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

2022 IEEE 15th International Conference on Cloud Computing (CLOUD) | 978-1-6654-8137-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLOUD55607.2022.00071

***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. When clients perform arbitrary move operations concurrently on different replicas, it could result in various bugs, making this operation challenging to implement. Previous work has revealed bugs such as data duplication and cycling in replicated trees. In this paper, 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 timestamps of operations. 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× to 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 clients at different geo-locations to ensure high uptime and availability. When clients perform concurrent operations, data replication at different replicas may cause consistency issues. Various consistency models have been implemented in the literature to ensure the mutability of replicated data. These consistency models are classified into different classes based on the consistency guarantees they provide, such as strong consistency, eventual consistency [1], and causal consistency [2]. The mutation occurs instantly across replicas in the strong consistency model; this is the strongest condition in an ideal setting. However, replicas may diverge when the network is partitioned; consequently, strong consistency is not easy to achieve with network partitions without sacrificing availability. Further, strong consistency suffers from performance overhead due to high synchronization costs when the network is reliable [3].

Strong consistency is the strongest form of consistency for any replicated system; unfortunately, it comes with a considerable performance penalty. As a result, it may not be suitable for replicated systems that require high availability, scalability with concurrent updates and convergence guarantee to a consistent state. As a result, systems designed based on weaker consistency models such as eventual consistency have become popular [2], [4]. In the eventual consistency model, replicas may diverge for various reasons; however, they eventually converge to the same state if no new updates are performed at any replica [5], [4], [6].

Concurrent updates to various replicas make it very difficult to converge and has been extensively studied in the literature. Numerous approaches have been developed to overcome this problem in several ways. The most prevalent techniques are *operational transformation* (OT) [7], [8], [9] and *conflict-free replicated data types* (CRDTs) [10], [11], [12]. OT requires a centralized server and an active server connection to modify the replicated file 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 applications that guarantee some form of eventual consistency [13]. 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 distributed file system runs a daemon on the client's machine that keeps track of changes by monitoring the designated directory [13], [14]. 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 distributed systems that often use the replicated tree data model.

*A poster version of this paper is accepted in 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid '22), Taormina (Messina), Italy, 16-19 May 2022.
Source code is available on Github at: https://github.com/anonymous1474
Author sequence follows the lexical order of last names.

2159-6190/22/$31.00 ©2022 IEEE
DOI 10.1109/CLOUD55607.2022.00071

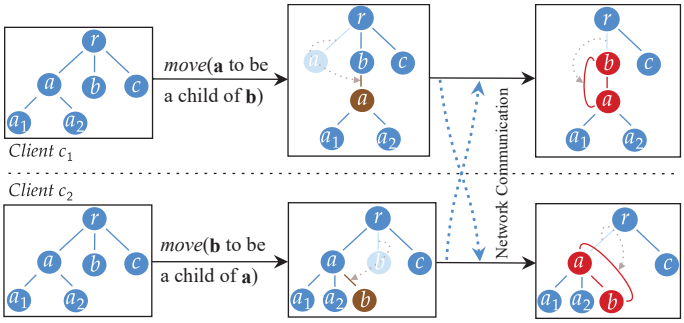


Fig. 1: Difficulty with the move operation on a replicated tree: let us assume a tree structure t rooted at r, and two clients c_1 and c_2, concurrently operating on t in their local replicas. Let say c_1 move *(a to be a child of b)* concurrently with client c_2 moving *(b to be a child of a)* in their local version of t without any coordination. Later, when these replicas are synchronized by propagating local operations, it may produce a cycle $a \leftrightarrow b$ disconnected from the root r.

The clients can read and update the files offline on their local system, which can later be synchronized with other replicas. Moving nodes is a common operation in such tree-based collaborative applications. In the file system example, the move operation moves files or directories to a new location within the tree. In a collaborative text editor that stores data using an XML or JSON data model, changing a paragraph to bullet points generates a new list and bullet point node. It then moves the paragraph nodes under the bullet point node. Another example is a collaborative graphical editor (Figma [15]) where grouping two objects lead to adding a new node in the tree [13].

