A highly-available move operation for replicated trees

Martin Kleppmann
mk428@cl.cam.ac.uk
Department of Computer Science and Technology
Cambridge, United Kingdom

Dominic P. Mulligan
Dominic.Mulligan@arm.com
Arm Research
Cambridge, United Kingdom

Victor B. F. Gomes
vb358@cl.cam.ac.uk
Department of Computer Science and Technology
Cambridge, United Kingdom

Alastair R. Beresford
arb33@cl.cam.ac.uk
Department of Computer Science and Technology
Cambridge, United Kingdom

ABSTRACT

Replicated tree data structures are found in distributed filesystems, such as Google Drive and Dropbox, and 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. Such a move operation is easy to implement in a centralised system, e.g. with a designated primary replica, but it is difficult in settings where different replicas can concurrently perform arbitrary move operations. For example, we find that Google Drive and Dropbox fail to correctly synchronise files when users on different devices concurrently perform certain move operations. In this paper we present an 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 eventually converge towards the same consistent state. We formally prove the correctness of our algorithm using the Isabelle/HOL interactive proof assistant, and we evaluate the performance of our formally verified implementation in a geo-replicated setting to demonstrate its viability in practice.

ACM Reference Format:
Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. 2019. A highly-available move operation for replicated trees. In . ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Many systems use a tree-structured data model. For example:

- The **filesystem** on most operating systems is a tree: directories are branch nodes, and files are leaf nodes. In Unix systems the nodes of the tree are known as *inodes*. Strictly speaking, filesystems with hardlinks form a DAG; we will focus on trees now and show in §3.6.1 how to support links.
- **Rich text editors** maintain a tree of textual structures such as paragraphs, lists, figures, sections, and so on. These can be nested: for example, a paragraph may appear inside a bulleted list item, which may in turn appear inside a numbered point of an enumeration. Rich text documents are typically represented in memory using the Document Object Model (DOM), and written to file in HTML, XML, or JSON format.
- **Vector graphics** and **presentation software** represents images using graphical objects such as text boxes, rectangles, lines, and so on. These objects are contained within nodes representing pages of a document, or slides of a presentation. Objects may be combined into a *group* and manipulated as a unit; this corresponds to making these objects children of a common parent node. Multiple objects and groups may be combined further into higher-level groups, forming a tree.
- **Note-taking** and **task-management tools** such as Org-mode for Emacs [15] or OmniOutliner [43] present the user with a tree structure that they can inspect and manipulate.

In this paper we consider applications that use such a tree data model, and that replicate this tree across multiple nodes. Moreover, we focus on *optimistic replication* [51], that is, systems in which any replica can autonomously make changes to the data, without waiting for communication or coordination with any other replicas. Such systems have the advantage that they can continue processing read and write requests even in the presence of arbitrary network partitions; in other words, they are *available* and *partition-tolerant* in the sense of the CAP theorem [16]. This approach is desirable since it enables disconnected operation in a mobile computing context, and high availability in geo-replicated settings.

As the tree structure may be concurrently modified on different replicas, the state of these replicas may temporarily diverge. In this paper we show how the replicas can nevertheless achieve *strong eventual consistency* [17, 53]: as the replicas communicate, we guarantee that they will converge towards a consistent state. Our algorithm for achieving convergence is an example of a *Conflict-free Replicated Data Type* or CRDT [52, 53].

In this paper we use an abstract model of a tree that can support any of the applications listed above, including distributed filesystems. We allow replicas to manipulate this tree in any way: by creating new nodes, deleting nodes, or moving subtrees to a new location within the tree. While there are many existing systems that support creating and deleting nodes (see §6), the key innovation of our algorithm is the support for moving subtrees. We explain in §2 why this move operation is so challenging. In summary, our contributions are:

- We define a Conflict-free Replicated Data Type for trees that supports a *move* operation, without requiring any coordination between replicas such as locking or consensus. As

, ,

discussed in §2.3, this has previously been thought to be impossible to achieve [38, 40].
- We formalise an abstract algorithm using Isabelle/HOL [58], an interactive proof assistant based on higher-order logic, and obtain a computer-checked proof of correctness of the algorithm. In particular, we prove that arbitrary concurrent modifications to the tree can be merged such that all replicas converge towards a consistent state (strong eventual consistency) while keeping the data in a tree structure.
- To demonstrate the practical viability of our approach, we refine our abstract algorithm to an executable implementation within Isabelle/HOL and prove the equivalence of the two. We then extract a formally verified Scala implementation from the Isabelle/HOL definitions and evaluate its performance with replicas across three continents.
- We perform experiments with replicated filesystem products such as 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, whereby all children of the moved node move along with it. For example:

- 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 DOM tree this corresponds to creating a new bullet list node, creating a new bullet point node within it, and then moving the paragraph node inside the bullet point.
- In presentation software, grouping two graphical objects corresponds to creating a new group node, and then moving the two objects to be children of 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 by example some of the 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 a 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 presentation software examples, such duplication is also undesirable.

Rather than duplicating the moved subtree, 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-compliant filesystems do not allow this outcome, since they allow hardlinks only to files, but not 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. For example, 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 [23]. All replicas pick the same operation as winner, and effectively ignore the conflicting operation. (The *timestamp* in this context need not come from a physical clock; it could also be logical, such as a Lamport timestamp [34].)

