the fewer conflict types exist. We designed and applied a four-step framework that derives how to *resolve* conflicts. It allows to build a *coherent* and *reflected* set of conflict resolution steps, superior to the *arbitrary* resolution steps we found in related academic works. We designed an *iterative* conflict resolution approach which solves one conflict at a time, manipulating the operation on the losing (rather than the winning) replica. This avoids negative side effects of the resolution on other conflicts which we observed in industrial synchronizers and keeps the implementation of resolving each individual conflict simple. Instead of limiting Syncpal to one specific resolution approach we designed and implemented several criteria for selecting the winner or loser replica. We are also the first in academia to discuss conflict *combinations* where multiple conflicts affect a specific object. To deal with such situations we extended our iterative resolution approach to first sort conflicts according to their type. We built an optimal sort order that maximizes the preservation of the users' intentions by analyzing all possible resolution orders for different conflict type combinations.

We found that Edit-Edit and Create-Create conflicts were most relevant in practice for our test-users. These conflicts were predominantly produced a single application, WSCAD, but not due to *offline* work, but by WSCAD which delays committing the user's work to files in parallel *online* work. We designed a tabular log that explains the conflicts and their resolution to the user in a graphical user interface. While leading industrial synchronizers provide little or no conflict-related information our conflict handling and their visualization improves the conflict recovery process for end-users.

9.2 Limitations

We start with presenting two major limitation that apply to *state-based* file synchronizers in general, followed by providing limitations specific to Syncpal.

The first limitation is the performance hit already encountered during the *update detection* stage. Memory and execution times depend on the number of objects n. Regardless how well an implementation is optimized, a user might want to synchronize a directory that has too many objects for the implementation to handle. To overcome this limitation, users either have to cooperate and synchronize only a sub-tree of the file system, or concepts such as *bubbling modification timestamps* [LKT05] can be used, which allow to identify the specific sub-trees that changed since the last synchronization, so that update detection can be limited to those sub-trees. The second inherent limitation is that files are *black-boxes*

[1] For instance, the *createdir* operation of a parent directory needs to precede the creation of its children.

that make it impossible for the synchronizer to merge conflicting *edit* operations. However, even if the synchronizer understood the data model of a file, automatic merging would still be undesired because it is limited to syntactic merging. Ultimately, merging is an AI-complete problem.

The following is a list of Syncpal- or BSync-specific limitations:

- When incompatibilities between two different file systems are found, BSync does not always implement the most user-friendly alternative to resolve them. For instance, when BSync detects objects with more than one link, it stops and asks the user to delete all but one, instead of adding links an *ignore list*.
- Replicas synchronized by Syncpal may not be arranged in *cyclic* configurations. For instance, given replicas X, Y, Z we may synchronize $X \leftrightarrow Y$ and $Y \leftrightarrow Z$. Thus $X \leftrightarrow Z$ implicitly holds, by transitivity. However, we may not synchronize $X \leftrightarrow Z$ explicitly. This may cause non-conflicting situations to be detected as real conflicts, because operations detected in X may reach Z several times, out of order. To overcome this limitation, Syncpal would need to keep a *history* of operations, to be able to identify out-of-order operations. We ignore this limitation, because not only would a solution incur additional time and space complexity, but most synchronization configurations (in practice) are set up using a client-server *star* topology, where this problem does not occur.
- Because Syncpal cannot lock file systems for exclusive access, there is a chance that concurrent file system modifications made by the user invalidate operations executed by Syncpal. We implemented some checks to minimize this chance, but they cannot entirely eliminate it.
- When a user edits *and* moves/renames a file while being offline, and the *edit* operation changes the file's ID (e.g. to safely replace it), Syncpal's state-based update detection incorrectly detects the two operations as *delete* + *create* operation.

9.3 Future work

Discussions with our users revealed several use-cases where synchronization needs to be *limited* to a sub-set of objects. For instance, when a user has only *read-only* permissions to one of the replicas, the synchronization *direction* should be uni-directional. This requires major adaptations to Syncpal, which is inherently built with bi-directional synchronization in mind. Users also requested the ability to save space, by limiting the synchronization to smaller files or directories. This could be realized by implementing an *ignore list* feature. An even better approach is the use of *placeholder files*, where all files and directories are visible on the user's local disk, but files are empty placeholders that are downloaded (and kept in sync) only on-demand. This approach has also been implemented by various industrial file synchronizers[2]. The *ignore list* is still needed for limitations that cannot be handled by placeholder files, such as ignoring extra links on file systems that support only one link per file, or ignoring files whose names are reserved.

Another requested feature was to improve *control* over the synchronization process such that it is no longer fully automatic. A user may not trust a fully automatic synchronizer [How93], or the connection to the server might be metered. This can be addressed with a user-controlled synchronization mode that is semi-automatic. It holds off from automatically synchronizing until some condition is met. For instance, the user might set a pause interval during which synchronization is completely stopped, or define that update detection stays active but synchronization only starts after the file system's state was quiescent for at least five minutes. Syncpal might also ask the user for confirmation based on a set of rules, e.g. to confirm the synchronization of *delete* operations.

A lot of future work is left to improve *conflict handling*, such as implementing additional user-proposed or cascaded criteria to choose the conflict winner, or retroactively applying different conflict resolutions

[2]See e.g. Microsoft OneDrive's *Files On-Demand* feature, Google's *Drive File Stream* application, or Dropbox's *Smart Sync* feature.

using learning techniques. We also plan to investigate approaches that *avoid* conflicts to begin with. This can be achieved by providing better *awareness* about concurrent user activity, e.g. a warning if another user opened the same file(s) as we did. Pessimistic concurrency control, such as locks, is another alternative. Integrating BSync into the operating system's file manager would allow to use overlay icons to convey status information and a file's lock state and offer custom actions (made available in the context menu), such as the ability to lock a file. Any changes affecting the user interface should be *evaluated with users*, to complement the rather *technical* evaluations we performed in this work.

Finally, some users were dissatisfied with BSync's performance during *propagation*. Propagation was particularly slow when transferring many small files, or transferring a large file where only small portions changed. The performance can be improved significantly by implementing *client capabilities* [BDM15] such as *chunking*, *delta encoding* and *bundling*. We note that this requires changes to both the client and server replicas, where changing the latter might not be possible. As work-around users suggested the ability to prioritize specific files or directories such that they are transferred first.

9.4 Closing words

Optimistic replication has many advantages, most notably improved availability, which enables users to work offline. State-based file synchronizers such as Syncpal are universally applicable, because they allow to replicate the state of *any* third-party application. However, the synchronization granularity stops at the *file* level, ignoring the *data model* of the third-party application that creates those files. There are a few tailored applications whose synchronization is aware of the data model, but they require the application's author to integrate the synchronization process natively into the software. To simplify this process several frameworks have emerged that make integration of data synchronization techniques such as Operational Transformation and CRDTs easier. They range from replicated databases such as Couchbase Lite or CouchDB[3], to web frameworks like Y.js[4] that run in the browser, to SDKs for mobile apps[5]. Mobile apps are an excellent use-case for replication to mitigate negative effects of intermittently losing connectivity due to unreliable cellular networks. However, integrating optimistic replication is inherently difficult. The application's data model and its operations may be *very* elaborate, which in turn requires complicated conflict handling. Many of these applications that integrated data synchronization, such as Google Docs, are known to misbehave under certain conditions and result in a divergent state [BMP17]. Having to transform and store the data model in a replicated database requires additional effort. Due to the increased complexity developers often shy away from using replication and synchronization. Consequently, file-based applications, which ignore these concepts, are here to stay, and file synchronizers will continue to play an important role in users' lives. As we have shown, the synchronizer's lacking ability to deal with the application's data model can be mitigated with the help of usability tweaks, like providing helpful awareness and intelligent conflict handling. Our suggestions are an important first step towards a better user experience for file synchronizers. By sparing users the work and frustration to find and fix synchronization errors, our work avoids cost-intensive iterations in cooperation processes for both academia and industry.

[3]See `https://www.couchbase.com/products/lite` and `http://couchdb.apache.org/`, retrieved July 21, 2019.

[4]See `http://y-js.org/`, retrieved July 21, 2019.

[5]See e.g. `https://aerogear.org/services/data-sync/`, `http://www.ensembles.io/`, `https://firebase.google.com/`, or `https://realm.io/`. Retrieved July 21, 2019.

Bibliography

[ABF08] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. “Interval Tree Clocks”. In: *Principles of Distributed Systems*. Ed. by Theodore P. Baker, Alain Bui, and Sébastien Tixeuil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 259–274. ISBN: 978-3-540-92221-6.

[AC08] Michał Antkiewicz and Krzysztof Czarnecki. “Design Space of Heterogeneous Synchronization”. In: *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 3–46. ISBN: 978-3-540-88643-3. DOI: `10.1007/978-3-540-88643-3_1`. URL: `https://doi.org/10.1007/978-3-540-88643-3_1`.

[Agu+08] Ng Agustina, Fei Liu, Steven Xia, Haifeng Shen, and Chengzheng Sun. “CoMaya. Incorporating Advanced Collaboration Capabilities into 3D Digital Media Design Tools”. In: *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*. CSCW ’08. New York, NY, USA: ACM, 2008, pp. 5–8. ISBN: 978-1-60558-007-4. DOI: `10.1145/1460563.1460566`. URL: `http://doi.acm.org/10.1145/1460563.1460566`.

[Ale+13] Christopher Alexander, Sara Ishikawa, Murray Silverstein, and Max Jacobson. *A pattern language. Towns, buildings, construction*. 36. print. Vol. 2. Center for Environmental Structure series. Ishikawa, Sara, (Author.) Silverstein, Murray, (Author.) New York, NY: Oxford Univ. Press, ca. 2013. XLIV, 1171 S. ISBN: 978-0195019193.

[Amj04] Hasan Amjad. “Combining model checking and theorem proving”. University of Cambridge, Computer Laboratory, 2004. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-601.pdf`.

[AMU12] Mehdi Ahmed-Nacer, Stéphane Martin, and Pascal Urso. “File system on CRDT”. In: *CoRR* abs/1207.5990 (2012).

[And03] George E. Andrews. *The theory of partitions*. Digital printing. Cambridge Mathematical Library. Cambridge: Cambridge university press, 2003. XVI, 255 s. ISBN: 0-521-63766-X.

[Ape+11] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. “Semistructured merge”. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE ’11*. the 19th ACM SIGSOFT symposium and the 13th European conference (Szeged, Hungary). Ed. by Tibor Gyimóthy and Andreas Zeller. New York, New York, USA: ACM Press, 2011, p. 190. ISBN: 9781450304436. DOI: `10.1145/2025113.2025141`.

[App04] Apple Inc. *Technical Note TN1150*. HFS Plus Volume Format. 2004. URL: `https://developer.apple.com/legacy/library/technotes/tn/tn1150.html`.

[App06] Apple Inc. *OS X 10.9 Man Pages: chflags*. 2006. URL: `https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/chflags.1.html`.

[App10] Apple Inc. *OS X 10.9 Man Pages: xattr*. 2010. URL: `https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/xattr.1.html`.

[App17] Apple Inc. *Apple File System Guide - FAQs*. 2017. URL: https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/APFS_Guide/FAQ/FAQ.html.

[Art+06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. "Testing telecoms software with quviq QuickCheck". In: *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. Portland, Oregon, USA: ACM, 2006, pp. 2–10. ISBN: 1-59593-490-1. DOI: 10.1145/1159789.1159792.

[ASB15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Efficient State-Based CRDTs by Delta-Mutation". In: *Networked Systems*. Ed. by Ahmed Bouajjani and Hugues Fauconnier. Cham: Springer International Publishing, 2015, pp. 62–76. ISBN: 978-3-319-26850-7.

[AT16] Marcos K. Aguilera and Douglas B. Terry. "The Many Faces of Consistency". In: *IEEE Data Eng. Bull.* 39 (1 2016), pp. 3–13.

[Bal+16] Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno Preguiça, Rodrigo Rodrigues, Marc Shapiro, and Viktor Vafeiadis. "Geo-Replication. Fast If Possible, Consistent If Necessary". In: *IEEE Data Engineering Bulletin* 39 (1 2016). Computer Science [cs]/Other [cs.OH]Journal articles, p. 12. URL: https://hal.inria.fr/hal-01350652.

[Bao+11] X. Bao, N. Xiao, W. Shi, F. Liu, H. Mao, and H. Zhang, eds. *SyncViews: Toward Consistent User Views in Cloud-Based File Synchronization Services*. 2011 Sixth Annual Chinagrid Conference. 2011 Sixth Annual Chinagrid Conference. 2011. 89-96. ISBN: 1949-131X. DOI: 10.1109/ChinaGrid.2011.35.

[Bas18] Basho. *RIAK Key Value Database Homepage*. 2018. URL: http://basho.com/products/riak-kv/.

[BDM15] Enrico Bocchi, Idilio Drago, and Marco Mellia. "Personal Cloud Storage Benchmarks and Comparison". In: *IEEE Transactions on Cloud Computing* 5 (4 2015). Oct, pp. 751–764. ISSN: 2168-7161. DOI: 10.1109/TCC.2015.2427191.

[BF93] Steve Benford and Lennart Fahlén. "A Spatial Model of Interaction in Large Virtual Environments". In: *Proceedings of the Third Conference on European Conference on Computer-Supported Cooperative Work*. ECSCW'93. Norwell, MA, USA: Kluwer Academic Publishers, 1993, pp. 109–124. ISBN: 0-7923-2447-1. URL: http://dl.acm.org/citation.cfm?id=1241934.1241942.

[BHT97] Richard Bentley, Thilo Horstmann, and Jonathan Trevor. "The World Wide Web as Enabling Technology for CSCW. The Case of BSCW". In: *Computer Supported Cooperative Work (CSCW)* 6 (2 1997). Jun, pp. 111–134. ISSN: 1573-7551. DOI: 10.1023/A:1008631823217. URL: https://doi.org/10.1023/A:1008631823217.

[Bil05] Philip Bille. "A survey on tree edit distance and related problems". In: *Theoretical Computer Science* 337 (1-3 2005), pp. 217–239. ISSN: 03043975. DOI: 10.1016/j.tcs.2004.12.030.

[Bjø07] Nikolaj Bjørner. "Models and Software Model Checking of a Distributed File Replication System". In: *Formal Methods and Hybrid Real-Time Systems*. Ed. by Cliff B. Jones, Zhiming Liu, and Jim Woodcock. Vol. 4700. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–23. ISBN: 978-3-540-75220-2. DOI: 10.1007/978-3-540-75221-9_1.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. Cambridge, Mass.: MIT Press, 2008. 1 online resource (xvii, 975. ISBN: 978-0-262-02649-9.

[BMM94] T. Berners-Lee, L. Masinter, and M. McCahill. *Uniform Resource Locators (URL)*. RFC 1738 (Proposed Standard) RFC dec, Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986, 6196, 6270, 8089. Fremont, CA, USA: RFC Editor and RFC Editor, 1994. DOI: 10.17487/RFC1738. URL: https://www.rfc-editor.org/rfc/rfc1738.txt.

[BMP17] Marina Billes, Anders Møller, and Michael Pradel. "Systematic black-box analysis of collaborative web applications". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017.* the 38th ACM SIGPLAN Conference (Barcelona, Spain). Ed. by Albert Cohen and Martin Vechev. New York, New York, USA: ACM Press, 2017, pp. 171–184. ISBN: 9781450349888. DOI: `10.1145/3062341.3062364`.

[BP98] S. Balasubramaniam and Benjamin C. Pierce. "What is a File Synchronizer?" In: *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking.* MobiCom '98. New York, NY, USA: ACM, 1998, pp. 98–108. ISBN: 1-58113-035-X. DOI: `10.1145/288235.288261`. URL: `http://doi.acm.org/10.1145/288235.288261`.

[BR94] Gordon S. Blair and Tom Rodden. "The Challenges of CSCW for Open Distributed Processing". In: *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing II.* Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co, 1994, pp. 127–140. ISBN: 0-444-81861-8. URL: `http://dl.acm.org/citation.cfm?id=648101.748729`.

[Bre00] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing.* Portland, Oregon, USA: ACM, 2000, p. 7. ISBN: 1-58113-183-6. DOI: `10.1145/343477.343502`.

[Bre12] E. Brewer. "CAP twelve years later: How the rules have changed". In: *Computer* 45 (2 2012), pp. 23–29. ISSN: 0018-9162. DOI: `10.1109/MC.2012.37`.

[Cel+16] Antonio Celesti, Maria Fazio, Massimo Villari, and Antonio Puliafito. "Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems". In: *Journal of Network and Computer Applications* 59 (2016), pp. 208–218. ISSN: 10848045. DOI: `10.1016/j.jnca.2014.09.021`.

[CH06] Yek Loong Chong and Youssef Hamadi. "Distributed log-based reconciliation". In: *ECAI.* Vol. 141. 2006, pp. 108–112.

[CJ05] Russ Cox and William Josephson. *File Synchronization with Vector Time Pairs.* 2005.

[CKD10] George F. Coulouris, Tim Kindberg, and Jean Dollimore. *Distributed systems. Concepts and design.* 4. ed., 6. impr. International computer science series. Harlow [u.a.]: Addison-Wesley, 2010. XIV, 927 S. ISBN: 0321263545.

[Cla08] Edmund M. Clarke. "The Birth of Model Checking". In: *25 Years of Model Checking: History, Achievements, Perspectives.* Ed. by Orna Grumberg and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–26. ISBN: 978-3-540-69850-0. DOI: `10.1007/978-3-540-69850-0_1`. URL: `https://doi.org/10.1007/978-3-540-69850-0_1`.

[CLD16] Y. Cui, Z. Lai, and N. Dai. "A first look at mobile cloud storage services. Architecture, experimentation, and challenges". In: *IEEE Network* 30 (4 2016), pp. 16–21. ISSN: 0890-8044. DOI: `10.1109/MNET.2016.7513859`.

[Cra08] Matt Craighead. *Windows vs. Unix File System Semantics.* 2008. URL: `http://www.conifersystems.com/2008/10/21/windows-vs-unix-file-system-semantics/`.

[Csi16] Elod Csirmaz. *Algebraic File Synchronization. Adequacy and Completeness.* jan, 2016. URL: `https://arxiv.org/pdf/1601.01736.pdf`.

[CSS09] Federico Cabitza, Carla Simone, and Marcello Sarini. "Leveraging Coordinative Conventions to Promote Collaboration Awareness". In: *Comput. Supported Coop. Work* 18 (4 2009), pp. 301–330. ISSN: 0925-9724. DOI: `10.1007/s10606-009-9093-z`.

[DB92] Paul Dourish and Victoria Bellotti. "Awareness and coordination in shared workspaces". In: *Proceedings of the 1992 ACM conference on Computer-supported cooperative work - CSCW '92.* the 1992 ACM conference (Toronto, Ontario, Canada). Ed. by Marilyn Mantel and Ron Baecker. New York, New York, USA: ACM Press, 1992, pp. 107–114. ISBN: 0897915429. DOI: `10.1145/143457.143468`.