This paper focuses on the move operation in the replicated tree CRDT due to its usefulness. The *move operation* moves a sub-tree within the tree. This operation is difficult to implement because concurrent operations by multiple clients may result in cycles; additionally, the tree structure may be broken [16], [13], [14]. Due to the concurrent operations, a concurrency control mechanism is required to ensure the data structure's correctness. Further, ensuring correctness while providing low latency, high throughput, and maintaining high availability can be very challenging.

An example in Figure 1 shows the difficulty associated with the move operation in the replicated setting. The tree structure is replicated on multiple systems. Different clients can perform concurrent operations, leading to various malformations in the tree, such as a cycle, duplication, and detachment from the parent node. Concurrent move operations on the same tree node cause the data duplication problem. Providing support for a concurrent move operation for the replicated tree that does not require continuous synchronization or centralized coordination is problematic because two operations that are individually safe at their local replicas, when combined, might produce a cycle. Prior works by Nair et al. [14] and Kleppmann et al. [13] have shown that Dropbox suffers from duplication, and Google Drive results in errors due to the formation of cycles.

We present an efficient protocol to perform move operations while maintaining the distributed 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. In case of conflicting operations, we follow the last write win approach based on timestamps computed using the Lamport clock [17]. The proposed protocol requires one compensation operation to undo the last moved node that causes the cycle. Essentially, we address the conflict resolution problem of the replicated tree by minimizing the number of undo and redo operations required to resolve conflicts.

The significant contributions are as follows:

- We propose a novel move operation on the distributed geo-replicated tree that is computationally efficient and offers low latency operations. 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 (see §IV).
- The proposed algorithm guarantees strong eventual consistency; correctness proofs are provided in §V to show the *convergence* of the replicas and the maintenance of the tree structure.
- The performance of the proposed approach is compared against the Kleppmann et al. [13]. The experiment results show that the proposed approach achieves an effective speedup between 14.6× to 68.19× over Kleppmann's approach for the remote move operations (§VI).

A brief overview of the related work aligned with the proposed approach is discussed in §II, while the system model is given in §III. §VII conclude with some future research directions.

II. Related Work

This section briefly discusses the related work that has been done in line with the proposed approach.

Algorithms on replicated data structures are classified into two classes: operational transformation (OT) [7], [18], [8], [9] and conflict-free replicated data types (CRDTs) [10], [11], [12]. Many of the proposed approaches mainly support two operations: insert and delete. Furthermore, the approaches based on OT require a central server and an active network connection between the client and the central server. It means it does not work when the replica is offline. In comparison, CRDT mitigates this issue by allowing asynchronous peer-to-peer communication between replicas using optimistic replication. In the presence of faults to reduce operation response time and increase availability, the optimistic replication [19] allows replicas to diverge temporarily.

The CRDTs are mainly categorized into two types: state-based [11] and operation-based [20], [11]. The former, alternatively referred to as convergent replicated data types, is more straightforward to design and implement. However, a significant disadvantage is that it requires the transmission of the entire state to every other replica. On the other hand, later, also referred to as commutative replicated data types, transmit only update operations to each replica, thus requiring less

network bandwidth. Nonetheless, it is built on the assumption of a reliable communication network, which means that no operations are dropped or duplicated. The data structures supported by these CRDTs are counters, lists, registers, sets, graphs, trees, etc. The proposed protocol implements efficient move operation on operation-based tree CRDT.

Extensive research has been done to implement geo-replicated distributed tree data structures, which are used in a variety of distributed applications, including Google Drive, Dropbox (file systems), Google Docs, Apple's Notes App (collaborative text editing), and Figma (collaborative graphical editor). For replicated tree CRDTs, numerous algorithms have been proposed. Several of these algorithms support only insert and delete operations and have a long response time or latency.

Martin et al. [21] proposed tree CRDT for XML data to support insert and delete operation. While supporting these two operations in JSON data format is proposed by Kleppmann et al. [22]. Insert and delete operations can be used to implement a move operation; however, this could lead to the data duplication problem and increase the number of computation steps. A replicated file system using tree CRDT is implemented in [23], [24]. These solutions result in data duplication (tree node duplication). When replicas perform operations concurrently, data duplication occurs, resulting in irreversible divergence between replicas or the need for manual intervention to restore replicas to a consistent state.

