



Article

Synchronizing Many Filesystems in Near Linear Time

Elod P. Csirmaz ^{1,*} and Laszlo Csirmaz ^{1,2} ¹ Alfréd Rényi Institute of Mathematics, 1053 Budapest, Hungary; csirmaz@renyi.hu² Institute of Information Theory and Automation, CZ-182 00 Prague, Czech Republic

* Correspondence: elod@epcsirmaz.com

Abstract: Finding a provably correct subquadratic synchronization algorithm for many filesystem replicas is one of the main theoretical problems in operational transformation (OT) and conflict-free replicated data types (CRDT) frameworks. Based on the algebraic theory of filesystems, which incorporates non-commutative filesystem commands natively, we developed and built a proof-of-concept implementation of an algorithm suite which synchronizes an arbitrary number of replicas. The result is provably correct, and the synchronized system is created in linear space and time after an initial sorting phase. It works by identifying conflicting command pairs and requesting one of the commands to be removed. The method can be guided to reach any of the theoretically possible synchronized states. The algorithm also allows asynchronous usage. After the client sends a synchronization request, the local replica remains available for further modifications. When the synchronization instructions arrive, they can be merged with the changes made since the synchronization request. The suite also works on filesystems with a directed acyclic graph-based path structure in place of the traditional tree-like arrangement. Consequently, our algorithms apply to filesystems with hard or soft links as long as the links create no loops.

Keywords: file synchronization; algebraic model; optimistic synchronization; linear complexity

MSC: 08A02; 08A70; 68M07; 68P05



Citation: Csirmaz, E.P.; Csirmaz, L. Synchronizing Many Filesystems in Near Linear Time. *Future Internet* **2023**, *15*, 198. <https://doi.org/10.3390/fi15060198>

Academic Editor: Davide Tosi

Received: 26 March 2023

Revised: 17 May 2023

Accepted: 24 May 2023

Published: 30 May 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction and Related Works

Synchronizing diverged copies of some data stored on a variety of devices and/or at different locations is an ubiquitous task. The last two decades saw a proliferation of practical and theoretical works addressing this problem. According to [1], “file synchronization is a feature usually included with backup software in order to make it easier to manage and recover data as and when required”. File synchronization is usually delivered through cloud services. Dedicated file synchronizing solutions frequently come with additional tools not just for managing the saved data but also to allow for file sharing and collaboration with stored files and documents.

These cloud storage services are easily accessible for the end user because the service front ends are very well integrated into web clients as well as desktop and mobile environments. Simple user interfaces hide the complex and sophisticated service back ends [2]. Collaboration services are frequently integrated into the “cloud storage” environment. For example, Google Docs is an application layer integrated into Google Drive storage, Office 365 is integrated with One Drive storage and Dropbox Paper service is an extension of Dropbox storage.

To address the emerging challenges in the more specific fields of distributed data storage and collaborative editors, two competing theoretical frameworks emerged: operational transformation (OT) and conflict-free (or commutative) replicated data types (CRDT). OT appeared in the seminal work of [3] and was refined later, among others, in [4–6]. The main applications are collaborative editors, the most notable example being Google Docs [7]. CRDT emerged as an alternative with a stronger theoretical background, see [8,9]. Both OT and CRDT have been applied successfully in a variety of synchronization tasks,

including file synchronizers [6,10,11]. Finding a provably correct, subquadratic synchronization algorithm, however, has remained one of the main open problems both in OT and CRDT [12].

The algebraic theory of filesystems [13,14] introduced algebraic manipulations of filesystem commands, and provided the foundations for automated checking of certain filesystem properties. It is reminiscent of both OT [6] and CRDT inasmuch as, instead of pure traditional filesystem commands, it uses operations enriched with contextual information. While these operations are not fully commutative—as would be requested by CRDT [15]—the non-commutative parts can be isolated systematically and handled separately. In [14], this framework is used successfully to create a provably correct theoretical filesystem synchronizer for two replicas together with a complete analysis of all possible synchronized states. The present work extends [14] significantly by the following:

- Providing the theoretical foundation for synchronizing an *arbitrary number* of replicas;
- Developing, for the first time, a provably correct synchronization algorithm which works in *linear time* after an initial sort, and thus in subquadratic total running time;
- Allowing *asynchronous usage*, namely, after requesting synchronization, the local replicas need not be locked;
- Allowing for *late comers* when a replica can be upgraded to the synchronized state without providing the local changes;
- Generalizing the traditional tree-like filesystem skeleton to arbitrary *acyclic graphs*, thus extending the applicability of the synchronization algorithm.

In this paper, we will use the term *near linear* to mean “linear up to a logarithmic factor”. Thus, sorting requires near linear time [16], and the total time required by the above synchronization algorithm is also near linear.

This paper follows the traditional paradigm of filesystem synchronization described in, for example, [17] and illustrated in Figure 1. The starting point is a set of identical copies of the same filesystem, possibly stored at different locations, on different hardware and architectures (cloud servers, mobile devices, laptop and desktop computers with various operating systems) or using different software implementations (e.g., ext4, btrfs, ZFS, NTFS, APFS, or database file systems). Each of these replicas is edited (modified) locally. At a certain time, the diverged copies call for *synchronization* by sending a description of the diverged state to a central server. After receiving the requests, the server computes filesystem commands, which transform each replica into a common synchronized state, and send them back to the replicas. The replicas execute the received commands on their local copy, transforming all diverged copies into a new identical synchronized state. At that point, the synchronization cycle can start again.

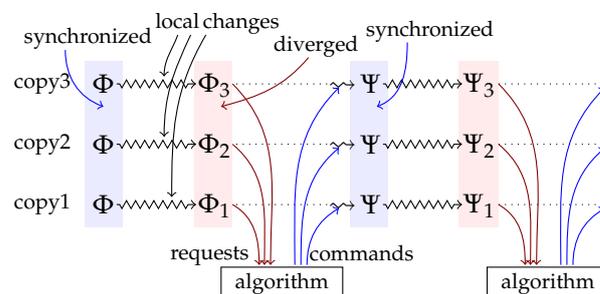


Figure 1. The synchronization cycle. Identical copies of the same filesystem are edited independently. Each replica sends the locally created update information to the synchronizer, which returns the commands to be executed on the local copy to update it to a common synchronized state.

Synchronizers typically require locking the replicas during the whole synchronization process, meaning that no modifications are allowed after the synchronization request is sent (the locked time period is indicated by the dotted lines on Figure 1). Asynchronous, or optimistic synchronization allows additional local modifications after the synchronization request is sent as depicted on the top of Figure 2. When the synchronization commands

arrive from the server, those commands are modified to reflect the additional changes, and then applied to the replica. The result should be the same as when performing synchronization without the additional changes, and then applying them to the synchronized filesystem afterwards—as indicated at the bottom of the figure.

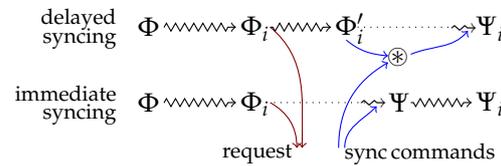


Figure 2. Asynchronous synchronization. Top line: after the synchronization request has been sent, additional local modifications are made to the filesystem. When receiving the synchronization commands, they are modified using the current state of the local filesystem. Bottom line: the end result should be the same as applying the synchronization commands immediately and then making the local modifications afterwards.

The main focus of this paper is filesystem synchronization, or the synchronization of data stored in the nodes of a tree or directed acyclic graph. The stored data are considered to be an indivisible unit, and the task of consolidating different versions of the same data is not considered. It should be solved by other methods specifically tailored to this task.

Some practical aspects of data synchronization are not touched and are out of the scope of this paper. Managing user access and permissions, when and how to allow file sharing and collaboration are especially important due to security considerations [1]. Additionally, file synchronization should use very strict security protocols to ensure that data are safely protected and secured at all times and to make data leaks and malicious access less likely.

Synchronizers should also minimize network traffic. Unlike the popular Rsync utility [18] available for comparing and synchronizing files, there is currently no such “middleware” utility for general datasets [19]. Hopefully our work is a small step in that direction.

The rest of this paper is organized as follows. Section 2 recalls the building blocks of the algebraic theory of filesystems, including the filesystem model, the augmented filesystem commands and their basic properties. This section does not contain new results, and its purpose is to give the reader a comprehensive summary of the topic, on which the rest of the paper relies. For more intuition, explanation and examples, please consult [14]. Section 3 is a high-level overview of how filesystem synchronization can be handled in the algebraic framework. This section *defines* what constitutes a synchronized state of several diverged replicas, rather than providing a method of creating it. The definition automatically guarantees many desired and required properties of the merged filesystem, an indication of the strength and adequacy of the algebraic framework. It is discussed how all synchronized states can be achieved by conflict resolution, paving the way towards the near linear synchronization algorithm discussed in Sections 4 and 6. Asynchronous (optimistic) synchronization is discussed at the end of Section 3.

The base algorithms of the synchronization suite are discussed in Section 4, including the one which generates the command sequence (called *merger*), which produces the merged state in subquadratic time. Supporting theoretical results are collected and proved in Section 5. Section 6 discusses how all possible synchronized states can be generated by a nondeterministic algorithm running in linear time. Section 7 presents some empirical results justifying the claims about the running time of the algorithms. Finally, Section 8 concludes the paper with some extensions and open problems. Most notably, our algorithms, with some modification, work not only on the tree-like filesystem skeletons as stipulated by the algebraic theory but also on filesystems based on directed acyclic graphs.

This work focuses mainly on algorithmic aspects, so many theoretical justifications are deliberately phrased in general terms. Rigorous proofs would require substantially more space and, in our opinion, would not provide additional insight. A proof-of-concept implementation of the algorithms presented in this paper in Python can be found at <https://github.com/csirmaz/algebraic-reconciler> (accessed on 10 May 2023).

2. Definitions

This section recalls the notions and basic results of the algebraic theory of filesystems [13], with some illustration of the concepts. The main component is a highly symmetric set of filesystem commands, which are enriched with contextual information. Devising such a command set which is amenable to algebraic manipulation was one of the main contributions of [13]. For more intuition and explanation on this, see [14].

2.1. Filesystems

The filesystem model reflects the most important high-level aspects of real-world filesystems. It is a mixture of identity- and path-based models [10,12]. The contents of the filesystem are stored at *nodes*, which are identified, or labeled, by a set of fixed and predetermined paths. The collection of all (virtually) available nodes is fixed in advance (while in a real-world filesystem, only a restricted subset of those paths is present). No path operations are considered; in particular, our model does not support the creation or deletion of *links*. In the basic case, no links are allowed at all; thus, the namespace—the set of available nodes or paths—forms a collection of rooted trees. An actual filesystem populates this fixed namespace with values. If Φ is a filesystem, the value stored at node n is denoted by $\Phi(n)$. Valid filesystems are required to have the *tree property* all the time, meaning that along any branch starting from a root node, there must be zero or more *directories*, zero or one *file*, followed by *empty* nodes only. If Φ does not have the tree property, then we say that Φ is *broken*.