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. The same restriction is required in any other tree structure that supports a move operation.

In an unreplicated tree it is easy to prevent cycles being introduced: if the node being moved is an ancestor of the destination node, then the operation is rejected. However, a new problem arises in a replicated setting: different replicas may perform operations that are individually safe, but whose combination leads to a cycle.

One example of such a scenario 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, in which A and B have become locked in a cycle and detached from the tree.

Another possible resolution 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 an operation timestamp.

[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 each sync a directory on the local filesystem with the corresponding cloud service. To test concurrent operations, we turned off the network interface on both computers, performed a move operation on the local filesystem of each computer, then reconnected both computers to the Internet and waited for them to sync.

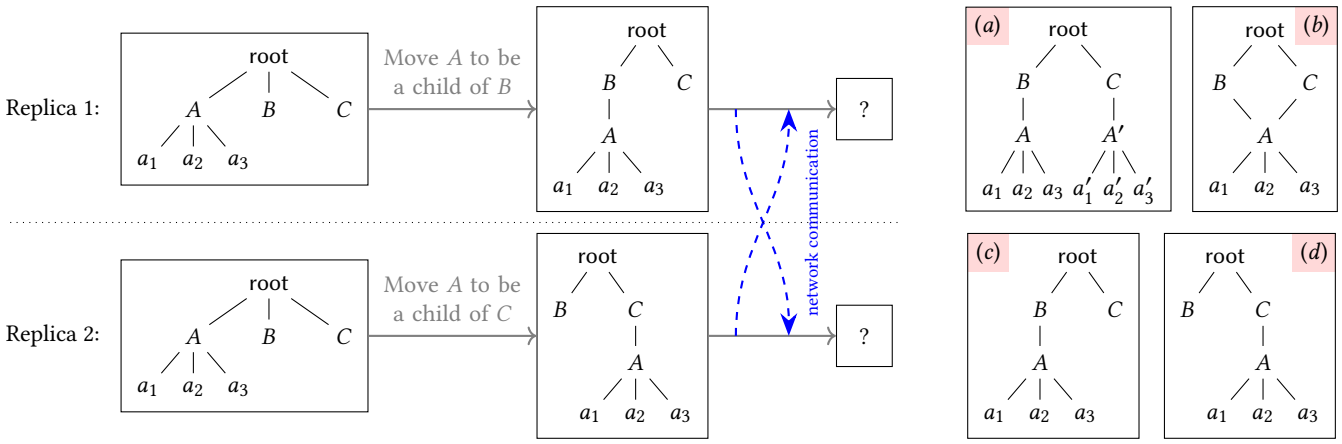

Figure 1: Replica 1 moves A to be a child of B, while concurrently replica 2 moves the same node A to be a child of C. Boxes (*a*) to (*d*) show possible outcomes after the replicas have communicated and merged their states.

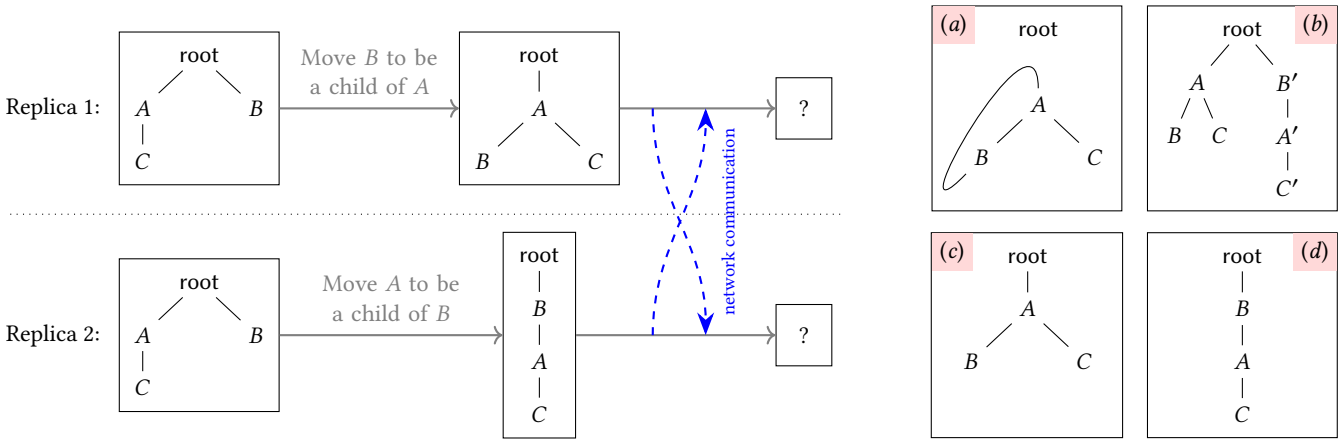

Figure 2: Initially, nodes A and B are siblings. Replica 1 moves B to be a child of A, while concurrently replica 2 moves A to be a child of B. Boxes (*a*) to (*d*) show possible outcomes after the replicas have communicated and merged their states.