Data duplication is a severe issue, and for large systems, manually handling this problem is very difficult. On the other hand, some techniques require extensive metadata exchanges between replicas to mitigate these issues, which increases network bandwidth requirements. The solution proposed in [25] results in a directed acyclic graph on concurrent moves. The approach proposed in [14] requires causal delivery of operations that may not be possible when a replica crash fails, or the network is unreliable.

As discussed earlier in §I, move operations are difficult to implement because two concurrent moves can produce a cycle and separate the node from its parent or ancestor. Moving a node in its descendent tree may produce a cycle and break the tree structure. Local move operations on a replica may or may not result in cycles. However, remote move operations may cause cycles that must be handled properly to preserve consistency. Hence, an efficient approach must be proposed to move the nodes and their subtrees to another location in the replicated tree.

The most recent work is proposed by Kleppmann et al. [13]. This approach is computation-intensive, requires many compensation operations to avoid the cycles in the replicated tree, and relies on making a total global ordering of operations and ensure strong eventual consistency. Eventual consistency focuses solely on a liveness guarantee, i.e., updates will be detected eventually. Strong eventual consistency, on the other hand, provides the security guarantee that any two nodes that have received the same unordered batch of updates will be in the same state.

In Kleppmann's approach, before applying any remote operation (operation received from remote replica), they first undo all operations applied with a higher timestamp than the received operation, then apply the received operation, finally, redo all those undone operations. They maintain total global order between the operations and ensure strong eventual consistency. Unlike their approach, which requires multiple undo and redo operations per remote move operation, the proposed approach requires only one undo and compensation operation per conflicting operation and avoids multiple undo and redo operations for non-conflicting move operations.

The proposed approach ensures strong eventual consistency. It avoids re-computation for non-conflicting changes to the tree by identifying which changes might cause problems to arise. In our approach for a remote operation that creates a problem (cycle), we undo one operation and send that as a compensation operation to all other replicas. By doing this, we save the time of re-computation for non-conflicting operations, as well 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 simple approach improves the performances significantly. Additionally, there is no data duplication in the proposed approach and does not result in directed acyclic graph on concurrent move operation that may lead to inconsistency or divergence between the replicas. Moreover, causal delivery and strict global total order while applying the operations for consistency are not required. So, we propose a novel coordination-free efficient move operation on replicated tree CRDT to support low latency and high availability operations.

III. System Model

Following system model in [13], 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.

We propose an efficient algorithm to maintain the replicated tree structure. 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:

- *Inserting* a new node in the tree.
- *Deleting* a node from the tree.
- *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 [17], 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 *present_log* 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.

We implement *insert* and *delete* operations as a *move* operation. *Insert* is implemented as a move operation, where the node being moved (n_j in the above example) does not already exist in the tree. For *delete*, we use a special node called *trash*—a child of root and the parent of all deleted nodes. When a delete on a node n_j is invoked, it is moved to the sub-tree of *trash*. We explain further details in §IV.

The proposed algorithm satisfies the strong eventual consistency property *convergence*—two replicas are said to converge or be in the same state if both of them have seen and applied the same set of operations. The replicas may apply the operations in any order due to reordering of messages and delays. This implies operations must be commutative. The formal proof is provided in §V.

IV. Proposed Algorithm

This section describes the proposed algorithm for performing efficient move operations on the replicated tree. 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, and the algorithm operates autonomously. The proposed protocol requires no central server or consensus mechanism for replica coordination (unlike OT), requires minimal metadata, and satisfies *strong eventual consistency*. A client generates and applies operations locally with Algorithm 4, then sends them asynchronously over the network to all other replicas with Algorithm 1. Due to page constraints, we have described the main idea while all the details are explained as pseudocode in the various algorithms.

The proposed algorithm supports *insert, delete* and *move* operations on a replicated tree. It can be shown that inserting and deleting involve changing various nodes' parents. *insert* can be viewed as the creation of a new node that is to be 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 denoted 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.

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. 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. The data structures are shown in Listing 1.