Formally, the paths form a forest-like *namespace*: a set \mathbb{N} endowed with the partial function $\uparrow : \mathbb{N} \rightarrow \mathbb{N}$ returning the parent of every non-root node, while this function is not defined on roots. If $n = \uparrow m$ then n is the parent of m , and m is a child of n . For two nodes $n, m \in \mathbb{N}$, we say that n is above m , or n is an ancestor of m , and write $n \prec m$, if $n = \uparrow^i m$ for some $i \geq 1$. As usual, $n \preceq m$ denotes $n \prec m$ or $n = m$. As the parent function \uparrow induces a tree-like structure on \mathbb{N} , the relation \preceq is a partial order. Two nodes $n, m \in \mathbb{N}$ are *comparable* if either $n \preceq m$ or $m \preceq n$, and they are *incomparable* or *independent* otherwise.

In practice, nodes of the filesystem are labeled by *complete paths*, where directory names are separated by the slash character. Thus, the root has the label $/$; nodes $/a$, $/xxx$ are on the first level just under the root; and $/a/b/cc/d$ is (the label of) a node on level four, whose ancestors are $/a/b/cc$, $/a/b$, $/a$, and $/$. The implementation of the algorithms also follows this convention.

As indicated above, the value stored by a filesystem at a node can be a *directory*, can be *empty* or can be a *file*. The type of this value is denoted by \mathbb{D} , \mathbb{O} and \mathbb{F} , respectively, corresponding to these possibilities. Regarding notation, we also use \mathbb{D} for the directory value, and \mathbb{O} for the empty value (where \mathbb{O} means “no value”, not to be confused with a file which has no content). When the node value is a file, then the node stores the complete file content (including the possibility that this content is empty). While this filesystem model allows only one directory value and only one empty value (see the discussion in Section 8 on relaxing this limitation), there are many different possible file values of type \mathbb{F} representing different file contents. The value types are *ordered*, with \mathbb{O} being the lowest, and \mathbb{D} being the highest, written as $\mathbb{O} < \mathbb{F} < \mathbb{D}$. The type of the filesystem value x is denoted by $\text{tp}(x)$.

2.2. Filesystem Commands

Real-life filesystems are usually manipulated by commands such as creating or deleting files and directories, modifying (editing, appending to) a file, or moving an existing file or directory to another location. Our model contains similar commands but with some modifications, the first of which is that we only consider commands which affect the filesystem at a single node. Thus, a *move* command should be represented as a sequence of *delete* and *create*.

Second, commands in our model include the complete new value to be stored in the filesystem. Even if file contents are just partially modified or are appended to, the full new

value must be supplied. This allows our model to use a unified representation of all single-node commands, as they can be fully specified by the node (path), at which the command acts, and the new value (including a directory and empty value) to be stored there.

Third, as was observed in [13], enriching filesystem commands with additional contextual information—in this case, the previous content at the affected node—makes them amenable to algebraic manipulation.

Definition 1 (Filesystem commands). *A filesystem command is a triplet $\sigma = \langle n, x, y \rangle$, where $n \in \mathbb{N}$ is the node on which σ acts, x is the content at node n before σ is executed (the contextual information, precondition), and y is the new content.*

It is clear that every real-life filesystem command acting on a single node can be easily (and automatically) transformed into this internal representation (Definition 1). For example, $rm\ dir(n)$ corresponds to $\langle n, \mathbb{D}, \mathbb{O} \rangle$, which replaces the directory value at n by the empty value. The command $\langle n, \mathbb{O}, \mathbb{D} \rangle$ creates a directory at n but only if the node n has no content, that is, there is no directory or file at n (a usual requirement when creating a directory). This command reflects the usual behavior or $mk\ dir$. For files f_1 and $f_2 \in \mathbb{F}$, the command $\langle n, f_1, f_2 \rangle$ replaces f_1 stored at n by the new content f_2 . This latter command can be considered an equivalent of $edit(n, f_2)$.

As an example, creating a copy of the file `/home/user/text` in the same directory under the name “copy” and then deleting the original file is represented by the following sequence of commands:

$$\langle /home/user/copy, \mathbb{O}, f_0 \rangle \text{ and } \langle /home/user/text, f_0, \mathbb{O} \rangle,$$

where f_0 is the file content at the “text” node.

Applying the command σ to a filesystem Φ is written as the left action $\sigma\Phi$. The command $\sigma = \langle n, x, y \rangle$ is *applicable* to Φ if Φ contains x at node n , that is, $\Phi(n) = x$ (the precondition holds), and after changing the content at n to y , the filesystem still has the tree property. If σ is not applicable to Φ , then we say that σ *breaks the filesystem*. If σ does not break Φ , then σ is applicable to Φ , and its execution changes Φ at node n only.

Command sequences are applied from left to right; thus, $(\sigma\alpha)\Phi = \alpha(\sigma\Phi)$, where α is a command sequence. The composition of sequences is written as the concatenation $\alpha\beta$, but occasionally we write $\alpha \circ \beta$ to emphasize that β is to be executed after α . A sequence breaks a filesystem if one of its commands breaks the filesystem when it is to be applied. The sequence α is *non-breaking* if there is at least one filesystem α that does not break; otherwise, it is *breaking*.

Sequences α and β are *semantically equivalent*, written as $\alpha \equiv \beta$, if they have the same effect on all filesystems, that is, $\alpha\Phi = \beta\Phi$ for all Φ . We write $\alpha \sqsubseteq \beta$ to denote that β semantically extends α , that is, $\alpha\Phi = \beta\Phi$ for all filesystems such that α does not break.

For example, the sequence which creates a file at some node n and then changes this file to a directory is equivalent to the single command which creates the directory directly:

$$\langle n, \mathbb{O}, f \rangle \circ \langle n, f, \mathbb{D} \rangle \equiv \langle n, \mathbb{O}, \mathbb{D} \rangle,$$

while creating the file and deleting it immediately is semantically strictly weaker than applying the “null” command $\langle n, \mathbb{O}, \mathbb{O} \rangle$, and thus we only have

$$\langle n, \mathbb{O}, f \rangle \circ \langle n, f, \mathbb{O} \rangle \sqsubseteq \langle n, \mathbb{O}, \mathbb{O} \rangle.$$

This is because the right-hand side is applicable when the parent of n contains a file, while the left-hand side would break such a filesystem.

The *inverse* of $\sigma = \langle n, x, y \rangle$ is $\sigma^{-1} = \langle n, y, x \rangle$. For a sequence α , its inverse α^{-1} consists of the inverses of the commands in α in reverse order. The inverse has the expected property:

if α does not break Φ , then $(\alpha^{-1}\alpha)\Phi = \Phi$, that is, α^{-1} rolls back the effects of α . Observe that α is non-breaking if and only if it is α^{-1} .

2.3. Command Types, Execution Order

The input and output *values* of $\sigma = \langle n, x, y \rangle$ are x and y , respectively, while the input and output *types* are $\text{tp}(x)$ and $\text{tp}(y)$. Commands are classified by their input and output types using *patterns*. The command $\langle n, x, y \rangle$ matches the pattern $\langle n, \mathcal{P}_x, \mathcal{P}_y \rangle$ if $\text{tp}(x)$ is listed in \mathcal{P}_x , and $\text{tp}(y)$ is listed in \mathcal{P}_y . In a pattern, the symbol \bullet matches any value. As an example, every command matches $\langle \bullet, \text{OFD}, \text{DFO} \rangle$.

Commands with identical input and output values are *null commands*. Null commands do not change the filesystem (but can break it if the precondition does not hold). *Structural commands* change the type of stored data. Structural commands are further split into *constructors* and *destructors*. A constructor increases the type of the stored value, while a destructor decreases it. Thus, a constructor matches either $\langle \bullet, \text{O}, \text{FD} \rangle$ or $\langle \bullet, \text{F}, \text{D} \rangle$, and a destructor matches either $\langle \bullet, \text{DF}, \text{O} \rangle$ or $\langle \bullet, \text{D}, \text{F} \rangle$. Observe that σ is a constructor if and only if σ^{-1} is a destructor. Finally, non-null commands matching $\langle \bullet, \text{F}, \text{F} \rangle$ are *edits*.

The binary relation $\sigma \ll \tau$ between commands on parent–child nodes captures the notion that σ must precede τ in the execution order.

Definition 2 (\ll relation, \ll -chain). *The relation $\sigma \ll \tau$ holds if the pair matches either $\langle n, \text{DF}, \text{O} \rangle \ll \langle \uparrow n, \text{D}, \text{FO} \rangle$, or matches $\langle \uparrow n, \text{OF}, \text{D} \rangle \ll \langle n, \text{O}, \text{FD} \rangle$. An \ll -chain is a sequence of \ll -related commands connecting its first and last element.*

The first case in Definition 2 corresponds to the requirement that before deleting a directory, its descendants should be deleted. The second case says that a file or directory can only be created under an existing directory. Observe that $\sigma \ll \tau$ if and only if $\tau^{-1} \ll \sigma^{-1}$, and in this case, either σ and τ are both constructors, or both are destructors.

2.4. Canonical Sets and Sequences

Commutativity is a core concept in command-based synchronization [17], where, in fact, the task is to determine in what order (and which) modifications made to other replicas can be applied to a particular replica. If two modifications or commands commute, that is, their result does not depend on the order in which they are applied, then they do not represent conflicting updates to the data, as they can be seen as independent. Unsurprisingly, then, commutativity plays a central role in CRDT (see [8,9]), where basic data types with special operators are devised so that executing the operators in different orders yields the same results.

While not all filesystem commands commute, non-commutative pairs can be isolated systematically. If σ and τ are on different nodes which are not in parent–child relation, then they commute ($\sigma\tau$ and $\tau\sigma$ are semantically equivalent). An example of this is creating a file in some directory and editing another existing file. The affected nodes where the changes are made are independent. If σ and τ are non-null commands on parent–child nodes, then either $\sigma\tau$ breaks every filesystem, or, necessarily, $\sigma \ll \tau$. If a command σ on node $/a/b/c$ is followed immediately by another command τ on the node $/a/b/c/d$ successfully, then σ must create a directory at $/a/b/c$ (whose location previously was either empty, or contained a file); thus $/a/b/c/d$ is empty, so τ must create either a file or a directory there. Consequently, we have $\sigma \ll \tau$ by Definition 2. Consecutive commands on the same node either break every filesystem (if the second command requires a different value than the output of the first command), or can be replaced by a single command (while extending the semantics). These easy facts imply some strong and intricate structural properties of non-breaking command sequences. Exploring and using these properties made possible the complete and thorough investigation of the synchronization process of two diverged replicas in [14], as well as devising the first provably correct subquadratic synchronization algorithm in this paper.

Intuitively, a canonical sequence is just the “clean” version of a non-breaking command sequence. An important property of canonical sequences is that their semantics is determined uniquely by the set of commands they contain, see Theorem 1. Command sets that can be arranged into canonical sequences are also called canonical. For the formal definitions, we need two more notions. A command sequence α honors \ll , if for any two commands $\sigma, \tau \in \alpha$, σ precedes τ in the sequence whenever $\sigma \ll \tau$. The command set A is \ll -connected if for any two commands $\sigma, \tau \in A$; if σ and τ are on different comparable nodes, then they are connected by an \ll -chain (Definition 2) consisting of commands in A . In particular, if A is \ll -connected and $\sigma, \tau \in A$ are on the comparable nodes n and m , then A has commands on each node between n and m .