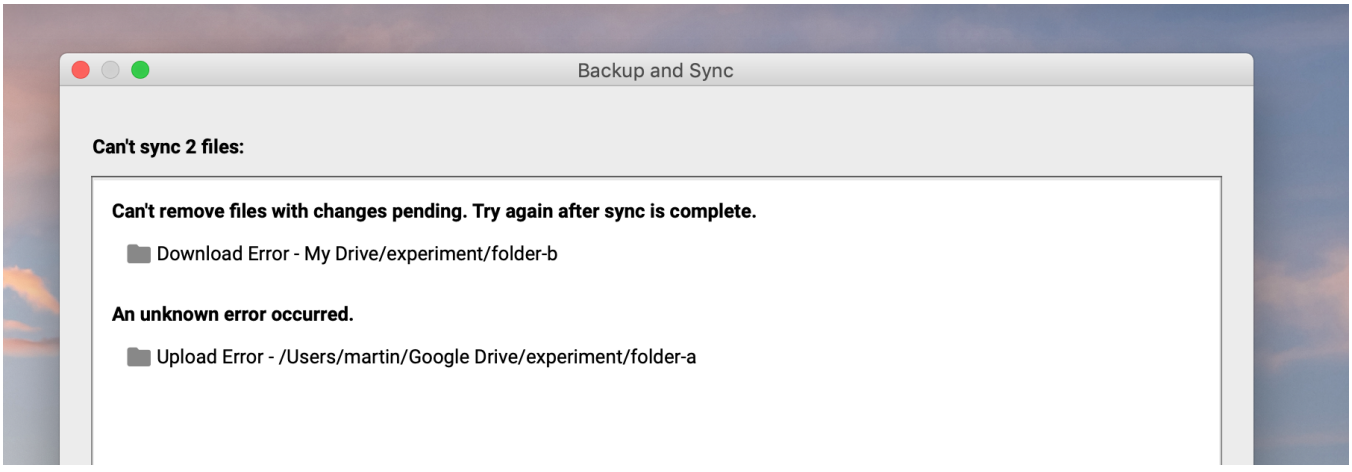

Figure 3: Error message produced by Google Drive Backup and Sync on Mac OS as a result of performing the operations shown in Figure 2.

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. The replica in an error state refused to sync the conflicting directory, and its filesystem state remained 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. On the other hand, Dropbox exhibited the duplication behaviour shown in Figure 2b.

2.3 The impossibility of a highly-available move operation

Najafzadeh et al. [38, 40] have previously implemented a replicated filesystem with a move operation, and analysed the case of concurrent move operations introducing a cycle, as in §2.2. Using the CISE ('Cause I'm Strong Enough) proof tool [18, 39] 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.

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" [40]. We revisit 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 MOVE OPERATION

We now introduce our algorithm for a replicated tree data structure 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 object and apply it on their local replica. Every operation is also asynchronously sent over the network to all other replicas, and applied on every remote replica using the same algorithm as for local operations. The network may deliver operations in any order; thus, the communication can be performed peer-to-peer, and it does not require any central server or consensus protocol. We only assume that eventually every operation is applied on every replica, provided that network partitions are of a finite duration.

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 is commutative.

Figure 4 gives the full source code for our algorithm, implemented using the Isabelle/HOL language [58]. 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 also allows executable code to be exported to common functional programming languages such as Scala, Haskell, or OCaml.

In this section we walk through the code in Figure 4 step by step, explaining features of the Isabelle/HOL syntax as we encounter them. Additional background documentation is available [42].

3.1 Operations and trees

We define only one kind of operation: `Move t p m c` (Figure 4, lines 1–5). A move operation is a record consisting of four fields: a timestamp `t` of type `'t`, a parent node `p` of type `'n`, a metadata field `m` of type `'m`, and a child node `c` of type `'n`. Here, `'t`, `'n` and `'m` are *type variables* that can be substituted for arbitrary types; our only requirement is that timestamps `'t` are globally unique and have an associated total order (e.g. Lamport timestamps [34] would suffice).

The meaning of an operation `Move t p m c` is that at time `t`, the child node `c` is moved to be a child of parent node `p`. The operation does not specify the old location of `c`; the algorithm simply "steals" `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`. For this reason we do not need a separate operation type to create tree nodes: simply performing a move with a fresh child node `c` is sufficient to create `c`. Nor do we need to define a remove operation: instead, we can designate a tree node as the "trash", and perform deletion of a node `c` by moving `c` to be a child of the trash node.

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 would be the inodes of a directory and a file within it, respectively, and the metadata would contain 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 unchanged inode of the existing parent directory, 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 to their local replica. The timestamp `t` of each operation must be globally unique; the other fields `p`, `m` and `c` can be chosen freely.

We can now represent the tree as a set of `(parent, meta, child)` triples, denoted in Isabelle as `('n × 'm × 'n) set`. When we have `(p, m, c) ∈ tree`, that means `c` is a child of `p` in the tree, with associated metadata `m`. Given a `tree`, we can move the child `c` to a new parent `p`, with associated metadata `m`, as follows:

```
{(p', m', c') ∈ tree. c' ≠ c} ∪ {(p, m, c)}
```

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 31 and 36 of Figure 4, as we shall explain shortly.

```
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)
```

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

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. A `LogMove t oldp p m c` record (lines 7–12) is similar to a record `Move t p m c`; the only difference is that the `LogMove` has an additional field `oldp` of type `('n × 'm) option`. This `option` type means the field can either take the value `None` (intuitively similar to null), or a pair of an inode 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`, that is: if there exist `p'` and `m'` such that `(p', m', c) ∈ tree`, then `oldp` is set to `Some (p', m')`. The `get_parent` function implements this (lines 16–20).