[DeC+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo. Amazon's highly available key-value store". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP '07*. twenty-first ACM SIGOPS symposium (Stevenson, Washington, USA). Ed. by Thomas C. Bressoud and M. Frans Kaashoek. New York, New York, USA: ACM Press, 2007, p. 205. ISBN: 9781595935915. DOI: 10.1145/1294261.1294281.

[Dou96] James Paul Dourish. "Open implementation and flexibility in CSCW toolkits". phd. London: University of London, 1996.

[DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[DP08] David Dearman and Jeffery S. Pierce. "It's on my other computer! Computing with multiple devices". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Florence, Italy: ACM, 2008, pp. 767–776. ISBN: 978-1-60558-011-1. DOI: 10.1145/1357054.1357177.

[Dro16] Dropbox Inc. *Celebrating half a billion users*. 2016. URL: https://blogs.dropbox.com/dropbox/2016/03/500-million/.

[Dro17] Dropbox Inc. *Dropbox for HTTP Developers. The Dropbox API v2*. 2017. URL: https://www.dropbox.com/developers/documentation/http/overview.

[Dro18] Dropbox Inc. *Dropbox Homepage*. 2018. URL: https://www.dropbox.com.

[DSL02] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. "Generalizing operational transformation to the standard general markup language". In: *Proceedings of the 2002 ACM conference on Computer supported cooperative work - CSCW '02*. the 2002 ACM conference (New Orleans, Louisiana, USA). Ed. by Elizabeth F. Churchill, Joe McCarthy, Christine Neuwirth, and Tom Rodden. New York, New York, USA: ACM Press, 2002, p. 58. ISBN: 1581135602. DOI: 10.1145/587078.587088.

[Dus07] L. Dusseault. *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. RFC 4918 (Proposed Standard) RFC jun, Updated by RFC 5689. Fremont, CA, USA: RFC Editor and RFC Editor, 2007. DOI: 10.17487/RFC4918. URL: https://www.rfc-editor.org/rfc/rfc4918.txt.

[EG89] C. A. Ellis and S. J. Gibbs. "Concurrency control in groupware systems". In: *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. Portland, Oregon, USA: ACM, 1989, pp. 399–407. ISBN: 0-89791-317-5. DOI: 10.1145/67544.66963.

[EN15] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems*. Pearson, 2015. ISBN: 0133970779.

[Ene+18] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. "Efficient Synchronization of State-based CRDTs". In: *CoRR* abs/1803.02750 (2018).

[EYL13] Frank I. Elijorde, Hyunho Yang, and Jaewan Lee. "Ubiquitous Workspace Synchronization in a Cloud-based Framework". In: *Journal of Korean Society for Internet Information* 14 (1 2013), pp. 53–62. ISSN: 1598-0170. DOI: 10.7472/jksii.2013.14.53.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process". In: *J. ACM* 32 (2 1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121.

[Fos+07] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. "Exploiting schemas in data synchronization". In: *Journal of Computer and System Sciences* 73 (4 2007), pp. 669–689. ISSN: 00220000. DOI: 10.1016/j.jcss.2006.10.024.

[FPP95] Ludwin Fuchs, Uta Pankoke-Babatz, and Wolfgang Prinz. “Supporting Cooperative Awareness with Local Event Mechanisms. The GroupDesk System”. In: *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work ECSCW ’95: 10-14 September, 1995, Stockholm, Sweden*. Ed. by Hans Marmolin, Yngve Sundblad, and Kjeld Schmidt. Dordrecht: Springer Netherlands, 1995, pp. 247–262. ISBN: 978-94-011-0349-7. DOI: 10.1007/978-94-011-0349-7_16. URL: https://doi.org/10.1007/978-94-011-0349-7_16.

[Fra09] Neil Fraser. “Differential synchronization”. In: *Proceedings of the 9th ACM symposium on Document engineering*. Munich, Germany: ACM, 2009, pp. 13–20. ISBN: 978-1-60558-575-8. DOI: 10.1145/1600193.1600198.

[Gam+77] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design patterns. Elements of reusable object-oriented software*. Forty-fourth printing. Addison-Wesley professional computing series. Gamma, Erich, (author.) Helm, Richard (author.) Johnson, Ralph E (author.) Vlissides, John, (author.) Boston, Mass.: Addison-Wesley, 1977. xv, 395. ISBN: 978-0201633610.

[GG02] Carl Gutwin and Saul Greenberg. “A Descriptive Framework of Workspace Awareness for Real-Time Groupware”. In: *Comput. Supported Coop. Work* 11 (3 2002). nov, pp. 411–446. ISSN: 0925-9724. DOI: 10.1023/A:1021271517844. URL: https://doi.org/10.1023/A:1021271517844.

[GHJ07] Alex Groce, Gerard Holzmann, and Rajeev Joshi. “Randomized Differential Testing as a Prelude to Formal Verification”. In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 621–631. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.68.

[Goo18] Google Inc. *Backup and Sync Homepage*. 2018. URL: https://www.google.com/drive/download/backup-and-sync/.

[Got+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. “’Cause I’m strong enough. Reasoning about consistency choices in distributed systems”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg, FL, USA: ACM, 2016, pp. 371–384. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837625.

[GPP93] R. G. Guy, G. J. Popek, and T. W. Page. “Consistency algorithms for optimistic replication”. In: *1993 International Conference on Network Protocols*. 1993 International Conference on Network Protocols (San Francisco, CA, USA). IEEE Comput. Soc. Press, 1993, pp. 250–261. ISBN: 0-8186-3670-X. DOI: 10.1109/ICNP.1993.340912.

[Gri14] Erin Griffith. *Who’s winning the consumer cloud storage wars?* 2014. URL: http://fortune.com/2014/11/06/dropbox-google-drive-microsoft-onedrive/.

[Guy+99] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. “Rumor. Mobile Data Access Through Optimistic Peer-to-Peer Replication”. In: *Advances in Database Technologies*. Ed. by Yahiko Kambayashi, Dik Lun Lee, Ee-Peng Lim, Mukesh Kumar Mohania, and Yoshifumi Masunaga. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 254–265. ISBN: 978-3-540-49121-7.

[Guy91] Richard Guy. “Ficus: A Very Large Scale Reliable Desitributed File System”. Ph.D. dissertation. UCLA, 1991.

[Hac17] Mark Hachmann. *Google Drive is being replaced by Backup and Sync: What to expect*. 2017. URL: https://www.pcworld.com/article/3223136/data-center-cloud/google-drive-is-being-replaced-by-backup-and-sync-what-to-expect.html (visited on 07/21/2019).

[Han+16] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall. “MetaSync. Coordinating Storage across Multiple File Synchronization Services”. In: *IEEE Internet Computing* 20 (3 2016), pp. 36–44. ISSN: 1089-7801. DOI: 10.1109/MIC.2016.44.

[Her+12] Frank Hermann, Hartmut Ehrig, Claudia Ermel, and Fernando Orejas. "Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars". In: *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* Ed. by Juan de Lara and Andrea Zisman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 178–193. ISBN: 978-3-642-28872-2. DOI: `10.1007/978-3-642-28872-2_13`. URL: `http://dx.doi.org/10.1007/978-3-642-28872-2_13`.

[HO48] Carl G. Hempel and Paul Oppenheim. "Studies in the Logic of Explanation". In: *Philosophy of science* 15 (2 1948), pp. 135–175.

[How93] J. H. Howard, ed. *Using reconciliation to share files between occasionally connected computers.* Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III. Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III. 1993. 56-60. DOI: `10.1109/WWOS.1993.348172`.

[Hug+16] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. "Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service". In: *Proceedings, 2016 IEEE International Conference on Software Testing, Verification and Validation. 10-15 April 2016, Chicago, Illinois.* 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST) (Chicago, IL, USA). Los Alamitos, California: IEEE Computer Society, Conference Publishing Services, 2016, pp. 135–145. ISBN: 978-1-5090-1827-7. DOI: `10.1109/ICST.2016.37`.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability. A correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12 (3 1990), pp. 463–492. ISSN: 0164-0925. DOI: `10.1145/78969.78972`.

[JLP13] Nils Jeners, Oleksandr Lobunets, and Wolfgang Prinz. "What groupware functionality do users really use? A study of collaboration within digital ecosystems". In: *2013 7th IEEE International Conference on Digital Ecosystems and Technologies (DEST).* July, 2013, pp. 49–54. DOI: `10.1109/DEST.2013.6611328`.

[JOO15] Tero Jokela, Jarno Ojala, and Thomas Olsson. "A Diary Study on Combining Multiple Information Devices in Everyday Activities and Tasks". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems.* Seoul, Republic of Korea: ACM, 2015, pp. 3903–3912. ISBN: 978-1-4503-3145-6. DOI: `10.1145/2702123.2702211`.

[JP14] Nils Jeners and Wolfgang Prinz. "Metrics for Cooperative Systems". In: *Proceedings of the 18th International Conference on Supporting Group Work.* Sanibel Island, Florida, USA: ACM, 2014, pp. 91–99. ISBN: 978-1-4503-3043-5. DOI: `10.1145/2660398.2660407`.

[JPV02] Trevor Jim, Benjamin C. Pierce, and Jérôme Vouillon. *How to build a file synchronizer.* 2002. URL: `https://www.cis.upenn.edu/~bcpierce/courses/dd/papers/unisonimpl.ps` (visited on 07/21/2019).

[JT75] Paul R. Johnson and Robert Thomas. *Maintenance of duplicate databases.* 1975. URL: `https://tools.ietf.org/html/rfc677`.

[KA10] Bettina Kemme and Gustavo Alonso. "Database replication. A tale of research across communities". In: *Proceedings of the VLDB Endowment* 3 (1-2 2010), pp. 5–12. ISSN: 21508097. DOI: `10.14778/1920841.1920847`.

[KB17] Martin Kleppmann and Alastair R. Beresford. "A Conflict-Free Replicated JSON Datatype". In: *IEEE Transactions on Parallel and Distributed Systems* 28 (10 2017), pp. 2733–2746. ISSN: 1045-9219. DOI: `10.1109/TPDS.2017.2697382`.

[Ker+01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. "The IceCube approach to the reconciliation of divergent replicas". In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing.* Newport, Rhode Island, USA: ACM, 2001, pp. 210–218. ISBN: 1-58113-383-9. DOI: `10.1145/383962.384020`.

[KK17] Abdullah Talha Kabakus and Resul Kara. “A Performance Evaluation of Dropbox in the light of Personal Cloud Storage Systems”. In: *International Journal of Computer Applications* 163 (5 2017), pp. 6–11.

[KKP07] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. “A Formal Investigation of Diff3”. In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science: 27th International Conference, New Delhi, India, December 12-14, 2007. Proceedings*. Ed. by V. Arvind and Sanjiva Prasad. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 485–496. ISBN: 978-3-540-77050-3. DOI: 10.1007/978-3-540-77050-3_40. URL: https://doi.org/10.1007/978-3-540-77050-3_40.

[Kol16] Felix Kollmar. *The Cloud Storage Report – Dropbox Owns Cloud Storage on Mobile*. 2016. URL: https://blog.cloudrail.com/cloud-storage-report-dropbox-owns-cloud-storage-mobile/ (visited on 07/21/2019).

[KS91] P. Kumar and M. Satyanarayanan. *Log-Based Directory Resolution in the Coda File System*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[KS92] James J. Kistler and M. Satyanarayanan. “Disconnected operation in the Coda File System”. In: *ACM Transactions on Computer Systems* 10 (1 1992), pp. 3–25. ISSN: 07342071. DOI: 10.1145/146941.146942.

[KWK03] Brent Byunghoon Kang, R. Wilensky, and J. Kubiatowicz, eds. *The hash history approach for reconciling mutual inconsistency*. 23rd International Conference on Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on Distributed Computing Systems, 2003. Proceedings. 2003. 670-677. ISBN: 1063-6927. DOI: 10.1109/ICDCS.2003.1203518.

[Lam78] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Commun. ACM* 21 (7 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[Li+12a] Q. Li, L. Zhu, S. Zeng, and W. Q. Shang, eds. *An Improved File System Synchronous Algorithm*. 2012 Eighth International Conference on Computational Intelligence and Security. 2012 Eighth International Conference on Computational Intelligence and Security. 2012. 520-524. DOI: 10.1109/CIS.2012.123.

[Li+12b] Qiang Li, Ligu Zhu, Wenqian Shang, and Saifeng Zeng. “CloudSync: Multi-nodes Directory Synchronization”. In: *International Conference on Industrial Control and Electronics Engineering (ICICEE 2012), 2012. Xi’an, China, 23 - 25 Aug. 2012*. 2012 International Conference on Industrial Control and Electronics Engineering (ICICEE) (Xi’an, China). International Conference on Industrial Control and Electronics Engineering and ICICEE. Piscataway, NJ and Piscataway, NJ: IEEE, 2012, pp. 1470–1473. ISBN: 978-1-4673-1450-3. DOI: 10.1109/ICICEE.2012.386.

[Lin01] Tancred Lindholm. “A 3-way Merging Algorithm for Synchronizing Ordered Trees. The 3DM merging and differencing tool for XML”. masterthesis. Helsinki, Finland: Helsinki University of Technology, 2001. URL: https://www.cs.hut.fi/~ctl/3dm/thesis.pdf.

[Lin04] Tancred Lindholm. “A three-way merge for XML documents”. In: *Proceedings of the 2004 ACM symposium on Document engineering*. Milwaukee, Wisconsin, USA: ACM, 2004, pp. 1–10. ISBN: 1-58113-938-1. DOI: 10.1145/1030397.1030399.

[LKT05] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. “A hybrid approach to optimistic file system directory tree synchronization”. In: *The 4th ACM international workshop on Data engineering for wireless and mobile access* (Baltimore, MD, USA). Ed. by Vijay Kumar, Arkady Zaslavsky, Ugur Cetintemel, and Alexandros Labrinidis. New York, NY, USA: ACM, 2005, pp. 49–56. ISBN: 1-59593-088-4. DOI: 10.1145/1065870.1065879.

[LM10] Avinash Lakshman and Prashant Malik. “Cassandra. A decentralized structured storage system”. In: *ACM SIGOPS Operating Systems Review* 44 (2 2010), p. 35. ISSN: 01635980. DOI: 10.1145/1773912.1773922.

[LS90] Eliezer Levy and Abraham Silberschatz. “Distributed file systems. Concepts and examples”. In: *ACM Computing Surveys* 22 (4 1990), pp. 321–374. ISSN: 03600300. DOI: 10.1145/98163.98169.

[LSL04] R. Li, C. Sun, and D. Li. “A Time Interval Based Consistency Control Algorithm for Interactive Groupware Applications”. In: *Parallel and Distributed Systems, International Conference on(ICPADS)*. Vol. 00. 07. 2004, p. 429. DOI: 10.1109/ICPADS.2004.1316123. URL: doi.ieeecomputersociety.org/10.1109/ICPADS.2004.1316123.

[Lv+17] Xiao Lv, Fazhi He, Weiwei Cai, Yuan Cheng, and Yiqi Wu. “CRDT-based Conflict Detection and Resolution for Massive-scale Real-time Collaborative CAD systems”. In: *Proceedings of the 12th Chinese Conference on Computer Supported Cooperative Work and Social Computing - ChineseCSCW ’17*. the 12th Chinese Conference (Chongqing, China). Ed. by Zili Zhang, Ning Gu, Shaozi Li, Tun Lu, and Li Li. New York, New York, USA: ACM Press, 2017, pp. 185–188. ISBN: 9781450353526. DOI: 10.1145/3127404.3127436.

[Lv+18] Xiao Lv, Fazhi He, Yuan Cheng, and Yiqi Wu. “A novel CRDT-based synchronization method for real-time collaborative CAD systems”. In: *Advanced Engineering Informatics* 38 (2018), pp. 381–391. ISSN: 14740346. DOI: 10.1016/j.aei.2018.08.008.

[LZD13] Z. Li, Z. L. Zhang, and Y. Dai. “Coarse-grained cloud synchronization mechanism design may lead to severe traffic overuse”. In: *Tsinghua Science and Technology* 18 (3 2013), pp. 286–297. DOI: 10.1109/TST.2013.6522587.

[Mat+89] Friedemann Mattern et al. “Virtual time and global states of distributed systems”. In: *Parallel and Distributed Algorithms* 1 (23 1989), pp. 215–226.

[Meh+14] Ahmed-Nacer Mehdi, Pascal Urso, Valter Balegas, Nuno Pergui&#231, and a. “Merging OT and CRDT algorithms”. In: *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. Amsterdam, The Netherlands: ACM, 2014, pp. 1–4. ISBN: 978-1-4503-2716-9. DOI: 10.1145/2596631.2596636.

[Men02] T. Mens. “A state-of-the-art survey on software merging”. In: *IEEE Transactions on Software Engineering* 28 (5 2002), pp. 449–462. ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1000449.

[Mic06] Microsoft Inc. *How To Use NTFS Alternate Data Streams*. 2006. URL: https://support.microsoft.com/en-us/help/105763/how-to-use-ntfs-alternate-data-streams.

[Mic18a] Microsoft Inc. *MSDN documentation: File Attribute Constants*. 2018. URL: https://docs.microsoft.com/en-us/windows/win32/fileio/file-attribute-constants.

[Mic18b] Microsoft Inc. *MSDN documentation: Naming Files, Paths, and Namespaces*. 2018. URL: https://docs.microsoft.com/en-us/windows/win32/fileio/naming-a-file.

[Mic19] Microsoft Inc. *OneDrive Homepage*. 2019. URL: https://onedrive.live.com.

[Mol+03] Pascal Molli, Gérald Oster, Hala Skaf-Molli, and Abdessamad Imine. “Using the transformational approach to build a safe and generic data synchronizer”. In: *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*. Sanibel Island, Florida, USA: ACM, 2003, pp. 212–220. ISBN: 1-58113-693-5. DOI: 10.1145/958160.958194.

[Mor+86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. “Andrew. A Distributed Personal Computing Environment”. In: *Commun. ACM* 29 (3 1986). mar, pp. 184–201. ISSN: 0001-0782. DOI: 10.1145/5666.5671. URL: http://doi.acm.org/10.1145/5666.5671.

[MT07] Dahlia Malkhi and Doug Terry. “Concise version vectors in WinFS”. In: *Distributed Computing* 20 (3 2007). Oct, pp. 209–219. ISSN: 1432-0452. DOI: 10.1007/s00446-007-0044-y. URL: https://doi.org/10.1007/s00446-007-0044-y.

[MT94] M. S. Mazer and J. J. Tardo, eds. *A Client-Side-Only Approach to Disconnected File Access*. 1994 First Workshop on Mobile Computing Systems and Applications. 1994 First Workshop on Mobile Computing Systems and Applications. 1994. 104-110. DOI: 10.1109/WMCSA.1994.1.

[Mye86] Eugene W. Myers. “An O(ND) difference algorithm and its variations”. In: *Algorithmica* 1 (1-4 1986), pp. 251–266. ISSN: 0178-4617. DOI: 10.1007/BF01840446.

[Naj16] Mahsa Najafzadeh. “The Analysis and Co-design of Weakly-Consistent Applications”. en. Université Pierre et Marie Curie, 2016. URL: https://hal.inria.fr/tel-01351187/document (visited on 07/21/2019).

