An Efficient Approach to Move Elements in a Distributed Geo-Replicated Tree

Parwat Singh Anjana[†], Adithya Rajesh Chandrassery[‡], and Sathya Peri[†]
[†]Department of Computer Science and Engineering, Indian Institute of Technology, Hyderabad, India
[‡]Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal, India
cs17resch11004@iith.ac.in, adithyarajesh.191cs203@nitk.edu.in, sathya_p@cse.iith.ac.in

***Abstract*—Replicated tree data structures are extensively used in collaborative applications and distributed file systems, where clients often perform move operations. Local move operations at different replicas may be safe. However, remote move operations may not be safe. We present an efficient algorithm to perform move operations on the distributed replicated tree while ensuring eventual consistency. The proposed technique is primarily concerned with resolving conflicts efficiently, requires no interaction between replicas, and works well with network partitions. We use the last write win semantics for conflict resolution based on globally unique operation timestamps. The proposed solution requires only one compensation operation to avoid cycles being formed when move operations are applied. The proposed approach achieves an effective speedup of 14.6−68.19× over the state-of-the-art approach in a geo-replicated setting.**

***Index Terms*—Conflict-free Replicated Data Types, Eventual Consistency, Distributed File Systems, Replicated Tree**

I. Introduction

Collaborative applications and distributed systems like Google Drive and Dropbox make considerable use of replicated data structures. Data is replicated onto several replicas closer to the users at different geo-locations to ensure high uptime and availability. Concurrent updates to various replicas by users (clients) make it very difficult to converge and lead to data consistency problems. Various approaches have been proposed to overcome these problems in several ways. The most prevalent techniques are *operational transformation* (OT) [2], [3], [4] and *conflict-free replicated data types* (CRDTs) [5], [6]. OT requires a centralized server and an active connection to modify the replicated files collaboratively. In contrast, CRDTs do not require a centralized server and allow peer-to-peer editing.

CRDTs have become an indispensable component of many modern distributed systems that guarantee some form of eventual consistency [7]. Clients update their replicas concurrently without coordination to provide high availability even when the network is partitioned. It allows users to operate locally with no lag, even if they are not connected to other replicas. The system eventually becomes consistent when a user synchronizes with other users and devices.

Popular distributed file systems such as Dropbox and Google Drive optimistically replicate data using a replicated tree data model. Clients interactively operate on the tree to perform various operations, such as updating, renaming, moving, deleting, and adding new files or directories. An interior node in the tree represents a directory, while a leaf node represents a file. This system runs a daemon on the client's machine that keeps track of changes by monitoring the designated directory [7], [8]. Clients can read and update files locally on their systems, which can then be synchronized with other replicas. Collaborative text editing and graphical editors are examples of such systems that often use the replicated tree data model.

Moving nodes is a common operation in such tree-based collaborative systems. In the file system, the move operation moves files or directories to a new location within the tree. Other examples are collaborative text editors that stores data using an XML or JSON data model and collaborative graphical editors such as Figma [9] where grouping two objects lead to adding a new node in the tree.

Move operations on the replicated tree are difficult to implement because concurrent operations by multiple clients may result in cycles; additionally, the tree structure may be broken [7], [8], [10]. Due to the concurrent operations, a concurrency control mechanism is required to ensure data correctness. Further, ensuring correctness while providing low latency, high throughput, and availability can be challenging. Kleppmann et al. [7] proposed an approach that first undoes all move operations applied with a higher timestamp (ts) than the ts of a move operation received from other replicas; applies the received operation, and finally, redoes the undone operations. They maintain total global order between the operations and ensure strong eventual consistency. This approach is computation-intensive, requires many compensation operations to avoid the cycles, and relies on making a total global order of operations.

In contrast, the proposed approach avoids re-computation for non-conflicting changes to the tree by identifying which changes might cause problems to arise. When a remote move operation leads to a conflict (a cycle), we undo one operation and send it as a compensation operation to all other replicas. By doing this, we save the time of re-computation for non-conflicting operations. Many operations that need to be undone and redone can be avoided. Essentially, the number of compensation operations is just 1 per cycle and 0 for safe operations. We observed that such a straightforward approach

Author sequence follows lexical order of last names.
Source code is available on Github at: https://github.com/anonymous1474
Full technical report is available at: https://arxiv.org/abs/2203.10285 [1]

significantly improves performance. Additionally, there is no data duplication in the proposed approach. It does not result in a directed acyclic graph on a concurrent move operation that may lead to inconsistency or divergence between the replicas and ensure strong eventual consistency.