An operation (local or remote) is applied using Algorithm 4 and Algorithm 5. Algorithm 5 first compares the operation timestamp (i.e., $o.ts$) with the timestamp of the node to be moved (i.e., $node.ts$). 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.

Listing 1: Data Structures

```
move {
  clock ts; /* Unique timestamp using Lamport
      clock. */
  int n; // Tree node being moved.
  int p; // New parent.
};
treeNode {
  int id; // Unique id of the tree node.
  clock ts; /* Timestamp of the last operation
      applied on the node. */
  int parent; // Parent node id.
};
lc_time - Lamport timestamp of a replica.
root - Original starting point of the tree.
present_log - Stores m unique previous parents of
     each node in the adjacency list form.
ts - Timestamp.
ch[] - Array of channels (size equal to the
    number of replicas).
conflict node - a special child of the root that
    cannot be moved.
```

Algorithm 1: `send`(channel ch, i): send local operations to other replicas.

```
Procedure send(channel ch, i):
    while true do
        move⟨ts, n, p⟩ ← ch.get(i)
        if ts == -1 then
            break /* Threads will join when
                condition becomes true.        */
        RPC.send(move⟨ts, n, p⟩)
```

Algorithm 2: `checkCycle`(n_i, n_j): detect cycle between two nodes in the tree 't'.

```
Procedure checkCycle(n_i, n_j):
    while n_j ≠ root do
        if n_j == n_i then
            return TRUE
        n_j ← get_node(root, n_j.parent)
    return FALSE
```

Algorithm 3: `findLast`(n_i, n_j): find node with highest timestamp in cycle.

```
29 Procedure findLast(n_i, n_j):
30     maxTS ← n_i.ts
31     undoNode ← n_i
32     while n_j ≠ n_i do
33         if n_j.ts > maxTS then
34             undoNode ← n_j
35             maxTS ← n_j.ts
36         n_j ← get_node(root, n_j.parent)
37     return undoNode
```

Algorithm 4: `applyLocal`(n, p): apply local operations and send them to other replicas.

```
38 Procedure applyLocal(n, p):
39     node_n ← get_node(n) /* Gets reference of the
           node with id n in t.                    */
40     node_p ← get_node(p) /* Gets reference of the
           node with id p in t.                    */
41     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.                                 */
42     ts ← ++lc_time
       /* The check cycle method returns true if it
           finds a cycle.                          */
43     if ¬checkCycle(node_n, node_p) then
44         present_log[node_n.id].add(node_n.parent) /* Update
               the current parent of node_n in the
               present_log.                        */
45         node_n.parent ← node_p.id
46         node_n.ts ← ts
           // Send move operation to other replicas
47         for j = 0 to numReplicas do
48             ch[j].add(move⟨ts, n, p⟩)
       /* If found cycle, then discard the move
           operation.                              */
49     Unlock()
```

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 (using Algorithm 2). As a result, 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 4 and Algorithm 5 ensure that all operations applied will not form a cycle. Algorithm 4 applies the local move operations, while Algorithm 5 applies remote move operations.[1] The procedure followed to apply a remote move operation by Algorithm 5 is explained here. Before applying

Algorithm 5: `applyRemote`(): receives and applies remote move operations.

```
50 Procedure applyRemote():
51     while true do
52         move⟨ts, n, p⟩ ← Stream.Receive()/* Receive
               remote move operation. If returns
               End, receiver threads will join.    */
53         if End then break
54         node_n ← get_node(n)// Get reference of n
55         node_p ← get_node(p)// Get reference of p
56         Lock()/* Lock so that at a time only one
               operation will be applied by threads
               (local/receiver) 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.             */
57         if ts < node_n.ts then
58             return
59         lc_time ← max(ts, lc_time)+1
60         present_log[node_n.id].add(node_n.parent) /* Update
               the current parent of node_n in the
               present_log.                        */
           /* The check cycle method returns true if
               it finds a cycle.                   */
61         if checkCycle(node_n, node_p) then
               /* Find the node between node_n-node_p
                   with highest ts.                */
62             undoNode ← findLast(node_n, node_p)
               /* Undo (move back) to a most recent
                   previous parent that is 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.                  */
63             while true do
64                 if present_log[undoNode.id] == NULL then
65                     undoParent ← conflict node
66                 else
                       /* Get and delete previous
                           parent from present_log for
                           undoNode.               */
67                     undoParent ←
                         present_log[undoNode.id].pop()
                   /* Check if a cycle exists between
                       n & undoParent if no cycle,
                       then found a safe node to move
                       back to breaks cycle between n
                       & p.                        */
68                 if ¬checkCycle(node_n, undoParent) then
69                     break
70             applyLocal(undoNode, undoParent)
           // Already applied a higher ts operation.
71         if undoNode == node_n then
72             Unlock()
73             return
74         node_n.parent ← node_p.id
75         node_n.ts ← ts
76         Unlock()
```

