A highly-available move operation for replicated trees

Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford

Abstract—Replicated tree data structures are a fundamental building block of distributed filesystems, such as Google Drive and Dropbox, and collaborative applications with a JSON or XML data model. These systems need to support a *move* operation that allows a subtree to be moved to a new location within the tree. However, such a move operation is difficult to implement correctly if different replicas can concurrently perform arbitrary move operations, and we demonstrate bugs in Google Drive and Dropbox that arise with concurrent moves. In this paper we present a CRDT algorithm that handles arbitrary concurrent modifications on trees, while ensuring that the tree structure remains valid (in particular, no cycles are introduced), and guaranteeing that all replicas converge towards the same consistent state. Our algorithm requires no synchronous coordination between replicas, making it highly available in the face of network partitions. We formally prove the correctness of our algorithm using the Isabelle/HOL proof assistant, and evaluate the performance of our formally verified implementation in a geo-replicated setting.

Index Terms—Conflict-free Replicated Data Types (CRDTs), formal verification, distributed filesystems, distributed collaboration

1 Introduction

Many applications use a tree-structured data model. Most filesystems are trees: directories are branch nodes, files are leaf nodes (we discuss hardlinks in §3.7). XML and JSON documents are also trees, and they are used in many applications to represent e.g. rich text (a tree of paragraphs, lists, figures, sections, etc.), vector graphics, CAD drawings, and many other types of data. Typically, a graphical user interface allows a user to edit this information interactively, resulting in updates to the underlying XML/JSON structure: adding or deleting nodes in the tree, and moving nodes from one position in the tree to another.

In distributed filesystems and collaborative multi-user software, this tree is replicated across multiple nodes. If users attempt to update the tree concurrently on different replicas, concurrency control is required. However, standard techniques such as two-phase locking require synchronous coordination between replicas. If the software is running on mobile devices with unreliable network connectivity, an application based on synchronous coordination becomes unresponsive during network interruptions, leaving users unable to work while they are offline.

If we want to allow offline work, we must allow the system to continue processing read and write requests even in the presence of arbitrary network partitions; in other words, we require high *availability* and *partition-tolerance* in the sense of the CAP theorem [1]. We can achieve this goal by using *optimistic replication* [2], which means that any replica can make changes to the data without waiting for communication with any other replicas; updates made while disconnected are sent to other replicas later when a network connection is available. Besides allowing the system to tolerate network partitions, this approach can also improve performance for end users because the response time of a user request is independent of any network latency.

Dropbox and Google Drive are widely-deployed examples of optimistically replicated filesystems: they run a daemon on the user's machine that watches a designated directory for changes. The user can read and arbitrarily modify the files on their local disk, even while their computer is offline. However, when the filesystem is concurrently updated on different computers, Google Drive and Dropbox exhibit bugs in their concurrency control, as we show in §2.

In this paper we introduce a novel algorithm for handling concurrent updates to a replicated tree, such as an XML document or filesystem. It allows replicas to manipulate the tree by creating nodes, deleting nodes, or moving subtrees to a new location within the tree. We rule out bugs like those in Google Drive by formally proving our algorithm correct using the Isabelle/HOL proof assistant.

Our algorithm supports optimistic replication, allowing replicas to temporarily diverge as they are updated, and ensuring that they always converge towards a consistent state. It is an example of a *Conflict-free Replicated Data Type* or CRDT [3], and it guarantees a consistency model called *strong eventual consistency* [3], [4]. Our contributions are:

- We define a Conflict-free Replicated Data Type for trees that allow *move* operations without any coordination between replicas such as locking or consensus. As discussed in §2.3, this has previously been thought to be impossible to achieve [5], [6].
- We formalise the algorithm using Isabelle/HOL [7], a proof assistant based on higher-order logic, and obtain a computer-checked proof of correctness. In particular, we prove that arbitrary concurrent modifications to the tree can be merged such that all replicas converge to a consistent state, while preserving

- *M. Kleppmann and A.R. Beresford are with the University of Cambridge.*
- *D.P. Mulligan is with Arm Research, Cambridge, UK.*
- *V.B.F. Gomes is with Google (this work was done while at the University of Cambridge).*

the tree structure. Our proof technique can also be applied to other distributed systems, making it of independent interest.

- To demonstrate its practical viability, we extract a formally verified Scala implementation of our algorithm from Isabelle. We compare its performance to a hand-optimised (not formally verified) implementation and to classic state machine replication. In a geo-replicated setup across three continents, state machine replication has approximately four times higher throughput than our algorithm, but our algorithm has up to 100,000 times lower latency.
- We perform experiments with Dropbox and Google Drive, and show that they exhibit problems that would be prevented by our algorithm.

2 Why a move operation is hard

Applications that rely on a tree data model often need to move a node from one location to another location within the tree, such that all of its children move along with it:

- In a filesystem, any file or directory can be moved to a different parent directory. Moreover, renaming a file or directory is equivalent to moving it to a new name without changing its parent directory.
- In a rich text editor, a paragraph can be turned into a bullet point. In the XML tree this corresponds to creating new list and bullet point nodes, and then moving the paragraph node inside the bullet point.
- In graphics software, grouping two objects corresponds to creating a new group node, and then moving the two objects into the new group node.

As these operations are so common, it is not obvious why a move operation should be difficult in a replicated setting. In this section we demonstrate some problems that arise with replicated trees, before proceeding to our solution in §3.

2.1 Concurrent moves of the same node

The first difficulty arises when the same node is concurrently moved into different locations on different replicas. This scenario is illustrated in Figure 1, where replica 1 moves node A to be a child of B, while concurrently replica 2 moves A to be a child of C. After the replicas communicate, what should the merged state of the tree be?

If a move operation is implemented by deleting the moved subtree from its old location, and then re-creating it at the new location, the merged state will be as shown in Figure 1a: the concurrent moves will duplicate the moved subtree, since each move independently recreates the subtree in each destination location. We believe that this duplication is undesirable, since subsequent edits to nodes in the duplicated subtree will apply to only one of the copies. Two users who believe they are collaborating on the same file may in fact be editing two different copies, which will then become inconsistent with each other. In the rich text editor and graphics software examples, such duplication is also undesirable.

Another possible resolution is for the destination locations of both moves to refer to the same node, as shown in Figure 1b. However, the result is a DAG, not a tree. POSIX filesystems do not allow this outcome, since they do not allow hardlinks to directories.

In our opinion, the only reasonable outcomes are those shown in Figure 1c and 1d: the moved subtree appears either in replica 1's destination location or in replica 2's destination location, but not in both. Which one of these two is picked is arbitrary, due to the symmetry between the two replicas. The "winning" location could be picked based on a timestamp in the operations, similarly to the "last writer wins" conflict resolution method of Thomas's write rule [8]. The *timestamp* in this context need not come from a physical clock; it could also be logical, such as a Lamport timestamp [9].

We tested this scenario with file sync products Dropbox and Google Drive by concurrently moving the same directory to two different destination directories.[1] Dropbox exhibited the undesirable duplication behaviour of Figure 1a, while the outcome on Google Drive was as in Figure 1c/d.

2.2 Moving a node to be a descendant of itself

On a filesystem, the destination directory of a move operation must not be a subdirectory of the directory being moved. For example, if `b` is a subdirectory of `a`, then the Unix shell command `mv a a/b/` will fail with an error. This restriction is required because allowing this operation would introduce a cycle into the directory graph, and so the filesystem would no longer be a tree. Any tree data structure that supports a move operation must handle this case.

In an unreplicated tree it is easy to prevent cycles being introduced: if the node being moved is an ancestor of the destination node, the operation is rejected. However, in a replicated setting, different replicas may perform operations that are individually safe, but whose combination leads to a cycle. One such example is illustrated in Figure 2. Here, replica 1 moves B to be a child of A, while concurrently replica 2 moves A to be a child of B. As each replica propagates its operation to the other replica, a careless implementation might end up in the state shown in Figure 2a: A and B have formed a cycle, detached from the tree.

Another possible outcome is shown in Figure 2b: the nodes involved in the concurrent moves (and their children) could be duplicated, so that both "A as a child of B" and "B as a child of A" can exist in the tree. However, such duplication is undesirable for the same reasons as in §2.1.

In our opinion, the best way of handling the conflicting operations of Figure 2 is to choose either Figure 2c or 2d: that is, either the result of applying replica 1's operation and ignoring replica 2's operation, or vice versa. Like in §2.1, the winning operation can be picked based on a timestamp.

As before, we tested this scenario with Google Drive and Dropbox. In Google Drive, one replica was able to successfully sync with the server, while the other replica displayed the "unknown error" message shown in Figure 3.

[1] Experiment setup: we installed the official Mac OS clients for Dropbox and Google Drive on two computers, logged into the same Dropbox/Google accounts, and configured them to sync a directory on the local filesystem. To test concurrent operations, we disconnected both computers from the Internet, performed a move operation on the local filesystem of each computer, then reconnected and waited for them to sync.