[Nex18] Nextcloud GmbH. *Homepage*. 2018. URL: https://nextcloud.com/.

[Nic+16] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. *Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types*. 2016. DOI: 10.1145/2957276.2957310. URL: http://dl.acm.org/ft_gateway.cfm?id=2957310&type=pdf.

[Nid12] Srinivas Nidhra. “Black Box and White Box Testing Techniques - A Literature Review”. In: *International Journal of Embedded Systems and Applications* 2 (2 2012), pp. 29–50. ISSN: 18395171. DOI: 10.5121/ijesa.2012.2204.

[NS16] Agustina Ng and Chengzheng Sun. “Operational Transformation for Real-time Synchronization of Shared Workspace in Cloud Storage”. In: *Proceedings of the 19th International Conference on Supporting Group Work*. Sanibel Island, Florida, USA: ACM, 2016, pp. 61–70. ISBN: 978-1-4503-4276-6. DOI: 10.1145/2957276.2957278.

[NSE18] Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster. “Co-Design and Verification of an Available File System”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Isil Dillig and Jens Palsberg. Cham: Springer International Publishing, 2018, pp. 358–381. ISBN: 978-3-319-73721-8.

[Ntz16] Gian Ntzik. “Reasoning about POSIX File Systems”. PhD thesis, Imperial College London, 2016. URL: https://www.doc.ic.ac.uk/~pg/publications/Ntzik2017Reasoning.pdf.

[Orb18] OrbiTeam Software GmbH & Co KG. *BSCW Social*. 2018. URL: https://www.bscw.de/social/.

[Ost+06a] Gérald Oster, Hala Skaf-Molli, Pascal Molli, and Hala Naja-Jazzar. *Supporting Collaborative Writing of XML Documents*. English. Computer Science [cs]/Other [cs.OH]Reports. 2006. URL: https://hal.inria.fr/inria-00108996.

[Ost+06b] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. “Data consistency for P2P collaborative editing”. In: *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work - CSCW ’06*. the 2006 20th anniversary conference (Banff, Alberta, Canada). Ed. by Pamela Hinds and David Martin. New York, New York, USA: ACM Press, 2006, p. 259. ISBN: 1595932496. DOI: 10.1145/1180875.1180916.

[own19] ownCloud GmbH. *ownCloud Homepage*. 2019. URL: https://owncloud.org.

[Par+83] D. S. Parker, 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”. In: *IEEE Transactions on Software Engineering* SE-9 (3 1983), pp. 240–247. ISSN: 0098-5589. DOI: 10.1109/TSE.1983.236733.

[PB03] Rachel A. Pottinger and Philip A. Bernstein. “Merging models based on given correspondences”. In: *Proceedings of the 29th international conference on Very large data bases - Volume 29*. Berlin, Germany: VLDB Endowment, 2003, pp. 862–873. ISBN: 0-12-722442-4.

[Pie18] Benjamin Pierce. *Unison File Synchronizer - Homepage*. 2018. URL: https://www.cis.upenn.edu/~bcpierce/unison/.

[Pop02] Karl Popper. *The logic of scientific discovery*. London: Routledge Classics, 2002. 513 pp. ISBN: 0-415-27844-9.

[Pre+09] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. "A Commutative Replicated Data Type for Cooperative Editing". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS) (Montreal, Quebec, Canada). IEEE, 2009, pp. 395–403. DOI: `10.1109/ICDCS.2009.20`.

[Pre18] Nuno M. Preguiça. "Conflict-free Replicated Data Types. An Overview". In: *CoRR* abs/1806.10254 (2018).

[Pri17] Rob Price. *Google Drive now hosts more than 2 trillion files*. 2017. URL: `http://www.businessinsider.de/2-trillion-files-google-drive-exec-prabhakar-raghavan-2017-5` (visited on 07/21/2019).

[Pri93] Wolfgang Prinz. "TOSCA Providing Organisational Information to CSCW Applications". In: *Proceedings of the Third Conference on European Conference on Computer-Supported Cooperative Work*. ECSCW'93. Norwell, MA, USA: Kluwer Academic Publishers, 1993, pp. 139–154. ISBN: 0-7923-2447-1. URL: `http://dl.acm.org/citation.cfm?id=1241934.1241944`.

[PSG04] Benjamin C. Pierce, Alan Schmitt, and Michael B. Greenwald. *Bringing Harmony to Optimism - An Experiment in Synchronizing Heterogeneous Tree-Structured Data*. 2004.

[PV04] Benjamin C. Pierce and Jérôme Vouillon. *What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer*. 2004.

[Qia04] Yuechen Qian. "Data synchronization and browsing for home environments". Eindhoven University of Technology, 2004.

[RC01] Norman Ramsey and Elod Csirmaz. "An algebraic approach to file synchronization". In: *the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium* (Vienna, Austria). Ed. by A. Min Tjoa and Volker Gruhn. 2001, p. 175. DOI: `10.1145/503209.503233`.

[Rei+94] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. "Resolving file conflicts in the Ficus file system". In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. Boston, Massachusetts: USENIX Association, 1994, p. 12.

[Rid+15] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. "SibylFS. Formal specification and oracle-based testing for POSIX and real-world file systems". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. Monterey, California: ACM, 2015, pp. 38–53. ISBN: 978-1-4503-3834-9. DOI: `10.1145/2815400.2815411`.

[Rij18] André dos Reis Martins Rijo. "Building Tunable CRDTs". 2018. URL: `https://run.unl.pt/handle/10362/55171`.

[Roh+11] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. "Replicated abstract data types. Building blocks for collaborative applications". In: *Journal of Parallel and Distributed Computing* 71 (3 2011), pp. 354–368. ISSN: 07437315. DOI: `10.1016/j.jpdc.2010.12.006`.

[RSK04] Pamela Ravasio, Sissel Guttormsen Schär, and Helmut Krueger. "In pursuit of desktop evolution". In: *ACM Transactions on Computer-Human Interaction* 11 (2 2004), pp. 156–180. ISSN: 10730516. DOI: `10.1145/1005361.1005363`.

[Sat+90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. "Coda. A highly available file system for a distributed workstation environment". In: *IEEE Transactions on Computers* 39 (4 1990), pp. 447–459. ISSN: 00189340. DOI: `10.1109/12.54838`.

[SCF97] Maher Suleiman, Michèle Cart, and Jean Ferrié. "Serialization of Concurrent Operations in a Distributed Collaborative Environment". In: *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*. GROUP '97. New York, NY, USA: ACM, 1997, pp. 435–445. ISBN: 0-89791-897-5. DOI: `10.1145/266838.267369`. URL: `http://doi.acm.org/10.1145/266838.267369`.

[SE98] Chengzheng Sun and Clarence Ellis. “Operational transformation in real-time group editors. Issues, algorithms, and achievements”. In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. Seattle, Washington, USA: ACM, 1998, pp. 59–68. ISBN: 1-58113-009-0. DOI: 10.1145/289444.289469.

[Sha+11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. English. This research was supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208), and a Google Research Award 2009. Marek Zawirski is a recipient of the Google Europe Fellowship in Distributed Computing, and this research is supported in part by this Google Fellowship. Carlos Baquero is partially supported by FCT project Castor (PTDC/EIA-EIA/104022/2008).) Computer Science [cs]/Other [cs.OH]Reports INRIA UNL U Minho LIP6. Inria – Centre Paris-Rocquencourt and INRIA, 2011. URL: https://hal.inria.fr/inria-00555588.

[Sha+11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-Free Replicated Data Types”. In: *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3. DOI: 10.1007/978-3-642-24550-3_29. URL: https://doi.org/10.1007/978-3-642-24550-3_29.

[She19] Marius Shekow. “Syncpal: A simple and iterative reconciliation algorithm for file synchronizers”. In: *Distributed Applications and Interoperable Systems - IFIP International Federation for Information Processing, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Lingby, Denmark, June 18-21, 2019, Proceedings*. Ed. by José Pereira and Laura Ricci. 2019.

[SLS14] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations. A study guide for the certified tester exam*. Rocky Nook, Inc, 2014.

[SM02] Torsten Suel and Nasir Memon. “Algorithms for Delta Compression and Remote File Synchronization”. In: *Lossless Compression Handbook*. Ed. by Khalid Sayood. Academic Press, 2002.

[SM94] Reinhard Schwarz and Friedemann Mattern. “Detecting causal relationships in distributed computations. In search of the holy grail”. In: *Distributed Computing* 7 (3 1994). Mar, pp. 149–174. ISSN: 1432-0452. DOI: 10.1007/BF02277859. URL: https://doi.org/10.1007/BF02277859.

[SP19] Marius Shekow and Wolfgang Prinz. “A capability analysis of groupware, cloud and desktop file systems for file synchronization”. In: *Proceedings of 17th European Conference on Computer-Supported Cooperative Work-Exploratory Papers. The International Venue on Practice-centred Computing an the Design of Cooperation Technologies*. European Society for Socially Embedded Technologies (EUSSET). 2019. DOI: 10.18420/ecscw2019_ep06. URL: http://dx.doi.org/10.18420/ecscw2019_ep06.

[SRK00] Marc Shapiro, Antony Rowstron, and Anne-Marie Kermarrec. “Application-independent reconciliation for nomadic applications”. In: *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*. Kolding, Denmark: ACM, 2000, pp. 1–6. DOI: 10.1145/566726.566728.

[SS05] Yasushi Saito and Marc Shapiro. “Optimistic replication”. In: *ACM Computing Surveys* 37 (1 2005), pp. 42–81. ISSN: 03600300. DOI: 10.1145/1057977.1057980.

[SS09] David Sun and Chengzheng Sun. “Context-Based Operational Transformation in Distributed Collaborative Editing Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 20 (10 2009), pp. 1454–1470. ISSN: 1045-9219. DOI: 10.1109/TPDS.2008.240.

[Sto15] Storage Networking Industry Association SNMP. *CDMI: Cloud Data Management Interface, v1.1*. 2015. URL: https://www.snia.org/cdmi.

[Sun+04] David Sun, Steven Xia, Chengzheng Sun, and David Chen. “Operational Transformation for Collaborative Word Processing”. In: *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work.* CSCW ’04. New York, NY, USA: ACM, 2004, pp. 437–446. ISBN: 1-58113-810-5. DOI: 10.1145/1031607.1031681. URL: http://doi.acm.org/10.1145/1031607.1031681.

[Sun+18] Chengzheng Sun, David Sun, Agustina, and Weiwei Cai. “Real Differences between OT and CRDT for Co-Editors”. In: *CoRR* abs/1810.02137 (2018).

[Sun+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. “Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems”. In: *ACM Transactions on Computer-Human Interaction* 5 (1 1998). mar, pp. 63–108. ISSN: 10730516. DOI: 10.1145/274444.274447. URL: http://doi.acm.org/10.1145/274444.274447.

[Sur17] Surur. *Microsoft celebrates 10 years of OneDrive, promises many more.* 2017. URL: https://mspoweruser.com/microsoft-celebrates-10-years-onedrive-promises-many/.

[SW13] Stephanie Santosa and Daniel Wigdor. “A field study of multi-device workflows in distributed workspaces”. In: *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing.* Zurich, Switzerland: ACM, 2013, pp. 63–72. ISBN: 978-1-4503-1770-2. DOI: 10.1145/2493432.2493476.

[Tan+15] Haowen Tang, Fangming Liu, Guobin Shen, Yuchen Jin, and Chuanxiong Guo. “UniDrive. Synergize Multiple Consumer Cloud Storage Services”. In: *Proceedings of the 16th Annual Middleware Conference.* Vancouver, BC, Canada: ACM, 2015, pp. 137–148. ISBN: 978-1-4503-3618-5. DOI: 10.1145/2814576.2814729.

[TBM13] John Tang, Jed Brubaker, and Catherine Marshall. “What do you see in the cloud? Understanding the cloud-based user experience through practices”. In: *14th International Conference on Human-Computer Interaction (INTERACT).* Springer. 2013, pp. 678–695.

[Ter+13] Doug Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. “Consistency-based service level agreements for cloud storage”. In: *Proceedings ACM Symposium on Operating Systems Principles.* November. ACM, 2013. URL: https://www.microsoft.com/en-us/research/publication/consistency-based-service-level-agreements-for-cloud-storage/.

[Ter+94] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. “Session guarantees for weakly consistent replicated data”. In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems.* Sep, 1994, pp. 140–149. DOI: 10.1109/PDIS.1994.331722.

[Ter+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. “Managing update conflicts in Bayou, a weakly connected replicated storage system”. In: *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP ’95.* the fifteenth ACM symposium (Copper Mountain, Colorado, United States). Ed. by Michael B. Jones. New York, New York, USA: ACM Press, 1995, pp. 172–182. ISBN: 0897917154. DOI: 10.1145/224056.224070.

[TH10] Linus Torvalds and Junio Hamano. *Git: Distributed Version Control.* 2010. URL: https://git-scm.com.

[TIH19] Tasuku Takahashi, Kengo Imae, and Naohiro Hayashibara. “Conflict-free Multi-user Collaborative Editing System for 3D Models”. In: *Complex, Intelligent, and Software Intensive Systems.* Ed. by Leonard Barolli, Nadeem Javaid, Makoto Ikeda, and Makoto Takizawa. Cham: Springer International Publishing, 2019, pp. 269–279. ISBN: 978-3-319-93659-8.

[TM+96] Andrew Tridgell, Paul Mackerras, et al. “The rsync algorithm”. In: (1996).

[TRN15] Vinh Tao, Vianney Rancurel, and João Neto. “A Name Is Not A Name”. In: *the 6th Asia-Pacific Workshop* (Tokyo, Japan). Ed. by Kenji Kono and Takahiro Shinagawa. 2015, pp. 1–8. DOI: 10.1145/2797022.2797034.

[TSR15] Vinh Tao, Marc Shapiro, and Vianney Rancurel. “Merging Semantics for Conflict Updates in Geo-distributed File Systems”. In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR ’15. New York, NY, USA: ACM, 2015, 10:1–10:12. ISBN: 978-1-4503-3607-9. DOI: 10.1145/2757667.2757683. URL: http://doi.acm.org/10.1145/2757667.2757683.

[UFB10] Sandesh Uppoor, Michail D. Flouris, and Angelos Bilas. “Cloud-based synchronization of distributed file system hierarchies”. In: *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*. 2010, pp. 1–4. ISBN: -2005. DOI: 10.1109/CLUSTERWKSP.2010.5613087. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5613087.

[Vid+00] Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. “Copies convergence in a distributed real-time collaborative environment”. In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work - CSCW ’00*. the 2000 ACM conference (Philadelphia, Pennsylvania, United States). Ed. by Wendy Kellogg and Steve Whittaker. New York, New York, USA: ACM Press, 2000, pp. 171–180. ISBN: 1581132220. DOI: 10.1145/358916.358988.

[vLP16] Albert van der Linde, João Leitão, and Nuno Preguiça. “Delta-CRDTs. Making delta-CRDTs delta-based”. In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data - PaPoC ’16*. the 2nd Workshop (London, United Kingdom). Ed. by Peter Alvaro and Alysson Bessani. New York, New York, USA: ACM Press, 2016, pp. 1–4. ISBN: 9781450342964. DOI: 10.1145/2911151.2911163.

[Voi+06] Stephen Voida, W. Keith Edwards, Mark W. Newman, Rebecca E. Grinter, and Nicolas Ducheneaut. “Share and Share Alike. Exploring the User Interface Affordances of File Sharing”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’06. New York, NY, USA: ACM, 2006, pp. 221–230. ISBN: 1-59593-372-7. DOI: 10.1145/1124772.1124806. URL: http://doi.acm.org/10.1145/1124772.1124806.

[VV16] Paolo Viotti and Marko Vukolić. “Consistency in Non-Transactional Distributed Storage Systems”. In: *ACM Computing Surveys* 49 (1 2016), pp. 1–34. ISSN: 03600300. DOI: 10.1145/2926965.

[Wal+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. “The LOCUS distributed operating system”. In: *Proceedings of the ninth ACM symposium on Operating systems principles - SOSP ’83*. the ninth ACM symposium (Bretton Woods, New Hampshire, United States). Ed. by Jerome Saltzer, Roy Levin, and David Redell. New York, New York, USA: ACM Press, 1983, pp. 49–70. ISBN: 0897911156. DOI: 10.1145/800217.806615.

[Wan+16] Haiyang Wang, Xiaoqiang Ma, Feng Wang, Jiangchuan Liu, Bharath Kumar Bommana, and Xin Liu. “Diving into cloud-based file synchronization with user collaboration”. In: *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)* (Beijing, China). New York, NY, USA: IEEE, 2016, pp. 1–9. DOI: 10.1109/IWQoS.2016.7590396.

[Wik17] Wikipedia. *Comparison of file systems*. 2017. URL: https://en.wikipedia.org/wiki/Comparison_of_file_systems.

[WRB01] An-I A. Wang, Peter Reiher, and Rajive Bagrodia. *Understanding the Behavior of the Conflict-Rate Metric in Optimistic Peer Replication*. Submitted for publication. 2001.

[Yan+16] Chao-Tung Yang, Wen-Chung Shih, Chih-Lin Huang, Fuu-Cheng Jiang, and William Cheng-Chung Chu. “On construction of a distributed data storage system in cloud”. In: *Computing* 98 (1 2016), pp. 93–118. ISSN: 1436-5057. DOI: 10.1007/s00607-014-0399-4. URL: https://doi.org/10.1007/s00607-014-0399-4.

[Zha+14] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “ViewBox. Integrating local file systems with cloud storage services”. In: *FAST’14 Proceedings of the 12th USENIX conference on File and Storage Technologies*. 2014, pp. 119–132.

Appendix A

Appendix

A.1 File system meta-data handling

A.1.1 System-generated meta-data

Meta-data is data that provides further information about objects. Conceptually, each object has a key-value store attached to it, representing its meta-data. Meta-data is not stored as part of the object, but at a separate location. Each file system examined in section 3.1 provides a different set of meta-data, where some values are managed (i.e. changed) only by the system, and some can be altered by the user. The following table gives an overview which system-generated meta-data can be retrieved from each file system.

	Windows	macOS	WebDAV[1]	BSCW	Dropbox
Device id	✓	✓	X	X	X
Object id	✓ (file ID)	✓ (inode)	X	✓ (oid)	✓ (id)
Type	✓	✓	✓ (Resource type)	✓ (Resource type and *class*, indicates e.g. calendar or other types)	✓
Create / last-modified timestamps	✓	✓	✓	✓	✓[2]
File size	✓	✓	✓	✓	✓
Read-only attribute	✓	✓	X	X	X
Hidden attribute	✓	✓	X	X	X
Other attributes or meta-data	archived, compressed, encrypted, omit content search indexing, system file, temporary file, others [Mic18a]	archived, nodump, opaque, append [App06]	Content-type[3], ETag[4]	MD5 checksum (for files), various others, such as owner's username, or event history	SHA-256 checksum, revision (unique string for a specific file, used to detect changes)

Row hints:

- **Device id**: unique identifier of the volume on which an object resides
- **Object id**: unique identifier of an object itself. When the identifier has a special name in the context of the file system, it is mentioned in parentheses.
- **Type**: indicates whether the object is a file, directory, or of another type (e.g. symbolic link)
- **Attributes**: a bitmask of flags. Some of the flags can be changed by the user and user-level applications, others are managed by the file system
  - Read-only / immutable: when set, applications cannot alter the content of a file
  - Hidden / invisible: indicates to file managers to hide the file from the user

A.1.2 Writing arbitrary meta-data

Aside from reading system-generated meta-data, most storage systems support one or more ways to write arbitrary, user-defined meta-data. The following table summarizes this capability:

[1] See section 15 of RFC 4918 for more details. Meta-data enforced by the server is referred to as *live properties* in the RFC.
[2] Separate last modified server + client timestamp.
[3] HTTPs Content-Type header which contains the MIME type, e.g. *application/json*
[4] Entity tag, contains arbitrary content used to determine whether the content of a file changed.