Definition 3 (Canonical sets and sequences). *The command set A is canonical if the following three conditions hold:*

- A does not contain null commands;
- A contains at most one command on each node;
- A is \ll -connected.

The command sequence α is canonical if the commands in α form a canonical set and, additionally, α honors \ll .

For example, the set consisting of the three commands $\langle /a/b, \mathbb{D}, f_5 \rangle, \langle /a/c, f_0, \mathbb{O} \rangle$, and $\langle /a/c, f_5, f_5 \rangle$ is not canonical for two reasons: the third command is a null command, and the second and the third commands are on the same node. The following command set is also not canonical:

$$\{ \langle /a/b/c/d, f_s, \mathbb{O} \rangle, \langle /a, \mathbb{D}, f_s \rangle, \langle /a/b, \mathbb{D}, \mathbb{O} \rangle \}$$

as its first and last elements are not \ll -connected (the second and third elements are \ll -connected). Adding the command $\langle /a/b/c, \mathbb{D}, \mathbb{O} \rangle$ to this set makes it canonical.

Theorem 1 (E.P. Csirmaz, [13]).

- (a) *If two canonical sequences share the same command set, then they are semantically equivalent. Actually, they can be transformed into each other using commutativity rules.*
- (b) *Canonical sequences are non-breaking.*
- (c) *Every non-breaking sequence α can be transformed into a canonical sequence $\alpha^* \supseteq \alpha$ (that is, α and α^* have the same effect on filesystems that α does not break, but α^* might work on more filesystems).*
- (d) *Canonical sets can be ordered to honor \ll , that is, to become canonical sequences. \square*

As an example, the only order honoring the \ll relation of the canonical set

$$E = \{ \langle /a/b/c/d, f_s, \mathbb{O} \rangle, \langle /a, \mathbb{D}, f_s \rangle, \langle /a/b, \mathbb{D}, \mathbb{O} \rangle, \langle /a/b/c, \mathbb{D}, \mathbb{O} \rangle \} \tag{1}$$

is the one which executes them “bottom up” starting with the command on $/a/b/c/d$ and ending with the command on node $/a$.

Using Algorithm 1 as an auxiliary tool, Algorithm 2 in Section 4 checks, in near linear time, whether a command set A is canonical. Algorithm 3 arranges a canonical set into a canonical sequence.

By virtue of Theorem 1 (a) and (d), the semantics of canonical sequences is determined by the unordered set of their commands, even if their order cannot be recovered uniquely (this happens when the set contains commands on incomparable nodes). Thus, when only the semantics is concerned, a canonical sequence can, and will, be replaced by the set of its commands. For example, when we write $A\Phi$, where A is a canonical command set, we mean that commands in the set A should be applied in some (or any) \ll -honoring order to the filesystem Φ . Similarly, $A \circ B$ means that, first, the commands in A are applied in some \ll -honoring order, followed by the commands in B , again in some \ll -honoring order.

The property that commands in a canonical set can be executed in different orders while preserving their semantics is a variant of the *commutativity principle* of CRDT [20]. Definition 4 below discusses a special case of the reordering when a subset of the commands is to be moved to the beginning of the execution line.

Definition 4 (Initial segment). *For a canonical set A , we write $B \subseteq A$ and say that B is an initial segment of A to indicate that B is not only a subset of A but can also be moved to the beginning of an ordering of A while keeping the semantics. In other words, $A \equiv B \circ (A \setminus B)$.*

We remark that if B is an initial segment of A , then both B and $A \setminus B$ are canonical (see Proposition 5 in [14]). For example, the canonical set E in Equation (1) has three proper initial segments:

$$\begin{aligned} & \{ \langle /a/b/c/d, f_s, \mathbb{O} \rangle, \langle /a/b, \mathbb{D}, \mathbb{O} \rangle, \langle /a/b/c, \mathbb{D}, \mathbb{O} \rangle \}, \\ & \{ \langle /a/b/c/d, f_s, \mathbb{O} \rangle, \langle /a/b/c, \mathbb{D}, \mathbb{O} \rangle \}, \text{ and} \\ & \{ \langle /a/b/c/d, f_s, \mathbb{O} \rangle \} \end{aligned}$$

and, of course, all of them are canonical.

2.5. Refluent Sets

Following the terminology of [14], canonical sets A and B are called *refluent* if there is at least one filesystem on which both of them work (neither of them breaks). In general, the canonical sets A_1, \dots, A_k are *jointly refluent* if there is a single filesystem on which all of them work. It is clear that if k canonical sets are jointly refluent, then they are pairwise refluent as well. The converse statement, that if the canonical sets A_i are pairwise refluent, then they are also jointly refluent, is stated and proved as Proposition 1 in Section 5. The concept of refluent arises naturally in file synchronization. Command sequences representing changes to the replicas are refluent, as they are applied to identical copies of the same filesystem.

3. Filesystem Synchronization

The synchronization paradigm used in this paper follows the traditional one described in, for example, [17] and depicted in Figure 1. At the beginning of the synchronization cycle, each replica stores an identical copy of the same filesystem Φ . Due to local modifications, the replicas diverge, and after some time, the filesystem in the i -th replica changes to Φ_i . At a certain moment, the replicas call for synchronization by sending information extracted by the *update detector* (run locally) to a central server, which hosts the reconciliation algorithm. After the server has received all update information, it determines what the common synchronized filesystem Ψ will be. Then, it sends instructions to the replicas separately telling them how to transform their local filesystem to the synchronized one. Finally, each replica executes the received instructions, which transforms their local copy to the synchronized state Ψ , optionally informing the user about some (or all) of the conflicts and how they have been resolved.

3.1. Update Detector

Depending on the data communicated by the replicas, synchronizers are categorized as either *state-based* or *operation-based* [9,12]. In state-based synchronization, replicas send the current state of their filesystems, or merely the differences between their current state and the last known synchronized state [21]. Frequently, the local copy does not have access to the original synchronized state because of its limited resources, and transmitting the whole current state is prohibitively expensive. It is an active research area to devise efficient transmission algorithms which transmit the differences only [18,22–24]. Operation-based synchronizers transmit the complete log (or trace) of all operations performed by the user [25]. It has been observed that in practice these logs are poorly maintained and are not always reliable [26,27].

Theorem 1 suggests that the updated information that the replicas send to the central server (or, rather, the data on which the synchronizer algorithm works) can be a canonical command set which transforms the original synchronized filesystem to the current replica. By Theorem 2 below, this set is not only a succinct representation of the differences but can also be generated in time proportional to the size of the filesystems (by traversing the original Φ and the modified filesystem Φ_i simultaneously).

Theorem 2 ((Theorem 19 in [14])). *Let Φ and Ψ be two filesystems. The command set*

$$A_{\Phi \rightarrow \Psi} = \{ \langle n, \Phi(n), \Psi(n) \rangle : n \in \mathbb{N} \text{ and } \Phi(n) \neq \Psi(n) \}$$

is canonical, and $A_{\Phi \rightarrow \Psi} \Phi = \Psi$.

If the replica has a complete log of the commands executed by the user (as in an operation-based update detector), that is, if the updates are collected in a command sequence α_i which transforms Φ to Φ_i , then α_i can be transformed into the requested canonical set as claimed by Theorem 1 (c). As detailed in Algorithm 4 in Section 4, this transformation can be performed in near linear time in the size of α_i , which can be much faster than traversing the whole filesystem.

3.2. Synchronization

The central server, having received the canonical sets from the replicas describing the local changes, must resolve all conflicts between the updates and generate a common, synchronized filesystem. Conflict resolution should be intuitively correct; thus, discarding all changes made by the replicas is not a viable alternative. While the majority of practical and theoretical synchronizers do not present any rationale to explain their specific conflict resolution approach [12], two notable exceptions [6,10] describe high-level consistency philosophies. In Ref. [10], the main principles are *no lost update* (preserve all updates on all replicas because these updates are equally valid) and *no side effects* (do not allow objects to unexpectedly disappear). While these principles make intuitive sense, neither can possibly be upheld for every conflict. In Ref. [6], the relevant consistency requirements are *intention-confined effect* (operations applied to the replicas by the synchronizer must be based on operations generated by the end user), and *aggressive effect preservation* (the effect of compatible operations should be preserved fully, and the effect of conflicting operations should be preserved as much as possible). These requirements are, in fact, variations of the OT consistency model [4]. Note that the other two OT principles—convergence and causality preservation—do not apply to filesystem synchronizers.

In keeping with the prescriptive nature of the above principles, we proceed by *defining* what a synchronized state is, rather than creating it by some ad hoc method. Suppose that the original filesystem is Φ , and the modified filesystem at the i^{th} replica is $\Phi_i = A_i \Phi$, where A_i is the canonical set submitted as input to the reconciler. The synchronized or *merged state* Ψ is determined by the canonical set M called the *merger* such that $\Psi = M\Phi$, where M satisfies the following two conditions:

- (1) Every command in M is submitted by one of the replicas;
- (2) The canonical set M is maximal with respect to the first condition.

The first condition ensures that the synchronization satisfies the *intention-confined effect*: there are no surprise changes in the merged filesystem. The *aggressive effect preservation* is guaranteed by the second condition. As M is maximal, it preserves as much of the intention of the users as possible. In general, there can be many different mergers satisfying these conditions. The reconciler must choose one of them either automatically using some heuristics, or manually as instructed by the user.

Observe that the canonical sets A_i describing the local changes are jointly refluent (see Section 2.5), as all of them can be (and were) applied to the original filesystem Φ . The formal definition of a merger is as follows.

Definition 5 (Merger). *The merger of the jointly refluent canonical sets A_1, \dots, A_k is a maximal canonical set $M \subseteq \bigcup_i A_i$. The corresponding synchronized state of the replicas $A_1\Phi, \dots, A_k\Phi$ is $M\Phi$.*

This definition requires the absolute minimum. Due to its simplicity, it is clear, intuitively appealing, and captures the desired properties of a synchronized state. It remains to be seen whether it is also sufficient or an oversimplification. Surprisingly, this definition is indeed sufficient. Mergers provided by Definition 5 satisfy many additional desirable properties without any further requirements. Some of these properties are discussed in the next subsections. For the rest of this section, we fix the canonical set A_i and the original filesystem Φ so that the current state of replica i is the valid filesystem $\Phi_i = A_i\Phi$. Consequently, none of the sets A_i breaks Φ , and therefore the command sets A_i are jointly refluent.