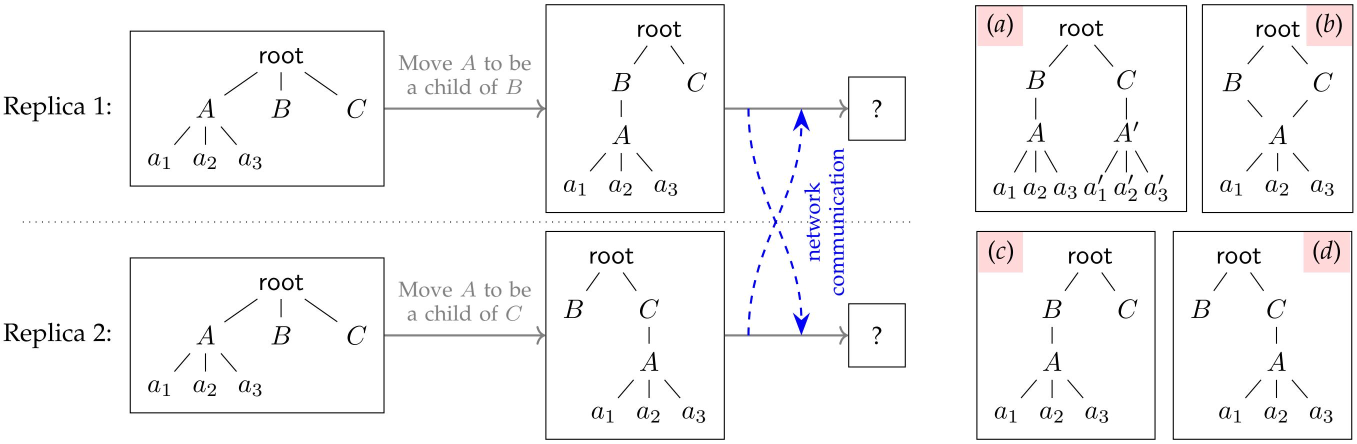


Fig. 1. Replica 1 moves A to be a child of B, while concurrently replica 2 moves the same node A to be a child of C. Boxes (*a*) to (*d*) show possible outcomes after the replicas have communicated and merged their states.

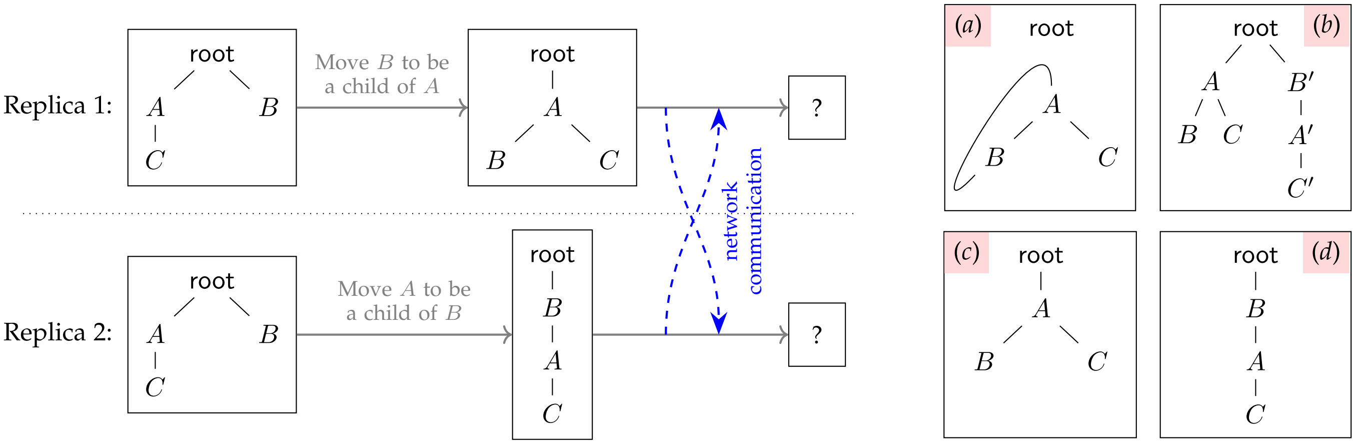
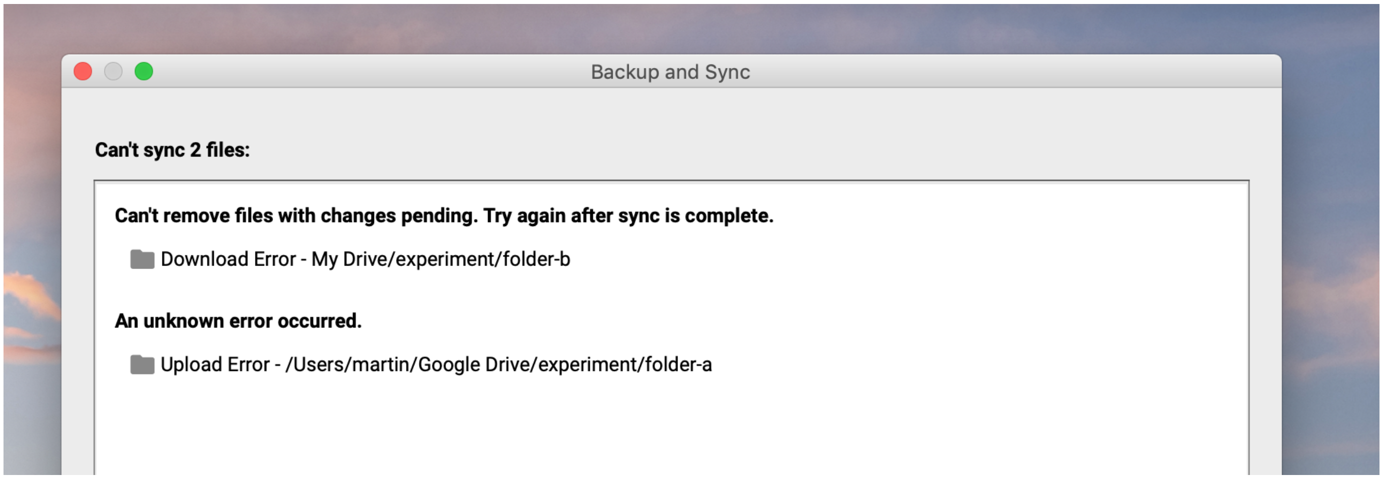


Fig. 2. Initially, nodes A and B are siblings. Replica 1 moves B to be a child of A, while concurrently replica 2 moves A to be a child of B. Boxes (*a*) to (*d*) show possible outcomes after the replicas have communicated and merged their states.



Can't sync 2 files:

Can't remove files with changes pending. Try again after sync is complete.

Download Error - My Drive/experiment/folder-b

An unknown error occurred.

Upload Error - /Users/martin/Google Drive/experiment/folder-a

Fig. 3. Error message produced by Google Drive Backup and Sync on Mac OS as a result of performing the operations shown in Figure 2, indicating a bug in the underlying replication algorithm.

The replica in an error state refused to sync the conflicting directory, and its filesystem state remained permanently inconsistent with the other replica. This error state persisted until the directories on the erroring replica were manually moved to match the state of the other replica. We have reported this bug to Google. On the other hand, Dropbox exhibited the duplication behaviour shown in Figure 2b.

2.3 Is a highly-available move operation impossible?

Najafzadeh et al. [5], [6] previously implemented a replicated filesystem with a move operation, and analysed the case of concurrent move operations introducing a cycle. Using the CISE proof tool [10], [11] the authors confirm that it is not sufficient for the replica that generates a move operation to check whether the operation introduces a cycle: like in Figure 2, two concurrent operations may be safe individually, but introduce a cycle when combined.

Najafzadeh et al. propose two solutions to this problem: either to duplicate tree nodes, as in Figure 2b, or to execute a synchronous locking protocol that prevents two move operations from concurrently modifying the same part of the tree. The downside of a locking protocol is that the move operation is no longer highly available in the presence of network partitions, since it must wait for synchronous communication with other replicas or a lock server.

While these solutions are valid, the authors go on to claim that "no file system can support an unsynchronised move without anomalies, such as loss or duplication" [6]. We refute that claim in this paper: our algorithm does not perform any locking, coordination or synchronisation among replicas, but it nevertheless ensures that the tree invariants are always satisfied (in particular, it never introduces cycles), and it never duplicates or loses any tree nodes. To our knowledge, our algorithm is the first to provide all of these properties simultaneously. We give a precise specification of our algorithm's consistency properties in §4.

3 The Replicated Tree Algorithm

We now introduce our algorithm for a replicated tree that supports a move operation. We model each replica as a state machine that transitions from one state to the next by applying an operation. The algorithm is executed independently on each replica with no shared memory between replicas.

When the user wants to make a change to the tree, they generate an operation and apply it to their local replica. Every operation is also asynchronously sent over the network to all other replicas, and applied by every remote replica using the same algorithm as for local operations. A replica may communicate directly with any other replica. The network may arbitrarily delay or reorder messages, but we assume that an underlying network protocol detects and retransmits lost messages, and suppresses duplicates. We do not assume any central server or consensus protocol. Any number of replicas may fail by crashing, and any non-crashed subset of replicas can continue executing operations.