In the first line of `get_parent`, the expression between `::` and **where** is the type signature of the function, in this case:

```
('n × 'm × 'n) set ⇒ 'n ⇒ ('n × 'm) option
```

This signature denotes a function that takes two arguments: a tree `('n × 'm × 'n) set` and a node `'n`. It then returns a `('n × 'm) option`. The operator ∃!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) ∈ tree` then `p` is an ancestor of `c` (line 23); moreover, if `a` is an ancestor of `p` and `(p, m, c) ∈ 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 26–31) 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 29, the `LogMove` record is constructed as described in §3.2, obtaining the prior parent and metadata of `c` using the `get_parent` function.

Line 30 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 31), 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 33–51).

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 43–51) 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 `linorder` type class, and they can therefore be compared with the < operator defining a total order. This comparison occurs on line 49.

Recall that the replica state includes the operation log (line 14), and we use this log to perform the undo-do-redo cycle. Lines 45–47 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 48), 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 *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 49) 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 50). Otherwise we perform the operation using `do_op`, and add the corresponding `LogMove` record as the head of the log (line 51).

This logic ensures that the log is maintained in descending timestamp order, with the greatest timestamp at the head. `undo_op` (lines 33–36) 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 38–41) 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 approach to the handling of conflicts:

- If two operations concurrently move the same node, then the "winning" operation is the one with the greater timestamp. Effectively, 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.
- If two operations would introduce a cycle when combined, as in §2.2, then the operation with the greater timestamp is ignored. This is the case 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.

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 have no effect on the tree.

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 Algorithm extensions

3.6.1 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).

3.6.2 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* [5].

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.

3.6.3 Ordering of sibling nodes. Another useful extension of the tree algorithm is to allow children within the same branch node to have an ordering. For example, in a graphics application, the order of nodes determines visibility (objects that are "further back" may be obscured by objects that are "closer to the front"). This can be implemented by maintaining a separate list CRDT for each branch node, e.g. using RGA [49], Treedoc [45], Logoot [57], or LSEQ [41]. These algorithms assign a unique identifier to each element of the list, and this unique identifier can be included in the metadata field of move operations in order to determine the order sibling nodes.

This approach to determining ordering also easily supports reordering of child nodes within a parent, e.g. to implement the "bring to front" and "send to back" feature of applications like PowerPoint. To move a node to a different position in a list, we use the list CRDT to generate a new identifier at the desired position in the sequence. Then we perform a move operation in which the parent node is unchanged, but the metadata is changed to this new identifier.

3.6.4 File merging. In a distributed filesystem, replication and conflict resolution is required not only for the directory structure, but also for the contents of individual files. This can be accomplished by using CRDTs for file contents as well. For example, we have explored CRDT algorithms for applications using a JSON data model [26, 30] and implemented them in a library called Automerge [29]. We discuss distributed filesystems further in §6.2.

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 we give only a brief sketch of the proofs in this paper, but the full details can be found in our open source Isabelle theory files [28].

To reason about the state of a replica we first define the function `apply_ops` on lines 53–54 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. To prove that our algorithm maintains a tree structure, no matter which operations are applied, we prove that these invariants hold. 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 multiple trees, if desired. For example, the "trash" node used for deletion (§3.1) can be separate from the main tree.

4.1.1 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 as follows, where `apply_ops_unique_parent` is the name we give to 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 56–57 of Figure 4: it states that 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. Proving it is easy because the do_op, undo_op and redo_op functions individually maintain the invariant unique_parent tree, and thus apply_op also maintains the invariant.

4.1.2 The graph contains no cycles. The second invariant we require is that the graph must not contain any cycles. In Isabelle:

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

The acyclic predicate is defined on lines 59–60 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.

Proof sketch: Assume do_op (op1, tree1) = (op2, tree2) for some op1, op2, tree1 and tree2. Further assume acyclic tree1. We can then prove by contradiction that tree2 is also acyclic. Assume to the contrary that tree2 contains a cycle. If do_op left the tree unchanged (line 30), we immediately have a contradiction. Otherwise we can assume ¬(ancestor tree c newp ∨ c = newp) and we have:

```
tree2 = {(p', m', c') ∈ tree1. c' ≠ c} ∪ {(newp, m, c)}
```

That is, do_op adds one edge {(newp, m, c)} to the graph, and perhaps removes another edge. Since tree1 is acyclic, the added edge must lie on the cycle in tree2. Moreover, since only one edge from newp to c was added, tree1 must contain a path from c to newp (which is then closed by the new edge to form a cycle), i.e. c must be an ancestor of newp in tree1. However, in that case, the predicate ancestor tree1 c newp on line 30 would have been true, which contradicts the earlier assumption. Hence, the do_op function preserves the acyclicity invariant of the tree.

Next, we show that for any tree produced by apply_ops, there exists a sequence of operations such that applying those operations using do_op invocations alone (not using undo_op or redo_op) results in the same tree. (This is the case when operations are applied in order of increasing timestamp.) Then, from the result of the previous paragraph, any tree produced by apply_ops is acyclic.

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 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). The assumption distinct (map move_time ops1) therefore states that in the list of operations 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.