[1]Locking methods (`Lock()` and `Unlock()` at Line 41, 49, 56, and 76) are used at a replica to synchronize local and remote move operation processes.

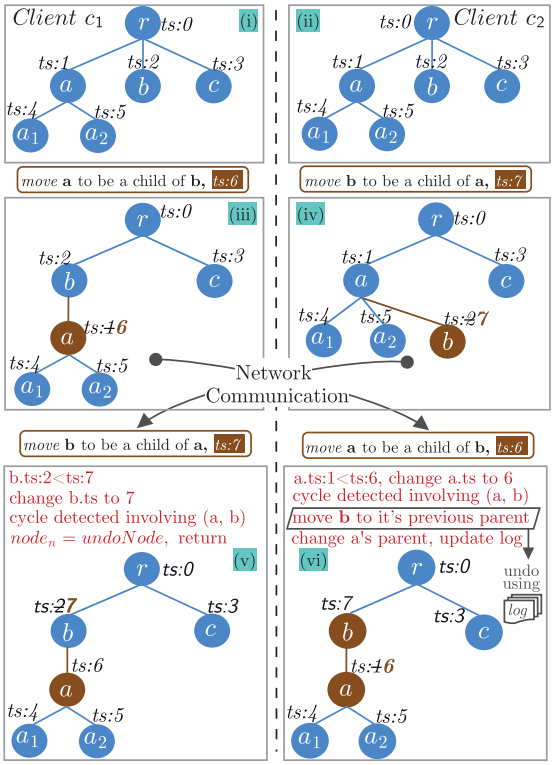


Fig. 2: Preventing cycles in proposed approach.

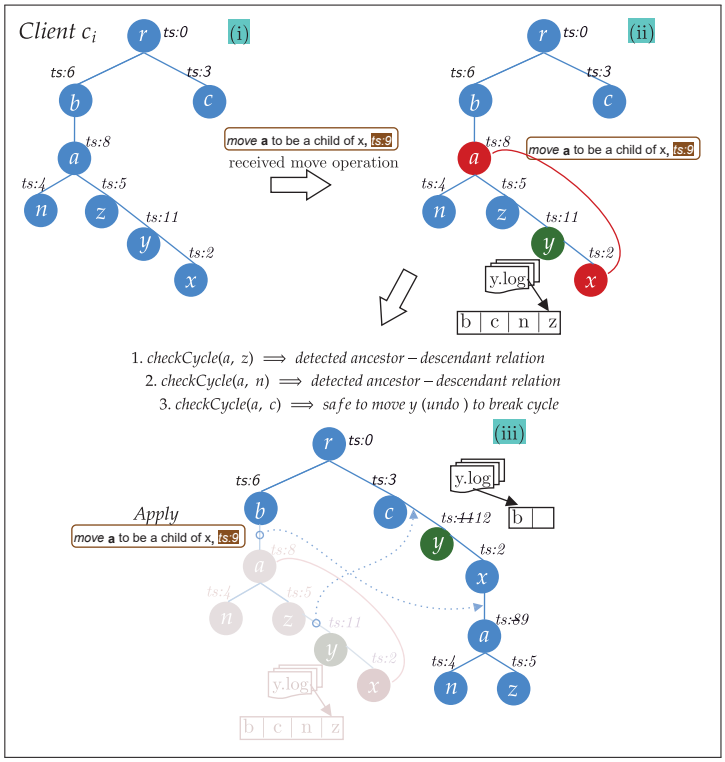


Fig. 3: Remote move operation handling at a replica.