The key consistency property of our algorithm is *convergence*: that is, whenever any two replicas have applied the same set of operations, then they must be in the same state—even if the operations were applied in a different order on different replicas. We prove this in §4 by showing that applying operations using our algorithm is commutative.

Figure 4 gives the full source code for our algorithm in the Isabelle/HOL language [7]. We choose this language because it combines the conciseness of pseudocode with the precision of mathematical notation. It supports formal reasoning, allowing us to prove the correctness of the algorithm (§4), and it can be exported to Scala, Haskell, or OCaml.

In this section we walk through the code step by step, explaining the Isabelle/HOL syntax as we encounter it. Additional background documentation is available [12].

3.1 Operations and trees

We allow the tree to be updated in three ways: by creating a new child of any parent node, by deleting a node, or by moving a node to be a child of a new parent. However, all three types of update can be represented by a move operation. To create a node, we generate a fresh ID for that node, and issue an operation to move this new ID to be a child of its desired parent; the node is then implicitly created. We also designate as "trash" some node ID that does not exist in the tree; then we can delete a node by moving it to be a child of the trash. Node creation and deletion are discussed further in §3.6.

Thus, we define one kind of operation: *Move t p m c* (Figure 4, lines 1–5). A move operation is a 4-tuple consisting of a timestamp t of type $'t$, a parent node ID p of type $'n$, a metadata field m of type $'m$, and a child node ID c of type $'n$. Here, $'t$, $'n$ and $'m$ are *type variables* that can be replaced with arbitrary types; we only require that node identifiers $'n$ are globally unique (e.g. UUIDs); timestamps $'t$ need to be globally unique and totally ordered (e.g. Lamport timestamps [9]).

The meaning of an operation *Move t p m c* is that at time t, the node with ID c is moved to be a child of the parent node with ID p. The operation does not specify the old location of c; the algorithm simply removes c from wherever it is currently located in the tree, and moves it to p. If c does not currently exist in the tree, it is created as a child of p.

The metadata field m in a move operation allows additional information to be associated with the parent-child relationship of p and c. For example, in a filesystem, the parent and child are the inodes of a directory and a file within it, respectively, and the metadata contains the filename of the child. Thus, a file with inode c can be renamed by performing a *Move t p m c*, where the new parent directory p is the inode of the existing parent (unchanged), but the metadata m contains the new filename.

When users want to make changes to the tree on their local replica, they generate new *Move t p m c* operations for these changes, and apply these operations using the algorithm described in the rest of this section.

We can represent the tree as a set of *(parent, meta, child)* triples, denoted in Isabelle/HOL as $('n \times 'm \times 'n)\ set$. When we have $(p, m, c) \in tree$, that means c is a child of p in the tree, with associated metadata m. Given a *tree*, we can construct a new *tree'* in which the child c is moved to a new parent p, with associated metadata m, as follows:

$$tree' = \{(p', m', c') \in tree.\ c' \neq c\} \cup \{(p, m, c)\}$$

```
datatype ('t, 'n, 'm) operation
  = Move (move_time: 't)
        (move_parent: 'n)
        (move_meta: 'm)
        (move_child: 'n)

datatype ('t, 'n, 'm) log_op
  = LogMove (log_time: 't)
           (old_parent: ('n × 'm) option)
           (new_parent: 'n)
           (log_meta: 'm)
           (log_child: 'n)

type_synonym ('t, 'n, 'm) state = ('t, 'n, 'm) log_op list × ('n × 'm × 'n) set

definition get_parent :: ('n × 'm × 'n) set ⇒ 'n ⇒ ('n × 'm) option where
  get_parent tree child ≡
    if ∃!parent. ∃!meta. (parent, meta, child) ∈ tree then
      Some (THE (parent, meta). (parent, meta, child) ∈ tree)
    else None

inductive ancestor :: ('n × 'm × 'n) set ⇒ 'n ⇒ 'n ⇒ bool where
  ⟦(parent, meta, child) ∈ tree⟧ ⟹ ancestor tree parent child |
  ⟦(parent, meta, child) ∈ tree; ancestor tree anc parent⟧ ⟹ ancestor tree anc child

fun do_op :: ('t, 'n, 'm) operation × ('n × 'm × 'n) set ⇒ ('t, 'n, 'm) log_op × ('n × 'm × 'n) set where
  do_op (Move t newp m c, tree) =
    (LogMove t (get_parent tree c) newp m c,
     if ancestor tree c newp ∨ c = newp then tree
     else {(p', m', c') ∈ tree. c' ≠ c} ∪ {(newp, m, c)})

fun undo_op :: ('t, 'n, 'm) log_op × ('n × 'm × 'n) set ⇒ ('n × 'm × 'n) set where
  undo_op (LogMove t None newp m c, tree) = {(p', m', c') ∈ tree. c' ≠ c} |
  undo_op (LogMove t (Some (oldp, oldm)) newp m c, tree) =
    {(p', m', c') ∈ tree. c' ≠ c} ∪ {(oldp, oldm, c)}

fun redo_op :: ('t, 'n, 'm) log_op ⇒ ('t, 'n, 'm) state ⇒ ('t, 'n, 'm) state where
  redo_op (LogMove t _ p m c) (ops, tree) =
    (let (op2, tree2) = do_op (Move t p m c, tree)
     in (op2 # ops, tree2))

fun apply_op :: ('t::{linorder}, 'n, 'm) operation ⇒ ('t, 'n, 'm) state ⇒ ('t, 'n, 'm) state where
  apply_op op1 ([], tree1) =
    (let (op2, tree2) = do_op (op1, tree1)
     in ([op2], tree2)) |
  apply_op op1 (logop # ops, tree1) =
    (if move_time op1 < log_time logop
     then redo_op logop (apply_op op1 (ops, undo_op (logop, tree1)))
     else let (op2, tree2) = do_op (op1, tree1) in (op2 # logop # ops, tree2))

definition apply_ops :: ('t::{linorder}, 'n, 'm) operation list ⇒ ('t, 'n, 'm) state where
  apply_ops ops ≡ foldl (λstate oper. apply_op oper state) ([], {}) ops

definition unique_parent :: ('n × 'm × 'n) set ⇒ bool where
  unique_parent tree ≡
   (∀p1 p2 m1 m2 c. (p1, m1, c) ∈ tree ∧ (p2, m2, c) ∈ tree ⟶ p1 = p2 ∧ m1 = m2)

definition acyclic :: ('n × 'm × 'n) set ⇒ bool where
  acyclic tree ≡ (∄n. ancestor tree n n)
```

Fig. 4. The move operation algorithm, implemented in the Isabelle/HOL language.

That is, we remove any existing parent-child relationship for *c* from the set *tree*, and then add $\{(p, m, c)\}$ to represent the new parent-child relationship. This expression appears on lines 30 and 35 of Figure 4, as we shall explain shortly.

3.2 Replica state and operation log

In order to correctly apply move operations, a replica needs to maintain not only the current state of the tree, but also an *operation log*. The log is a list of *LogMove* records in descending timestamp order. *LogMove t oldp p m c* (lines 7–12) is similar to *Move t p m c*; the difference is that *LogMove* has an additional field *oldp* of type $('n \times 'm)$ *option*. This *option* type means the field can either take the value *None* (similar to null), or a pair of a node ID and a metadata field.

When a replica applies a *Move* operation to its tree, it also records a corresponding *LogMove* operation in its log. The *t*, *p*, *m* and *c* fields are taken directly from the *Move* record, while the *oldp* field is filled in based on the state of the tree before the move. If *c* did not exist in the tree, *oldp* is set to *None*. Otherwise, *oldp* records the previous parent and metadata of *c*: if there exist *p′* and *m′* such that $(p', m', c) \in tree$, then *oldp* is set to *Some* (p', m').

The *get_parent* function (lines 16–20) implements this. In the first line of *get_parent*, the expression between *::* and **where** is the type signature of the function, in this case:

$$('n \times 'm \times 'n)\ set \Rightarrow 'n \Rightarrow ('n \times 'm)\ option$$

This signature denotes a function that takes two arguments: a tree $('n \times 'm \times 'n)$ *set* and a node ID *′n*. It then returns a $('n \times 'm)$ *option*. The operator $\exists!x$ means “there exists a unique value *x* such that. . . ”, while *THE x* means “*choose* the unique value *x* such that. . . ”.

In line 14 we define the datatype for the state of a replica: a pair *(log, tree)* where *log* is a list of *LogMove* records, and *tree* is a set of *(parent, meta, child)* triples as before.

3.3 Preventing cycles

Recall from §2.2 that in order to prevent a cycle being introduced, the node being moved must not be an ancestor of the destination node. To implement this we first define the *ancestor* relation in lines 22–24. It is the transitive closure of a tree’s parent-child relation: if $(p, m, c) \in tree$ then *p* is an ancestor of *c* (line 23); moreover, if *a* is an ancestor of *p* and $(p, m, c) \in tree$, then *a* is also an ancestor of *c* (line 24). The **inductive** keyword indicates that this recursive definition is iterated until the least fixed point is reached.