4.2.1 Commutativity of apply_op. To prove this theorem we must first show that undo_op performs the inverse of do_op, that is:

```
lemma do_undo_op_inv:
  assumes unique_parent tree
  shows undo_op (do_op (oper, tree)) = tree
```

apply_ops_commutes then follows from the following lemma, which proves that two calls of apply_op commute, provided that the timestamps in the operations and in the log are all distinct:

```
lemma apply_op_commute2:
  assumes unique_parent tree
    and distinct ((map log_time log) @
                  (map move_time [oper1, oper2]))
  shows apply_op oper2 (apply_op oper1 (log, tree)) =
        apply_op oper1 (apply_op oper2 (log, tree))
```

Here the @ operator concatenates two lists. Let t1 = move_time oper1 and t2 = move_time oper2 be the timestamps of the two operations oper1 and oper2. Assume t1 < t2 without loss of generality. We can now prove lemma apply_op_commute2 by induction over log.

For the base case, where log is empty, apply_op oper2 (apply_op oper1 ([], tree)) evaluates to first applying oper1 using do_op, then applying oper2 using do_op. On the other hand, apply_op oper1 (apply_op oper2 ([], tree)) first applies oper2 using do_op, then undoes oper2 again to yield the original tree (per do_undo_op_inv), then applies oper1 using do_op, and finally reapplies oper2 using redo_op. The final states are the same on both sides of the equality.

For the inductive step, let the log be logop # ops (where # is the *cons* operator). We now need to show that

```
apply_op oper2 (apply_op oper1 (logop # ops, tree)) =
apply_op oper1 (apply_op oper2 (logop # ops, tree)).
```

Let t3 = log_time logop be the timestamp of the operation at the head of the log. We now consider three cases:

(1) t3 < t1 < t2. Then neither applying oper1 nor applying oper2 will undo logop. Thus, the two operations are applied in the same way as in the base case.
(2) t1 < t3 < t2. Then apply_op oper2 (apply_op oper1 (logop # ops, tree)) first undoes logop, then applies oper1 using do_op, then reapplies logop using redo_op, and finally applies oper2 using do_op. On the other hand, apply_op oper1 (apply_op oper2 (logop # ops, tree)) first applies oper2 using do_op, then undoes it again, then undoes logop, applies oper1 and reapplies logop as before, and finally reapplies oper2 using redo_op. The end result is the same.
(3) t1 < t2 < t3. Then both applying oper1 and applying oper2 first undoes logop, yielding tree' = undo_op (logop, tree). From the inductive hypothesis we have

    ```
    apply_op oper2 (apply_op oper1 (ops, tree')) =
    apply_op oper1 (apply_op oper2 (ops, tree'))
    ```

    We now apply oper1 and oper2 as in the base case, and finally reapply logop to both sides using redo_op, resulting in the same state on both sides.

4.2.2 Strong eventual consistency. Gomes et al. [17] define a framework in Isabelle/HOL for proving the strong eventual consistency properties of CRDTs. Using our commutativity proof above we integrate our tree datatype with this framework, and thus prove that our move operation on trees does indeed guarantee strong eventual consistency. The details appear in our Isabelle theory files [28].

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 [21]. 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 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, metadata, child) triples, these definitions use a hash-map in which the keys are child nodes, and the values are (metadata, parent) pairs, written in Isabelle 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 (§4.1.1). The hash-map implementation is from the Isabelle Collections Framework [33].

We say a hash-map `t` of type `('n, 'm × 'n) hm` *simulates* a set `T` of type `('n × 'm × 'n) set` when they have the same entries:

```
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 versions of each function. For example, `executable_do_op` is the equivalent of the `do_op` function based on a hash-map:

```
lemma executable_do_op_simulates:
  assumes simulates t T
    and executable_do_op (oper, t) = (log1, u)
    and do_op (oper, T) = (log2, U)
  shows log1 = log2 ∧ simulates u U
```

Similarly, we can show this simulation relation holds for each function in our algorithm, all the way to `apply_ops`:

```
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 then they produce identical logs, and also produce trees that contain the same set of key-value bindings—i.e. the two trees are extensionally equivalent, despite having very different in-memory representations. Moreover, we can prove corollaries of `apply_ops_commutes` and `apply_ops_acyclic`, respectively, for the hash-map-based implementation.

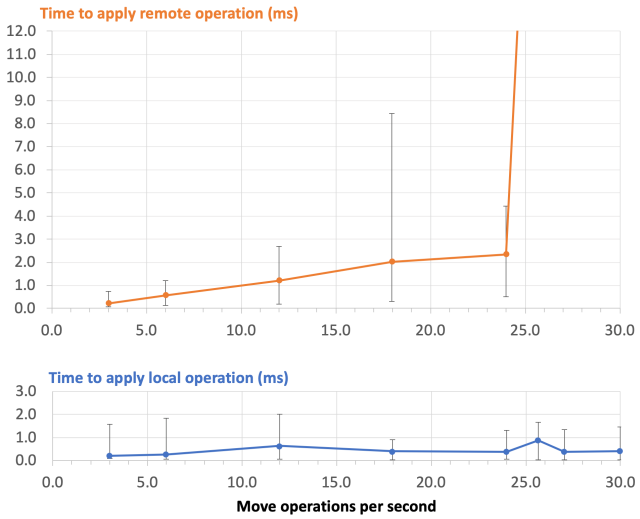

Figure 5: Median execution time for applying an operation to the replica state, on an Amazon EC2 t2.small instance, at varying operation throughput rates. Error bars indicate the minimum and 95th percentile execution times.