As an example, suppose that Φ contains directories at $/a$, $/a/b$, and single file at $/a/b/c$. The first replica deletes the file and all directories above it. The second replica creates a file below $/a/b$. The third replica creates the same file but with different content, and also creates a copy of that file below $/a$. The canonical sequences describing these changes are

$$\begin{aligned} A_1 &= \{\sigma_1, \sigma_2, \sigma_3\}, \text{ where } \sigma_1 = \langle /a/b/c, f_o, \mathbb{O} \rangle, \sigma_2 = \langle /a/b, \mathbb{D}, \mathbb{O} \rangle, \sigma_3 = \langle /a, \mathbb{D}, \mathbb{O} \rangle; \\ A_2 &= \{\tau\}, \text{ where } \tau = \langle /a/b/z, \mathbb{O}, f_z \rangle; \\ A_3 &= \{\rho_1, \rho_2\}, \text{ where } \rho_1 = \langle /a/z, \mathbb{O}, f_u \rangle, \text{ and } \rho_2 = \langle /a/b/z, \mathbb{O}, f_u \rangle. \end{aligned}$$

It is clear that σ_3 is in conflict with all commands in A_2 and A_3 ; σ_2 is in conflict with τ and ρ_2 . Commands τ and ρ_2 are also in conflict; and a merger containing σ_3 must also contain σ_2 . σ_1 is compatible with all other commands, and thus it will be in each possible merger. If σ_3 is in the merger, then it can contain no commands from either A_2 or A_3 . If σ_2 is not present, then only the conflict τ vs. ρ_2 remains. Thus, there are four mergers, namely

$$M_1 = \{\sigma_1, \sigma_2, \sigma_3\}, \quad M_2 = \{\sigma_1, \sigma_2, \rho_1\}, \quad M_3 = \{\sigma_1, \tau, \rho_1\}, \quad \text{and} \quad M_4 = \{\sigma_1, \rho_1, \rho_2\}.$$

Each merger describes a possible synchronized state, and each one can be the desired one under the right circumstances.

3.3. Mergers Are Applicable to the Filesystem

While Definition 5 does not require M to work on the original filesystem Φ , it never breaks it as proved in Proposition 3 in Section 5. Every merger creates a meaningful synchronized state and never breaks the original filesystem.

3.4. Mergers Can Be Created in Near Linear Time

Mergers can be created by a simple greedy algorithm. Proposition 4 in Section 5 states that a non-maximal canonical subset of $\bigcup_i A_i$ can always be extended by some command from $\bigcup_i A_i$ so that it remains canonical. Consequently, starting from the empty set and adding commands from $\bigcup_i A_i$ one by one while keeping the set canonical produces a merger. In particular, any canonical subset of $\bigcup_i A_i$ can be extended to a merger. Since checking whether a command set is canonical takes linear time (Algorithm 2), this naïve approach requires cubic time. Algorithm 5 creates a merger in near linear time. To generate all mergers in nondeterministic linear time, we need a more sophisticated algorithm as discussed in Section 6.

3.5. Mergers Have an Operational Characterization

The synchronized state defined by the merger M as $\Psi = M\Phi$ has a clear operational characterization. The local replica Φ_i can be transformed into the merged state by first rolling back some of the local operations executed on that replica, then applying additional commands executed on other replicas. By Proposition 3, $M \cap A_i$ is an initial segment of both A_i and M , and thus,

$$A_i \equiv (M \cap A_i) \circ (A_i \setminus M).$$

Rolling back the commands in $A_i \setminus M$, that is, executing the canonical set $(A_i \setminus M)^{-1}$ on Φ_i , gives $(M \cap A_i)\Phi$. Then, applying the canonical set $M \setminus A_i$ yields the filesystem

$$(M \cap A_i) \circ (M \setminus A_i)\Phi = M\Phi = \Psi.$$

In summary, the i -th replica should execute the command set

$$(A_i \setminus M)^{-1} \circ (M \setminus A_i)$$

on its local copy Φ_i to transform it into the synchronized state Ψ . The rolled back commands in $A_i \setminus M$ give a clear indication of the local changes that are discarded. This command set could be presented to the user to decide whether some of them should be reintroduced.

Choosing the merger $M_3 = \{\sigma_1, \tau, \rho_1\}$ in the example above, the first replica should roll back σ_2 and σ_3 by executing $\{\sigma_2^{-1}, \sigma_3^{-1}\}$. These commands restore the directories at $/a$ and $/a/b$. They are followed by executing $\{\tau, \rho_1\}$ which adds the two files. To reach the same synchronized state, the second replica need not roll back any of its commands. Executing $\{\sigma_1, \rho_1\}$ directly deletes the file at $/a/b/c$ and creates the new file at $/a/z$. Finally, the third replica should roll back ρ_2 (after this, the precondition in τ holds, so τ will not break the filesystem when executed), and then execute the commands τ and σ_1 in any order.

Observe that after synchronization, none of the rolled-back commands is applicable anymore. σ_2 and σ_3 would delete directories which are not empty, while ρ_2 would create a file which already exists. This is true in general. The maximality of the merger set M implies that none of the rolled-back commands can be executed directly on the synchronized state Ψ . Either its input condition would fail (modification by some other replica on that node took precedence), or the command would destroy the tree property (deleting a non-empty directory, or creating a file under a non-existent directory). Therefore, the changes represented by the rolled-back commands can only be reintroduced in a different form.

3.6. Mergers Can Be Created via Conflict Resolution

Synchronizers typically work by identifying and resolving conflicts until no conflicts remain. While Definition 5 specifies the synchronized state directly, clearly two commands are in conflict if they cannot occur together in the same canonical set. In our case, however, a weaker notion of conflict also works.

Definition 6. *The commands $\sigma, \tau \in \cup_i A_i$ are in conflict if either of the following is true:*

- (a) *They are different commands on the same node.*
- (b) *The node of σ is above the node of τ , σ creates a non-directory and τ creates non-empty content.*

Commands $\tau = \langle /a/b/z, \mathbb{O}, f_z \rangle$ and $\rho_2 = \langle /a/b/z, \mathbb{O}, f_u \rangle$ from the example above are in conflict, as they are different commands on the same node. Command $\sigma_3 = \langle /a, \mathbb{D}, \mathbb{O} \rangle$ creates a non-directory; thus, it is in conflict with every command below it, which creates a file—that is, all commands in A_2 and A_3 —but it is not in conflict with $\sigma_1 = \langle /a/b/c, f_o, \mathbb{O} \rangle$ and $\sigma_2 = \langle /a/b, \mathbb{D}, \mathbb{O} \rangle$, which create empty content.

By Proposition 2, canonical sets (and thus mergers) do not have conflicts at all, while Theorem 4 claims that a maximal command set of $\cup_i A_i$ without conflicts is a merger. The synchronizer, having received the command sets A_i , can create a *conflict graph*, whose vertices are the commands in $\cup_i A_i$ and edges connect conflicting commands. Mergers correspond to the maximal independent vertex sets of this graph. Creating a merger via conflict resolution can therefore be performed using the following procedure: pick an edge of the graph representing a conflict between, say, σ and τ . Choose either σ or τ as the winner, and delete all vertices connected to the winner (i.e., those vertices which cannot be in the same merger as the winner). When there are no more edges, the remaining vertices form a maximal independent set, a merger.

Using this conflict graph, the synchronizer can make smart decisions, a feature which is painfully missing in commercial and other theoretical synchronizers. When choosing the

conflict to be resolved and the winner of the conflict, the decision can take into account not only local information (the conflicting command pair) but also the effect of the decision on other conflicts.

Creating and manipulating the conflict graph can be done in quadratic time [28]. As this graph has a very special structure and not too many edges, the actual running time could be better. As our main interest in this paper was developing subquadratic algorithms, we did not pursue this line of research further. Using the conflict resolution strategy, Algorithm 5 in Section 4 creates a merger in near linear time. Unfortunately, this algorithm, as explained later, cannot generate all mergers in nondeterministic linear time. For that task, we need further ideas explored in Section 6.

3.7. Mergers Support Asynchronous and Offline Synchronization

As usual, let the filesystem at the i -th replica be $\Phi_i = A_i\Phi$, with the replica sending the canonical command set A_i to the server for synchronization. Section 3.5 discusses that when the server returns the merger M , the replica should execute

$$(A_i \setminus M)^{-1} \circ (M \setminus A_i)$$

on the local copy to transform it to the synchronized state $\Psi = M\Phi$. Optionally, the replica could present the conflicting command set $A_i \setminus M$ to the user for inspection.

Suppose the replica Φ_i has not been locked, and by the time the reply M arrives from the server, it has changed to $\Phi'_i = A'_i\Phi$, where A'_i is a new canonical set describing the differences between the current state Φ'_i and the original common state Φ ; see Figure 2. In this case, the local machine should transform the replica to the synchronized state Ψ and carry over those extra changes that are still executable. To this end, it invokes the synchronization algorithm for the canonical sets A'_i and M , making sure that the returned merger M^* contains M (this can be achieved, for example, by constructing the merger via conflict resolution and making sure that each conflict is resolved in favor of a command in M). By Claim 3, M will be an initial segment of M^* , and thus, $M^* = M \circ M'$, where M' is the canonical set $M' = M^* \setminus M$. Apply the commands

$$C'_i = (A'_i \setminus M^*)^{-1} \circ (M^* \setminus A'_i)$$

to the filesystem Φ'_i and return $A'_i \setminus M^*$ as the conflicting command set. At this moment, the local filesystem is

$$C'_i\Phi'_i = C'_i(A'_i\Phi) = M^*\Phi = M'(M\Phi) = M'\Psi,$$

which is exactly the synchronized filesystem Ψ , to which the command set M' has been applied. The commands in M' can be incorporated easily into the next round of synchronization.

In fact, the same method works, even if the replica does not take part in the process of determining the merger M . Thus, *latecomers* or *offline* replicas, who did not participate in determining the merged state, can still upgrade to it without losing their ability to take part in subsequent synchronization rounds.

4. Algorithms

Let us begin with some of the properties, assumptions and decisions that the algorithms rely on. In our model, a filesystem command has three components: the node it operates on, and the input and output values. Each component is stored in some constant space (using pointers, if necessary). Commands can be sorted using any lexicographic order on the nodes, which is consistent with the “parent” function. In the standard example, a node (path) name is a sequence of identifiers separated by the slash character. With the assumption that comparing two path strings lexicographically takes constant time, sorting n filesystem commands can be done deterministically in $O(n \log n)$ time [16]. We also assume that other path-manipulating algorithms, such as returning the parent of a node,

or deciding whether a node is above another one, also take constant time. Similarly, we presuppose that operations on filesystem values (determining their type, checking their equality or comparing them for sorting) can also be done in constant time.

Almost all algorithms assume that the input commands are in a doubly linked list sorted lexicographically by the nodes of the commands. The time and space complexity estimates of the algorithms typically exclude this sorting time. A proof-of-concept implementation of the algorithms presented in this paper in Python can be found at <https://github.com/csirmaz/algebraic-reconciler> (accessed on 10 May 2023).

4.1. The up Structure

Our first algorithm will be used many times, frequently tacitly, as an auxiliary tool. It enhances a set of nodes by adding an extra up pointer between the nodes. This pointer at node n is \perp if no other node in the set is above n ; otherwise, it points to the node in the set, which is its lowest ancestor. In particular, if the parent of n is also in the set, then $\text{up}(n)$ is the parent of n .