The *do_op* function (lines 26–30) now performs the actual work of applying a move operation. This function takes as argument a pair consisting of a *Move* operation and the current tree, and it returns a pair consisting of a *LogMove* operation (which will be added to the log) and an updated tree. In line 28, the *LogMove* record is constructed as described in §3.2, obtaining the prior parent and metadata of *c* using the *get_parent* function.

Line 29 performs the check that ensures no cycles are introduced: if *ancestor tree c newp*, i.e. if the node *c* is being moved, and *c* is an ancestor of the new parent *newp*, then the tree is returned unmodified—in other words, the operation is ignored. Similarly, the operation is also ignored if *c* = *newp*. Otherwise (line 30), the tree is updated by removing *c* from its existing parent, if any, and adding the new parent-child relationship *(newp, m, c)* to the tree.

3.4 Applying operations in any order

The *do_op* function is sufficient for applying operations if all replicas apply operations in the same order. However, in an optimistic replication setting, each replica may apply the operations in a different order, and we need to ensure that the replica state nevertheless converges towards a consistent state. This goal is accomplished by the *undo_op*, *redo_op*, and *apply_op* functions (lines 32–49).

When a replica needs to apply an operation with timestamp *t*, it first undoes the effect of any operations with a timestamp greater than *t*, then performs the new operation, and finally re-applies the undone operations. As a result, the state of the tree is as if the operations had all been applied in order of increasing timestamp, even though in fact they might have been applied in any order.

The *apply_op* function (lines 42–49) takes two arguments: a *Move* operation to apply and the current replica state; and it returns the new replica state. The constraint *′t::*{*linorder*} in the type signature indicates that timestamps *′t* are instances of the *linorder* type class, and they can therefore be compared with the < operator defining a linear (or total) order. This comparison occurs on line 47.

Recall that the replica state includes the operation log (line 14), and we use this log to perform the undo-do-redo cycle. Lines 43–45 handle the case where the log is empty: in this case, we simply perform the operation using *do_op*, and return the new tree along with a log containing a single *LogMove* record. If the log is nonempty (line 46), we take *logop* to be the first element of the log, and *ops* to be the rest. (The hash character in *logop* # *ops* is the *list cons* operator that adds one element to the head of a list.) If the timestamp of *logop* is greater than the timestamp of the new operation (line 47) we first undo *logop* with *undo_op*, then recursively apply the new operation to the remaining log, and finally reapply *logop* with *redo_op* (line 48). Otherwise we perform the operation using *do_op*, and add the corresponding *LogMove* record as the head of the log (line 49).

This logic ensures that the log is maintained in descending timestamp order, with the greatest timestamp at the head. *undo_op* (lines 32–35) inverts the effect of a previous move operation by restoring the prior parent and metadata that were recorded in the *LogMove*’s additional field. *redo_op* (lines 37–40) uses *do_op* to perform an operation again and recomputes the *LogMove* record (which might have changed due to the effect of the new operation).

3.5 Handling conflicts

Due to the undo-do-redo cycle, the state of the tree is as if all operations had been applied using *do_op* in increasing timestamp order, regardless of the order in which they were actually applied. This provides a clear and consistent approach to the handling of conflicts:

- If two operations concurrently move the same node, the operation with the lower timestamp moves the node first, and then the operation with the greater timestamp moves it again, so the final parent is determined by the latter. Since move operations do not specify the old location of a node, but only the new location, this sequential execution of concurrent operations is well-defined. In Figure 1, the outcome is (c) if the operation from replica 1 has the greater timestamp, or (d) if the operation from replica 2 has the greater timestamp.

- If two operations would introduce a cycle when combined, as in §2.2, then the operation with the greater timestamp is ignored because *do_op* checks for cycles based on the tree created by all operations with a lower timestamp. The lower of two conflicting operations will take effect, since that operation by itself is safe. When the higher-timestamped operation is applied, *do_op* detects that it would introduce a cycle, and therefore ignores the operation. In the example of Figure 2, the outcome is (c) if the operation from replica 1 has the lower timestamp, or (d) if the operation from replica 2 has the lower timestamp.

This resolution of a conflict between two operations can be generalised to any number of conflicting operations by repeatedly applying the rules pairwise.

Note that the safety of an operation (whether or not it would introduce a cycle) may change as subsequent operations with lower timestamps are applied. For example, an operation may initially be regarded as safe, and then be reclassified as unsafe after applying a conflicting operation with a lower timestamp. The opposite is also possible: an operation previously regarded as unsafe may become safe through the application of an operation that removes the risk of introducing a cycle. For this reason, the operation log must include all operations, even those that were ignored.

One final type of conflict that we have not discussed so far is multiple child nodes with the same parent and the same metadata. For example, in a filesystem, two users could concurrently create files with the same name in the same directory. Our algorithm does not prevent such a conflict, but simply retains both child nodes. In practice, the collision would be resolved by making the filenames distinct, e.g. by appending a replica identifier to the filenames.

3.6 Node creation and deletion

As discussed previously, no separate operations for node creation and deletion are needed, since a node is implicitly created when it is first moved, and a node can be deleted by moving it to a designated trash node outside of the tree. By avoiding a distinction between different operation types we simplify the algorithm and proofs of correctness, but we potentially sacrifice performance in applications that mostly create and delete nodes, and only occasionally move nodes. Ideally, we would want to incur the costs of the undo-do-redo process only for genuine move operations, and not for node creation and deletion.

This optimisation is possible for operations that create new tree nodes: such operations can be directly applied to the tree without any undo/redo by using *do_op* instead of *apply_op*, and they do not need to recorded in the log. We have proved in Isabelle that this optimisation is safe, under the following additional assumptions: 1. the parent node in any move/create operation must exist in the tree; 2. the creation operation for a given node must be unique (i.e. there cannot be two operations that both create the same node); 3. a node creation operation is applied before any operation that moves the created node. These assumptions are easily satisfied in practice.

For node deletion we have not been able to find a similar optimisation. Deletion operations must go through the undo-do-redo process because deleting a node may cause a previously unsafe move operation with a greater timestamp to become safe by breaking a particular ancestor relationship; and as the previously ignored operation takes effect, we must re-evaluate all operations with greater timestamps to determine whether they are safe or not.

When a node is deleted, any children of the deleted node remain in the tree, but since they are no longer reachable from the root, they effectively become invisible to the application. We do not recursively remove children, since a move operation concurrent to the deletion operation might move the deleted subtree back out of the trash and into the visible tree; in this scenario we want all children of the moved subtree to be preserved unmodified.

3.7 Algorithm extensions

Hardlinks and symlinks. Unix filesystems support hardlinks, which allow the same file inode to be referenced from multiple locations in a tree. Our tree data structure can easily be extended to support this: rather than placing file data directly in the leaf nodes of the tree, the leaf node must reference the file inode. Thus, references to the same file inode can appear in multiple leaf nodes of the tree. Symlinks are also easy to support, since they are just leaf nodes containing a path (not a reference to an inode).

Log truncation. The algorithm as specified in Figure 4 retains operations in the log indefinitely, so the memory use grows without bound. However, in practice it is easy to truncate the log, because *apply_op* only examines the log prefix of operations whose timestamp is greater than that of the operation being applied. Thus, once it is known that all future operations will have a timestamp greater than t, then operations with timestamp t or less can be discarded from the log. In this case, we say that t is *causally stable* [13].

A similar approach can be used to garbage-collect any tree nodes in the trash (§3.1). Initially, trashed nodes must be retained because a concurrent move operation may move them back out of the trash. However, once the operation that moved a node to the trash is causally stable, we know that no future operations will refer to this node, and so the trashed node and its descendants can be discarded.

We can determine causal stability in a system where the set of replicas is known, where each replica generates operations with monotonically increasing timestamps, and where the communication link between any pair of replicas is FIFO (messages are received in the order in which they are sent, as implemented e.g. by TCP). In this case, we can keep track of the most recent timestamp we have seen from each replica (including our own), and the minimum of these timestamps is the causally stable threshold.

Ordering of sibling nodes. Another useful extension of the tree algorithm is to allow children of the same parent node to have an explicit ordering. For example, in XML, the set of children of an element is ordered. This can be implemented by maintaining an additional list CRDT for each branch node, e.g. using RGA [14] or Logoot [15]. These algorithms assign a unique ID to each element of the list, and this ID can be included in the metadata field of move operations in order to determine the order of sibling nodes.

This approach to determining ordering also easily supports reordering of child nodes within a parent: to move a node to a different position in a list, we use the list CRDT to generate a new ID at the desired position in the sequence [16]. Then we perform a move operation in which the parent node is unchanged, and this new ID is used as metadata.

4 Proof of correctness

We now discuss the correctness properties of the algorithm from §3. All theorems stated here have been formally proved and mechanically checked using Isabelle. For space reasons this paper gives only the statements that were proved, but elides a discussion of the reasoning steps. The Isabelle files containing the full details are open source [17], and are included as supplementary material with this paper.