Windows: ✓	Extended Attributes (EA)[5]: Attribute data is stored in the master file table (MFT), values are limited to 64 KB in length, per entry.
	Alternate Data Streams (ADS) [Mic06]: ADS allows to attach additional text files (name and content) of arbitrary length to an existing file (of arbitrary type).
macOS: ✓	xattr [App10]: xattr allows to set key-value pairs. Values can be text or binary data.
	Resource forks[6]: Similar to ADS on Windows, macOS allows to store alternative data streams for a file. The main content is the *data fork*, whereas additional (named) streams are *resource forks*.
WebDAV: ✓	Using the PROPPATCH command (see section 9.2 of [Dus07]), most servers support setting arbitrary meta-data (referred to as *dead properties*) for any resource in XML format.
BSCW: ✓	Via WebDAV's PROPPATCH.
Dropbox: X	Writing arbitrary meta-data is not supported. Note: the documentation mentions an alpha-stage *properties* API, which allows to define a property template (which likely is the property's schema) and then use it to set, update, or remove instances of the template. However, there is no API function to add new templates.

A.1.3 Authorization

Authorization limits actions an authenticated[7] user is allowed to perform on file system objects. We assume that a user is always authenticated by some means provided by the file system API.

[5]See `https://github.com/jschicht/EaTools` for a summary of the mechanism, or e.g. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff625895` for a concrete API function. Retrieved July 21, 2019.

[6]See `http://xahlee.info/UnixResource_dir/macosx.html`, retrieved July 21, 2019.

[7]Authentication refers to unique *identifying* a user, whereas authorization defines what a specific, identified user is *allowed to do*.

Windows	Access control list (ACL) mechanism that allows to selectively grant or deny specific users or groups (identified by the *security identifier*, or *SID*) a set of permissions[8]. When setting new permissions for a directory, one can define whether these are inherited to sub-elements.
macOS	UNIX permissions: a mechanism available to UNIX systems and its descendant such as macOS. The OS has a set of users and groups. Users can be in one or more groups. Each object in the file system is assigned a specific owner (user) and group. Each object has three permission masks assigned to it, one that applies to the assigned owner, one to the assigned group and one to everyone else. The permissions that can be set in a permission mask are *Read*, *Write* and *Execute*. • For files, the meaning of each permission is self-explanatory. *Execute* means that the file is an executable file and the caller is allowed to execute its binary code or script content. If the *Write* permission is not set, this means the content of the file cannot be changed, but the file can still be moved or deleted! • For directories *Read* means that the user is able to list the contents of the dir, *Write* means that the user may move the dir or create, move/rename or delete immediate children inside of it. *Execute* allows to change to the directory and list its contents.
	ACL: macOS has an ACL mechanism that is similar to the one of Windows. See the note below regarding ACLs vs. UNIX permissions.
WebDAV	WebDAV doesn't specify how authentication and authorization has to be implemented. WebDAV is typically offered as a machine-to-machine interface by a backend system that then actually implements user and permission management, such as BSCW or owncloud. Since WebDAV is implemented over HTTP, clients typically authenticate themselves using the *WWW-Authenticate* header to transmit their credentials (see RFC 2617) or by sending session cookies. Authorization mechanisms are then left to the implementation. However, to allow only specific users to modify resources, section 6.2 of [Dus07] advises to use *shared locks*.
BSCW	BSCW offers an elaborate user and permission management system via the *workspaces* and *roles* concepts. A directory becomes a *workspace* if there is at least one user assigned to it via a *role*. A *role* is a set of permissions. There are a few pre-installed roles, such as "Member" (has most permissions, like read and write) or "Restricted member" (has only read permissions) to choose from, and server administrators can modify these roles or create new ones. Roles and their permissions are automatically inherited to sub-objects, unless a sub-object has another role assigned to it for the authenticated user.
Dropbox	While the user always has full permissions for objects in her own directories, Dropbox offers a "can view" permission when *sharing* a folder with another user.

ACLs are generally more powerful than UNIX permissions because permissions can be granted or denied for *multiple* specific users or groups in different ways, whereas permissions can only be set for the owning user and *one* specific group. While ACLs under Windows and macOS are similar, their implementations do differ in detail. The available permissions are slightly different, and on macOS ACL entries cannot automatically be inherited for new objects created within a dir which already has an ACL entry.

[8]The available permissionsre are can be found at `https://technet.microsoft.com/de-de/library/cc753525(v=ws.10).aspx`, retrieved July 21, 2019.

A.2 Alternative file system definitions

A.2.1 File system definition for H-All

This section formally defines the file system *H-All* from table 2.1 on page 16. The main differences to the *NH-MD* file system we formally defined in section 3.2 are:

- *Directories* are objects that store a list of $(ID, name)$ tuples,
- *Files* can have one or more incoming edges in the arborescence,
- The *name* is no longer an attribute of the node but of the edge,
- There are two new operations, $link$ and $unlink$, that add or remove links (edges) between an existing directory and file. $unlink$ replaces $deletefile$ or $deletedir$.

We refer to table A.1 for redefined basic functions.

Function	Description
$list(i)$	Returns the set of (j,ω) tuples of the *immediate* child nodes for the directory node with ID i. $j \in I$ are the IDs of the child nodes, $\omega \in \Sigma^+$ are the corresponding names, with $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$.
$id(i_{parent},\omega)$	Helper function used in $id(path)$. Returns i if $(i,\omega) \in list(i_{parent})$, *error* otherwise.
$name(i)$	No longer defined.
$path(i)$	No longer defined.
$paths(i)$	Performs a tree search and returns the set of paths $\{path_k\}_{k=1}^n$ s.t. $\forall k : id(path_k) = i$.
$basename(path)$	Returns the last list element of *split(path)*, e.g. *basename('/home/user/foo') = 'foo'*. Formally, if we can split $path = \pi/\omega$ s.t. $\pi = parent(path)$ and $\omega = basename(path)$ then the following holds: $path = \pi/\omega \iff \exists i,u \in I, \omega \in \Sigma^+ : (i,\omega) \in list(u) \wedge \{\pi\} = paths(u)$

Table A.1: Functions for working with file system IDs (H-All)

All invariants from section 3.2.1 have to be adapted, except for eq. 3.6 which remains the same. The adapted invariants are as follows:

$$\forall i,j \in I, \omega \in \Sigma^+ : (i,\omega) \in list(j) \implies type(j) = dir \tag{A.1}$$

$$\forall i \in I, \omega \in \Sigma^+ : (i,\omega) \notin list(i) \tag{A.2}$$

$$\forall i,j,k \in I, \omega, v \in \Sigma^+ : j \neq k \wedge type(i) = dir \wedge (i,\omega) \in list(j) \implies (i,v) \notin list(k) \tag{A.3}$$

$$\forall i \in I, \omega \in \Sigma^+ : (i_{root},\omega) \notin list(i) \tag{A.4}$$

$$\forall i \in I \setminus \{i_{root}\} : type(i) \neq error \iff ancestor(i_{root}, i) \tag{A.5}$$

$$\forall i,j,k \in I, \omega, v \in \Sigma^+ : j \neq k \wedge (j,\omega) \in list(i) \wedge (k,v) \in list(i) \implies \omega \neq v \tag{A.6}$$

With an adapted version of $ancestor(i,j)$:

$$ancestor(i,j) = \begin{cases} true & \exists \omega \in \Sigma^+ : (j,\omega) \in list(i) \\ true & \exists \omega \in \Sigma^+, k \in I : (k,\omega) \in list(i) \wedge ancestor(k,j) \\ false & \text{otherwise} \end{cases}$$

The list of operations is shown in table A.2.

Operation	Description, pre- and post-conditions
$link(path, newpath)$, $link(i, u, name)$	For the existing file with ID $i = id(path)$ this operation creates a new link with name $\omega = basename(newpath)$, located in the parent directory node with ID $u = id(parent(newpath))$. Precondition: $type(i) = file \wedge ancestor(i_{root}, u) \wedge type(u) = dir \wedge id(u,\omega) = error$ Postcondition: $(i,\omega) \in list(u)$
$unlink(path)$, $unlink(i, u, name)$	Removes the link to the node with ID i and name $\omega = basename(path)$ from parent directory node with ID $u = id(parent(path))$. If $type(i) = dir$, the directory must be empty! Precondition: $id(path) \neq error \wedge (i,\omega) \in list(u)$ $\wedge(type(i) = file \vee [type(i) = dir \wedge list(i) = \{\}])$ Postcondition: $(i,\omega) \notin list(u) \wedge id(path) = error$
$createdir(path)$, $createdir(i, u, name)$	See table 3.6
$createfile(path)$, $createfile(i, u, name)$	See table 3.6
$move(source, dest)$, $move(i, u, v, name)$	Moves a directory node with ID $i = id(source)$. This may change the parent from ID $u = id(parent(source))$ to $v = id(parent(dest))$ when moving it to a different directory, or the name from $\omega = basename(source)$ to $v = basename(dest)$, or both. Precondition: $type(u) = dir \wedge (i,\omega) \in list(u) \wedge type(v) = dir$ $\wedge type(i) = dir \wedge id(dest) = error \wedge \neg ancestor(i, v)$ Explanation: $\neg ancestor(i, v)$ ensures that the user cannot move a directory to a destination dir below it, e.g. $source$ = '/A' cannot be moved to $dest$ = '/A/x'. Postcondition: $id(source) = error \wedge id(dest) = i$
$edit(path, op)$, $edit(i, op)$	See table 3.6

Table A.2: File system operations

A.2.2 Conflict definitions for H-All

The following list specifies all conflicts that apply to this file system definition. Only those aspects are explained which deviate from section 5.5 describing the conflicts of the *NH-MD* file system. For simplicity, we assume that object IDs are equal on both replicas, and we ignore optimizations such as summarizing a *deletefile* and a *createfile* operation to *edit* or considering two *createdir* operations as pseudo-conflicting.

- **Create-Create:** See *NH-MD*.
  - Definition: $create_X(i_X, u_X, name_X) \otimes create_Y(i_Y, u_Y, name_Y) = [u_X = u_Y] \wedge [name_X = name_Y] \wedge [type_X(i_X) = dir \vee type_Y(i_Y) = dir \vee content_X(i_X) \neq content_Y(i_Y)]$
- **Link-Link:**
  - **Description:** On both replicas a new link to an already synchronized file is created with the same name under the same parent directory.
  - **Associated pattern:** Name clash conflict
  - **Definition:** $link_X(i_X, u_X, name_X) \otimes link_Y(i_Y, u_Y, name_Y) = (i_X \neq i_Y) \wedge (u_X = u_Y) \wedge (name_X = name_Y)$
  - **Violated precondition:** $id(u,\omega) = error$
  - **Resolution:** See name clash conflict. For any *name clash* conflict in the *H-All* file system we suggest that the loser operation is also chosen based on the amount of work of the operations. Exemplary, the increasing order may be: unlink, link, move, create/edit.

- **Link-Unlink:**
  - **Description:** On one replica all links of a synchronized file were removed (s.t. the file is deleted), on the other replica an additional link is created for the same file.
  - **Associated pattern:** Delete conflict
  - **Definition:** $link_X(i_X, u_X, name_X) \otimes unlink_Y(i_Y, u_Y, name_Y) = (i_X = i_Y) \wedge paths_Y(i_Y) = \emptyset$
  - **Violated precondition:** $type(i) = file$
  - **Resolution:** See delete conflict, where occurrences of *edit* or *move* are to be replaced with *link*.
- **Create-Link:**
  - **Description:** On one replica a new file or dir with name *name* is created in parent dir *v*, on the other replica a new link for a synchronized file is created with name *name* in *v*.
  - **Associated pattern:** Name clash conflict
  - **Definition:** $create_X(i_X, u_X, name_X) \otimes link_Y(i_Y, u_Y, name_Y) = (i_X \neq i_Y) \wedge (u_X = u_Y) \wedge (name_X = name_Y)$ with $create := createdir \vee createfile$
  - **Violated precondition:** $id(u,\omega) = error$
  - **Resolution:** See name clash conflict
- **Move-Link:**
  - **Description:** On one replica a directory is moved to directory *v* with new name *name*, on the other replica a new link for a synchronized file is created with name *name* in *v*.
  - **Associated pattern:** Name clash conflict
  - **Definition:** $move_X(i_X, u_X, v_X, name_X) \otimes link_Y(i_Y, u_Y, name_Y) = (i_X \neq i_Y) \wedge (v_X = u_Y) \wedge (name_X = name_Y)$
  - **Violated precondition:** $id(u,\omega) = error$
  - **Resolution:** See name clash conflict
- **Edit-Edit:** See *NH-MD*.
- **Move-Create:** See *NH-MD*. Only affects moved directories.
- **Edit-Delete:**
  - **Description:** On one replica the content of an already synchronized file was changed, on the other replica all links to that file were removed.
  - **Definition:** $edit_X(i_X, op_X) \otimes unlink_Y(i_Y, u_Y, name_Y) = (i_X = i_Y) \wedge paths_Y(i_Y) = \emptyset$
- **Move-Delete:**
  - **Description:** On one replica a directory was moved, on the other replica the only existing link to that directory was removed.
  - **Definition:** $move_X(i_X, u_X, v_X, name_X) \otimes unlink_Y(i_Y, u_Y, name_Y) = (i_X = i_Y)$
- **Move-Move (Source):** See *NH-MD*. Only affects moved directories.
- **Move-Move (Dest):** See *NH-MD*. Only affects moved directories.
- **Move-ParentDelete:**
  - **Description:** On one replica a dir was deleted, on the other replica an object was moved to be an immediate child of the corresponding dir.

  - **Definition:** $move_X(i_X, u_X, v_X, name_X) \otimes unlink_Y(i_Y, u_Y, name_Y) = paths_Y(i_X) \neq \emptyset \land (v_X = i_Y)$

- **Create-ParentDelete:**
  - **Description:** On one replica the only existing link to a dir d was removed, on the other replica a file or directory was created as an immediate children of d.
  - **Definition:** $create_X(i_X, u_X, name_X) \otimes unlink_Y(i_Y, u_Y, name_Y) = (u_X = i_Y)$ with $create := createdir \lor createfile$

- **Move-Move (Cycle):** See *NH-MD*. Only affects moved directories.

A.2.3 File system definition for NED-All

The *NED-All* file system consists of a *non-hierarchical* set of *files* where each file has a ID $i \in I$, a path $p \in P$, byte-content $b \in B$ and the *lastmodified* meta-datum $l \in L$. I is the set of unique IDs, P is the set of valid paths, B is the set of arbitrary byte sequences and L is the set of all valid lastmodified meta-datum values (e.g. $\mathbb{N}$). The file system FS is thus a set of (i, p, b, l) tuples. Directories are not part of the model and can be considered to be just a visualization computed at run-time by a file manager that helps the user to locate files. We define the following helper predicates and functions:

$$i \in FS = \begin{cases} true & \exists p \in P, b \in B, l \in L : (i, p, b, l) \in FS \\ false & \text{otherwise} \end{cases}$$

$$p \in FS = \begin{cases} true & \exists i \in I, b \in B, l \in L : (i, p, b, l) \in FS \\ false & \text{otherwise} \end{cases}$$

$$path(i) = \begin{cases} p & \exists p \in P, b \in B, l \in L : (i, p, b, l) \in FS \\ error & \text{otherwise} \end{cases}$$

$$id(path) = \begin{cases} i & \exists i \in I, b \in B, l \in L : (i, path, b, l) \in FS \\ error & \text{otherwise} \end{cases}$$

$$lastmodified(i) = \begin{cases} l & \exists p \in P, b \in B, l \in L : (i, p, b, l) \in FS \\ error & \text{otherwise} \end{cases}$$

The following two invariants hold for *NED-All*:

$$\forall i, j \in I : i \in FS \land j \in FS \implies [(i = j \land path(i) = path(j)) \lor (i \neq j \land path(i) \neq path(j)] \tag{A.7}$$

$$\forall i \in I, r \in R \nexists j \in I : i \in FS \land j \in FS \land i \neq j \land path(j) = path(i)/r \tag{A.8}$$

with R being the set of *relative paths* (i.e., paths that do not start with *'/'*), such as *'file.ext'* or *'somedir/-file.ext'*. The first invariant expresses that every file must have a different path and ID, while the second one illustrates that if a file exists in FS with path p, there cannot be another file with a path that starts with p but has an non-empty suffix r, because that would mean that p is both a directory and file path at the same time.

The list of operations is shown in table A.3.

Operation	Description, pre- and post-conditions
$createfile(path)$	Creates a file at $path$, generating its ID i and *lastmodified* meta-datum l. Its byte-content $b = \emptyset$ is empty. Precondition: $i \notin FS \wedge [\forall r \in R \nexists q \in P : q \in FS \wedge (q = path/r \vee path = q/r)]$ Postcondition: $(i, path, b, l) \in FS$
$movefile(source, dest)$, $movefile(i, dest)$	Moves the file with ID $i = id(source)$ to destination path $dest$ Precondition: $i \neq error \wedge dest \notin FS$ $\wedge[\forall r \in R \nexists q \in P : q \in FS \wedge (q = dest/r \vee dest = q/r)]$ Postcondition: $id(source) = error \wedge id(dest) = i$
$deletefile(path)$, $deletefile(i)$	Deletes the file with ID $i = id(path)$ Precondition: $i \neq error$ Postcondition: $path \notin FS$
$edit(path, op)$, $edit(i, op)$	Opens a handle for the file with ID $i = id(path)$ for *writing*, performs the operation op (e.g. adding, removing or changing bytes at specific positions within the file), then closes the handle again. Precondition: $i \neq error$. Let $l_{pre} = lastmodified(i)$ Postcondition: $id(path) \neq error \wedge lastmodified(i) \neq l_{pre}$

Table A.3: File system operations

A.2.4 Conflict definitions for NED-All

The following list provides the formal specification of all conflicts that apply to the NED-All file system definition. Descriptions, associated pattern, resolution and other aspects are not repeated, see section 5.5.