Table 1: Median response times of a RPC request between replicas in different regions.

	California	Ireland	Singapore
California		395 ms	491 ms
Ireland	395 ms		547 ms
Singapore	491 ms	547 ms	

5 EVALUATION

5.1 Experimental setup and results

We used Isabelle/HOL's code generation mechanism to extract executable Scala code from our HOL definitions, as discussed in §4.3, and wrapped this implementation in a network service using the Akka [35] actor framework, using gRPC [12] for network communication. We deployed three replicas of this service on Amazon EC2 `t2.small` instances in different regions worldwide: one in Northern California (region `us-west-1`), one in Ireland (`eu-west-1`), and one in Singapore (`ap-southeast-1`). The network delays for communication between these regions are substantial; see Table 1.

We use a synthetic workload in which each replica generates move operations at the same rate; for each move operation the generating replica chooses parent and child nodes uniformly at random from a set of 1,000 tree nodes. For a given operation rate we run the system for 10 minutes to reach a steady state. We use Lamport timestamps [34] as operation timestamps, and string values to identify tree nodes. When a replica generates an operation, it immediately applies that operation to its local state, and asynchronously sends the operation to the other two replicas via gRPC requests. 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.

We varied the rate at which replicas generated operations, and measured the execution time of our algorithm for applying each operation. Figure 5 shows the median execution times for local and remote operations respectively. The execution time for local operations is near-constant (median times mostly between 0.2 ms and 0.6 ms), while the execution time for remote operations increases approximately linearly from 0.2 ms to 2.3 ms over the range of 3 to 24 operations/sec. Above 25 operations/sec the execution time for remote operations suddenly increases to over 30 ms (off the scale of Figure 5). We do not plot the exact values for execution times above 12 ms because they are less meaningful, as explained below.

5.2 Interpretation of results

A locally generated operation always has a timestamp greater than any existing operation at the generating replica at that time (by definition of Lamport timestamps). Applying that operation requires only a `do_op`, and no `undo_op` or `redo_op`, hence the near-constant execution time. However, for remote operations, a number of undos and redos must be performed, depending on the degree of concurrency. As the rate of operations increases, the time interval between successive operations becomes smaller than the network delay between replicas; thus, there are more operations "in flight" at the same time, and more calls to `undo_op` and `redo_op` are required to order operations by timestamp.

The time taken to apply a remote operation is proportional to the number of undos and redos required. Above 25 operations/sec the CPU utilisation of our virtual machines grows to the point that incoming network requests from other replicas start queueing before they are processed. When a remote operation spends time queueing, more concurrent operations have been applied by the time it is processed, and thus more undos and redos are required to apply the incoming operation, which in turn increases the CPU utilisation and further increases queueing. This self-reinforcing effect explains the sudden dramatic increase in Figure 5.

Above 25 operations/sec the queueing delay increases so much that some RPC requests time out, and thus not all operations are received by all replicas. As the remote operation execution times above this threshold depend more on the network queueing properties than our algorithm, we discard these datapoints.

5.2.1 Practical relevance of results. A hand-optimised implementation of our algorithm could achieve higher throughput than our 24 operations/sec, since code generated from HOL definitions contains inefficiencies. The generated code could serve as a reference implementation against which other implementations could be tested.

Nevertheless, our evaluation sends a positive signal about the practical applicability of our algorithm. Note that when a user is interacting with a system, they only have to wait for local operations to execute; any remote operations can be applied in the background without affecting user interaction. Since our algorithm need not perform any undos or redos for local operations, these user interactions are consistently fast.

The alternative to our optimistic replication approach would be to use a central server (master replica) or consensus protocol that processes all operations in a serialisable order. Performing an operation then requires waiting for synchronous communication with a server potentially located on a different continent, taking hundreds of milliseconds (Table 1); by contrast, a local operation taking a fraction of a millisecond is orders of magnitude faster. Moreover, a local operation requires no waiting for network communication, and thus it supports disconnected operation: mobile devices can read and modify the replicated tree even while disconnected from the Internet, which is not the case with centralised systems.

6 RELATED WORK

Many replicated data systems use optimistic replication [51], which allows the state of replicas to temporarily diverge, in order to achieve better performance and better availability in the presence of faults than strongly consistent systems [2, 16]. 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 [10] leave any conflicts for the user to resolve manually. Some systems, such as Dynamo [14] and Bayou [56], rely on the application programmer to provide explicit conflict resolution logic; however, such logic is difficult to get right [3, 8, 17]. Hence, we focus on systems that automatically ensure that all replicas converge towards a consistent state, without requiring custom application logic—a consistency model known as *strong eventual consistency* [17, 53].

6.1 Conflict-free Replicated Data Types

Our algorithm is an example of an operation-based Conflict-free Replicated Data Type or CRDT [9, 52, 53]. 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. CRDTs have been defined for various abstract datatypes, such as registers and counters [52, 53], maps [4, 26], sets [6, 7], lists (sequences) [41, 49], and text [45, 57]. For tree data structures, several approaches have been proposed:

- Martin et al. [36] define a CRDT for XML data, and Kleppmann and Beresford [26] define a CRDT for JSON. However, these algorithms only deal with insertion and deletion of tree nodes, and do not support a move operation.
- As discussed in §2.3, Najafzadeh et al. [38, 40] 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).
- Ahmed-Nacer et al. [1] survey approaches to handling conflicts on trees without providing concrete algorithms.
- Tao et al. [55] 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.
- In our previous unpublished work [27] we develop the foundations for the algorithm described in this paper.