To reason about the state of a replica we first define the function *apply_ops* on lines 51–52 of Figure 4. It takes a list of operations *ops* and returns the state of a replica after it has applied all the operations in *ops*. The *apply_ops* function works by starting in the initial state *([], {})* consisting of the empty operation log *[]* and the tree represented by the empty set {}, and then applying the operations one by one to the state using the *apply_op* function (introduced in §3.4). The *foldl* function from the Isabelle/HOL standard library performs the iteration over the list of operations.

4.1 Tree invariants

A tree is an acyclic graph in which every node has exactly one parent, except for the root, which has no parent. In fact, we slightly generalise this property and allow more than one root to exist, so the graph represents a forest, allowing an application to move nodes between different trees if desired. For example, the trash node used for deletion can be separate from the main tree. To prove that our algorithm maintains a forest structure, no matter which operations are applied, we demonstrate several invariants.

Each node's parent is unique. The first invariant we prove is that each tree node has either no parent (if it is the root of a tree) or exactly one parent (if it is a non-root node). We state this theorem in Isabelle/HOL as follows, where *apply_ops_unique_parent* is the name of this theorem:

> **theorem** *apply_ops_unique_parent*:
> **assumes** *apply_ops ops* = (*log*, *tree*)
> **shows** *unique_parent tree*

That is, we consider any list of operations *ops* and define *(log, tree)* to be the replica state after *ops* have been applied. We then prove that "*unique_parent tree*" holds, where the *unique_parent* predicate is defined on lines 54–56 of Figure 4: whenever the tree contains a triple whose third element is the child node *c*, then the first and second elements of the triple (the parent node and the metadata) are uniquely defined. As we make no assumptions about *ops*, this theorem holds for any replica state that can be reached by applying any number of operations.

The graph contains no cycles. This second invariant is expressed as follows in Isabelle/HOL:

> **theorem** *apply_ops_acyclic*:
> **assumes** *apply_ops ops* = (*log*, *tree*)
> **shows** *acyclic tree*

The *acyclic* predicate is defined on lines 58–59 of Figure 4, using the *ancestor* relation (the transitive closure of the graph's edges): a graph contains no cycles if no node is an ancestor of itself.

Other correctness properties. There are further criteria we might use to determine if our algorithm is correct. For example, we might demonstrate that a single-replica system operates with the usual sequential semantics of a tree. In a system with multiple disjoint trees, we could prove that the trees don't get tangled together. We conjecture that those properties hold for our algorithm, but leave the proof out of scope for this paper.

4.2 Convergence

As discussed in §3.4, we require that when replicas apply the same set of operations, they converge towards the same state, regardless of the order in which the operations are applied. We formalise this in Isabelle/HOL as follows:

> **theorem** *apply_ops_commutes*:
> **assumes** *set ops1* = *set ops2*
> **and** *distinct* (*map move_time ops1*)
> **and** *distinct* (*map move_time ops2*)
> **shows** *apply_ops ops1* = *apply_ops ops2*

The predicate *distinct* takes a list as argument, and returns true if all elements of the list are distinct (i.e. no value occurs more than once in the list). Therefore, the assumption *distinct (map move_time ops1)* states that in the list *ops1*, there are no two operations with the same timestamp.

The function *set* takes a list and turns it into an unordered set with the same elements. Thus, the assumption *set ops1 = set ops2* means that the lists *ops1* and *ops2* contain the same elements, but perhaps in a different order—in other words, *ops1* is a permutation of *ops2*. Under these assumptions, *apply_ops_commutes* proves that applying the list of operations *ops1* results in the same replica state as applying the list of operations *ops2*.

Strong eventual consistency. Gomes et al. [4] define a framework in Isabelle/HOL for proving the strong eventual consistency properties of CRDTs. Using our convergence proof above we integrate our tree datatype with this framework, and thus demonstrate that our move operation on trees does indeed guarantee strong eventual consistency. The details appear in our Isabelle theory files [17].

4.3 Making the HOL definitions executable

Isabelle/HOL can generate executable Haskell, OCaml, Scala, and Standard ML code from HOL definitions using a sophisticated code generation mechanism [18]. However, not all definitions can be realised in executable form: for example, the use of choice principles (like in *get_parent*) and inductively defined relations (e.g. *ancestor*) cause problems. Moreover, HOL's *set* type allows infinite sets. Whilst these

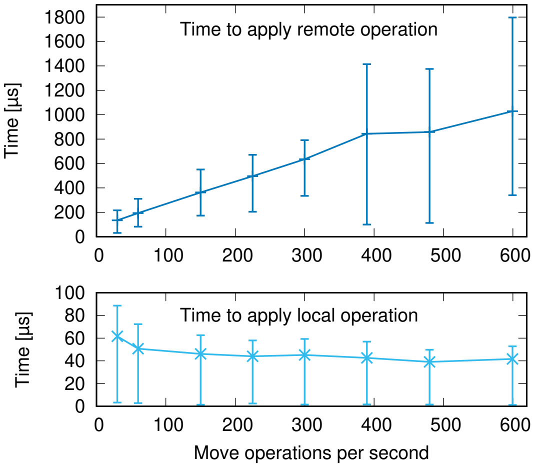


Fig. 5. Median execution time to apply an operation to the replica state, using Scala code generated by Isabelle. Error bars indicate the minimum and 95th percentile.

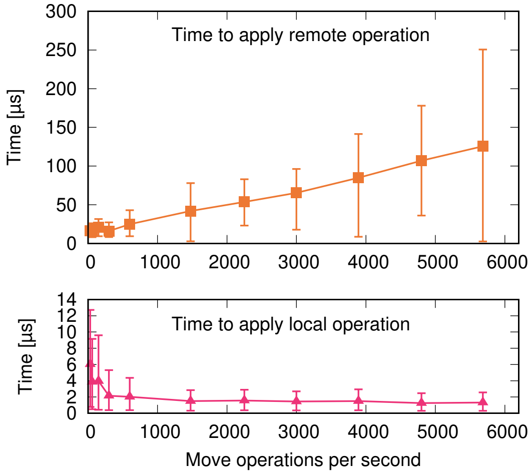


Fig. 6. Median execution time to apply an operation to the replica state, using a hand-optimised (not formally verified) Scala implementation. Error bars indicate the minimum and 95th percentile.

constructs are convenient for theorem proving, they do not translate well to executable code.

We therefore produce variants of the definitions in Figure 4 that are designed for execution rather than theorem proving. Rather than representing the tree as a set of (parent, meta, child) triples, these definitions use a hash-map in which the keys are child nodes, and the values are (meta, parent) pairs, written in Isabelle/HOL as *(’n::{hashable}, ’m × ’n) hm*. The *hashable* type class means that keys must have a hashing function. In effect, this hash-map is an index over the set of triples, using the fact that the parent and metadata for a given child are unique (§4.1). The hash-map implementation is from the Isabelle Collections Framework [19].

A hash-map *t* of type *(’n, ’m × ’n) hm simulates* a set *T* of type *(’n × ’m × ’n) set* when their entries are the same:

definition *simulates* **where** *simulates t T* ≡
(∀ *p m c*. *hm.lookup c t* = *Some* (*m*, *p*) ⟷ (*p*, *m*, *c*) ∈ *T*)

where *hm.lookup c t* looks up the key *c* in the hash-map *t*, returning *Some x* if *c* maps to the value *x*, and returning *None* if *c* does not appear in the hash-map. We can now prove the equivalence of the set-based and the hash-map-based implementations:

lemma *executable_apply_ops_simulates*:
assumes *executable_apply_ops ops* = (*log1*, *t*)
and *apply_ops ops* = (*log2*, *T*)
shows *log1* = *log2* ∧ *simulates t T*

That is, if *executable_apply_ops* and *apply_ops* are applied to the same list of operations, they produce identical logs, and also produce trees that contain the same set of key-value bindings—i.e. the trees are extensionally equivalent, despite having very different in-memory representations. We can also prove corollaries of *apply_ops_commutes* and *apply_ops_acyclic* for the hash-map-based implementation.

5 Evaluation

5.1 Performance of move operation

With our algorithm, the worst-case cost of applying a move operation is $O(nd)$, where n is the number of operations in the log that need to be undone and redone, and d is the depth of the tree (the number of parent relationships that need to be traversed to check whether the operation would introduce a cycle). To demonstrate that this cost is acceptable in practice, we evaluated the performance of two implementations of our algorithm. The first implementation is Scala code extracted from our formally verified HOL definitions using Isabelle/HOL’s code generation mechanism. The second implementation is handwritten Scala code that we believe to be functionally equivalent, but which we have not formally verified. By comparing these implementations we can distinguish between inefficiencies introduced by code generation and costs that are inherent to our algorithm.

We wrapped both implementations in a simple network service and deployed three replicas on Amazon EC2 `c5.large` instances in Northern California (`us-west-1`), Ireland (`eu-west-1`), and Singapore (`ap-southeast-1`). The network delays between these regions are substantial: a round trip from Ireland to California takes a median of 145 ms, and from Singapore to California takes 176 ms.