Contributions: We propose a novel coordination-free, computationally efficient, and low latency move operation on the replicated tree. The proposed approach supports optimistic replication, which allows replicas to temporarily diverge during updates but always converges to a consistent state in the absence of new updates (§III). The performance of the proposed approach is compared against the Kleppmann et al. [7]. The experiment results show that the proposed approach achieves an effective speedup of $14.6 - 68.19\times$ over Kleppmann's approach for the remote move operation (§IV).

II. System Model

Following system model in [7], there are n *replicas* $(r_1, ..., r_n)$ communicate with each other in a completely asynchronous in a peer-to-peer fashion. We assume that replicas can go offline, crash, or fail unexpectedly. Each replica is associated with a *client*. Each client performs operations on their local replicas. Each operation is then communicated to all other replicas asynchronously via messages. A message may suffer an arbitrary network delay or be delivered out of order. Clients can read and update data on their local replica even when the network is partitioned or their replica is offline.

We consider a replicated tree structure t rooted at *root* to which clients add new nodes, delete nodes and move the nodes to the new location within the tree. In a file system, an internal node of the tree represents directories, while a leaf node represents files. In collaborative text editing, different sections, paragraphs, sentences, words, etc., in the document can be represented as tree nodes.

The proposed algorithm is executed on each replica r_i without any distributed shared memory to operate on tree t. Clients generate the operations, apply them on their local replica, and communicate them asynchronously via the network to all other replicas. On receiving an operation, remote replicas apply them using the same algorithm. The proposed algorithm supports three operations on the tree: (1) *Inserting* a new node in the tree; (2) *Deleting* a node from the tree; and (3) *Moving* a node along with a sub-tree to a child of a new parent in the tree.

All three operations can be implemented as a move operation. The *move* operation is a tuple consisting of timestamp ts, node n, and new parent p, i.e., *move*$\langle ts, n, p\rangle$. The timestamp ts is unique and generated using Lamport timestamps [11], the node n is the tree node being moved, and the parent p is the location of the tree node to which it will be moved. We represent node timestamp as $node.ts$, and operation timestamp as $o.ts$. For example, $move_x\langle ts_i,\ n_j,\ p_k\rangle$ means that a node with id n_j is moved as a child of a parent p_k in the tree t at time ts_i in replica r_x. The additional information about the old parents of the node being moved is also logged in the *parentLog* used to undo the cyclic operations when cycles are formed due to the move of the n_j by the clients at different replicas. The move operation removes n_j from the current parent and moves it under the new parent p_k along with the sub-tree of n_j; however, if n_j does not exist in the tree, a new node with n_j is created as a child of p_k.

III. Proposed Algorithm

This section describes the proposed algorithm to perform move operations while maintaining the replicated tree structure and ensuring that replicas are eventually consistent.

The proposed approach does not require cross-replica coordination and hence is highly available even when the network is partitioned. Each replica is modeled as a state machine that transitions from one state to the next by performing an operation. There is no shared memory between replicas; the algorithm operates autonomously without a central server or consensus mechanism for replica coordination; requires minimal metadata; and satisfies *strong eventual consistency*. In the case of conflicting operations, it follows the last write win approach based on timestamps computed using the Lamport clock [11] and requires one compensation operation to undo the last moved node that caused the cycle. Essentially, we try to keep the number of undos and redos to a minimum when dealing with conflicts in the replicated tree.

The proposed algorithm supports *insert, delete* and *move* operations on a replicated tree. Insert and delete involve changing various nodes' parents. An *insert* can be viewed as the creation of a new node that is moved to be the child of a specified parent. For a *delete* operation, the node is moved to be a child of a special node designated as the *trash.* Thus, the move operation can be used to implement the other two operations. Hence, in this discussion, we only consider an efficient way of moving nodes.