Besides CRDTs, another family of algorithms for concurrent modification of data structures is *Operational Transformation* (OT) [54]. Several authors have defined concurrent tree structures using OT [13, 22, 24], but they only consider insertion and deletion of nodes, and do not support a move operation.

Molli et al. [37] define an OT tree structure with a move operation. However, it requires that all communication between replicas is performed via total order broadcast, which in turn requires the use of a sequencer server or a consensus algorithm [11]. 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.

6.2 Distributed filesystems

Many distributed filesystems, such as NFS, rely on synchronous interaction with a server. While this has the advantage of not requiring conflict resolution, it rules out users working while offline.

Coda is a client-server filesystem that allows clients to locally cache copies of files stored in a server-side data repository [25]. 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 [32], similarly to Bayou's approach [56]. 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" [31]. 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 [48] 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" [20]. Unfortunately we were unable to find a precise definition of the algorithm used in any of the available publications.

Rumor [19, 47] 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 [46] 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 [44]. It permits offline updates to both replicas. Like Rumor, Unison is a userspace process that works by comparing 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.

7 CONCLUSIONS

In this paper 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 and many other applications that use a tree-structured data model. Our approach ensures that all replicas converge to the same consistent state without requiring any manual conflict resolution, and without needing 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 in the background. This approach means that user interaction is consistently fast, even in the face of unbounded communication delays between replicas, e.g. during disconnected operation of mobile devices.

While existing systems exhibit anomalies (such as duplicating nodes or introducing cycles) or error messages (like in Figure 3) in the face of concurrent move operations, our algorithm is free from such problems. 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).

We also extracted a formally verified Scala implementation from our Isabelle/HOL definitions, and evaluated its performance across three replicas in California, Ireland and Singapore. Our implementation applies local updates in under 1 ms, orders of magnitude faster than is possible with a central server or consensus protocol operating over these distances. It is able to sustain 24 updates per second; we plan to develop a more scalable implementation in future work.

ACKNOWLEDGMENTS

The authors wish to acknowledge the support of The Boeing Company and the EPSRC "REMS: Rigorous Engineering for Mainstream Systems" programme grant (EP/K008528).

REFERENCES

[1] Mehdi Ahmed-Nacer, Stéphane Martin, and Pascal Urso. 2012. *File system on CRDT*. Technical Report RR-8027. INRIA. https://hal.inria.fr/hal-00720681/

[2] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[3] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *ACM Queue* 11, 3 (March 2013). https://doi.org/10.1145/2460276.2462076

[4] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. 2016. The problem with embedded CRDT counters and a solution. In *2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*. https://doi.org/10.1145/2911151.2911159

[5] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-based CRDTs Operation-based. In *14th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. 126–140. https://doi.org/10.1007/978-3-662-43352-2_11

[6] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. Brief Announcement: Semantics of Eventually Consistent Replicated Sets. In *26th International Symposium on Distributed Computing (DISC)*. 441–442. https://doi.org/10.1007/978-3-642-33651-5_48

[7] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. *An Optimized Conflict-free Replicated Set*. Technical Report RR-8083. INRIA. http://arxiv.org/abs/1210.3368

[8] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages* 1, 1-2 (Oct. 2014), 1–150. https://doi.org/10.1561/2500000011

[9] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 271–284. https://doi.org/10.1145/2535838.2535848

[10] Scott Chacon and Ben Straub. 2014. *Pro Git* (second ed.). Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-0076-6

[11] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[12] Cloud Native Computing Foundation. [n.d.]. gRPC. https://www.grpc.io

[13] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. 2002. Generalizing Operational Transformation to the Standard General Markup Language. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 58–67. https://doi.org/10.1145/587078.587088

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *21st ACM Symposium on Operating Systems Principles (SOSP)*. 205–220. https://doi.org/10.1145/1294261.1294281

[15] Carsten Dominik, Bastien Guerry, et al. [n.d.]. Org-mode. https://orgmode.org

[16] Seth Gilbert and Nancy A Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News* 33, 2 (2002), 51–59. https://doi.org/10.1145/564585.564601

[17] Victor B F Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, OOPSLA (Oct. 2017). https://doi.org/10.1145/3133933

[18] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *43rd ACM Symposium on Principles of Programming Languages (POPL)*. 371–384. https://doi.org/10.1145/2837614.2837625

[19] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. 1999. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Advances in Database Technologies, LNCS 1552*. Springer Berlin Heidelberg, 254–265. https://doi.org/10.1007/978-3-540-49121-7_22

[20] Richard G Guy, John S Heidemann, Wai-Kei Mak, Thomas W Page Jr, Gerald J Popek, Dieter Rothmeier, et al. 1990. Implementation of the Ficus Replicated File System. In *Summer USENIX Conference*. 63–72.