We use a synthetic workload in which each replica starts with an empty tree and generates move operations at a fixed rate; for each move operation the generating replica chooses parent and child nodes uniformly at random from a set of 1,000 tree nodes. A tree node is created by the first operation that refers to it, and for simplicity we do not use the optimisation discussed in § 3.6. We use Lamport clocks [9] as operation timestamps, and 64-bit integers to identify tree nodes. When a replica generates an operation, it immediately applies that operation to its local state, and it asynchronously sends the operation to the other two replicas via TCP connections. When a replica receives an operation from a remote replica, it also applies that operation to its own replica state. Thus, of the operations applied by each replica, approximately one third are locally generated, and two thirds are received from the other two replicas. A replica does not wait for a response for the previous operation before generating and sending the next, so there may be many operations “in flight” at the same time.

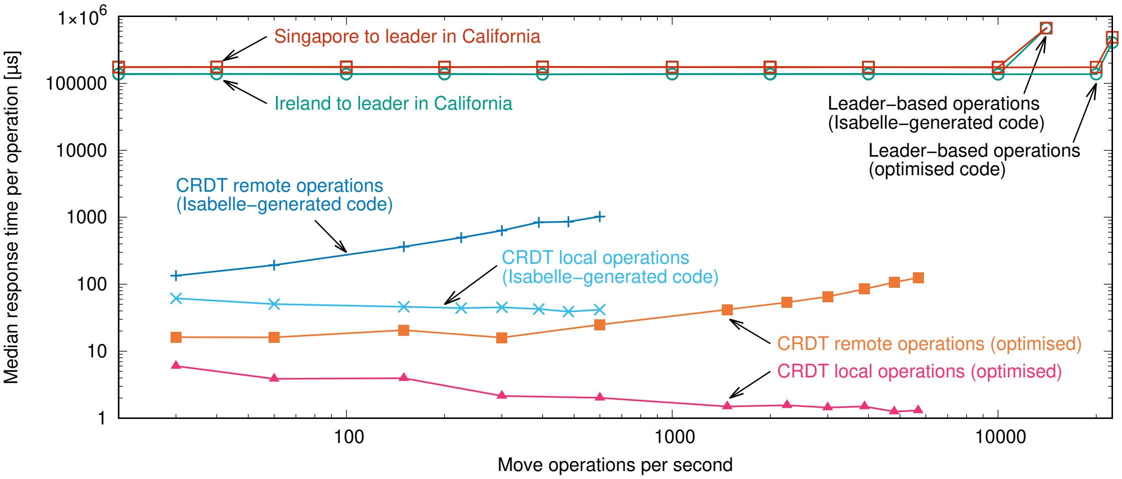


Fig. 7. Median time to apply a local CRDT operation compared to the median time to perform a move operation using state machine replication, with a leader located in another region. Note the log scale on both axes.

For a given operation rate we ran the system for 10 minutes to reach a steady state. We repeated the experiment with different operation rates, and measured the execution time of our algorithm for applying each operation. Figures 5 and 6 show the results for the Isabelle-generated code and the handwritten code respectively. In each figure, the upper plot shows the distribution of execution times to apply one operation received from another replica, and the lower plot shows the time to apply one locally generated operation.

Both implementations exhibit qualitatively similar behaviour, but the absolute numbers differ significantly. A local operation takes near-constant time to apply because its timestamp is always greater than any existing operation at the generating replica (by definition of Lamport clocks), so it does not require any undo or redo. The median time to apply a local operation is around 50 µs for the Isabelle-generated code and around 1–2 µs for the handwritten code. The reduced performance at low operation rates can be explained by the JVM delaying JIT optimisations until a method has been executed a certain number of times.

For remote operations, the execution time increases proportionally to the rate at which operations are generated: as the time interval between successive operations decreases relative to the network delay, more operations are in flight at the same time, and more undos/redos are required to put operations in timestamp order. At some point, each single-threaded replica is fully utilising one CPU core, and increasing the rate at which operations are generated does not increase throughput any further. The Isabelle-generated code is saturated at a rate of 600 operations/sec (median remote operations taking ≈1 ms), while the optimised code is saturated at 5,700 operations/sec. This factor of 9.5 performance difference is solely due to inefficient code generation; functionally both implementations perform the same work.

At peak, the optimised implementation is performing on average 200 undos and redos per remote operation. The throughput could be increased by applying a batch of operations at once, which would allow the cost of the undos and redos to be amortised over the size of the batch. We leave an evaluation of this optimisation for future work.

5.2 Comparison to locking/state machine replication

An alternative to our CRDT algorithm is to use a locking protocol (§ 2.3). For example, a replica can acquire a lock from a lock server, perform an update, and then release the lock again. The minimum time for which a lock will be held after it is acquired is then the round-trip time. If the lock server is in California and the replica is in Singapore, and if one move operation is performed per lock acquisition, this system could only perform $1/176\,\mathrm{ms} = 5.7$ operations/sec. The throughput of this locking-based scheme is therefore three orders of magnitude lower than our optimised algorithm's throughput of 5,700 operations/sec.

A better alternative to our CRDT algorithm is to use state machine replication [20]: that is, we use a leader replica or consensus algorithm to impose a total order on all operations, and then execute operations in that same order on all replicas. For a move operation on trees, the state machine replication algorithm is much simpler than the CRDT: it still needs to check for cycles, but it never needs to undo or redo any operations because they are never applied out-of-order.

To compare the performance of our algorithm to the state machine approach we ran another set of experiments on the same three replicas in California, Ireland, and Singapore. In this experiment, the Californian replica was designated leader; it totally ordered all operations it received, and sent them to all other replicas in the same order. The Irish and Singaporean replicas generated operations, sent them to the leader, and applied them to their local tree in the order they were received from the leader. In order to ensure a fair comparison to the CRDT algorithm, we ran these experiments using the same two implementations (Isabelle-generated and handwritten) and the same networking code.

The results are shown in Figure 7. Using the Isabelle-generated code, the leader-based approach is able to sustain 14,000 move operations per second (23 times the CRDT's throughput of 600 ops/sec). Using the handwritten code, the leader-based throughput is 22,000 ops/sec (4 times the CRDT's throughput of 5,700 ops/sec). The downside of state machine replication is that performing an operation requires waiting for a round-trip to the leader, which implies around five orders of magnitude higher latency (145–176 ms) than the 1–2 µs it takes to execute local CRDT operations.

Therefore, we have a clear trade-off: in applications that need to maximise throughput, a state machine replication approach is preferable; in applications that need to minimise response times to user requests or that need to continue to be available during network interruptions, our CRDT algorithm is preferable. In the leader-based approach, clients cannot make updates while offline.

5.3 Evaluation of formal proof

The formalisation of our algorithm, and the proofs of its properties as described in §4, have been formally checked by Isabelle/HOL. Our proofs contain no unproven assumptions (i.e. no occurrences of the **sorry** keyword). Checking all of the proofs takes 3 minutes on a 2018 MacBook Pro.

Besides the 59 lines of definitions given in Figure 4, our Isabelle/HOL formalisation consists of a further 2,495 lines of proof code. Of this, we use 203 lines to prove that every node has a unique parent, 443 lines to prove that the tree contains no cycles, 450 lines to prove that move operations commute and replicas converge, 327 lines to prove the strong eventual consistency property, 743 lines to define the executable variant of our algorithm and prove its equivalence to the definitions of Figure 4, and 779 lines to prove the safety of the optimisation in §3.6.

6 Related work

Many replicated data systems use optimistic replication [2], which allows the state of replicas to temporarily diverge, in order to achieve better performance and availability in the presence of faults than strongly consistent systems [1], [21]. As a consequence, these systems require a mechanism for merging or reconciling conflicting updates that were made concurrently on different replicas. For example, version control systems such as Git [22] leave conflicts for the user to resolve manually. Databases such as Dynamo [23] and Bayou [24] rely on the application programmer to provide explicit conflict resolution logic; however, such logic is difficult to get right [4], [25], [26]. Hence, we want to automatically ensure that all replicas converge towards a consistent state, without requiring custom application logic—a consistency model known as *strong eventual consistency* [3], [4].

6.1 Conflict-free Replicated Data Types

Our algorithm is an example of an operation-based Conflict-free Replicated Data Type or CRDT [3], [27]. All CRDTs share the property that concurrent changes on different replicas can be merged in any order; any two replicas that have seen the same set of updates are guaranteed to be in the same state, regardless of the order in which they processed these updates. Several CRDTs for trees have been proposed:

- Martin et al. [28], [29] define a CRDT for XML data, and Kleppmann and Beresford [30] define a CRDT for JSON. However, these algorithms only deal with insertion and deletion of tree nodes, and do not support moves. A move operation can be emulated by deleting and re-inserting the moved node, but this approach suffers from the duplication problem demonstrated in §2.1.
- As discussed in §2.3, Najafzadeh et al. [5], [6] propose two implementations for a replicated filesystem: a CRDT in which conflicting moves are handled by duplicating tree nodes (as in Figures 1b and 2b), and a centralised implementation in which move operations must obtain a lock before executing (not a CRDT since it relies on synchronous coordination). We discuss the performance costs of locking in §5.2.
- Ahmed-Nacer et al. [31] outline approaches to handling conflicts on trees, but provide no algorithms.
- Tao et al. [32] propose handling conflicting move operations by allowing the same object to appear in more than one location; thus, their datatype is strictly a DAG, not a tree. Some conflicts are handled by duplicating tree nodes. Tao et al. also perform experiments with Dropbox, Google Drive, and OneDrive, similar to our experiments discussed in §2.
- Nair et al. [33] develop a CRDT tree with move operation. This work is concurrent to ours and it is unpublished at the time of writing, so we have not been able to conduct a detailed comparison.