A client generates and applies operations locally with Algorithm 1, then sends them asynchronously over the network to all other replicas.

Each move operation takes as arguments: the node to move and the new parent. Further, each tree node also maintains the timestamp (ts) of the last operation applied which is passed to the move operation. Hence, a move operation is formally defined as: *move*$\langle ts,\ n,\ p\rangle$. Here ts is the timestamp of the move operation, n is the node to be moved and made as a child of p. An operation (local or remote) is applied using Algorithm 1 and Algorithm 2. Algorithm 2 first compares the operation timestamp ($o.ts$) with the timestamp of the node to be moved ($node.ts$). It applies the operation only if the $o.ts$ is greater than the $node.ts$. We prove convergence by showing that all the tree nodes across the replicas will get attached to the parent with the latest ts. Next, we explain how the cycle is prevented in the proposed approach.

Preventing Cycles: Recall from §I that a cycle is formed when an ancestor tree node becomes a child of its descendant tree node. Preventing cycles is difficult because concurrent move operations on different replicas may be safe independently. However, a cycle may be formed when move operations from different replicas are merged. To avoid cycles, the proposed algorithm uses timestamps and compensation operations. We check for cycles prior to performing any operation by determining whether the node to be moved is an ancestor of the new parent. We check for each operation to avoid the formation of a cycle and maintain the tree structure during concurrent moves. Another check identifies the node with the latest timestamp when a cycle is detected. Hence, the node with the most recent timestamp is returned to its previous parent, which is safe. A previous parent is said to be safe if it is not in the sub-tree of the node to be moved in the move operation.

Algorithm 1 and Algorithm 2 apply the local and remote move operations, respectively, and ensure that all operations applied will not form a cycle. The procedure followed to apply a remote move operation by Algorithm 2 is explained here. Before applying a move operation (i.e., *move*⟨*ts, n, p*⟩) Algorithm 2 checks if the operation's timestamp is greater than the previous move operation's timestamp applied on $node_n$ at Line 20. If the timestamp of the operation to be applied is smaller, it ignores the operation at Line 21. The following steps occur when a cycle is detected. The algorithm finds the node with the highest timestamp in the cycle and assigns it to *undoNode* at Line 25. Then find a safe previous parent for the *undoNode* in the while loop from Line 26. If no more previous parents are left, keep the previous parent as a conflict node at Line 28. Then move the *undoNode* to be a child of safe previous parent *undoParent* using Algorithm 1 at Line 33. This internally updates the Lamport timestamp, which will be used for the undo operation at Line 5. The operation is applied in the local replica at Line 6 and Line 9. After that, it sends the compensation operation to other replicas at Line 35. If *undoNode* is the same as $node_n$ at Line 36 then return at Line 38. Since it already applied an operation with a higher timestamp on $node_n$ as the undo operation. Otherwise, apply the operation to change the parent of $node_n$ at Line 39 and Line 40.

Storing fixed m previous parents for each node will be adequate by considering storage; moreover, increasing the value of m increases the search time. For current experiments, m is fixed to 5. Identifying the optimal number of previous parents (i.e., m) is left as future work. If previous parents in *parentLog* are too small for the node to be moved to break the cycle, it is moved under a special node known as the *conflict node* (a child of the root) to break the cycle. The *conflict node* is special node that cannot be moved, it ensures that it will always be free from cycles. In case if the number of previous parents (or m) for a node to be moved is deleted (by moving under *trash*), we still can move the node under the previous parent (deleted node in this case). Next, the clients have the choice to change location. Since the nodes attached to *trash* have not been deleted permanently. Even when all previous parents are permanently deleted or $parentLog_x$ for a node x is empty, we can still move that