Algorithm 1 (Adding up pointers, Code 1). *Sort the nodes lexicographically, and check them in increasing order. Suppose we have finished processing node n , and the next node in the list is m . Find the first node in the sequence $n, \text{up}(n), \text{up}(\text{up}(n)), \text{etc.}$, which is above m . If found, set $\text{up}(m)$ to this node. If none of them is above m or m is the first node, then set $\text{up}(m)$ to \perp .*

Code 1. Given a lexicographically sorted sequence of commands, add the up pointers.

```

1  for each command in sequence do
2      if this is the first command then
3          | command.up ← ⊥
4      else
5          | upCommand ← previousCommand
6          | loop
7              | if upCommand = ⊥ or
8                  | upCommand.node is an ancestor of command.node then
9                  | command.up ← upCommand
10                 | exit loop
11             else
12                 | upCommand ← upCommand.up
13             end
14         end
15 end

```

For correctness, observe that in the namespace, the sequence $n, \text{up}(n), \text{up}(\text{up}(n)), \text{etc.}$, defines the right boundary of the nodes processed up to n . Since the next node m is to the right of the earlier nodes, its ancestors in the given set must be in this list. After sorting, the running time is linear, as each up link is compared and discarded at most once, and each up link is filled exactly once. We are grateful to Gábor Tardos for devising this algorithm; it is used here with his permission.

4.2. Checking and Ordering Canonical Sets

This algorithm checks whether the command set A is canonical, assuming that it is sorted lexicographically according to the nodes of the commands. It checks the first two conditions of Definition 3, directly. Instead of the third condition (A is \ll -connected), the following clearly equivalent conditions are verified:

1. If A contains a command on the node n and also on an ancestor of n , then it contains a command on the parent of n ;
2. If $\sigma, \tau \in A$ are on parent-child nodes, then either $\sigma \ll \tau$ or $\tau \ll \sigma$.

Algorithm 2 (Determining if A is canonical, Code 2). Start with all commands in A arranged in a doubly linked list according to a lexicographic order of the command nodes. Loop through the commands and check that there is one command on each node, and none of the commands is a null-command. Run Algorithm 1 to define the up pointers. Loop through the commands. If the up pointer is not \perp , then it must point to the parent node; moreover, the current command and the command at the parent must be \ll -related.

Code 2. Check whether a set of commands is canonical.

```

1 – sort sequence lexicographically by the nodes of the commands
2 – add up pointers using Algorithm 1
3 for each command in sequence do
4   | if previousCommand.node = command.node then
5   |   | return false           not canonical, as multiple commands are on the same node
6   | end
7   | if command.up  $\neq \perp$  and not (
8   |   |  $\langle$ command.up, command $\rangle$  is a constructor pair or
9   |   |  $\langle$ command, command.up $\rangle$  is a destructor pair ) then
10  |   | return false not canonical because the closest command on an ancestor is not on the
11  |   | parent, or they do not form a valid pair
12  | end
13 end
14 return true                                     this is a canonical set

```

A canonical set A can always be ordered to honor \ll . Perhaps the simplest way to obtain such an ordering is to make two passes through the lexicographically sorted set A , as done by Algorithm 3.

Algorithm 3 (Ordering a canonical set, Code 3). Sort commands in a canonical set lexicographically. First, scan the commands forward (top–down), extracting the constructor commands, and place them at the beginning of the output sequence. Second, place the remaining commands on the output sequence in reverse lexicographical order (bottom–up). This includes destructors and edit commands matching $\langle \bullet, \mathbb{F}, \mathbb{F} \rangle$. It is clear that this sequence order honors the \ll relation.

Code 3. Order a canonical command set and return a canonical sequence.

```

1 sequence  $\leftarrow$  commandSet in some order
2 – sort sequence lexicographically by the nodes of the commands
3 for each command in sequence do
4   | if command is a constructor then
5   |   | push command on output
6   | end
7 end
8 for each command in sequence backwards do
9   | if command is not a constructor then
10  |   | push command on output
11  | end
12 end
13 return output

```

Both Algorithm 2 and Algorithm 3 of this section clearly run in near linear time.

4.3. Transforming a Sequence to a Canonical Set

Given a non-breaking command sequence α , Algorithm 4 creates a canonical set A , which semantically extends α in near linear running time.

Algorithm 4 (Command sequence to canonical set, Code 4). Sort the commands in α in a lexicographic order by their nodes, retaining the original order where they are on the same node. Process them from left to right. For any consecutive sequence of commands that are on the same node (including one-element sequences), define a replacement command that has the input value of the first command and the output value from the last. If the two values are different, add the replacement command to the result set.

If α may be breaking, it is easy to check that the output and input values of neighboring commands on the same node are equal, or if the resulting set is indeed canonical. The failure of these checks implies that α is breaking, though the algorithm may also successfully convert a breaking sequence to a non-breaking canonical set.

Code 4. Return the canonical command set that is the semantic extension of this sequence.

```

1  – sort sequence lexicographically by the nodes of the commands; in case of equality, keep the
   original order
2  for each command in sequence do
3  |   if this is the first command then
4  |   |   input  $\leftarrow$  command.inputValue
5  |   else
6  |   |   if command.node  $\neq$  prevCommand.node then
7  |   |   |   newCmd  $\leftarrow$  (prevCommand.node, input, prevCommand.outputValue)
8  |   |   |   if newCmd is not a null command then
9  |   |   |   |   output  $\leftarrow$  output  $\cup$  {newCmd}
10 |   |   |   end
11 |   |   |   input  $\leftarrow$  command.inputValue
12 |   |   end
13 |   end
14 end
15 if sequence was not empty then
16 |   newCmd  $\leftarrow$  (lastCommand.node, input, lastCommand.outputValue)
17 |   if newCmd is not a null command then
18 |   |   output  $\leftarrow$  output  $\cup$  {newCmd}
19 |   end
20 end
21 return output

```

4.4. Generating a Merger in Near Linear Time

Theorem 4 characterizes a merger of the jointly refluent command sets A_i as a maximal subset of $\cup_i A_i$ without conflicts. This characterization can be turned into a greedy algorithm which generates a merger in near linear time. Actually, the algorithm finds a maximal independent vertex set of the conflict graph (discussed in Section 3.6) exploiting some special properties of this graph.

According to Definition 6, commands σ and τ are in conflict if either they are acting on the same node; or if their nodes are comparable, the upper command creates a non-directory, and the lower command creates a non-empty value. Loop through the commands in $\cup_i A_i$ in a top-down order. At command σ , if σ is marked as being in conflict with some earlier command, then skip it. Otherwise, keep σ and mark commands which are in conflict with σ as conflicting. It follows that if σ is not skipped, it is not in conflict with commands preceding it, and so conflicting commands are either on the same node, or below the node of σ . To achieve the desired speed, instead of scanning all subsequent commands immediately, we use lazy bookkeeping. In essence, if σ is selected, we flag its node to remember to delete conflicting commands on descendant nodes. At each subsequent node, we check whether its parent has this flag. If yes, we flag that node as well and process any conflicts

accordingly. By Theorem 3, the node set of jointly refluent canonical sets is connected; thus, this flag percolates properly to the descendants.

Algorithm 5 (Generating a merger, Code 5). *The inputs are the jointly refluent canonical sets A_i ; the output is a merger M . Sort the commands in $\cup_i A_i$ lexicographically and then use Algorithm 1 to create the up pointers. Add a “delete conflicts down” flag to the node of each command, initially unset.*

Loop through the commands of $\cup_i A_i$ in lexicographic order. At command σ at node n , check if the node of the command up points to having the “delete conflicts down” flag set. If yes, then set this flag at n as well. If, additionally, σ creates some non-empty content (it is in conflict with a final command above it), then delete σ . If σ is not deleted, then mark it as “final” and delete all subsequent commands on the same node n . If σ is marked “final” and it creates a non-directory value, set the “delete conflicts down” flag at n .

Commands marked as “final” form a maximal command set without conflicts; thus, they form a merger.

Code 5. Given a set of jointly refluent canonical command sets, generate a merger.

```

1 sequence ← union of commands in the command sets
2 – sort sequence lexicographically by the nodes of the commands
3 – add up pointers using Algorithm 1
4 for each command in sequence do
5   if command.node = deleteOnNode then
6     | continue skip this command
7   end
8   if command.up ≠ ⊥ and command.up.node.delConflictsDown then
9     | command.node.delConflictsDown ← true
10    | if command.output ≠ ∅ then creates a non-empty content
11    | | continue skip this command
12    | end
13  end
14  merger ← merger ∪ {command}
15  deleteOnNode ← command.node
16  if command.output ≠ ∅ then creates a non-directory content
17  | command.node.delConflictsDown ← true
18  end
19 end
20 return merger

```

Unfortunately, this algorithm cannot generate all possible mergers. The only non-deterministic choice it can make is picking the winner among commands on the same node, which are not in conflict with previous commands. (Algorithm 5 chooses the first such command). Otherwise, when the algorithm encounters a command for the first time, it puts it into the final list, even if there might be mergers which do not contain this command. The more sophisticated Algorithm 7 generates all mergers in nondeterministic near linear time.

The second step in asynchronous synchronization discussed in Section 3.7 requires not only a merger, but a merger which extends a given canonical subset C of $\cup_i A_i$. With some tweaks, Algorithm 5 can be used for this task as well. The idea is that commands in $\cup_i A_i$ are scanned twice. First, all commands are deleted, which are in conflict with some command in C . Second, use the remaining commands only and proceed as in Algorithm 5. The first scan requires, however, not only a “conflicts down” flag, but also a “conflicts up” flag. To ensure that the algorithm spends linear time handling the upward conflicts, it should check whether this flag is set first, and if yes, quit the upward processing. Otherwise, it should set the flag, process the node, and continue processing at the parent node. We leave it to the interested reader to work out the details.

5. Theory

This section contains supporting theoretical results from the algebraic theory of filesystems. Some of the results were used to justify the correctness of algorithms presented in Section 4. Algorithm 6, that checks whether some canonical sets are refluent is presented in this section as it uses the specific characterization given in Theorem 3. First, we look at conditions which guarantee that a canonical set is applicable to a filesystem. Then, these conditions will be used to characterize refluent canonical sets.

Claim 1. *The canonical set A is applicable to the filesystem Φ if and only if the following conditions hold for every command $\sigma = \langle n, x, y \rangle \in A$:*

- (a) $\Phi(n) = x$;
- (b) *If σ is a destructor, then $\Phi(n') = \emptyset$ at every node n' below n not mentioned in A ;*
- (c) *If σ is a constructor, then $\Phi(n') = \mathbb{D}$ at every node n' above n not mentioned in A .*

Proof. The conditions are necessary. Condition (a) is clear. For (b) and (c), note that no command in A changes the filesystem value at n' , and after executing σ , the value at n' must be empty (or directory in case c), respectively. To show that the conditions are sufficient, let $\sigma \in A$, for which there is no $\tau \in A$ where $\tau \ll \sigma$. Then σ can be executed on Φ as $\Phi(n) = x$ by condition (a), and because if σ is a constructor, then no commands on nodes above n are in A , and thus the values at those nodes are \mathbb{D} ; if σ is a destructor, no command below n is in A , and thus all nodes there contain the empty value. Furthermore, conditions (a)–(c) are clearly inherited in the filesystem $\alpha\Phi$ and the command set $A \setminus \alpha$. \square

5.1. Characterizing Refluent Sets

Claim 2. *The canonical sets A and B are refluent if and only if the following conditions hold:*

- (a) *If $\sigma \in A$ and $\tau \in B$ are on the same node, then their input values are the same.*
- (b) *If $\sigma, \tau \in A \cup B$ are on comparable nodes, then for each node n' between them, there is a command in $A \cup B$ on n' .*
- (c) *Suppose $\sigma, \tau \in A \cup B$ are on nodes $\uparrow n$ and n , respectively. If one of the sets mentions n but not $\uparrow n$, then the input of σ is \mathbb{D} ; if one of the sets mentions $\uparrow n$ but not n , then the input of τ is \emptyset .*

Proof. The conditions are necessary. It is clear for (a). For the other two conditions, suppose A can be applied to Φ . Observe that according to Claim 1, if A has a command on node n but not on nodes above n , then on nodes above n , the filesystem must contain directories; if there are no commands in A below n , then all nodes below n must be empty.