Several other CRDTs, such as Treedoc [34] and LSEQ [35], use a tree structure internally. However, their data model is a linear sequence; the tree structure is not accessible to the application (for example, an application cannot freely choose the parent node of a new tree node), and they do not provide a move operation.

Besides CRDTs, another family of algorithms for concurrent modification of data structures is *Operational Transformation* (OT) [36]. Several authors have defined concurrent tree structures using OT [37], [38], [39], but they only handle insertion and deletion of nodes, and do not support moves.

Molli et al. [40] define an OT tree structure with a move operation. However, it requires that all communication between replicas is performed via total order broadcast, which requires a leader replica or consensus algorithm, like in §5.2. Our algorithm has better availability characteristics in the presence of network partitions because it allows messages to be delivered in any order, e.g. via peer-to-peer protocols.

Collaborative graphics software Figma uses an approach inspired by CRDTs, but prevents cycles in their object tree by relying on a central server; its replication protocol allows objects to temporarily disappear while syncing [41].

6.2 Distributed filesystems

Many distributed filesystems, such as NFS, rely on synchronous interaction with a server. This avoids the need for conflict resolution, but rules out users working offline.

Coda is a client-server filesystem that allows clients to locally cache copies of files stored in a server-side data repository [42]. Clients can edit data in the cache while offline, during which time a kernel module keeps track of all updates. When the client comes back online it attempts to resynchronise changes with the server. To resolve conflicts due to concurrent updates, Coda uses application-specific resolvers [43], similarly to Bayou's approach [24]. Concurrent renaming and move operations have been considered, but the authors note that they do "not address transparent resolution of cross-directory renames [i.e. move operations] in [their] current implementation" [44]. Furthermore, while the authors consider a number of conflicts associated with directory move operations, they do not highlight the potential for the creation of cycles.

Ficus [45] is an in-kernel SunOS-based replicated peer-to-peer filesystem. Ficus supports updates to replicas during periods of network partition and claims "conflicting updates to directories are detected and automatically repaired" [46]. Unfortunately we were unable to find a precise definition of the algorithm used in any of the available publications.

Rumor [47], [48] is the successor to Ficus. While previous work uses the kernel filesystem interface, Rumor is a userspace process that is invoked periodically by the user or by a daemon; when run, it compares the state of the replicas. The original version of Rumor was unable to scale beyond 20 replicas, but an extension called Roam [49] allowed better scaling. In an attempt to test Rumor's conflict handling we obtained the source code from *archive.org* [50]; however, we were unable to get it running after modest effort.

Unison is a file synchronisation tool with a formal specification that allows two replicas to synchronise the state of a directory [51]. It permits offline updates to both replicas. Like Rumor, Unison is a userspace process that compares replica states. Whenever it is run, Unison records a summary of the filesystem state on each replica, and it uses this summary to determine the changes made since the last synchronisation. When presented with the move operations described in Figure 1, Unison duplicates the files, resulting in the outcome shown in Figure 1a. Unison is unable to automatically synchronise the move operations shown in Figure 2 and instead asks the user to choose one of four possible resolutions: those shown in Figure 2b, 2c, 2d, or to delete both directories.

Hughes et al. [52] test Dropbox and Google Drive against a formal specification, but they do not consider moving files, and thus they do not find the issue described in §2.2.

Bjørner [53] discusses the development of the Distributed File System Replication (DFS-R) component of Windows Server, during which a model checker found an issue with concurrent moves similar to Figure 2a. Bjørner outlines several possible solutions, but notes that model-checking their algorithm was not feasible due to state space explosion. Our use of proof by induction, rather than model-checking, allows us to verify the correctness of our algorithm in unbounded executions.

After we performed the experiments described in §2, the Dropbox engineering team published a blog post [54] acknowledging the problems of cycles and duplication due to concurrent moves.

6.3 Totally ordered operation log

Our approach of ordering operations by a timestamp, and undoing/redoing them as necessary so that they take effect in ascending timestamp order, is conceptually very simple. Similar ideas appear in many other systems, including the Bayou database [24], Jefferson's *Time Warp* mechanism [55], and Burckhardt's *standard conflict resolution* [26, §4.3.3]. The concept of undo-redo is also well known from write-ahead logging [56].

The SECRO approach [57] allows arbitrary CRDTs to be defined by having each replica execute deterministic update functions in the same total order, and replaying operation history if necessary. In principle, our algorithm could be expressed as a SECRO, but we provide optimisations that go beyond the SECRO model. SECRO always processes operations in forward direction, whereas our use of undo/redo is more efficient on long operation histories if replicas have most of the operation log in common. (SECRO allows long operation histories to be truncated, but this mechanism may result in discarded operations if replicas are disconnected for extended periods of time.) Moreover, our optimisation of node creation operations (§3.6) is specific to our algorithm, and is not possible in the general SECRO model.

As an example of the SECRO approach, De Porre et al. present an AVL tree CRDT [57]. Even though this data structure is internally a tree, the interface it exposes is an ordered set datatype, not a tree in which the user can choose to create, delete, and move arbitrary nodes.

To our knowledge, the approach of ordering operations by timestamp has not previously been applied to the problem of replicated trees, and in particular not a move operation. We are not aware of previous mechanised proofs (using Isabelle or other tools) that formalise this approach.

7 Conclusions

We have defined a novel algorithm that handles arbitrary concurrent modifications of a tree data structure—adding, moving, and removing nodes—in a peer-to-peer replication setting. It is applicable to distributed filesystems, databases, and applications that use a tree-structured data model. Our approach ensures that all replicas converge to the same consistent state without needing any manual conflict resolution, and without requiring application developers to implement conflict handling logic. Updates made to a local replica take effect immediately, while operations from remote replicas can be propagated and applied asynchronously. This approach means that user interaction is consistently fast, even in the face of unbounded communication delays or during disconnected operation of mobile devices.

The principle behind our algorithm is easy to understand: undoing and redoing operations so that they are effectively executed in timestamp order. Nevertheless, it solves a real problem that has not been solved correctly in widely deployed software such as Google Drive and Dropbox, as we demonstrated in §2. To rule out such bugs, we formally verified the correctness of our algorithm using the Isabelle/HOL proof assistant; our theorems show that replicas can apply operations in any order, and that the result is always a valid tree (nodes have at most one parent, and the graph does not contain any cycles). Moreover, these results apply to unbounded executions and an arbitrary number of replicas—an advantage of using a proof assistant over other formal approaches such as model checking.

One might wonder whether our algorithm's consistency model is strong enough for practical use. On this point, we note that this model is what Google Drive and Dropbox use today [52] (apart from the aforementioned bugs).

We also evaluated the performance of two implementations of our algorithm (one formally verified, the other optimised for performance) across three replicas in California, Ireland and Singapore. Our optimised implementation applies local updates in 1–2 µs, five orders of magnitude faster than is possible with a leader replica or consensus protocol operating over these distances, and our algorithm remains available in the face of network interruptions. However, compared to leader-based replication our algorithm has four times lower throughput.

Our work is especially interesting due to the ubiquity of tree data models across many different types of applications and databases. In future work we hope to integrate our algorithm into CRDT libraries such as Automerge,[2] and use it to build novel collaborative applications.

We are also exploring whether the undo-do-redo approach can be used in other concurrent data structures; for example, there is an open problem in collaborative text editing [16] that might be solved by this approach. Our Isabelle/HOL formalisation, which is open source [17], can be used for future work in this area.

Acknowledgements

The authors wish to acknowledge the support of The Boeing Company and the EPSRC "REMS: Rigorous Engineering for Mainstream Systems" programme grant (EP/K008528). Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, RelationalAI, SoftwareMill, Talent Formation Network, and Adam Wiggins. Our evaluation was conducted using AWS credits from the AWS Educate program. Thank you to Marc Shapiro for feedback on a draft of this paper.

References

[1] S. Gilbert and N. A. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[2] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys*, vol. 37, no. 1, pp. 42–81, Mar. 2005.

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Oct. 2011, pp. 386–400.

[4] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. OOPSLA, Oct. 2017.

[5] M. Najafzadeh, "The analysis and co-design of weakly-consistent applications," Ph.D. dissertation, Université Pierre et Marie Curie, Aug. 2016. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01351187v1

[6] M. Najafzadeh, M. Shapiro, and P. Eugster, "Co-design and verification of an available file system," in *19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Jan. 2018, pp. 358–381.

[7] M. Wenzel, L. C. Paulson, and T. Nipkow, "The Isabelle framework," in *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Aug. 2008, pp. 33–38.