a move operation (i.e., *move*$\langle ts, n, p\rangle$) Algorithm 5 checks if the operation's timestamp is greater than the previous move operation's timestamp applied on $node_n$ at Line 57 (Line 57). If the timestamp of the operation to be applied is smaller, it ignores the operation at Line 58. 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 62. Then find a safe previous parent for the *undoNode* in the while loop from Line 63. If there are no more previous parents left, then keep the previous parent as a *conflict node* at Line 65. Then move the *undoNode* to be a child of safe previous parent *undoParent* using Algorithm 4 at Line 70. This internally updates the Lamport timestamp, which will be used for the undo operation at Line 42. Then the operation is applied in the local replica at Line 43 and Line 48. After that, it send the compensation operation to other replicas. If *undoNode* is the same as $node_n$ at Line 71 then return at Line 73. 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 Lines 74 and 75.

Working Example: From Figure 1, let us assume that initially, both clients (replicas) consist of the same tree with timestamp as shown in Figure 2 (sub-figure (i) and (ii)). Each operation is assigned unique timestamp. Client c_1 generates and applies the local operation $move_1\langle ts{:}6,\ n{:}a,\ p{:}b\rangle$, i.e., *move* (a) *to be a child of* (b) with timestamp $move_1.ts$: 6 as shown in (iii). Similarly, client c_2 generates and applies the local operation $move_2\langle 7,\ b,\ a\rangle$, i.e., *move* (b) *to be a child of* (a) with timestamp $move_2.ts$: 7 as shown in (iv). As shown in Figure 2 (v) when client c_1 receives the $move_2\langle 7,\ b,\ a\rangle$ operation from c_2, it executes the following steps:

1) Operation timestamp ($move_2.ts$: 7) is not less then move node timestamp ($b.ts$: 2).
2) Change move node timestamp to operation timestamp, i.e., $b.ts$: 7.
3) Cycle is detected involving (a, b).
4) The node in the cycle with the highest timestamp is (b) that is found using Algorithm 3.
5) The node (b) is moved to its previous parent which is (r).
6) The compensation operation is propagated to other replicas to ensure that every replica has seen and applied the same set of operations.
7) Since $node_n$ (i.e., node received in move operation) is same as node to be moved back, so algorithm returns.

Similar steps are followed when operation $move_1\langle 6,\ a,\ b\rangle$ from c_1 is received at c_2 (see Figure 2 (vi)). Except for case (7), where $node_n$ is not the same *undoNode*. Operation is applied on node (b). In summary, node (b) is moved back to its previous parent (r) using information stored in the *present_log*, then received operation is applied, i.e., (a) is moved as a child of (b), and the *present_log* is updated.

Note that when a cycle is detected, the algorithm identifies the node with the latest timestamp and move that node back (e.g., (b) in Figure 2 (vi)) to the previous parent, where it is safe. In the Algorithm 5: Line 63 – Line 69 tries to identify the previous parent of the node (e.g., (b) in Figure 2 (vi)) where it can be moved back safely, and cycle can be broken. We are storing m (a constant number) previous parent for tree node

in an adjacency list in the *present_log*.

Let us consider another example, as shown in Figure 3(i), when a replica (i.e., $Client\ c_i$) receive an operation to *move* a *to be a child of* x. As shown in Figure 3(ii), since a is an ancestor of x it will be detected that this operation can form a cycle. So algorithm finds the node with the highest timestamp in the potential cycle between a and x. Here, y has the highest timestamp; therefore, y can be moved to one of its previous parents to break the cycle. The algorithm, checks if any of its previous parents are safe (i.e., the previous parent is not a descendant of a). The check fails for n and z but c is a safe previous parent. Hence move y to be a child of c, as shown in Figure 3(iii). As a result of this, x is no longer a descendant of a. So apply the original operation to move a to be a child of x.

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 *present_log* 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 again as the nodes attached to *trash* have not been deleted permanently. Even when all previous parents are permanently deleted or $present\_log_x$ for a node x is empty, we can still move that node under *conflict node* to break the cycle.