Algorithm 1: `applyLocal`(n, p): apply local operations and send them to remote replicas.

```
1  Procedure applyLocal(n, p):
2      $node_n$ ← get_node(root, n) // Gets reference of the node with id $n$ in $t$.
3      $node_p$ ← get_node(root, p) // Gets reference of the node with id $p$ in $t$.
4      Lock()// Get lock, so that at a time only one operation will be applied by threads (local thread or receiver thread) on the tree $t$.
5      ts ← ++lc_time
       // Check for the cycle returns true if found.
6      if ¬checkCycle($node_n$, $node_p$) then
7          parentLog[$node_n$.id].add($node_n$.parent); // Update the current parent of $node_n$ in the parentLog.
8          $node_n$.parent ← $node_p$.id
9          $node_n$.ts ← ts
10     Unlock();
       // Send move operation to other replicas.
11     for j = 0 to numReplicas do
12         ch[j].add(move⟨ts, n, p⟩)
```

Algorithm 2: `applyRemote`(): receives and applies remote move operations.

```
13 Procedure applyRemote():
14     while true do
15         move⟨ts, n, p⟩ ← Stream.Receive()// Receive remote move operation. If returns End, receiver threads will join.
16         if End then break
17         $node_n$ ← get_node(root, n)// Get reference of $n$.
18         $node_p$ ← get_node(root, p)// Get reference of $p$.
19         Lock()// Lock so that at a time only one operation will be applied by threads (local or receiver thread) on the tree $t$.
           // Already applied an operation with higher ts on node $n$ then ignore received operation with smaller ts as node ts will be higher.
20         if ts < $node_n$.ts then
21             return
22         lc_time ← max(ts, lc_time)
23         parentLog[$node_n$.id].add($node_n$.parent) // Update the current parent of $node_n$ in the parentLog.
           // Check for the cycle returns true if found.
24         if checkCycle($node_n$, $node_p$) then
               // Find the node between $node_n$-$node_p$ with highest ts.
25             $undoNode$ ← findLast($node_n$, $node_p$)
               // Undo (move back) to a previous parent, not in the sub-tree of $node_n$. Keep searching for a suitable node to undo; if not found safe previous parent, then move $undoNode$ under conflict node.
26             while true do
27                 if parentLog[undoNode.id] == NULL then
28                     $undoParent$ ← conflict node
29                 else
                       // Get and delete previous parent from parentLog for undoNode.
30                     $undoParent$ ← parentLog[$undoNode$.id].pop()
                   // Check if a cycle exists between $n$ and undoParent; if no cycle, then found a safe node to move back to breaks cycle between $n$ & $p$.
31                 if ¬checkCycle($node_n$, undoParent) then
32                     break
33             applyLocal($undoNode$, $undoParent$)
               // Send compensation opr to other replicas.
34             for j = 0 to numReplicas do
35                 ch[j].add(move⟨ts, n, p⟩)
           // Already applied a higher ts operation.
36         if undoNode == $node_n$ then
37             Unlock()
38             return
39         $node_n$.parent ← $node_p$.id
40         $node_n$.ts ← ts
41         Unlock()
```

node under *conflict node* to break the cycle.

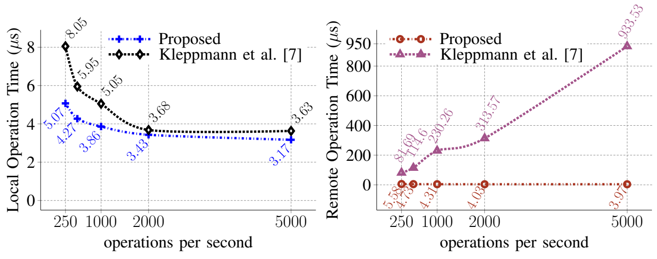
IV. Performance Evaluation

We have implemented our algorithm in Golang [12] and wrapped it in gRPC [13] network service to deploy at three different geo-location (Western Europe, Southeast Asia, and East US) on Microsoft Azure Standard E2s_v3 VM instances, each consisting of 2 vCPU(s), 16 GiB of memory, and 32 GiB of storage running Ubuntu 20.04 and Intel Xeon Platinum 8272CL processor. The proposed approach is compared with the approach proposed by Kleppmann et al. [7].

Each replica generates and applies local operations, subsequently asynchronously propagating and receiving the operations to/from the other two replicas. By selecting tree nodes uniformly at random from the tree size, each replica generates $(\frac{1}{3})^{rd}$ of total operations to apply and send, while receiving $(\frac{2}{3})^{rd}$ of operations from two other replicas. When a replica receives a remote operation, it applies. Any undo operation due to cycle identifies the previous parent. Once the appropriate parent is identified, the node is moved as a child and the undo operation is sent to other replicas. Note that our experimental workload is more conservative and contains more conflict than the real-time workload. Further, move operations conflict only with the move or delete operations and not with operations that update a node value or insert a new subtree.