- **Create-Create:** $createfile_X(path_X) \otimes createfile_Y(path_Y) = (path_X = path_Y) \wedge [content_X(path_X) \neq content_Y(path_Y)]$
- **Edit-Edit:** See *NH-MD.*
- **Move-Create:** $createfile_X(path) \otimes movefile_Y(i, dest) = (path = dest)$
- **Edit-Delete:** $edit_X(i_X, op_X) \otimes deletefile_Y(i_Y) = (i_X = i_Y)$
- **Move-Delete:** $movefile_X(i_X, dest_X) \otimes deletefile_Y(i_Y) = (i_X = i_Y)$
- **Move-Move (Source):** $movefile_X(i_X, dest_X) \otimes movefile_Y(i_Y, dest_Y) = (i_X = i_Y) \wedge (dest_X \neq dest_Y)$
- **Move-Move (Dest):** $movefile_X(i_X, dest_X) \otimes movefile_Y(i_Y, dest_Y) = (dest_X = dest_Y)$
- **Node-typing:**
  - **Description:** On replica X the user creates or moves a file to path p, on replica Y the user creates/moves a file to some path q where $q = p/\ldots$. Consequently, p points to a file on X but to a directory on Y.
  - **Associated pattern:** Indirect conflict
  - **Definition:** $createormove_X(path_X) \otimes createormove_Y(path_Y) = \exists r \in R : (path_X = path_Y/r) \vee (path_Y = path_X/r)$ with $createormove(path) := createfile(path) \vee movefile(i, path)$
  - **Violated precondition:** $\forall r \in R \nexists q \in P : q \in FS \wedge (q = dest/r \vee dest = q/r)$
  - **Resolution:** One of the paths has to be automatically renamed. We suggest to rename the shorter of the two paths, because this gives higher priority (and therefore stability) to directory paths.

A.3 Proof for impossible move operation cycles

This section addresses pure move operation cycles (see section 4.2.7.2 on page 57). It proves that it is impossible to have cycles that exclusively consist of *move* operations connected only by order dependency rule 8.

To reiterate, rule 8 states that a *move* operation o_i affecting directory *A* has to be executed *after* another *move* operation o_j affecting directory *B* iff in the database snapshot, *B* is below *A*, and in the current snapshot, *A* is below *B*.

Formally, let *O=compute_ops(db, snapshot)*, where *db* and *snapshot* are taken for replica *X* at times t_1 and t_2 respectively. Let $\bar{O}$ be a *list* created from set *O*. Then rule 8 formally states the following:

$$\forall o_i, o_j \in \bar{O}, o_i \neq o_j : optype(o_i) = move \land optype(o_j) = move \land i > j \\ \iff ancestor(db, id(o_i), id(o_j)) \land ancestor(snapshot, id(o_j), id(o_i)) \quad \text{(A.9)}$$

Figure A.1 illustrates an example, where o_j affects path 'a/b' (moved to 'b') and o_i affects path 'a' (moved to 'b/a'), therefore, according to equation A.9, $\bar{O} = [o_j, o_i]$ must hold.

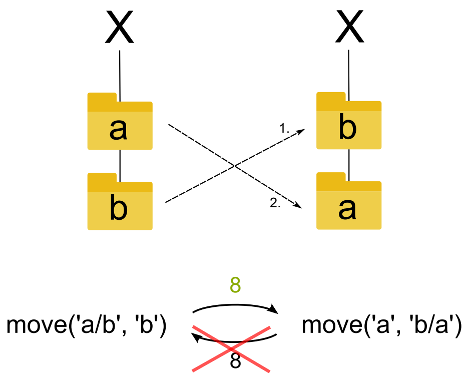


Figure A.1: Move operation cycle example

Theorem 9. *It is impossible to build a cycle* $\bar{C} = [o_k]_{k=1}^{n}$ *which is a subset of* $\bar{O}$*, s.t.* $\forall o_k \in \bar{C} : o_k \in \bar{O} \land type(o_k) = move$ *where equation A.9 is true for each pair of adjacent operations* $(o_j, o_i) \in \bar{C}$ *with*

$$i = \begin{cases} j+1 & \text{if } i < n \\ 1 & \text{otherwise, i.e., } j = n \end{cases}$$

Informally: it is impossible to have move operation cycles whose move operations are connected only by rule 8.

Proof. by contradiction: suppose you built $\bar{C} = [o_1, ..., o_n]$ s.t. it contains *n* move operations where equation A.9 holds for every pair $(o_1, o_2), ..., (o_{n-1}, o_n)$. At this point $\bar{C}$ is still a chain, not a cycle. Since the $ancestor()$ predicate is *transitive*[9], and since equation A.9 holds for these pairs (as stated above) then

$$\forall k \in [2, ..., n] : ancestor(db, id(o_k), id(o_1)) \quad \text{(A.10)}$$

must also hold. To turn $\bar{C}$ into a *closed* cycle, equation A.9 would also have to hold for one specific r s.t. $(o_n, o_r), r \in [1, ..., n-1]$, i.e., $ancestor(db, id(o_r), id(o_n))$ would have to hold. This contradicts with equation A.10, because it's impossible for $ancestor(db, id(o_r), id(o_n))$ and $ancestor(db, id(o_n), id(o_{k=r}))$ to be true at the same time. □

[9] $ancestor(id_X, id_Y) \land ancestor(id_Y, id_c) \implies ancestor(id_X, id_c)$

A.4 Operation reordering methods

Here we present algorithms 5-12. Each function detects and reorders incorrectly ordered operations.

```
def fix_delete_before_move(ops, sorted_ops, db, snapshot):
    foreach op in ops where optype(op) == delete:
        foreach move_op in ops where optype(move_op) == move:
            delete_parent_id = id(db, parent(path(op)))
            (source, dest) = path(move_op)
            move_dest_parent_id = id(snapshot, parent(dest))
            if delete_parent_id == move_dest_parent_id:
                if basename(path(op)) == basename(dest):
                    if sorted_ops.index(op) > sorted_ops.index(move_op):
                        move_first_after_second(move_op, op, sorted_ops)
```

Algorithmus 5 : Pseudo-code for *fix_delete_before_move()*

```
def fix_move_before_create(ops, sorted_ops, db, snapshot):
    foreach op in ops where optype(op) == move:
        foreach create_op in ops where optype(create_op) == create:
            (source, dest) = path(op)
            source_parent_id = id(db, parent(source))
            create_parent_id = id(snapshot, parent(path(create_op)))
            if source_parent_id == create_parent_id:
                if basename(source) == basename(path(create_op)):
                    if sorted_ops.index(op) > sorted_ops.index(create_op):
                        move_first_after_second(create_op, op, sorted_ops)
```

Algorithmus 6 : Pseudo-code for *fix_move_before_create()*

```
def fix_move_before_delete(ops, sorted_ops, db, snapshot):
    foreach op in ops where optype(op) == delete and type(op) == dir:
        foreach move_op in ops where optype(move_op) == move:
            (source, dest) = path(move_op)
            delete_dir_path = path(op)
            if source.startswith(delete_dir_path + '/'):
                if sorted_ops.index(move_op) > sorted_ops.index(op):
                    move_first_after_second(op, move_op, sorted_ops)
```

Algorithmus 7 : Pseudo-code for *fix_move_before_delete()*

```
def fix_create_before_move(ops, sorted_ops, db, snapshot):
    foreach op in ops where optype(op) == create and type(op) == dir:
        foreach move_op in ops where optype(move_op) == move:
            (source, dest) = path(move_op)
            move_dest_parent_id = id(snapshot, parent(dest))
            if move_dest_parent_id == id(op):
                if sorted_ops.index(op) > sorted_ops.index(move_op):
                    move_first_after_second(move_op, op, sorted_ops)
```

Algorithmus 8 : Pseudo-code for *fix_create_before_move()*

```
def fix_delete_before_create(ops, sorted_ops, db, snapshot):
    foreach op in ops where optype(op) == delete:
        foreach create_op in ops where optype(create_op) == create:
            delete_parent_id = id(db, parent(path(op)))
            create_parent_id = id(snapshot, parent(path(create_op)))
            if delete_parent_id == create_parent_id:
                if basename(path(op)) == basename(path(create_op)):
                    if sorted_ops.index(op) > sorted_ops.index(create_op):
                        move_first_after_second(create_op, op, sorted_ops)
```

Algorithmus 9 : Pseudo-code for *fix_delete_before_create()*

```
def fix_move_before_move_occupied(ops, sorted_ops, db, snapshot):
    foreach op in ops where optype(op) == move:
        foreach move_op in ops where optype(move_op) == move:
            if op == move_op:
                continue
            (source, dest) = path(move_op)
            source_parent_id = id(db, parent(source))
            (other_source, other_dest) = path(move_op)
            move_dest_parent_id = id(snapshot, parent(other_dest))
            if source_parent_id == move_dest_parent_id:
                if basename(source) == basename(other_dest):
                    if sorted_ops.index(op) > sorted_ops.index(move_op):
                        move_first_after_second(move_op, op, sorted_ops)
```

Algorithmus 10 : Pseudo-code for *fix_move_before_move_occupied()*

```
def fix_create_before_create(ops, sorted_ops, db, snapshot):
    foreach op in ops where optype(op) == create:
        foreach cd_op in ops where optype(cd_op) == create and type(cd_op) ==
            dir:
            if op == cd_op:
                continue
            if parent(path(op)) == path(cd_op):
                if sorted_ops.index(cd_op) > sorted_ops.index(op):
                    move_first_after_second(op, cd_op, sorted_ops)
```

Algorithmus 11 : Pseudo-code for *fix_create_before_create()*

```
def fix_move_before_move_parent_child_flip(ops, sorted_ops, db, snapshot):
    foreach x_op in ops where optype(x_op) == move and type(x_op) == dir:
        foreach y_op in ops where optype(y_op) == move and type(y_op) == dir:
            if x_op == y_op:
                continue
            (x_source, x_dest) = path(x_op)
            (y_source, y_dest) = path(y_op)
            is_x_below_y = x_dest.startswith(y_dest + '/')
            if not is_x_below_y:
                continue
            is_y_below_x_in_db = y_source.startswith(x_source + '/')
            if is_y_below_x_in_db:
                if sorted_ops.index(y_op) > sorted_ops.index(x_op):
                    move_first_after_second(x_op, y_op, sorted_ops)
```

Algorithmus 12 : Pseudo-code for *fix_move_before_move_parent_child_flip()*

A.5 Finding conflicts in snapshots

A.5.1 Corresponding object id

The following algorithm of function `corresponding_object_id(i, r)` which is given an ID `i` of an object on replica `r`, and finds the ID of the corresponding object (should it exist) on the other replica by using the replica-specific IDs and the closest i_{db}. Ideally, the object with ID `i` is known in the database. If it is not, the algorithm finds the closest parent directory known in the database and attempts to find the corresponding parent, using object *names* to fill the gaps.

```
Global data: snapshot_x, snapshot_y, db: the current state of each replica and the
    database state
Input: i, r: ID and replica identifier for which the ID is given
Output: corresponding ID on the other replica, or error

if r is X:
    snapshot = snapshot_x
    snapshot_other = snapshot_y
    other_replica = Y
else:
    snapshot = snapshot_y
    snapshot_other = snapshot_x
    other_replica = X

id_db = dbid(db, i, r)
if id_db is error:
    names = []  # initialize empty list
    current_path = path(snapshot, i)
    # Find parent dir known in the database:
    while [current_id_db = dbid(db, id(snapshot, current_path), r)] is error:
        names += basename(current_path)
        current_path = parent(current_path)

    names = reverse(names)
    # builds string of all names with '/' in-between
    relative_traversed_path = '/'.join(names)

    corresponding_parent_id = id(db, current_id_db, other_replica)
    corresponding_parent_path = path(snapshot_other, corresponding_parent_id)
    if corresponding_parent_path is error:
        return error
    corresponding_path = corresponding_parent_path + '/' + relative_traversed_path
    corresponding_id = id(snapshot_other, corresponding_path)  # may be error
    return corresponding_id

else:
    other_site_id = id(db, id_db, other_replica)
    return other_site_id
```

A.5.2 Corresponding object id (direct)

This section presents function `cid(i, r)`, a variant of the above algorithm that requires that the object for which to find the corresponding object in the other replica is known to the database.

```
Global data: db: the database state
```

```
Input: i, r: ID and replica identifier for which the ID is given
Output: corresponding ID on the other replica, or error

if r is X:
    other_replica = Y
else:
    other_replica = X

id_db = dbid(db, i, r)
if id_db is error:
    return error
other_site_id = id(db, id_db, other_replica)
return other_site_id
```

A.6 Finding conflicts in update trees

A.6.1 Finding conflicts

The following algorithm is given the update tree structures from the local and remote replica and finds conflicts by iterating over all nodes in a breadth-first approach:

```
Input: local_update_tree , remote_update_tree
Output: list of conflict objects to be sorted and resolved by the resolver

conflicts = []
local_move_dir_nodes = []
remote_move_dir_nodes = []

foreach node in (local_update_tree + remote_update_tree):
    if current_node.is_dir and ChangeEvent.Move in current_node.events:
        if current_node.side is local:
            local_move_dir_nodes += current_node
        else:
            remote_move_dir_nodes += current_node

    if ChangeEvent.Create in node.events and ConflictType.Create_Create not in node.
         conflicts_already_considered:
        conflicts += check_create_create_conflict(node)
    if ChangeEvent.Edit in node.events and ConflictType.Edit_Edit not in node.
         conflicts_already_considered:
        conflicts += check_edit_edit_conflict(node)
    if ChangeEvent.Delete in node.events:
        if node.is_dir:
            conflicts += check_move_parentdelete_conflict(node)
            conflicts += check_create_parentdelete_conflict(node)
        conflicts += check_move_delete_conflict(node
        conflicts += check_edit_delete_conflict(node)
    if ChangeEvent.Move in node.events
        conflicts += check_move_create_conflict(node
        if ConflictType.Move_Move_Dest not in node.conflicts_already_considered:
            conflicts += check_move_move_dest_conflict(node
        if ConflictType.Move_Move_Source not in node.conflicts_already_considered
            conflicts += check_move_move_source_conflict(node

conflicts += determine_move_move_cycle_conflicts(local_move_dir_nodes, remote_move_dir_nodes)
```

Each *Conflict* data structure consists of the conflict type, and the local and remote nodes whose operations are conflicting. The `conflicts_already_considered` list is maintained for every node, to avoid that *symmetric* conflicts, such as Edit-Edit or Move-Move(Source) conflicts are detected twice. The function `corresponding_node_in_other_tree()` is implemented exactly like `corresponding_object_id()` in section A.5.1, but is adapted to work on *update trees* instead of *snapshots*. `corresponding_node_direct()` works like `cid()` in section A.5.2, also adapted to work on *update tree* nodes. The algorithms that find individual conflicts, e.g. `check_create_create_conflict()`, are explained in the subsequent subsections.

A.6.2 Create-Create conflict

```
Input: create_node
Output: conflict (optional)

if first_sync:
    corresponding_parent_node = corresponding_node_in_other_tree(create_node.parent)
else:
    corresponding_parent_node = corresponding_node_direct(create_node.parent)
if corresponding_parent_node == error:
    return
```

```
child_request = ChildNodeRequest.ReturnOnlyNonDelete
corresponding_create_node = corresponding_parent_node.get_child(create_node.name, child_request)

if corresponding_create_node != error and ChangeEvent.Create in corresponding_create_node.events:
    if !is_pseudo_conflict(node):
        conflict = make_create_create_conflict(create_node, corresponding_create_node)
        corresponding_create_node.conflicts_already_considered += ConflictType.Create_Create
        return conflict
```

Finding Create-Create conflicts is straightforward and shown in the algorithm above. Instead of providing pseudo-code for `is_pseudo_conflict()` we refer to the FOL definition in section 5.5.1 on page 80 for more details.

A.6.3 Edit-Edit conflict

```
Input: edit_node
Output: conflict (optional)

corresponding_node = corresponding_node_direct(edit_node)
if corresponding_node != error and ChangeEvent.Edit in corresponding_node.events:
    if !is_pseudo_conflict(edit_node):
        conflict = make_edit_edit_conflict(edit_node, corresponding_node)
        corresponding_node.conflicts_already_considered += ConflictType.Edit_Edit
        return conflict
```

A.6.4 Move-Create

```
Input: move_node
Output: conflict (optional)

move_parent_node = move_node.parent
corresponding_parent_node = corresponding_node_direct(move_parent_node)
if corresponding_parent_node != error:
        child_request = ChildNodeRequest.ReturnOnlyNonDelete
        potential_create_child_node = corresponding_parent_node.get_child(move_node.name,
            child_request)
        if potential_create_child_node != error and ChangeEvent.Create in potential_create_child_node
            .events:
            conflict = make_move_create_conflict(move_node, potential_create_child_node)
            return conflict
```

A.6.5 Edit-Delete

```
Input: delete_node
Output: conflict (optional)

if delete_node.is_dir:
    return

corresponding_edit_node = corresponding_node_direct(delete_node)
if corresponding_edit_node != error and ChangeEvent.Edit in corresponding_edit_node.events:
    conflict = make_edit_edit_conflict(delete_node, corresponding_edit_node)
    return conflict
```

A.6.6 Move-Delete

```
Input: delete_node
Output: conflict (optional)

corresponding_move_node = corresponding_node_direct(delete_node)
```

```
if corresponding_move_node == error or ChangeEvent.Move not in corresponding_move_node.events:
    return

conflict = make_move_delete_conflict(delete_node, corresponding_move_node)
return conflict
```

A.6.7 Move-ParentDelete

To find Move-ParentDelete conflicts we iterate over all those sub-nodes of the corresponding node of the deleted dir which have the Move change-event. If at least one such move node exists, a Move-ParentDelete conflict is found.

```
Input: delete_node (only directory nodes)
Output: conflict (optional)

corresponding_dir_node = corresponding_node_direct(delete_node)

if corresponding_dir_node != error:
    if ChangeEvent.Delete in corresponding_dir_node.events:
        return

    move_nodes = []
    foreach sub_node in corresponding_dir_node.children:
        if ChangeEvent.Move in sub_node.events:
            move_nodes += move_node

    if not move_nodes.empty():
        conflict = make_move_parentdelete_conflict(delete_node, move_nodes)
        return conflict
```

A.6.8 Create-ParentDelete

Finding Create-ParentDelete conflicts works very similar to finding Move-ParentDelete conflicts. Instead of iterating over sub-nodes with a *Move* change-event, we iterate over those with a *Create* change-event.

```
Input: delete_node (only directory nodes)
Output: conflict (optional)

corresponding_dir_node = corresponding_node_direct(delete_node)
if corresponding_dir_node != error:
    if ChangeEvent.Delete in corresponding_dir_node.events:
        return

    create_nodes = []
    foreach sub_node in corresponding_dir_node.children:
        if ChangeEvent.Create in sub_node.events:
            create_nodes += sub_node

    if not create_nodes.empty():
        conflict = make_create_parentdelete_conflict(delete_node, create_nodes)
        return conflict
```

A.6.9 Move-Move (Source)

```
Input: move_node
Output: conflict (optional)

corresponding_move_node = corresponding_node_direct(move_node)
```

```
if ChangeEvent.Move not in corresponding_move_node.events:
    return
if corresponding_move_node != error:
    corresponding_move_node.conflicts_already_considered.append(ConflictType.Move_Move_Source)
    if move_node.name != corresponding_move_node.name or move_node.parent != corresponding_move_node.
        parent:
        conflict = make_move_move_source_conflict(move_node, corresponding_move_node)
        return conflict
```

A.6.10 Move-Move (Dest)

```
Input: move_node
Output: conflict (optional)

node_parent_in_other_tree = corresponding_node_direct(move_node.parent)
if node_parent_in_other_tree != error:
        child_request = ChildNodeRequest.ReturnOnlyNonDelete
        potential_move_child = node_parent_in_other_tree.get_child(move_node.name, child_request=
            child_request)
        if potential_move_child != error && ChangeEvent.Move in potential_move_child.events &&
            potential_move_child.i_db != move_node.i_db:
            conflict = make_move_move_dest_conflict(move_node, potential_move_child)
            potential_move_child.conflicts_already_considered.append(ConflictType.Move_Move_Dest)
            return conflict
```

A.6.11 Move-Move (Cycle)

```
Input: local_move_dir_nodes, remote_move_dir_nodes
Output: list of conflicts (optional)

conflicts = []

for local_node in local_move_dir_nodes:
    for remote_node in remote_move_dir_nodes:
        if local_node.i_db == remote_node.i_db:
            continue
        local_db_path = local_db_snapshot.get_relative_path_for_db_id(local_node.i_db)
        remote_db_path = remote_db_snapshot.get_relative_path_for_db_id(remote_node.i_db)
        if local_db_path.startswith(remote_db_path + '/') or remote_db_path.startswith(local_db_path
            + '/'):
            continue
        # The paths are independent. Check if they are now in a cyclic relationship
        corresponding_local_node = corresponding_node_direct(remote_node)
        corresponding_remote_node = corresponding_node_direct(local_node)
        if is_a_below_b(a=local_node, b=corresponding_local_node) and is_a_below_b(a=remote_node, b=
            corresponding_remote_node):
            conflicts += make_move_move_cycle_conflict(local_node, remote_node)

return conflicts
```

A.7 Resolving conflicts in update trees

A.7.1 Undoing a move

The `undo_move(move_node)` function is used in the resolution of Move-Delete, Move-Move(Source), Move-Move(Cycle) and Move-ParentDelete conflicts. Table A.4 shows caveats that need to be considered. They are illustrated by example that starts with the initial situation shown in figure A.2, attempting to undo the operation *move("A/B", "B_moved")*.