For the other direction, we use Claim 1, too. Set the content at each node mentioned in $A \cup B$ to the common input value. Additionally, set the content to directory at each node above a non-empty node and set the content to empty below every non-directory node. Furthermore, for each constructor command in $A \cup B$ on node n , set every node above n not mentioned in $A \cup B$ to a directory. Similarly, for each destructor command in $A \cup B$ on node m , set every node below m not mentioned in $A \cup B$ to empty. Observe that values at nodes in $A \cup B$ do not change due to (b) and (c). These assignments produce a valid filesystem which satisfies the conditions of Claim 1. \square

Proposition 1. *If the canonical sets $A_i : i \leq k$ are pairwise refluent, then they are jointly refluent.*

Proof. Mimicking the proof of Claim 2, construct the filesystem Φ as follows. Start with all empty nodes. For each command in A_i , set the value at the node of the command to its input value. Each node obtains the same value, as the sets A_i are pairwise refluent. For the same reason, if a node obtains a non-empty value, then all nodes above it can be set to be a directory. Next, if $\sigma \in A_i$ is a destructor and n' below n is not mentioned in A_i but n' is not empty, then it is set by some A_j , and then A_i and A_j are not refluent. Finally, if $\sigma \in A_i$ is a constructor, and n' is above n , not mentioned in A_i , then n' should be set to a directory. If it cannot be done because either $\Phi(n')$ has been set to a different value, or some node

above n' has been set to a non-directory, then again we obtain A_j such that A_i and A_j are not refluent. \square

Claim 3. *Suppose the canonical sets A and B are refluent. Then $A \cap B$ is an initial segment of A . In particular, $A \equiv (A \cap B) \circ (A \setminus B)$.*

Proof. Suppose $A \cap B$ is not empty and let σ be one of the common commands. By Claim 2, if $\tau \in A$ and $\tau \ll \sigma$, then τ must be in B as well. Thus, $A \cap B$ contains a command, which is an initial segment both in A and in B . Delete this command from A and B and apply this claim recursively to the remaining commands. \square

Algorithm 6 below checks whether the collection $\{A_i : i \leq k\}$ of canonical sets is jointly refluent using the characterization proved in Theorem 3. For stating the theorem, define, for any node $n \in \mathbb{N}$ and for $i \leq k$, the index set I_n as

$$I_n = \{i : \text{there is a command in } A_i \text{ on node } n\}.$$

Assuming further that all commands in $\cup_i A_i$ on node n have the same input value, this common value is denoted by $x(n)$.

Theorem 3. *The canonical sets A_i are jointly refluent if and only if the following conditions hold:*

- (a) *All commands on node n have the same input value;*
- (b) *If m is above n and neither I_n nor I_m are empty, then $I_{\uparrow n}$ is non-empty as well;*
- (c) *If $x(\uparrow n) \neq \mathbb{D}$, then $I_n \subseteq I_{\uparrow n}$;*
- (d) *If $x(n) \neq \mathbb{O}$, then $I_{\uparrow n} \subseteq I_n$.*

Proof. Let us remark that condition (b) is equivalent to requesting that if n and m are comparable, neither I_m nor I_n are empty, then $I_{n'}$ is not empty for nodes between n and m .

To check that the conditions are necessary, let Φ be a filesystem, on which all A_i work. Then, $\Phi(n) = x(n)$ for all nodes mentioned in $\cup_i A_i$, given condition (a). Condition (b) follows from part (b) of Claim 2 applied to the refluent sets A_i and A_j , where $i \in I_n$ and $j \in I_m$. To check (c), assume $x(\uparrow n) \neq \mathbb{D}$. Then, $\Phi(n) = \mathbb{O}$, and so if A_i has a command on n (that is, $i \in I_n$), then A_i changes $\Phi(n)$ to a non-empty value, and thus, at the end, the filesystem must have a directory at $\uparrow n$. As A_i does not break Φ and $\Phi(\uparrow n)$ is not a directory, it must contain a command on $\uparrow n$, and thus $i \in I_{\uparrow n}$, as required by (c). Similarly, if $x(n) \neq \mathbb{O}$ (in which case $\Phi(\uparrow n) = \mathbb{D}$) and A_i has a command on $\uparrow n$ (that is, A_i sets $\Phi(\uparrow n)$ to be a non-directory), then A_i must also set $\Phi(n)$ to be empty, and thus $i \in I_n$, as required by (d).

For the reverse implication, it suffices to show that Claim 2 is true for every pair A_i and A_j , and then apply Proposition 1. Condition (a) of Claim 2 is immediate from (a). For the rest, we first remark that if m is the parent of n and none of I_m and I_n are empty, then $\text{tp } x(m) \geq \text{tp } x(n)$. Indeed, if $x(m) \neq \mathbb{D}$, then $I_n \subseteq I_m$ by (c), and thus, there is a canonical A_k , which has commands on both n and m , consequently, we must have $x(n) = \mathbb{O}$. Similarly, if $x(n) \neq \mathbb{O}$ then $I_m \subseteq I_n$, which implies similarly that $x(m) = \mathbb{D}$.

Returning to checking conditions in Claim 2 for sets A_i and A_j , suppose $i \in I_m$ and $j \in I_n$ and m is above n . Consider the path between m and n . By condition (b), there are commands on every node between m and n , and by the previous paragraph, the input types on these nodes are non-increasing. If the next node below m on the $m-n$ path is not \mathbb{O} , then (d) gives that A_i also has a command on that node, too. Similarly, if the node immediately above n is not \mathbb{D} , then by (c), A_j has a command on that node. Consequently, either there is a node between m and n on which both A_i and A_j have a command, or otherwise there is a command from A_i and a command from A_j on parent-child nodes such that the former has input value \mathbb{D} (as it is not in A_j), and the latter has input value \mathbb{O} (as it is not in A_i). In all cases, conditions (b) and (c) of Claim 2 hold, as required. \square

5.2. Mergers by Conflict Resolution

This section presents a proof of the claim that the mergers are exactly the maximal conflict-free subsets. Recall from Definition 6 that two different commands in $\cup_i A_i$ are in conflict if either (a) they are on the same node, or (b) they are on comparable nodes, the node on the higher node creates a non-directory and the command on the lower node creates a non-empty content.

Proposition 2. *There are no conflicts in a canonical set.*

Proof. A canonical set contains at most one command on each node, so assume σ and τ are on comparable nodes. Then, there is an \ll -chain between them (see Definition 3); thus, either $\sigma \ll \dots \ll \tau$, or $\tau \ll \dots \ll \sigma$. In both cases, either the command on the higher node creates a directory, or the command on the lower node creates an empty content. \square

Theorem 4. *Suppose the command sets A_i are jointly refluent. $M \subseteq \cup_i A_i$ is a merger if and only if M is maximal without conflicts.*

Proof. By Proposition 2, a merger does not contain conflicts; thus, it suffices to show that a maximal conflict-free set M is canonical. M contains, at most, one command on each node by condition (a) of Definition 6. Let $\sigma, \tau \in M$ be on nodes n and m , respectively, such that n is above m . Moreover, let $\sigma \in A_i$ and $\tau \in A_j$. We want to show that there is a \ll -chain in M between σ and τ . We know that σ and τ are not in conflict.

Consider first the case when σ creates a directory, that is, it matches $\langle n, \mathbb{O}, \mathbb{D} \rangle$. As A_i and A_j are refluent, let Φ be any filesystem on which both A_i and A_j work. Since $\Phi(n)$ is not a directory, all nodes in Φ below n are empty, in particular, τ matches $\langle m, \mathbb{O}, \mathbb{FD} \rangle$. As the canonical A_j does not break Φ , A_j must contain commands on all nodes between n and m , including m . If n is a parent of m , then $\tau \ll \sigma$, and we are done. If n is strictly above m , then we may assume that there are no commands in M on nodes between n and m , and thus no command on $\uparrow m$ either. However, A_j contains a command $\tau' = \langle \uparrow m, \mathbb{O}, \mathbb{D} \rangle$ (as A_j does not break Φ), and $M \cup \{\tau'\}$ is conflict-free, contradicting the maximality of M .

The second case is when τ creates an empty node, that is, it matches $\langle m, \mathbb{DF}, \mathbb{O} \rangle$. Similarly to the above, σ matches $\langle n, \mathbb{D}, \mathbb{FO} \rangle$, and then either $\tau \ll \sigma$, or otherwise, A_i contains the command $\tau' = \langle \uparrow m, \mathbb{D}, \mathbb{O} \rangle$, which can be added to M . \square

Proposition 3. *Let A_i be canonical sets and $M \subseteq \cup_i A_i$ be a merger. If none of A_i breaks Φ , then neither does M .*

Proof. We use the conditions in Claim 1 to show that M does not break Φ . To this end, let $\sigma = \langle n, x, y \rangle \in M$ so that $\sigma \in A_i$. Since A_i does not break Φ , condition (a) follows. To check (b), suppose σ is a destructor command, n' is below n , and it is not mentioned in M . If n' is not mentioned in A_i either, then condition (b) holds, as A_i is applicable to Φ . So suppose $\tau' \in A_i$ is on the node n' . As $\tau' \notin M$, there must be a command $\tau \in M$ on node m , which is in conflict with τ' . Since M has no command on n' (but has a command above n'), m must be above n' . By Definition 6, τ' creates a non-empty content. Since σ and τ' are not in conflict (both are in the canonical set A_i), σ must create a directory. However, this contradicts the assumption that σ is a destructor.

The case when σ is a constructor and n' is above n is similar. \square

Proposition 4. *Let A_i be refluent canonical sets, and $C \subseteq \cup_i A_i$ be canonical. There is a merger extending C .*

Proof. As commands in C are not in conflict by Proposition 2, C can be extended to be a maximal conflict-free subset of $\cup_i A_i$. However, this set is a merger by Theorem 4. \square

6. Generating All Mergers

Algorithm 5 in Section 4.4 cannot generate all possible mergers in nondeterministic linear time. The modified algorithm, which creates a merger extending a given canonical subset C , can, however, be used for this purpose as follows. Pick a random subset C of the commands in $\cup_i A_i$ and check if C is canonical using Algorithm 2. If yes, use the modified version of Algorithm 5 to create a merger extending C ; otherwise, use the original version to create a merger.