Fig. 1 depicts the average time to apply a local and remote move operation. The number of operations/second is varied from 250 to 5K, while the number of nodes in the tree is kept constant at 500. Fig. 1(A) shows that the average time to apply a local move operation at a replica is not significant between both approaches with increasing operations. The apply time for a local move operation drops as the number of operations increases. However, the time to apply a remote move operation is almost constant in the proposed approach, while Kleppmann's approach has the opposite trend; the time increases with operations per second for remote move operations as shown in Fig. 1(B). There is a significant performance gap for the remote operation in both the approaches; this is because the number of compensation operations (undo/redo) by Kleppmann's approach is ≈ 200 undo and redo operations for every remote operation a replica receives while in our case it is only 1 that too whenever there is a cycle (conflict).

Fig. 1(B) shows that Kleppmann's approach attains a maximum apply time of 933.53μs over a remote move operation at 5K operations per second. In comparison, the minimum is 81.69μs at 250 operations per second; the minimum time is $\approx 14.63\times$ higher than the maximum time of 5.58μs at 250 operations by the proposed approach. It can be seen that the proposed approach achieves, on average, a speedup of $1.34\times$ for the local move operation, while $14.6\times$ to $68.19\times$ speedup for the remote move operation over Kleppmann's approach. Hence, the proposed approach is much more efficient in applying remote operations. The difference in time only increases with an increase in the rate of operations per second. It shows the performance benefits of the proposed approach. In Kleppmann's approach, as the rate of operation generation



(A) Local move operation apply time (B) Remote move operation apply time

Fig. 1: Average time to apply a move operation.

increases, the number of operations in flight also increases. Hence, the compensation cost will be high. In contrast, the compensation cost remains the same in the proposed approach.

V. Conclusion

We proposed a novel algorithm for coordination-free, computationally efficient, and low latency move operation on the replicated tree. The proposed technique requires a single compensating operation to undo the effect of the cyclic operation. It achieves a maximum speedup of $68.19\times$ for remote move operation over Kleppmann's approach [7]. We have stored a constant number of the previous parents for every node, hence, identifying the optimal number of previous parents is left as future work. Implementing an efficient move operation on other CRDTs could be an exciting direction to explore.

References

[1] P. S. Anjana, A. R. Chandrassery, and S. Peri, "An efficient approach to move elements in a distributed geo-replicated tree," *arXiv preprint arXiv:2203.10285*, 2022.

[2] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the ACM SIGMOD international conference on Management of data*, 1989, pp. 399–407.

[3] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping, "High-latency, low-bandwidth windowing in the jupiter collaboration system," in *Proceedings of the 8th annual ACM symposium on User interface and software technology*, 1995, pp. 111–120.

[4] T. Seifried, C. Rendl, M. Haller, and S. Scott, "Regional undo/redo techniques for large interactive surfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012, pp. 2855–2864.

[5] N. Preguiça, "Conflict-free replicated data types: An overview," *arXiv preprint arXiv:1806.10254*, 2018.

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Ph.D. dissertation, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[7] M. Kleppmann, D. P. Mulligan, V. B. F. Gomes, and A. Beresford, "A highly-available move operation for replicated trees," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021.

[8] S. Nair, F. Meirim, M. Pereira, C. Ferreira, and M. Shapiro, "A coordination-free, convergent, and safe replicated tree," *arXiv preprint arXiv:2103.04828*, 2021.

[9] Figma, "Figma: the collaborative interface design tool," https://www.figma.com/, [Accessed 27-07-2021].

[10] N. Bjørner, *Models and software model checking of a distributed file replication system*. Springer, 2007, pp. 1–23.

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

[12] "Go," https://golang.org/, [Accessed 31-03-2022].

[13] "grpc: a high performance, open source universal rpc framework," https://grpc.io/, [Accessed 31-03-2022].