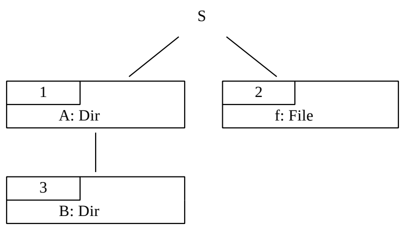


Figure A.2: Initial situation for illustrating problems for `undo_move()`

The algorithm solving all these issues is shown below:

Pseudo-code for `undo_move()`:

```
Input: move_node, well-known root_dir_id
Output: a Move operation that undoes the move of move_node


origin_dir_node = move_node.get_move_origin_parent_node()
move_origin_filename = basename(move_node.move_origin)
is_undoing_move_possible = True
if is_a_below_b(a=origin_dir_node, b=move_node):
    is_undoing_move_possible = False
elif ChangeEvent.Delete in origin_dir_node.events:
    is_undoing_move_possible = False
else:
    potential_origin_node = origin_dir_node.get_child(move_origin_filename, child_request=
        ChildNodeRequest.ReturnOnlyNonDelete)
    if potential_origin_node != error and potential_origin_node.change_events.contains_any([
        ChangeEvent.Create, ChangeEvent.Move]):
        is_undoing_move_possible = False

target_parent_id = origin_dir_node if is_undoing_move_possible else root_dir_id
target_name = move_origin_filename if is_undoing_move_possible else add_conflict_suffix(
    move_origin_filename)
also_update_db = not is_undoing_move_possible

return MoveOperation(move_node, target_parent_id, target_name, also_update_db)
```

A.7.2 Move-Delete

```
Input: move_node, delete_node
Output: MoveOperation or DeleteOperation that resolves the conflict

if configured option == Move Wins:
    deleted_child_nodes = delete_node.get_child_nodes_recursively()
    deleted_child_node_db_ids = []
    foreach node in deleted_child_nodes:
```

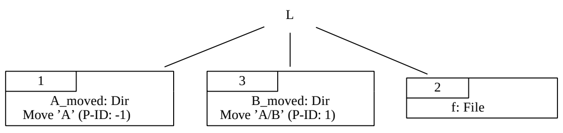
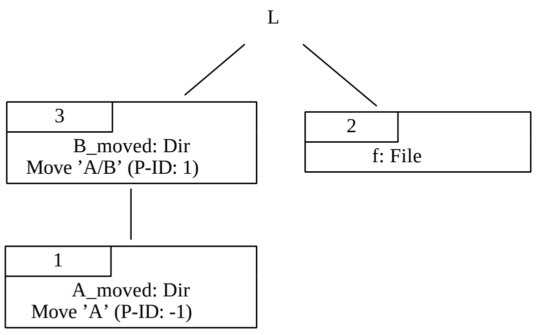
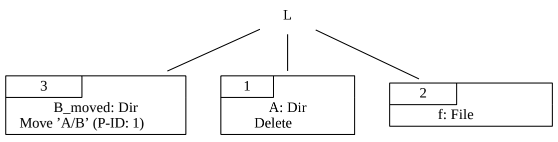
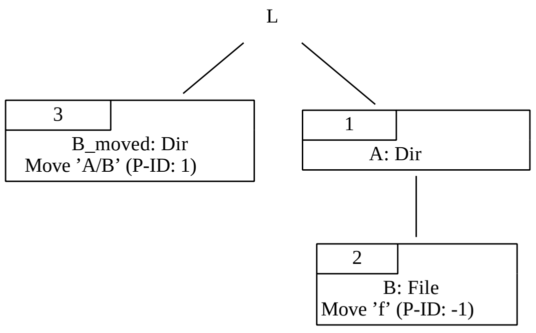
Caveat	Illustration	Resolution
Path of move origin parent dir has changed	 L 1 A_moved: Dir Move 'A' (P-ID: -1) 3 B_moved: Dir Move 'A/B' (P-ID: 1) 2 f: File	Use IDs to locate the current path of the move origin parent dir, instead of the path from the snapshot
Move origin parent dir is now below the move node	 L 3 B_moved: Dir Move 'A/B' (P-ID: 1) 2 f: File 1 A_moved: Dir Move 'A' (P-ID: -1)	Move the object whose move should be undone to the root directory, appending a unique conflict suffix.
Move origin parent dir was deleted	 L 3 B_moved: Dir Move 'A/B' (P-ID: 1) 1 A: Dir Delete 2 f: File	
The original name of the object under the origin parent dir is already in use	 L 3 B_moved: Dir Move 'A/B' (P-ID: 1) 1 A: Dir 2 B: File Move 'f' (P-ID: -1)	

Table A.4: Caveats to consider in `undo_move()`

```
            deleted_child_node_db_ids += node.i_db
        db_modifications = []
        if delete_node.is_dir:
            all_child_node_db_ids = database.get_child_node_db_ids(delete_node.i_db)
            orphan_node_db_ids = all_child_node_db_ids − deleted_child_node_db_ids
            for o_db_id in orphan_node_db_ids:
                orphan_node = delete_node.root.get_node_by_db_id(o_db_id)
                new_name = orphan_node.name() + get_conflict_suffix()
                db_modifications += database.build_update_path_query_for_db_id(o_db_id,
                    new_name)
            winner_side = delete_node.side
            conflict_resolver.register_orphans(orphan_node_db_ids, winner_side)
        db_modifications += database.build_delete_rows_query(delete_node.i_db)
        db_modifications += database.build_delete_rows_query(deleted_child_node_db_ids)
        return DeleteOperation(delete_node, db_modifications, omit=True)
else:
    # delete wins
    if delete_node.is_dir:
        return undo_move(move_node)
    else:
        db_modifications = database.build_delete_rows_query(delete_node.i_db)
        return DeleteOperation(delete_node, db_modifications, omit=False)
```

The necessary steps for resolving Move-Delete conflicts are shown in the above algorithm. While the steps are straightforward in case the strategy is “delete wins”, the steps for “move wins” are more involved. As outlined in subsection 5.5.5, the goal is that if the *delete* replica moved objects from below *delete_node* to a location outside of *delete_node* prior to deleting the directory, that these moves should be synchronized on the mover’s replica eventually.

To realize this, our implementation starts by marking the rows for *delete_node* and all those child nodes that were actually deleted on the deleter’s replica to be removed from the database (removal of these rows is executed by our propagator component when executing the `DeleteOperation`). Those corresponding files and dirs that still physically exist on the mover’s replica will therefore be detected as new and be synchronized in subsequent sync iterations. The deletions of rows is problematic, however, in cases where the delete replica moved objects outside of *delete_node*. Their rows should remain in the database, but they would become *orphaned*, as they lack some of their parent paths. Not fixing these orphan paths can lead to multiple issues in consecutive synchronization iterations. We therefore bend the paths of orphans to a unique path (with conflict suffix) on the *root* level in the database. For instance, an orphan node with path “`/dir_that_deleter_replica_deleted/some_dir`” is set to “`/some_dir-conflict-<datetime>-<random string>`”. This will cause a Move-Move(Source) conflict in the subsequent sync iteration for each orphan node, because physically the corresponding file or dir isn’t at this location on either replica. Since our goal is that the orphans end up in the same location as they are on the deleter’s replica, we temporarily register the affected nodes in an in-memory *registry* so that the resolver automatically resolves them in favor of the deleter’s replica (because it finds the nodes the registry). This overrules the default resolution option for Move-Move (Source) conflicts in this case. Move-Move (Source) conflicts resolved in this way are not presented as conflicts to the user, because it wasn’t the user who produced them, but the synchronizer.

A.8 File synchronizer comparative test details

A.8.1 Conflict-free operations

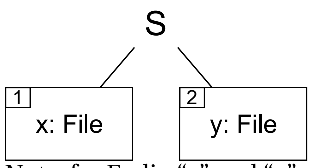
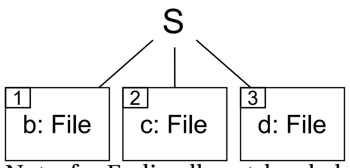
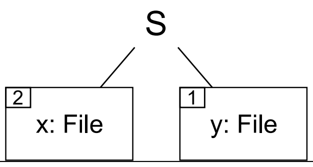
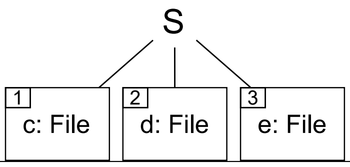
A.8.1.1 Result remarks

- A) OneDrive's macOS *remote* client does not pick up the individual *delete* operation while the client is *online* in a timely manner. The operation is only synchronized either when waiting for 30-120 seconds, restarting the remote client, or when performing additional *create* or *move* operations on the *remote replica*.
- B) Low network traffic indicates that *move* operations are detected by the remote and local Dropbox client correctly, but the local client does generally *not* move objects (in any test!). It instead creates a copy of the object at the destination, followed by deleting the source (or moving it to an internal cache). This behavior is particularly inefficient if the *move* operation targets a large file or a directory with many (large) sub-files.
- C) When the *local* client of Google Backup and Sync is on a macOS VM and the *move* operation targets a non-empty *directory*, the local client does not move it but instead creates a new directory at the target location, moves all immediate child objects of the source directory into the target location and finally deletes the source directory. If the local client is on a Windows VM, or when moving a *file*, the client moves the object as expected.
- D) This test (and follow-up tests) shows that Unison does not support *move* operations, which is also described in their corresponding paper [BP98]. The *remote* client detects the *move* operation as *delete* operation of the source file and an additional *create* operation at the destination path. Similar to Dropbox (see remark B), Unison maintains a temporary cache of files during synchronization, s.t. the *local client* actually *copies* the moved object instead of deleting and retransmitting it.
- E) Various issues occur, depending on the concrete test. For some tests, file system objects have a " (1)" appended to their name in one replica but not in the other. In other tests, the remote client GUI indicates errors but states that the cause is "unknown". In some tests, the replica's final state only becomes synchronized when restarting both clients. More detailed results are found in the appendix section A.8.1.2.
- F) Dropbox duplicates directory structures, see appendix section A.8.1.3.
- G) Final file system structure contains two files: the original file at the destination path and the updated file at the source path.
- H) Since the *remote* client synchronizes the *move* operation as *delete* + *create* operation to the server replica, the *local client* detects an *Edit-Delete* conflict for that file and asks the user to resolve the conflict.

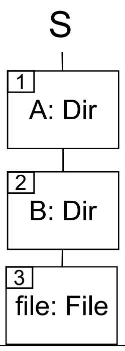
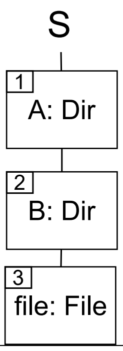
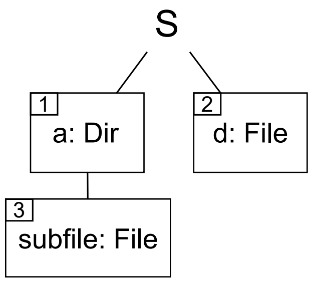
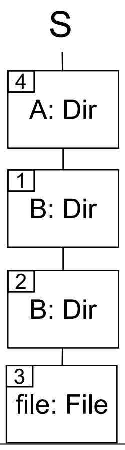
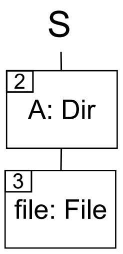
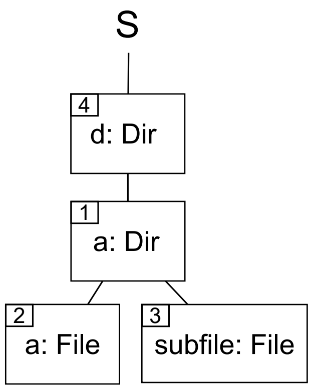
A.8.1.2 Complex single-replica operations

Here we present the five concrete tests whose collection we labeled "Complex single-replica operations", together with the detailed results.

A.8.1.2.1 Test description The following two tables provide a description of the five tests:

Name	Move-Swap	Move-Chain
Description	Two objects are swapped by applying three *move* operations. A state-based update detector computes only *two* operations, which the client cannot apply to the other replica without special handling.	The names of three objects are shifted along the alphabet using chained move operations. The client on the other replica must apply the operation chain in the same order. We tested *forward* and *backward* shifting to find out whether internal client sorting correctly deals with both directions.
Parameters	O, F	O, F
Base scenario	 Note, for F=dir, "x" and "y" are directories with each having a subfile "x" and "y" respectively.	 Note, for F=dir, all root-level objects are directories with each having a subfile "b", "c" and "d" respectively.
Applied operations	*Move(x, temp), Move(y, x), Move(temp, y)*	*Move(d, e), Move(c, d), Move(b, c)*
Computed operations (state-based)	*Move(x, y), Move(y, x)*	*Move(d, e), Move(c, d), Move(b, c)*
Expected result		

The remaining three tests are labeled *"Move-Occupied"* because they involve *move* operations to destinations that are already occupied in the replica. The applied operations are challenging for a synchronizer to apply, in particular if parameter O=*offline*.

Name	Move-Occupied 1	Move-Occupied 2	Move-Occupied 3
Parameters	O	O	O
Base scenario			
Applied operations	*Move (A, temp), CreateDir(A), Move(temp, A/B)*	*Move(A, temp), Move(temp/B, A), Delete(temp)*	*Move(d, a/a), CreateDir(d), Move(a, d/a)*
Computed operations (state-based)	*Move(A, A/B), CreateDir(A)*	*Move(A/B, A), Delete(A)*	*CreateDir(d), Move(a, d/a), Move(d, d/a/a)*
Expected result	 S 4 A: Dir 1 B: Dir 2 B: Dir 3 file: File		 S 4 d: Dir 1 a: Dir 2 a: File 3 subfile: File

Refer to figure 4.1 on page 51 for an explanation of the update tree figures used throughout this chapter.

A.8.1.2.2 Results The following table shows detailed results. It only contains those file synchronizers which showed unexpected behavior in at least one of the tests.

Name	Param.	Dropbox	Backup and Sync	NextCloud
Move-Swap, see figure A.3 for details	O=online, F=file	✓ A)	X Name mismatch	✓ B)
	O=online, F=dir	✓ A)	X Name mismatch	✓
	O=offline, F=file	✓ A)	X Out of sync, error message C)	✓ D)
	O=offline, F=dir	✓ A)	Name mismatch	✓
Move-Chain, see figure A.4 for details	O=online, F=file	✓ A)	X Name mismatch	✓ D)
	O=online, F=dir	✓ A)	X Name mismatch	✓ D)
	O=offline, F=file	X Out of sync	X /✓ Requires client restart E)	X Out of sync
	O=offline, F=dir	X Out of sync	X Name mismatch	X Out of sync
Move-Occupied 1, see figure A.5 for details	O=online	✓ A)	X Name mismatch	✓
	O=offline	✓ A)	X Out of sync, remote client detects no changes F)	✓
Move-Occupied 2, see figure A.6 for details	O=online	✓ A)	X Name mismatch	✓
	O=offline	✓ A)	X Name mismatch (macOS) / client crash (Windows)	✓
Move-Occupied 3, see figure A.7 for details	O=online	✓ A)	X Name mismatch	✓ B)
	O=offline	✓ A)	X /✓ Requires client restart G)	✓ D)

✓ indicates that the final file system structure matches the expected one and that no file payloads were unnecessarily transmitted (unless a remark states otherwise).

Remarks:

- *Name mismatch*: indicates that the final file system structure on the *local* and *remote* replica is generally equal, but the *names* of some individual nodes on the *local* replica do not match the corresponding ones on the *remote* replica.
- A) Although the final structure matches the expected one, this was *not* achieved by the local client *moving* the same set of objects on the *local replica* as our test code did on the *remote replica*. Instead, the local client *copies* files on the *local* replica, and leaves directories in place, only moving sub-files where necessary.
- B) The *remote* client correctly uploads just meta-data (containing the *move* operations), but the *local* client re-downloads file payloads of both files (which is inefficient).
- C) The *remote* client GUI displays an error message “Can’t sync 2 files: an unknown error occurred.”, where a error-details-dialog reveals an “Upload Error” for both files.

- D) The *remote* client uploads the payload of all files (inefficient), consequently the *local* client also re-downloads file payloads.
- E) After starting the *remote* client, it only transmits a sub-set of the operations (which ones depend on the host-platform and whether the chain applies a forward or backward shift) to the *server* replica. The GUI displays an error message "Can't sync 2 files". The synchronization only recovers (and produces a correct result) after restarting the remote client. This includes (re-)transmission of file payloads by both the *local* and *remote* client.
- F) The *remote* client does not detect (or transmit) any changes. Consequently the *local* replica remains unchanged.
- G) After starting the *remote* client, it only transmits the *Move(d, a/a)* operation. The synchronization only recovers (and produces a correct result) after restarting the remote client.