While this algorithm clearly generates all mergers in nondeterministic linear time, it is not satisfactory, as it blindly guesses the final merger. In this section, we develop a more appealing approach by further exploiting the structure of refluent canonical sets. Let us fix the jointly refluent canonical sets $\{A_i : i \leq k\}$, and consider all nodes mentioned in the command set $\cup_i A_i$. Since the A_i canonical sets are refluent, we know that the input values of the commands on the same nodes are equal.

Observe that if there are any conflicts among the commands, then there is a conflict of one or more of the following special types:

- (1) Multiple different commands on the same node with a file input value;
- (2) A pair of commands matching $\langle \uparrow n, \mathbb{D}, \mathbb{OF} \rangle$ and $\langle n, \mathbb{O}, \mathbb{FD} \rangle$;
- (3) Multiple different commands with an empty input value on the same node;
- (4) Multiple different commands with a directory input value on the same node.

We eliminate these conflicts in this order. First, we consider conflicts of type (1). They are necessarily on incomparable nodes, as in any filesystem, file nodes are on such nodes. Of the commands on the same node, we choose a winner. If the winner is a destructor or an edit (matching $\langle n, \mathbb{F}, \mathbb{OF} \rangle$), we delete all commands below n which create non-empty content. If the winner is a constructor or an edit (matching $\langle n, \mathbb{F}, \mathbb{FD} \rangle$), we delete all destructor commands above n . Since these conflicts are on incomparable nodes, deletions triggered by one do not affect the conflicts on another. Additionally, since all deleted commands are in conflict with the winner, an element of the merger, we know that the merger will be maximal.

Next, we consider conflicts of type (2). We mark all parent nodes with a directory value that have a destructor command and which have a constructor on an empty node on one of their children. We consider such parent nodes in bottom-up order, and either keep the destructor command(s) on the parent, or all the constructor commands on the children, without choosing a winner yet. If the destructors are kept, we delete all commands below $\uparrow n$, which create non-empty content. If the constructors are kept, we delete all destructors on and above $\uparrow n$. Since the empty child nodes in these conflicts are on incomparable nodes, the deletions there are independent. The deletions of commands creating non-empty content on other children can only affect commands matching $\langle n, \mathbb{D}, \mathbb{F} \rangle$, which may be part of conflicts of this type that are already resolved. However, since there is a destructor on $\uparrow n$, there must be a $\langle n, \mathbb{D}, \mathbb{O} \rangle$ command on such children, so the resolution of earlier conflicts are not affected by these deletions.

The deletions of destructors upwards are not independent, but as we proceed in bottom-up order, they may resolve yet unresolved conflicts of type (2) but will never interfere with conflicts already processed. We note that the subsequent steps in the algorithm always choose a destructor or a constructor command as the winner on a node if, at this stage, it has at least one. This means that all commands deleted here are in conflict with a command that will be part of the merger, ensuring its maximality.

Conflicts of type (3) are considered in a top-down order. We choose a single winner command on each node. If the winner matches $\langle n, \mathbb{O}, \mathbb{F} \rangle$, then we delete all constructor commands below n . As the deletions are downwards, and we proceed top-down, they may resolve yet unresolved conflicts of type (3) but will not interfere with the winners already chosen. Additionally, deleted commands are clearly in conflict with the winner.

Finally, conflicts of type (4) are processed in bottom-up order. We again choose a single winner on each node. If it matches $\langle n, \mathbb{D}, \mathbb{F} \rangle$, then we delete all destructors above

n . It is again true that the deletions do not affect winners already chosen, and that the maximality of the merger is guaranteed.

Since we know that any conflict entails a conflict of one of the above types, and as we removed all such conflicts, we also know that the merger constructed is not only maximal but also conflict-free.

The algorithm sketched below realizes this idea, and thus generates all mergers in randomized near linear time. It assumes that the input command sets A_i are canonical and refluent.

Algorithm 7 (Generating all mergers). *Arrange the commands in $\cup_i A_i$ lexicographically and add the up pointers as in Algorithm 6. Make several passes over the commands dealing with conflicts (1)–(4) as indicated above. Each pass handles commands either in top–down order of their nodes, or in the reverse bottom–up order. Handling the command at node n (which is the parent of another node in case (2)) may result in deleting those commands at node n which satisfy a certain property, deleting all commands above n which satisfy some other property, and deleting all commands below n satisfying a third property, or some combination of these possibilities. The algorithm assumes that those deletions are performed before proceeding to the next node.*

In summary, make four passes through the commands, alternating top–down and bottom–up orders, and handle cases (1) to (4) in each pass. Return the set of the final, non-deleted commands as the merger. The correctness and that the algorithm can actually create all mergers by making appropriate choices that follow from the discussion above.

The running time is linear if each pass can finish processing in linear time. To ensure this, we keep additional flags at each node noting either that required upward deletions are performed at and above this node, or that downward deletions should be performed as necessary. Upward deletions are performed immediately following the up pointers, but they abort upon encountering a node, in which the relevant flag is already set. This ensures that each node is visited at most once for this purpose, keeping the running time linear.

Flags for downward deletions are checked whenever visiting a node in a top–down pass. If the flag is set on the parent, set the flag on the current node, and delete the necessary commands there. This ensures that the latest deletions are applied just in time. During bottom–up passes, downward deletions are not performed immediately, as descendant nodes are not processed again, but rather they are delayed until the next top–down pass or an additional pass is executed for this purpose. As each flag is set at most once, the running time is guaranteed to be linear. An easily accessible implementation of this algorithm in Python can be found at <https://github.com/csirmaz/algebraic-reconciler> (accessed on 10 May 2023).

7. Empirical Results

The performance of the algorithms was tested on several synthetic data sets. These sets consist of the collection of the canonical command sequences the replicas executed on a common filesystem. This initial filesystem is determined by two integer parameters S and T . It has non-empty nodes on the topmost three levels only. These nodes are labeled by the paths $/i$, $/i/j$ and $/i/j/k$, where i , j and k are numbers between 0 and $S - 1$ such that (i, j) and (j, k) are not farther from each other modulo S than T . The filesystem contains different files at the non-empty nodes $/i/j/k$, and contains directories at all other non-empty nodes. Typical parameter values are $T = 2$ and $S = 10$.

The set of command sequences to be synchronized also depends on the number of users (replicas), which varies between 2 and $S - 1$. User u for $0 \leq u \leq S - 1$ makes the following extensive changes on the filesystem:

- (1) Deletes all existing files at $/i/u/k$ for all i and k ;
- (2) Removes (the now empty) directories at $/i/u$ for all i ;
- (3) For each x in $(u - 1, u, u + 1)$ modulo S and for all i and for all $j \neq u$ changes the file at $/i/j/x$ (if exists) to a directory;

- (4) Under each newly created directory creates S new files with unique content. These files are placed at the nodes with paths $/i/j/x/l$, where $0 \leq l < S$.

Depending on the parameter values S and T , the number of necessary filesystem commands achieving these changes varies between 100 and 8000 per user. The instructions were chosen so that there are both a large number of conflicts and also a large number of non-conflicting command pairs, forcing any conflict-based synchronizer to spend quadratic time, even to check the existence of conflicts. The command sequences have many symmetries to ensure that the order in which the commands or command pairs are processed has little or no effect on the running time.

In the experiments, we tried 20 different filesystems with the parameter S running from 5 to 14 (inclusive), T running from 1 to $\lfloor (S - 1) / 2 \rfloor$, and the number of users running from 2 to $S - 1$, inclusive. The general synchronization Algorithm 7 from Section 6 was called on the resulting collection of sequences to generate the first and the first three possible synchronized states. Each run was repeated 10 times to smooth out the effects of other programs running on the same server. Figure 3 depicts the average time used to generate a single synchronizing command set. The input size on the x axis is the total number of different commands in the sequences to be synchronized. The average running time on the y axis is in seconds. The non-optimized Python program was running on a desktop machine with an Intel(R) Core(TM) i5-8250U CPU @ 1.60 GHz processor and 8 G memory. The different colors represent the number of replicas. The results confirm that the running time is subquadratic in the total input size and does not depend on the number of replicas.

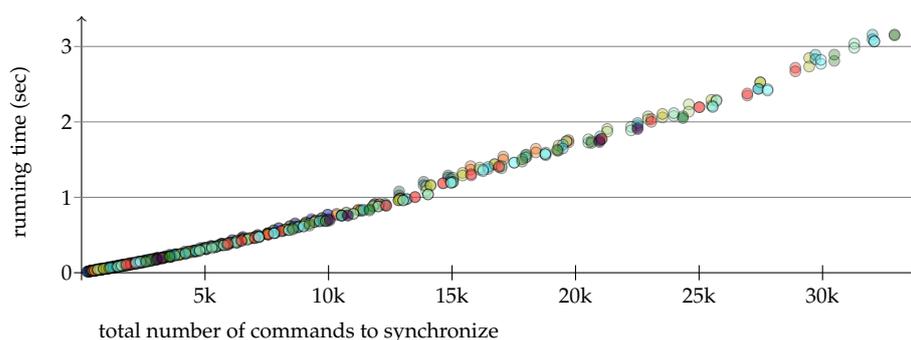


Figure 3. Running time for generating the synchronized state on several synthetic data sets. The running time depends only on the total number of filesystem commands (x axis), and not on the number of replicas (color).

8. Conclusions

This paper presented a provably correct synchronization algorithm running in subquadratic time, which can synchronize an arbitrary number of replicas. The existence of such an algorithm was a long-standing open problem [12] in the fields of operation transformation (OT) [5] and conflict-free replicated data types (CRDT) [9]. Our work is based on the algebraic theory of filesystems (ATF) [14], which, in many respects, resembles both OT and CRDT. Instead of traditional filesystem commands, ATF uses operations enriched with contextual information similarly to OT and CRDT. It also favors commutativity, but instead of requesting all operations to be commutative, their non-commutative part can be isolated systematically and handled separately. As a consequence, similarly to OT and CRDT, ATF can deal with command sets instead of sequences, where the execution order is not specified, but the semantics of executing the commands *in some order* is defined unambiguously.

The underlying filesystem model, while arguably simplistic, retains the most important high-level and platform-independent properties of real-life filesystems; see Section 2.1 for a more detailed discussion. The two most prominent omissions of this model are the lack of *directory attributes* (the model handles all directories as equal), and *links* breaking the

regular tree-like structure of filesystem paths. Both of these shortcomings, and how to circumvent them, are discussed below.

Filesystem synchronization starts with *update detection* run locally, which extracts information encoding the state of the modified replica as discussed in Section 3.1. In the ATF framework, this information is a *canonical command set* describing how to construct the replica from the original filesystem.

The task of the synchronizer is to create a common, merged filesystem after considering what changes have been applied to the replicas. In the ATF framework, this task amounts to creating another canonical command set, the *merger*, which transforms the original filesystem into the merged filesystem. What this merger can be is described by two simple and intuitively appealing principles: the *intention-confined effect*, where operations in the merger should come from those supplied by the replicas, and *aggressive effect preservation*, where the merger should contain as many of those commands as possible. Definition 5 formalizes this idea and defines what a synchronized state is. Sections 3.3 and 3.5 discuss that this goal-driven definition automatically implies many operational properties such that a merger command set always defines a valid synchronized state; synchronization can be achieved from the local copy by rolling back some of the local commands and executing additional ones originating from other replicas. Section 3.7 shows that with minimal effort the synchronization paradigm can be extended to tolerate replicas which do not lock their filesystem—allowing for asynchronous or optimistic synchronization [9]. Even replicas missing a synchronization cycle can later upgrade to the synchronized state.