[8] P. R. Johnson and R. H. Thomas. (1975, Jan.) RFC 677: The maintenance of duplicate databases. [Online]. Available: https://tools.ietf.org/html/rfc677

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[10] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'Cause I'm strong enough: reasoning about consistency choices in distributed systems," in *43rd ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2016, pp. 371–384.

[11] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro, "The CISE tool: Proving weakly-consistent applications correct," in *2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*, Apr. 2016.

[12] T. Nipkow and G. Klein, *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. [Online]. Available: http://concrete-semantics.org/

[13] C. Baquero, P. S. Almeida, and A. Shoker, "Making operation-based CRDTs operation-based," in *14th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, Jun. 2014, pp. 126–140.

[14] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011.

[15] S. Weiss, P. Urso, and P. Molli, "Logoot-Undo: Distributed collaborative editing system on P2P networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1162–1174, Jan. 2010.

[16] M. Kleppmann, "Moving elements in list CRDTs," in *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*, Apr. 2020.

[17] M. Kleppmann, D. P. Mulligan, V. B. F. Gomes, and A. R. Beresford, "Source code accompanying 'A highly-available move operation for replicated trees'," 2021. [Online]. Available: https://github.com/trvedata/move-op

[18] F. Haftmann and T. Nipkow, "Code generation via higher-order rewrite systems," in *10th International Symposium on Functional and Logic Programming (FLOPS)*, 2010, pp. 103–117.

[19] P. Lammich and A. Lochbihler, "The Isabelle Collections Framework," in *1st International Conference on Interactive Theorem Proving (ITP)*, 2010, pp. 339–354.

[20] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[21] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 185–196, Nov. 2014.

[22] S. Chacon and B. Straub, *Pro Git*, 2nd ed. Berkeley, CA: Apress, 2014. [Online]. Available: https://git-scm.com/book/en/v2

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007, pp. 205–220.

[24] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995, pp. 172–182.

[25] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *ACM Queue*, vol. 11, no. 3, Mar. 2013.

[26] S. Burckhardt, "Principles of eventual consistency," *Foundations and Trends in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, Oct. 2014.

[27] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: Specification, verification, optimality," in *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 2014, pp. 271–284.

[28] S. Martin, P. Urso, and S. Weiss, "Scalable XML collaborative editing with undo," in *On the Move to Meaningful Internet Systems*, Oct. 2010, pp. 507–514.

[29] S. Martin, M. Ahmed-Nacer, and P. Urso, "Abstract unordered and ordered trees CRDT," INRIA, Research Report RR-7825, Dec. 2011. [Online]. Available: https://hal.inria.fr/hal-00648106

[30] M. Kleppmann and A. R. Beresford, "A conflict-free replicated JSON datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, Apr. 2017.

[31] M. Ahmed-Nacer, S. Martin, and P. Urso, "File system on CRDT," INRIA, Tech. Rep. RR-8027, Jul. 2012. [Online]. Available: https://hal.inria.fr/hal-00720681/

[2] https://github.com/automerge/automerge

[32] V. Tao, M. Shapiro, and V. Rancurel, "Merging semantics for conflict updates in geo-distributed file systems," in *8th ACM International Systems and Storage Conference (SYSTOR)*, May 2015.

[33] S. S. Nair, F. Meirim, M. Pereira, C. Ferreira, and M. Shapiro, "A coordination-free, convergent, and safe replicated tree," 2021, unpublished draft.

[34] N. Preguiça, J. Manuel Marquès, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2009.

[35] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils, "LSEQ: an adaptive structure for sequences in distributed collaborative editing," in *13th ACM Symposium on Document Engineering (DocEng)*, Sep. 2013, pp. 37–46.

[36] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *ACM Conference on Computer Supported Cooperative Work (CSCW)*, Nov. 1998, pp. 59–68.

[37] T. Jungnickel and T. Herb, "Simultaneous editing of JSON objects via operational transformation," in *31st Annual ACM Symposium on Applied Computing (SAC)*, Apr. 2016, pp. 812–815.

[38] C.-L. Ignat and M. C. Norrie, "Customizable collaborative editor relying on treeOPT algorithm," in *8th European Conference on Computer-Supported Cooperative Work (ECSCW)*, Sep. 2003, pp. 315–334.

[39] A. H. Davis, C. Sun, and J. Lu, "Generalizing operational transformation to the Standard General Markup Language," in *ACM Conference on Computer Supported Cooperative Work (CSCW)*, Nov. 2002, pp. 58–67.

[40] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine, "Using the transformational approach to build a safe and generic data synchronizer," in *2003 International ACM SIGGROUP Conference on Supporting Group Work*, Nov. 2003, pp. 212–220.

[41] E. Wallace. (2019, Oct.) How Figma's multiplayer technology works. Archived at https://perma.cc/79TM-6FEE. [Online]. Available: https://www.figma.com/blog/how-figmas-multiplayer-technology-works/

[42] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992.

[43] P. Kumar and M. Satyanarayanan, "Flexible and safe resolution of file conflicts," in *USENIX Winter Technical Conference*, Jan. 1995.

[44] ——, "Log-based directory resolution in the coda file system," in *2nd International Conference on Parallel and Distributed Information Systems (PDIS)*, 1993, pp. 202–213.

[45] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. J. Popek, "Resolving file conflicts in the Ficus file system," in *Summer USENIX Conference*, Jun. 1994, pp. 183–195.

[46] R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. Page Jr, G. J. Popek, D. Rothmeier *et al.*, "Implementation of the Ficus replicated file system," in *Summer USENIX Conference*, 1990, pp. 63–72.

[47] R. Guy, P. Reiher, D. Ratner, M. Gunter, W. Ma, and G. Popek, "Rumor: Mobile data access through optimistic peer-to-peer replication," in *Advances in Database Technologies, LNCS 1552*. Springer Berlin Heidelberg, Nov. 1999, pp. 254–265.

[48] P. Reiher, "Rumor 1.0 user's manual," 1998. [Online]. Available: https://web.archive.org/web/20170705140958/ftp://ftp.cs.ucla.edu/pub/rumor/rumor_users_manual.ps

[49] D. Ratner, P. Reiher, and G. J. Popek, "Roam: a scalable replication system for mobile computing," in *10th International Workshop on Database and Expert Systems Applications (DEXA)*. IEEE, Sep. 1999, pp. 96–104.

[50] Rumor development team at UCLA, "Rumor 1.0.2 source code," 1998. [Online]. Available: https://web.archive.org/web/20170705140950/ftp://ftp.cs.ucla.edu/pub/rumor/rumor.src.release-1.0.2.tar.gz

[51] B. C. Pierce and J. Vouillon, "What's in Unison? A formal specification and reference implementation of a file synchronizer," Dept. of Computer and Information Science, University of Pennsylvania, Tech. Rep. MS-CIS-03-36, 2004. [Online]. Available: http://www.cis.upenn.edu/~bcpierce/papers/unisonspec.pdf

[52] J. Hughes, B. C. Pierce, T. Arts, and U. Norell, "Mysteries of Dropbox: Property-based testing of a distributed synchronization service," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2016, pp. 135–145.

[53] N. Bjørner, "Models and software model checking of a distributed file replication system," in *Formal Methods and Hybrid Real-Time Systems, LNCS 4700*. Springer Berlin Heidelberg, 2007, pp. 1–23.

[54] S. Jayakar, "Rewriting the heart of our sync engine," Mar. 2020, archived at https://perma.cc/HU2D-H9X4. [Online]. Available: https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

[55] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[56] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[57] K. De Porre, F. Myter, C. De Troyer, C. Scholliers, W. De Meuter, and E. Gonzalez Boix, "Putting order in strong eventual consistency," in *IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Springer, Jun. 2019, pp. 36–56.

Martin Kleppmann is a Senior Research Associate at the University of Cambridge. His research spans distributed systems, databases, security, and formal verification, with a focus on decentralised collaboration software and CRDTs. His book *Designing Data-Intensive Applications* was published in 2017 and has been translated into six languages. Previously, he was a software engineer and entrepreneur at Internet companies including Rapportive and LinkedIn.

Dominic P. Mulligan is a Principal Research Engineer in the Systems Group at Arm Research, UK. Prior to moving to Arm, he worked as a postdoctoral researcher at the Universities of Cambridge and Bologna, Italy, investigating the formal specification and verification of systems software such as C compilers and linkers. His interests include formal verification, distributed systems, and privacy and security.

Victor B. F. Gomes received a diplôme d'ingénieur from INSA Lyon and a PhD degree from the University of Sheffield. He was a research associate at the University of Cambridge from 2016 to 2019 and he is currently working at Google. His main research interests are semantics of programming languages, formal verification via theorem prover assistants and algebraic approaches for program verification.

Alastair R. Beresford is Professor of Computer Security in the Department of Computer Science at the University of Cambridge. His research work explores the security and privacy of large-scale distributed systems, with a particular focus on networked mobile devices such as smartphones, tablets and laptops. He looks at the security and privacy of the devices themselves, as well as the security and privacy problems induced by the interaction between mobile devices and cloud-based Internet services.