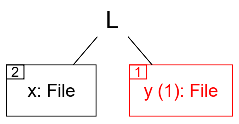
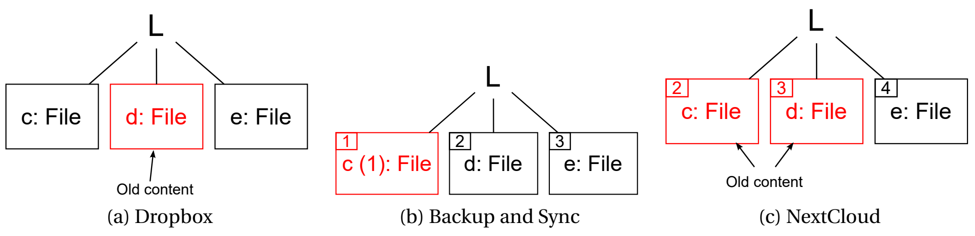


Figure A.3: Incorrect result for test Move-Swap, Backup and Sync

The result shows that the local Backup and Sync client attempts the *Move(x, y)* operation, but y is already occupied, thus it falls back to *Move("x", "y (1)")*, followed by *Move("y", "x")*. The local client keeps this inconsistent "y (1)" name indefinitely. The *remote* replica remains unchanged.
Note: any *red* nodes shown in the remainder of this chapter indicate incorrect nodes.



(a) Dropbox (b) Backup and Sync (c) NextCloud

Figure A.4: Incorrect results for test Move-Chain

Subfigures a-c show the incorrect results on the *local* replica of the Move-Chain test for the corresponding clients. Subfigure *a* lacks IDs because Dropbox never moves objects but instead copies them. Consequently, files lose their (file system) ID in any case. The corresponding *remote* replica remains unchanged.

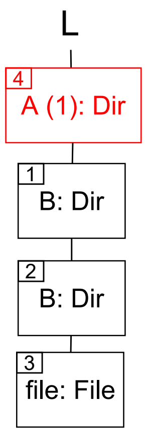
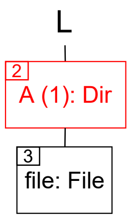
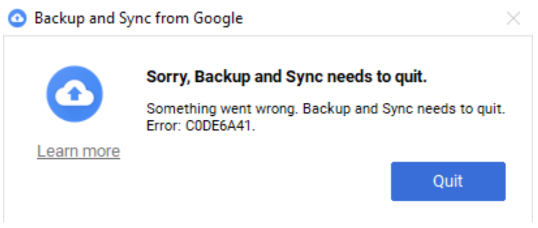


Figure A.5: Incorrect results for test Move-Occupied 1, Backup and Sync

We note that the corresponding *remote* replica remains unchanged.



(a) Name mismatch



Sorry, Backup and Sync needs to quit.

Something went wrong. Backup and Sync needs to quit.
Error: C0DE6A41.

(b) O=Offline (Windows) crash dialog

Figure A.6: Incorrect results for test Move-Occupied 2, Backup and Sync

Subfigure *a* shows the name mismatch produced for parameter value O=online (for *local* client running on macOS or Windows) or O=offline (given that the *local* client runs on macOS). The corresponding *remote* replica remains unchanged. Subfigure *b* shows the crash dialog of the *local* client if O=offline and *local* client runs on Windows. In this case, both *local* and *remote* replica remain unchanged and are permanently out of sync.

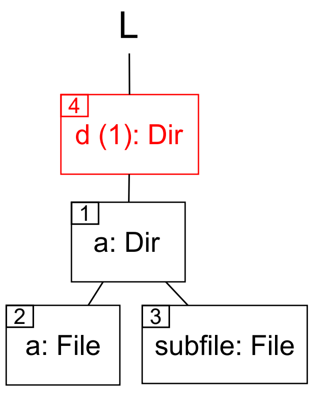


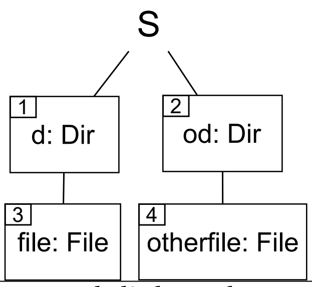
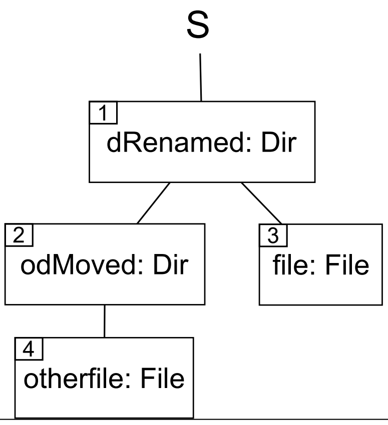
Figure A.7: Incorrect results for test Move-Occupied 3, Backup and Sync

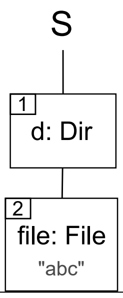
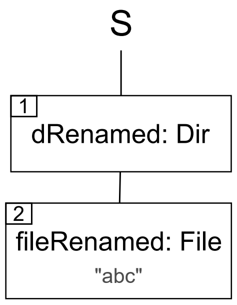
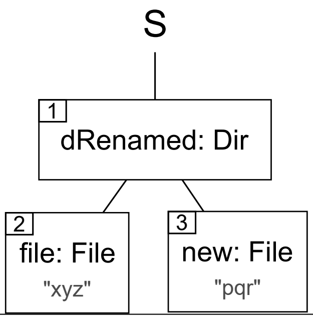
We note that the corresponding *remote* replica remains unchanged.

A.8.1.3 Multi-level operations

Here we present the three concrete tests whose collection we labeled “Multi-level operations”, together with the detailed results.

A.8.1.3.1 Test description The following two tables provide a description of the tests:

Name	Rename parent move outside to inside
Description	One operation set renames the parent directory at path “d”, the other operation set moves another directory “od” into the parent directory. If parameter D=distributed, the *rename* operation is applied to the *remote replica* and the *move* operation to the *local replica*.
Parameters	O, D
Base scenario	 S 1 d: Dir 2 od: Dir 3 file: File 4 otherfile: File
Operation set 1	*Move(od, d/odMoved)*
Operation set 2	*Move(d, dRenamed)*
Expected result	 S 1 dRenamed: Dir 2 odMoved: Dir 3 file: File 4 otherfile: File

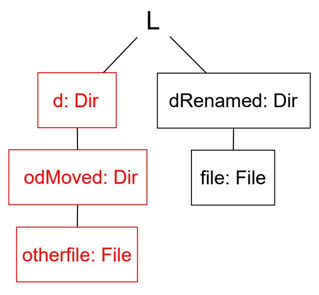
Name	Rename parent and sub-file	Rename parent and update/create sub-file
Description	One operation set renames the parent directory at path "d", the other operation renames a sub-file within the parent directory. If parameter value D=distributed, the parent *rename* operation is applied to the *remote* replica and the sub-file *rename* operation to the *local* replica.	One operation set renames the parent directory at path "d" on the *local* replica, the other operation set edits the content of a sub-file and creates a new one within the parent directory, on the *remote* replica.
Parameters	O, D	O
Base scenario		
Operation set 1	*Move(d, dRenamed)*	
Operation set 2	*Move(d/file, d/fileRenamed)*	*Edit(d/file), CreateFile(d/newfile)*
Expected state		 S 1 dRenamed: Dir 2 file: File "xyz" 3 new: File "pqr"

A.8.1.3.2 Results The following table shows detailed results. It only contains those file synchronizers which showed unexpected behavior in at least one of the tests.

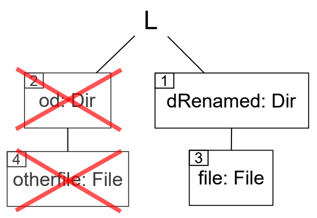
Name	Param.	Dropbox	Backup and Sync	NextCloud	Unison
Rename parent move outside to inside, see figure A.8 for details	O=any, D=same replica	✓	✓	✓	✓
	O=any, D=distributed	X Duplicated structure	X / ✓ Requires client restart A)	X Out of sync B)	X Detects Create-ParentDelete conflict C)
Rename parent and sub-file, see figure A.9 for details	O=any, D=same replica	✓	✓	✓	✓
	O=any, D=distributed	X Duplicated structure	✓	X Duplicated structure	X Duplicated structure
Rename parent and update/create sub-file, see figure A.10 for details	–	X Duplicated structure	✓	✓	X Duplicated structure

Remarks:

- A) After starting the *local* client, it only deletes “od”. The synchronization only recovers (and produces a correct result) after restarting the local client.
- B) The *local* client displays an error “od: Unknown error” for a brief period. But the error message then disappears, leaving the *local* replica in an inconsistent state.
- C) The *local* client detects “od” as remotely deleted and “d” as locally deleted but remotely as “changed” (due to the remotely applied *move* operation into “d”). This is equivalent to detecting a *Create-ParentDelete* conflict. The final result depends on how the user chooses to resolve the conflict.



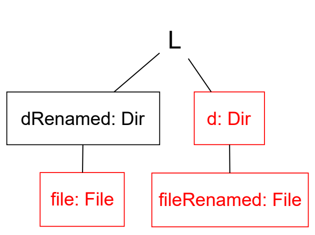
(a) Dropbox



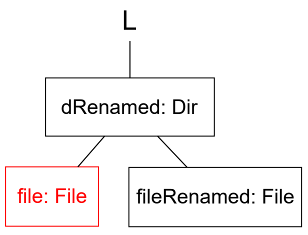
(b) NextCloud

Figure A.8: Incorrect results for test ’Rename parent move outside to inside’

Subfigures a-b show the incorrect results on the *local* replica of the test *Rename parent move outside to inside* for the corresponding clients.



(a) Dropbox, Unison



(b) NextCloud

Figure A.9: Incorrect results for test 'Rename parent and sub-file'

Subfigures a-b show the incorrect results on the *local* replica of the test *Rename parent and sub-file* for the corresponding clients.

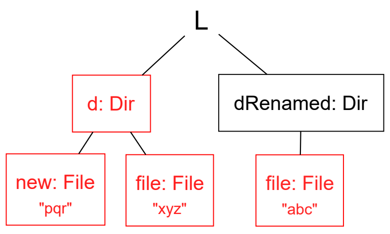


Figure A.10: Incorrect results for test 'Rename parent and update/create sub-file'

Shows the incorrect results on the *local* replica of the test *Rename parent and sub-file* for the Dropbox and Unison clients, which coincidentally produce the same result.

A.8.2 Conflict operations

A.8.2.1 Expected result alternatives

Conflict name	Expected result alternatives
Create-Create	• *Rename*: One of the objects is renamed. • *Merge*: only one object remains. In case of F=directory-directory this means that directory contents are recursively merged. In case of F=file-file and R=deterministic, this implies that no data needs to be transmitted over the network, because the client can detect that file payloads are equal based on its check sum.

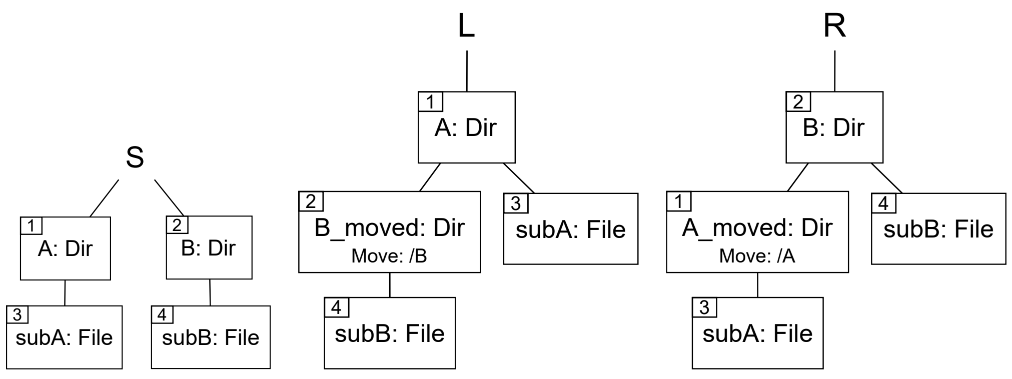
Conflict name	Expected result alternatives
Edit-Edit	• *Overwrite*: changes of one replica are overwritten by the changes of the other replica (with a prior back up to avoid losing changes). • *Merge*: only when R=deterministic: the file remains unchanged, no data is transmitted over the network. • *Duplicate*: one of the files is coped to a “conflict” file name placed next to the original file (e.g. “file” is copied to “file <hostname>’s conflicting copy”). Only makes sense if the content of both files is different. Two sub-variants are possible: – *Sync*: the conflicting copy is synchronized to all other users. – *HostOnly*: the conflicting copy remains only on the host where it was created.
Name clash	• *Rename*: one of the objects is renamed, e.g. by appending a “conflict” string. • *Merge*: only for parameter F=directory-directory. The contents of both directories are merged.
Edit-Delete of file	• *Restore*: the *edit* operation takes precedence and the file is restored on the replica where it was deleted. • *Delete*: the *delete* operation takes precedence, the edited file is deleted on all replicas. • *Local*, *Remote*: whichever operation was done on the remote (or local) replica takes precedence.
Move-Delete	• *Restore*: the *move* operation takes precedence and the file (or directory) is restored on the replica where it was deleted (with all sub-contents in case of a directory). • *Delete*: the *delete* operation takes precedence, the moved file or directory is deleted on all replicas. • *Local*, *Remote*: whichever operation was done on the remote (or local) replica takes precedence.

Conflict name	Expected result alternatives
Move-ParentDeleted	• *Restore*: the *move* operation takes precedence. The directory "test" is recreated to make the *move* operation possible. Other sub-objects of "test" (that were not manipulated by the replica on which the *move* operation was executed) are deleted. • *Delete*: the *delete* operation takes precedence. The *move* operation is first undone, then "test" is deleted. • *Local, Remote*: whichever operation was done on the remote (or local) replica takes precedence.
Create-ParentDeleted	• *Restore*: see Move-ParentDeleted. Replace "move" with "create" or "edit" respectively. • *Delete*: the *delete* operation takes precedence. To avoid data loss, all created objects or updated files are backed up prior to executing the delete operation. • *Local, Remote*: whichever operation was done on the remote (or local) replica takes precedence.
Move-Move (Source)	• *Local*: the *move* operation applied to the *local replica* takes precedence. • *Remote*: the *move* operation applied to the *remote replica* takes precedence. • *Duplicate*: the file or directory is copied, s.t. both "testMoved" and "testOtherMoved" exist. Note that this is not a favorable outcome!
Move-Move (Cycle)	• *Local*: the final file system structure resembles the one of the *local* replica. Ideally, the synchronization client undoes the conflicting *move* operation detected on the *remote* replica and then applies the *local* replica's *move* operation. • *Remote*: the final file system structure resembles the one of the *remote* replica. • *Duplicate*: the directories are copied, s.t. the directories "A", "A/B_moved", "B" and "B/A_moved" exist. Note that this is not a favorable outcome!

A.8.2.2 Result remarks

- A) "test" is renamed to "test-conflict-<date and time>-<5 random characters>" (our implementation).

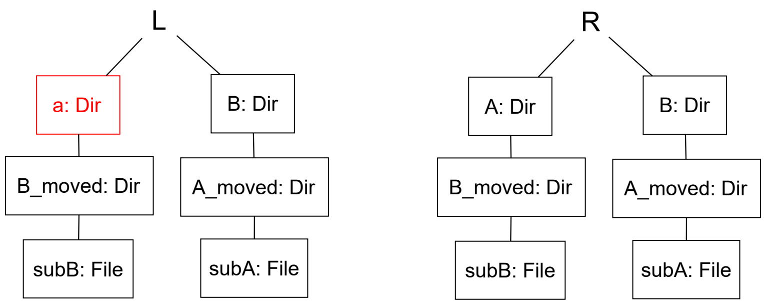
- B) “test” is renamed to “test (<hostname>’s conflicted copy <date>)” (Dropbox).
- C) “test” is renamed to “test (1)” (Backup and Sync).
- D) “test” is renamed to “test-<hostname>” (OneDrive).
- E) “test” is renamed to “test_conflict-<date and time>” (NextCloud).
- F) Unison generally does not automatically *resolve* conflicts. It *detects* the conflict and prompts the user for a choice which operation should take precedence in its graphical user interface.
- G) NextCloud does re-download the file from the *server* replica but then correctly merges the result. In case of F=dir-dir, the client re-downloads the directory’s sub-files.
- H) The *local* and *remote* replica keep their respective file or directory. Windows client does not show any error message, while the macOS client shows a notification that there are problems with synchronizing files.
- I) *Local* client renames one directory to “test (1)” in the *local* replica. After both *local* and *remote* client have finished synchronizing changes, the directory “test (1)” of the *local* replica corresponds to “test” on the *remote* replica, and “test” of the *local* replica corresponds to “test (1)” on the *remote* replica. Using the Google Drive’s web interface we found that both directories are named “test”, which clarifies that Google Drive does not follow the same namespace limitation rules as Windows or macOS disks, where a name may only be used once within a directory.
- J) The *local* client moves the locally created conflicting object “dest” to “dest_conflict-<date and time>”. However, the client implementation generally seems to *remove* all objects that end with this conflict-pattern from the server and all other client replicas.
- K) Detects the conflict as *Create-Create* conflict, due to the inability to understand *move* operations.
- L) A restart of both clients is required to establish a stable synchronization in case the *delete* operation is applied to the *remote* replica. In this case, the *entire* “test” directory is retransmitted to the server replica.
- M) Unison detects the conflict as expected. If the user selects the *move* / *content-update* operation to have precedence, the *entire* “test” directory is retransmitted to the server replica, i.e., Unison does not just create the missing “test” directory itself but also sub-objects that previously existed within “test”.
- N) Once our test code established the concurrent changes on the local and server replica and started the local Backup And Sync client, its GUI immediately shows an error “Can’t sync 1 item”. In case the *delete* operation was performed on the *local* replica, the client immediately recovers from this error (it disappears) and sends the *delete* operation to the server replica. But if instead the *delete* operation was applied to the *remote* replica, the error remains until the local client is restarted manually.
- O) If the *local* client runs on Windows, the remote *move* operation takes precedence. If it runs on macOS, a duplicate is created. This indicates that both implementations are not using the same code base and have implemented different conflict resolution options.
- P) See figures A.11 and A.12 for more details.
- Q) Unison detects two *Create-ParentDelete* conflicts and resolves them as chosen by the user, see remark *M*.