Algorithms described in Section 4 give a high-level description of a proof-of-concept implementation at <https://github.com/csirmaz/algebraic-reconciler> (accessed on 10 May 2023). Algorithm 5 creates a merger in linear time after sorting the canonical sets sent by the replicas. While this algorithm can make some nondeterministic decisions, it cannot generate all possible mergers. Two nondeterministic algorithms which can do so are sketched in Section 6; both of them run in near linear time. The first algorithm uses the fact that Algorithm 5 can recognize mergers. It first creates a random subset of the supplied commands blindly, and then checks if it is a merger. The second one, described as Algorithm 7, is more elaborate. It exploits the structure of refluent command sets, and can assist in the synchronization process by highlighting the consequences of different conflict resolutions.

8.1. Node Attributes

The important task of consolidating different versions of the same document (file value) was not considered, as processing the internal structure of file contents is outside the scope of filesystem synchronization. Files different in content are considered to be different, and the synchronization algorithm forces the choosing of one or the other. There is, however, an easy way of incorporating third-party content-merging applications. This can be done by pretending that there is only one possible file content, and using the ATF framework to synchronize the *structure* of the filesystems. When it becomes clear which nodes contain file values, check for the commands which modified the actual content there, and use the external application to determine the final file content. A similar approach can handle node attributes by considering the changes made by the replicas at some node and consolidating them. This approach, however, should be followed carefully. To illustrate the problem, consider a directory at node n , which originally had the “private” attribute. Replica A changes this attribute to “public”, while replica B , under the impression that the directory is private, creates a file under it. When merging the attributes, the change at node n is carried over, making the directory publicly available. This, however, is clearly unacceptable. It is an interesting open problem to incorporate node attributes into the ATF synchronization paradigm.

8.2. Filesystems on Directed Acyclic Graphs

From the user's perspective, a (hard or soft) *link* between the nodes n and n' is a promise, or a commitment, that the filesystem at and below n is exactly the same at and below n' . In other words, the filesystem acts as if the nodes n and n' in the filesystem skeleton are glued together.

If the filesystem has many links and the links do not form loops, then after this gluing, the skeleton becomes a directed acyclic graph (DAG) with many sources (the roots in the original skeleton). The gluing works in the other direction, too: given any DAG with one or more sources, it can be "unfolded" into a forest. The paths of a tree-like filesystem can be identified with the directed paths starting from a source, and two nodes are "linked" if the directed paths in the DAG lead to the same vertex. A DAG vertex v represents the collection of all nodes in the unfolded filesystem, which are determined by the directed paths in the DAG, which lead to v . Two nodes of this unfolded filesystem are *equivalent*, written as $n_1 \simeq n_2$ if the corresponding directed DAG paths lead to the same vertex. It is clear that \simeq is an equivalence relation, and factoring the filesystem by \simeq yields the DAG. If every vertex in the DAG has a finite indegree, then the equivalence classes are also finite.

Operations on a DAG-based filesystem can be mimicked on the unfolded filesystem by simply requesting that an operation performed on the DAG vertex v be done on all nodes represented by v . Similarly, a command set A on the unfolded filesystem corresponds to the command set A/\simeq on the DAG-based filesystem if with every command $\sigma \in A$ all commands \simeq -equivalent to σ are also in A . We call these command sets \simeq -invariant. Requesting all command sets to be \simeq -invariant, the claims, propositions and theorems in this paper remain true (remark that in this case, the definition of a merger should require M to be \simeq -invariant). Similarly, all algorithms continue to work, but they must handle not commands but sets of \simeq -equivalent commands. Consequently, time estimates are no longer valid. In summary, our results and algorithms remain valid on filesystems based on arbitrary DAGs. It is an open question as to whether the algorithms can be implemented in linear time in the general case.

Author Contributions: All authors contributed equally to this work. All authors have read and agreed to the published version of the manuscript.

Funding: The work of the second author (L.Cs) was partially supported by the ERC Advanced Grant ERMiD.

Data Availability Statement: The Python program generating the synthetic data sets our algorithms were tested on is available at <https://github.com/csirmaz/algebraic-reconciler> (accessed on 25 May 2023).

Acknowledgments: Gábor Tardos' contribution for devising Algorithm 1 is gratefully acknowledged.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Athow, D.; Turner, B. Best File Syncing Solutions of 2023, 2023. Available online: <https://www.techradar.com/best/best-file-syncing-solution> (accessed on 10 May 2023).
2. Mościcki, J.T.; Mascetti, L. Cloud storage services for file synchronization and sharing in science, education and research. *Future Gener. Comput. Syst.* **2018**, *78*, 1052–1054. [[CrossRef](#)]
3. Sun, C.; Jia, X.; Zhang, Y.; Yang, Y.; Chen, D. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Trans. Comput. Hum. Interact.* **1998**, *5*, 63–108. [[CrossRef](#)]
4. Sun, C.; Ellis, C.A. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work, Seattle, WA, USA, 14–18 November 1998; Poltrock, S.E., Grudin, J., Eds.; ACM: New York, NY, USA, 1998; pp. 59–68. [[CrossRef](#)]
5. Shao, B.; Li, D.; Lu, T.; Gu, N. An Operational Transformation Based Synchronization Protocol for Web 2.0 Applications. In Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work, Hangzhou, China, 19–23 March 2011, CSCW '11; Association for Computing Machinery: New York, NY, USA, 2011; pp. 563–572. [[CrossRef](#)]

6. Ng, A.; Sun, C. 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, FL, USA, 13–16 November 2016; Lukosch, S.G., Sarcevic, A., Lewkowicz, M., Muller, M.J., Eds.; ACM: New York, NY, USA, 2016; pp. 61–70. [CrossRef]
7. Day-Richter, J. What’s Different about the New Google Docs: Making Collaboration Fast, 2010. Available online: <https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html> (accessed on 12 January 2023).
8. Shapiro, M.; Preguiça, N.M.; Baquero, C.; Zawirski, M. Conflict-Free Replicated Data Types. In Proceedings of the Stabilization, Safety, and Security of Distributed Systems-13th International Symposium, SSS 2011, Grenoble, France, 10–12 October 2011; Défago, X., Petit, F., Villain, V., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6976 , pp. 386–400. [CrossRef]
9. Preguiça, N.M. Conflict-free Replicated Data Types: An Overview. *arXiv* **2018**, arXiv:1806.10254.
10. Tao, V.; Shapiro, M.; Rancurel, V. Merging semantics for conflict updates in geo-distributed file systems. In Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, 26–28 May 2015; Naor, D., Heiser, G., Keidar, I., Eds; ACM: New York, NY, USA, 2015; pp. 1–12. [CrossRef]
11. Liu, E. A CRDT-Based File Synchronization System. Master’s Thesis, Norwegian University of Science and Technology, Trondheim, Norway , 2021.
12. Shekow, M. Syncpal: A Simple and Iterative Reconciliation Algorithm for File Synchronizers. Ph.D. Thesis, RWTH Aachen University, Aachen, Germany, 2019.
13. Csirmaz, E.P. Algebraic File Synchronization: Adequacy and Completeness. *arXiv* **2016**, arXiv:1601.01736.
14. Csirmaz, E.P.; Csirmaz, L. Data Synchronization: A Complete Theoretical Solution for Filesystems. *Future Internet* **2022**, *14*, 344. [CrossRef]
15. Shapiro, M.; Preguiça, N.; Baquero, C.; Zawirski, M. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*; Technical Report 7506; INRIA, Inria-Centre Paris-Rocquencourt: Le Chesnay-Rocquencourt, France , 2011.
16. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed.; Addison-Wesley: Boston, MA, USA , 1997.
17. Balasubramaniam, S.; Pierce, B.C. What is a File Synchronizer? In Proceedings of the MOBICOM ’98, The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Dallas, TX, USA, 25–30 October 1998; Osborne, W.P., Moghe, D.B., Eds.; ACM: New York, NY, USA, 1998; pp. 98–108. [CrossRef]
18. Tridgell, A.; Mackerras, P. *The Rsync Algorithm*; Australian National University: Canberra, Australia , 1996.
19. Boškov, N.; Trachtenberg, A.; Starobinski, D. Enabling Cost-Benefit Analysis of Data Sync Protocols. *arXiv* **2023**, arXiv:2303.17530.
20. Preguiça, N.; Marques, J.M.; Shapiro, M.; Letia, M. A Commutative Replicated Data Type for Cooperative Editing. In Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems, Montreal, QC, Canada, 22–26 June 2009; ICDCS ’09; IEEE Computer Society: Washington, DC, USA, 2009; pp. 395–403. [CrossRef]
21. Antkiewicz, M.; Czarnecki, K. Design Space of Heterogeneous Synchronization. In Proceedings of the Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, 2–7 July 2007; Revised Papers; Lämmel, R., Visser, J., Saraiva, J., Eds.; Springer: Berlin, Heidelberg, 2008; pp. 3–46. [CrossRef]
22. Li, Z.; Wilson, C.; Jiang, Z.; Liu, Y.; Zhao, B.Y.; Jin, C.; Zhang, Z.L.; Dai, Y. Efficient Batched Synchronization in Dropbox-Like Cloud Storage Services. In Proceedings of the Middleware 2013, Beijing, China, 9–13 December 2013; Eysers, D., Schwan, K., Eds.; Springer: Berlin, Heidelberg, 2013; pp. 307–327.
23. Petroni, A.; Cuomo, F.; Schepis, L.; Biagi, M.; Listanti, M.; Scarano, G. Adaptive Data Synchronization Algorithm for IoT-Oriented Low-Power Wide-Area Networks. *Sensors* **2018**, *18*, 4053. [CrossRef] [PubMed]
24. Feng, J.; Qiao, X.; Li, Y. The research of synchronization and consistency of data in mobile environment. In Proceedings of the 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems, Hangzhou, China, 30 October–1 November 2012; Volume 2, pp. 869–874. [CrossRef]
25. Klophaus, R. Riak Core: Building Distributed Applications without Shared State. In Proceedings of the ACM SIGPLAN Commercial Users of Functional Programming, Baltimore, MD, USA, 1–2 October 2010, CUFP ’10; Association for Computing Machinery: New York, NY, USA, 2010. [CrossRef]
26. Qian, Y. Data Synchronization and Browsing for Home Environments. Ph.D. Thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands , 2004. [CrossRef]
27. Zhang, Y.; Dragga, C.; Arpaci-Dusseu, A.; Arpaci-Dusseu, R. *-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems, Renton, WA, USA, 8–9 July 2019, HotStorage’13; USENIX Association: Berkeley, CA, USA, 2013; p. 2.
28. Even, S. *Graph Algorithms*, 2nd ed.; Cambridge University Press: Cambridge, UK , 2011. [CrossRef]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.