[21] Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *10th International Symposium on Functional and Logic Programming (FLOPS)*. 103–117. https://doi.org/10.1007/978-3-642-12251-4_9

[22] Claudia-Lavinia Ignat and Moira C Norrie. 2003. Customizable Collaborative Editor Relying on treeOPT Algorithm. In *8th European Conference on Computer-Supported Cooperative Work (ECSCW)*. 315–334. https://doi.org/10.1007/978-94-010-0068-0_17

[23] Paul R Johnson and Robert H Thomas. 1975. RFC 677: The Maintenance of Duplicate Databases. https://tools.ietf.org/html/rfc677

[24] Tim Jungnickel and Tobias Herb. 2016. Simultaneous editing of JSON objects via operational transformation. In *31st Annual ACM Symposium on Applied Computing (SAC)*. 812–815. https://doi.org/10.1145/2851613.2852003

[25] James J Kistler and Mahadev Satyanarayanan. 1992. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 3–25. https://doi.org/10.1145/146941.146942

[26] Martin Kleppmann and Alastair R Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (April 2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

[27] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2018. OpSets: Sequential Specifications for Replicated Datatypes (Extended Version). https://arxiv.org/abs/1805.04263

[28] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. 2019. Source code accompanying "A highly-available move operation for replicated trees". https://github.com/trvedata/move-op

[29] Martin Kleppmann, Peter van Hardenberg, and Orion Henry. [n.d.]. Automerge. https://github.com/automerge/automerge

[30] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-first software: You own your data, in spite of the cloud. https://www.inkandswitch.com/local-first.html

[31] Puneet Kumar and Mahadev Satyanarayanan. 1993. Log-based directory resolution in the Coda file system. In *2nd International Conference on Parallel and Distributed Information Systems (PDIS)*. 202–213. https://doi.org/10.1109/PDIS.1993.253092

[32] Puneet Kumar and Mahadev Satyanarayanan. 1995. Flexible and Safe Resolution of File Conflicts. In *USENIX Winter Technical Conference*.

[33] Peter Lammich and Andreas Lochbihler. 2010. The Isabelle Collections Framework. In *1st International Conference on Interactive Theorem Proving (ITP)*. 339–354. https://doi.org/10.1007/978-3-642-14052-5_24

[34] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[35] Lightbend, Inc. [n.d.]. Akka. https://akka.io

[36] Stéphane Martin, Pascal Urso, and Stéphane Weiss. 2010. Scalable XML Collaborative Editing with Undo. In *On the Move to Meaningful Internet Systems*. 507–514. https://doi.org/10.1007/978-3-642-16934-2_37

[37] Pascal Molli, Gérald Oster, Hala Skaf-Molli, and Abdessamad Imine. 2003. Using the transformational approach to build a safe and generic data synchronizer. In *2003 International ACM SIGGROUP Conference on Supporting Group Work*. 212–220. https://doi.org/10.1145/958160.958194

[38] Mahsa Najafzadeh. 2016. *The Analysis and Co-design of Weakly-Consistent Applications*. Ph.D. Dissertation. Université Pierre et Marie Curie. https://tel.archives-ouvertes.fr/tel-01351187v1

[39] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE Tool: Proving Weakly-Consistent Applications Correct. In *2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*. https://doi.org/10.1145/2911151.2911160

[40] Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster. 2018. Co-Design and Verification of an Available File System. In *19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 358–381. https://doi.org/10.1007/978-3-319-73721-8_17

[41] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *13th ACM Symposium on Document Engineering (DocEng)*. 37–46. https://doi.org/10.1145/2494266.2494278

[42] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics - With Isabelle/HOL*. Springer. https://doi.org/10.1007/978-3-319-10542-0

[43] Omni Group. [n.d.]. OmniOutliner. https://www.omnigroup.com/omnioutliner/

[44] Benjamin C. Pierce and Jérôme Vouillon. 2004. *What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer*. Technical Report MS-CIS-03-36. Dept. of Computer and Information Science, University of Pennsylvania. http://www.cis.upenn.edu/~bcpierce/papers/unisonspec.pdf

[45] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. https://doi.org/10.1109/ICDCS.2009.20

[46] David Ratner, Peter Reiher, and Gerald J Popek. 1999. Roam: a scalable replication system for mobile computing. In *10th International Workshop on Database and Expert Systems Applications (DEXA)*. IEEE, 96–104. https://doi.org/10.1109/DEXA.1999.795151

[47] Peter Reiher. 1998. Rumor 1.0 User's Manual. https://web.archive.org/web/20170705140958/ftp://ftp.cs.ucla.edu/pub/rumor/rumor_users_manual.ps

[48] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald J Popek. 1994. Resolving File Conflicts in the Ficus File System. In *Summer USENIX Conference*. 183–195.

[49] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

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

[51] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *Comput. Surveys* 37, 1 (March 2005), 42–81. https://doi.org/10.1145/1057977.1057980

[52] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

[53] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[54] Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 59–68. https://doi.org/10.1145/289444.289469

[55] Vinh Tao, Marc Shapiro, and Vianney Rancurel. 2015. Merging semantics for conflict updates in geo-distributed file systems. In *8th ACM International Systems and Storage Conference (SYSTOR)*. https://doi.org/10.1145/2757667.2757683

[56] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th ACM Symposium on Operating Systems Principles (SOSP)*. 172–182. https://doi.org/10.1145/224056.224070

[57] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-Undo: Distributed Collaborative Editing System on P2P networks. *IEEE Transactions on Parallel and Distributed Systems* 21, 8 (Jan. 2010), 1162–1174. https://doi.org/10.1109/TPDS.2009.173

[58] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. 2008. The Isabelle Framework. In *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 33–38. https://doi.org/10.1007/978-3-540-71067-7_7