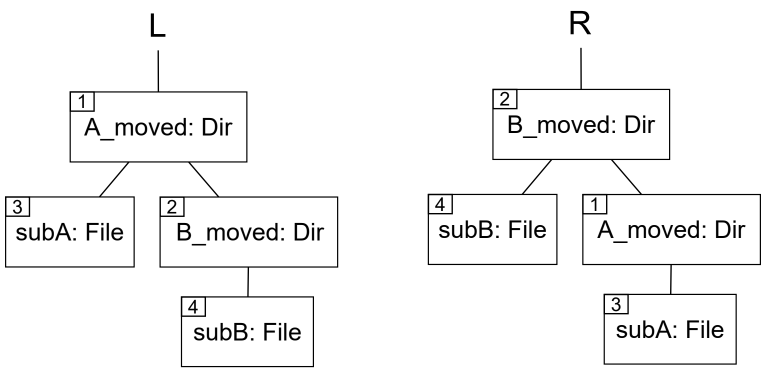
(a) Initial situation (b) Local replica changes (c) Remote replica changes

Figure A.11: Initial situation for test 'Move-Move Cycle'

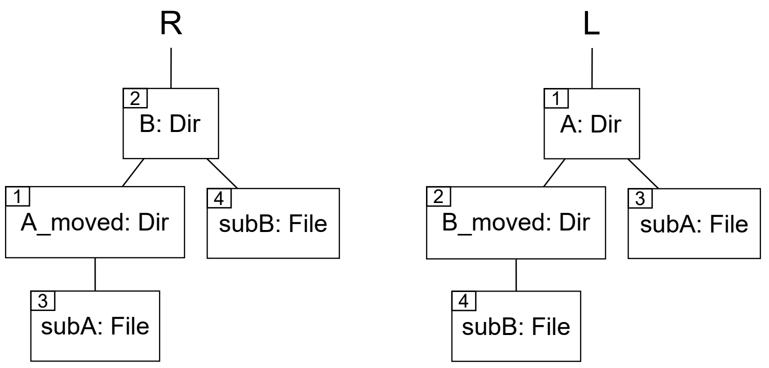
Subfigures a-c show the initial file system tree and the concurrent changes made on the *local* and *remote* replica.



(a) Dropbox



(b) Backup and Sync



(c) NextCloud

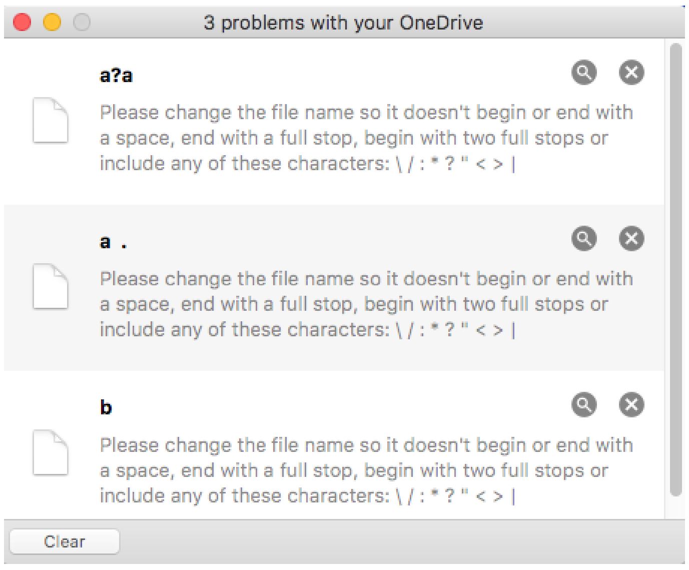
Figure A.12: Incorrect results for test 'Move-Move Cycle'

Subfigures a-c show the incorrect results for the respective clients. Dropbox almost achieves consistency by *duplicating* the directory structures, but inexplicably uses a lower-case name for the directory "A" on just one replica. Google's Backup and Sync never achieves consistency. Its *local* client keeps retrying an upload-operation of "B_moved", which continuously fails. For NextCloud, neither the *local* nor *remote* replica are changed by the respective clients. The *local* client's tray icon indicates an error and a brief error message is shown, indicating that *A* could not be synchronized due to an error. *A* also appears in the UI's *Not synced* tab, with "Unknown error" being shown as rationale for not synchronizing the directory.

A.8.3 Cross-platform issues

A.8.3.1 Result remarks

- *Remote rename*: here the *remote* client automatically applies *move* operations that rename objects. In the event of a *case-sensitivity* or *Unicode normalization clash* this means that *one* of the objects keeps the name that the user intended it to have, while the other(s) are renamed by the *remote* client. In the event of reserved characters or names, the objects are renamed to replace invalid with valid characters or names. The advantage of this approach is that both the *local* and *remote* replica finally contain an *equal* set of files and directories. The disadvantage is that this approach assumes the lowest common feature set of *all* potentially existing replicas. For instance, Dropbox and our implementation assumes that file systems are case-*in*sensitive in general, only because there *may* be Windows clients where this is the case. If all replicas, however, were installed on a case-sensitive macOS disk volume, these rename operations would have been inadequate.
- *Unchanged / No conflict*: All objects are transmitted from the remote to the local replica as they are. No automatic renaming or omission of objects is applied. This is only possible if all replicas have equal capabilities, e.g. when both local and remote replica are on the same operating system.
- SFN = Short File Name.
- A) The configuration assistant of the macOS OneDrive client does not allow to choose a directory on a case-sensitive disk volume.
- B) Unisons remote client skips the synchronization of both the upper- and lower-case object. The following message is shown: "Two or more files on a case-sensitive system have names identical except for case. They cannot be synchronized to a case-insensitive file system. No updates to propagate". Given that *server* replica is on a case-sensitive Linux disk volume, this behavior is sub-optimal.
- C) The object using upper-case on macOS ("FILE", "DIR") is named "FILE (1)", "DIR (1)" on the *local* replica on Windows.
- D) The *remote* client uploads both objects to the server replica. The *local* client downloads the lower-case object and then fails to download the upper-case object. The client's tray icon briefly indicates an error, and the user can inspect the error details in the UI ("Not synced" tab).
- E) The object using lower-case on macOS ("file") is named "file (1)" on the *local replica* on Windows.
- F) The *local* client on Windows does not detect the conflict. The client's tray icon and UI show the same behavior as in remark *D*.
- G) Unison generally does not automatically resolve conflicts. The *local* client detects the conflict and prompts the user for a choice which operation should take precedence in its graphical user interface.
- H) Only the file encoded using the NFC-normalization is sent to the *server* replica and downloaded to the *local* replica. No notification is shown to the user.
- I) Like in remark *B*, the remote client skips the synchronization of both "ä" files. It also shows the exact same error message, which is obviously incorrect, because the files do not differ by *case* but by *Unicode normalization*.
- J) On the *remote* replica, both files keep their name. On the *local* replica the client creates files "ä" and "ä (1)", where the latter corresponds to the NFC-normalized file on the *remote* replica.
- K) The file is automatically renamed by the *remote* client to NFC normalization on the *remote* replica. Consequently, it is also created using NFC normalization on the *local* replica.



3 problems with your OneDrive

a?a

Please change the file name so it doesn't begin or end with a space, end with a full stop, begin with two full stops or include any of these characters: \ / : * ? " < > |

a .

Please change the file name so it doesn't begin or end with a space, end with a full stop, begin with two full stops or include any of these characters: \ / : * ? " < > |

b

Please change the file name so it doesn't begin or end with a space, end with a full stop, begin with two full stops or include any of these characters: \ / : * ? " < > |

Figure A.13: OneDrive macOS client cross-platform issues information

Information window shown by the OneDrive client on macOS while running the test *Reserved characters and names*. Assuming that these hints are given to achieve compatibility with Windows, several hints are incorrect. (1) Windows does allow files to *begin* with spaces (but they may not *end* with them). (2) Windows files may *begin* with ".." as long as there are valid follow-up characters. For instance, "..test" is a valid name. However, ".." is not.

- L) The file "ä" keeps the NFD normalization on the *remote* replica but is converted to NFC normalization on the *local* replica.
- M) This leads to problems in practice. A Windows user can concurrently create a file "ä" which leads to having two files whose name *look exactly equal* in the Windows file manager. Users of an early BSync version (which did not handle this case yet) were confronted with this issue.
- N) The *remote* client renames the file "b " to "b" on the *remote* replica, but keeps the names of all other files. After the rename operation, the client synchronizes all files to the *server* replica. If the *local* client is on macOS, all files are downloaded to the *local* replica.
- O) The *remote* client synchronizes files with reserved *names* ("LPT1", "LPT1.foo.bar") to the *server* replica, but *skips* synchronization for the three files we created whose names contain reserved *characters* (or end with spaces and "."). The *remote* client notifies the user that these files were not synchronized and how she can rename them herself to make them synchronizable. Interestingly, some of that advice is technically incorrect. See figure A.13 for more details. If the *local* replica is on *Windows*, files with reserved *name* (that do exist on the *server* replica) are not downloaded, without any notification shown to the user. If the *local* replica is on *macOS*, these files are downloaded.
- P) None of the files that have reserved characters or names are downloaded to the *local* replica by the Windows client. The user is not notified. Dropbox offers a *Check bad files* tool[10], a web site that allows Windows users to inspect the list of files that exist on the server but cannot be synchronized to their Windows machine.

[10]See `https://www.dropbox.com/bad_files_check`, retrieved July 21, 2019.

- Q) The *remote* client synchronizes *all* files to the *server* replica. The *local* client on Windows automatically renames files that contain reserved *characters* by replacing special characters with "_", or renaming files that end with space or "." s.t. they end with "_" on the *local* replica. For reserved *names*, the Windows client uses the *UNC path* technique[11] to create those files anyway! This is problematic, because such files and directories cannot be opened, moved or deleted by the user using Windows explorer. Special tools or expert command-line knowledge is required to delete these objects. Interestingly, the Backup and Sync Windows client does *not* seem to use *UNC paths* when *deleting* objects. Consequently, the user cannot remove such files by removing them on Google Drive (or any macOS client), because the file will still remain on the Windows machine.

- R) The *remote* client transmits *all* files to the *server* replica. The *local* client in turn does not download any of the files that contain reserved characters or names.

- S) The synchronization fails silently, causing both the *local* and *remote* replica to remain unchanged.

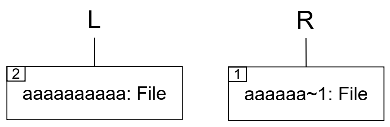
- T) After the synchronization finished, both files exist on both replicas, but with different names. Each replica keeps its originally created file at the original path. On the *local* replica, the remotely created file is found with name "a....a (1)" (length of 255 characters), i.e. the client applies name truncation. On the *remote* replica, the locally created file is found with name "a...a" with an inexplicable length of *224* characters.

- U) If our test performs the operations on the *remote* replica while the *remote* client is *offline*, then the *remote* client hangs after starting it, showing the status as "synchronizing" for an indefinite time period. If our test instead performs the operations while the remote client is *online*, the operations are successfully synchronized with the *server* replica. In this case the *local* client hangs indefinitely.

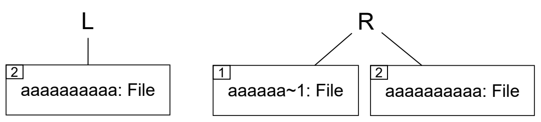
- V) The *remote* client automatically renames the file on the *remote* replica, replacing the "~" character with "-".

- W) The *local* client does not recognize the issue that the short file name cannot be created during *update detection*, but only recognizes this during *propagation*. It displays an incorrect error message "Destination updated during synchronization: The file aaaaaa~1 has been created". The *local* client most likely assumes that the file with the short name really exists and that it was created after the update detection phase had finished.

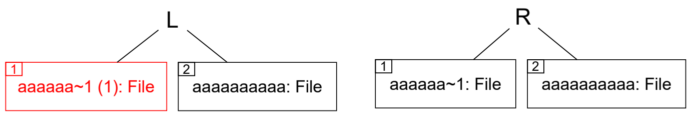
[11]While Windows APIs forbid operations such as *CreateFile("C:\LPT1")*, it does permit *CreateFile("\\?\C:\LPT1")*. The UNC name prefix [Mic18b] disables any parsing of the path, making such operations possible.



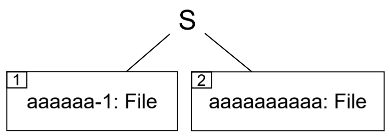
(a) Initial situation



(b) Dropbox / NextCloud / Unison



(c) Backup and Sync



(d) OneDrive / Our implementation

Figure A.14: Results for Windows 8.3 file names - distributed creation

Subfigure a shows the initial situation created by the test. Subfigures b-d show the final file system structure once each implementation has completed the synchronization procedure. Note the replacement of "~" with "-" in subfigure d. Regarding subfigure b, we should point out that Windows' file system API only forbids the operation sequence *CreateFile("aaaaaaaaaa") + CreateFile("aaaaaa~1")*, but allows *CreateFile("aaaaaa~1") + CreateFile("aaaaaaaaaa")*. Consequently, the *remote* replica can have *both* files.

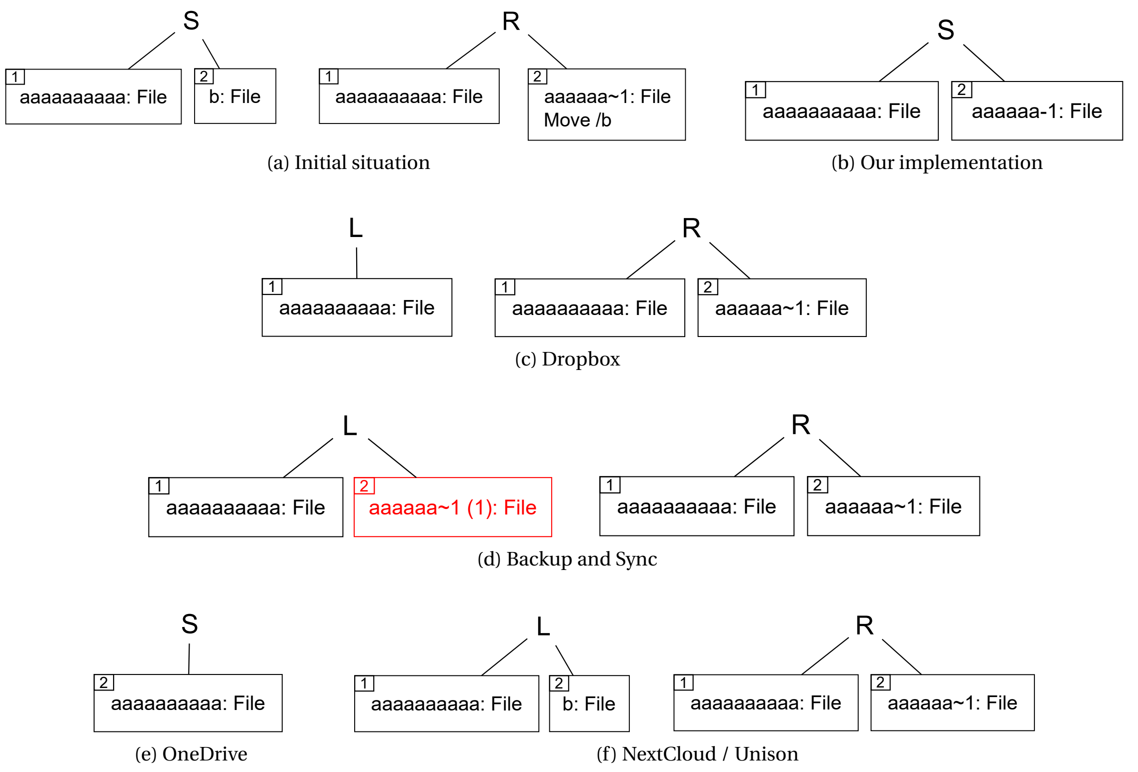


Figure A.15: Results for Windows 8.3 file names - macOS clash

Subfigure a shows the initial situation created by the test, as well as the *move* operation applied on the remote replica. Subfigures b-f show the final file system structure once each implementation has completed the synchronization procedure. Note that in subfigure e the original file “aaaaaaaaaa” with ID=1 has been completely removed from all replicas unexpectedly. We observed that the *local* client first deleted the file “aaaaaaaaaa”, followed by executing a *Move(b, aaaaaaaaaa)* operation. The *local* client showed a confusing (and obviously incorrect) error message: “You now have two copies of a file; We couldn’t merge the changes in aaaaaa~1-<local-host-hame> so we created another copy of it.”. Because the *delete* and *move* operations are synchronized to all other replicas, this leads to the complete loss of the file with the long name (ID=1).

Publications of the author

Publications related to this thesis

Marius Shekow. “Syncpal: A simple and iterative reconciliation algorithm for file synchronizers”. In: *Distributed Applications and Interoperable Systems - IFIP International Federation for Information Processing, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Lingby, Denmark, June 18-21, 2019, Proceedings.* Ed. by José Pereira and Laura Ricci. 2019

Marius Shekow and Wolfgang Prinz. “A capability analysis of groupware, cloud and desktop file systems for file synchronization”. In: *Proceedings of 17th European Conference on Computer-Supported Cooperative Work-Exploratory Papers. The International Venue on Practicecentred Computing an the Design of Cooperation Technologies. European Society for Socially Embedded Technologies (EUSSET). 2019.* DOI: 10.18420/ecscw2019_ep06. URL: `http://dx.doi.org/10.18420/ecscw2019_ep06`.

Further publications

Marius Shekow and Leif Oppermann. “On maximum geometric finger-tip recognition distance using depth sensors”. In: *WSCG 2014: communication papers proceedings: 22nd International Conference in Central Europeon Computer Graphics, Visualization and Computer Vision. In co-operation with EUROGRAPHICS Association.* WSCG. Ed. by Václav Skala. Václav Skala - UNION Agency, 2014, pp. 83–89. ISBN: 978-80-86943-71-8. URL: `https://dspace5.zcu.cz/handle/11025/26381`.

Leif Oppermann, Marius Shekow, and Deniz Bicer. “Mobile cross-media visualisations made from building information modelling data”. In: *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct - MobileHCI ’16. the 18th International Conference (Florence, Italy).* Ed. by Fabio Paternò and Kaisa Väänänen. New York, New York, USA: ACM Press, 2016, pp. 823–830. ISBN: 9781450344135. DOI: 10.1145/2957265.2961852.

Leif Oppermann, Lisa Blum, and Marius Shekow. “Playing on AREEF”. In: *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services - MobileHCI ’16. the 18th International Conference (Florence, Italy).* Ed. by Fabio Paternò and Kaisa Väänänen. New York, New York, USA: ACM Press, 2016, pp. 330–340. ISBN: 9781450344081. DOI: 10.1145/2935334.2935368.

Curriculum Vitae

Personal information	
Name	Marius Alwin Shekow
Date & place of birth	September 6, 1985, Tettnang, Germany

Education	
10/2010 - 03/2013	RWTH Aachen University M.Sc. Media Informatics, B-IT
10/2006 - 09/2010	Karlsruhe University of Applied Sciences B.Sc. Computer Science
2002 - 2005	Claude Dornier Schule Friedrichshafen Secondary school, Abitur (German university entrance diploma)

Professional experience	
02/2013 - now	Fraunhofer FIT, MARS group, Sankt Augustin, Germany Research associate, Software engineer with focus on Distributed Systems (File Synchronization) and Virtual/Augmented Reality
02/2012 - 02/2013	Fraunhofer FIT, MARS group, Sankt Augustin, Germany Working student, Software engineer and M.Sc. thesis in Computer Vision (hand gesture recognition)
09/2011 - 01/2012	Fraunhofer FIT, UCC group, Sankt Augustin, Germany Working student, Software engineer with focus on Distributed Systems and Android
03/2010 - 08/2010	CERN, Meyrin, Switzerland Technical student program, B.Sc. thesis on automation software for linear particle accelerator
03/2008 - 08/2008	Cluetec GmbH, Karlsruhe, Germany Working student, Enterprise Java software, J2ME
07/2007 - 02/2008	Webzooms AG (now Citrix), Karlsruhe, Germany Working student, web development