The following important question is, how do we know which tree nodes are part of the cycle? As previously mentioned, the cycle is formed when the node moves to an ancestor of its new parent. Hence the cycle will lie between the new parent and the node to be moved. Since operations are applied one by one, an operation can almost form only one cycle. When a user sees nodes attached to *conflict node*, they understand it resulted in the formation of a cycle and had to be resolved. A valid question could be why not just prevent the last operation for each replica that results in the cycle instead of looking for the global last operation. It is possible that since operations are applied across replicas in different orders, they will result in each replica moving a different tree node to *conflict node* / *previous parent*. This means that a large number of operations could be ignored. However, by looking for the global last operation, only the effect of one operation gets ignored.

Globally Unique Timestamps: We use Lamport clock [17] on each replica for timestamping operations. However, this alone will not make it globally unique. Hence, we use the *replica id* for tiebreakers when the Lamport timestamps are equal; together, they globally unique. As an alternative to the Lamport clock, the hybrid logical clock [26] can be used that provides the unique timestamps.

Difficulties: *Trash* can grow indefinitely. There can be a permanent *delete* that recursively deletes from the leaf nodes to maintain consistency. Similar to the *rm -r* command in Linux. However, this can lead to the case where a permanently deleted node is in the log of previous parents of other nodes. We will have to skip that parent and continue to the next previous parent in such cases. If none of the previous parents are safe or are permanently deleted, we move the node to be the child of *conflict node*. As we are storing only the last m previous parents and then moving to *conflict node*, we are missing out on moving to the older positions if none are safe. A sound argument can be made that instead of moving a node to ancient locations, it is better to move to the *conflict node* to notify the user that the *move* operation on the following node was unsafe. Users can find another suitable location to move it to. Implementing permanent delete is left as future work.

Another important point is that, in the case of a cycle due to remote operations, the proposed approach requires propagating the undo operation to other replicas (requires one undo operation per remote move operation) that may be an additional message cost. However, it decreases many undo and redo operations to just one compensation operation at each replica, compensating for additional message costs. To summarize, we provide a replicated tree that can support efficient and highly available move operation.

V. Proof of Correctness

This section provides the formal proof that all the replicas eventually converge to the same tree (state) and maintain the tree structure through optimistic replication. All our operations (*insert, delete, move*) are just changing the parent to which the node is attached.

Lemma 1 (Duplication): A tree node will never be located at multiple different positions.

Proofsketch. For every node, we only store a single value for its parent. It must have more than one parent to be duplicated at multiple positions. However, that is not possible in our approach since we only store a single parent based on the last written win semantic.

Lemma 2 (Cycle): No operation will result in the formation of a cycle.

Proofsketch. Before applying any operation, our algorithm tries to detect if it will form a cycle. Assume we have an operation of the form $\langle ts, n, p \rangle$ where ts is a timestamp, n is the tree node to be moved, p is the parent. As previously stated, the operation will only form a cycle if n is an ancestor of p.

We traverse all the way up from p to *root*, and in case we do not find n in that path, it implies n is not an ancestor of p. Hence the operation is safe to apply and will not form a cycle. If we find n is an ancestor of p, we will apply an alternate compensation operation. Hence we never apply an operation where n is the ancestor of p.

So now, for a cycle to exist, it needs to be formed from an operation where n is not the ancestor of p. How to prove that if n is not an ancestor, it will not form a cycle?

If n and p need to form a cycle, there should be a path from n to p and p to n. However, applying the operation will only create a path from n to p. The path from p to n needs to exist before, and such a path will only happen if n is an ancestor of p.

Lemma 3 (Forest): The tree will never be split into multiple forests.

Proofsketch. We always maintain a parent for every tree node other than *root*, and we do not allow operations that have the same parent and tree node value. The tree could be split into forests if there is some node without a parent or a cycle. We have already shown that cycles cannot be formed, and our algorithms always maintain a parent for each tree node other than *root*.

Theorem 4 (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.

Proofsketch. Since the node with the highest timestamp in the cycle is moved back to the previous parent, i.e., the previous parent must not be in the sub-tree of the node to be moved in a move operation, moreover, when there is no previous parent such that it is not in the sub-tree, then *conflict node* (special child of the *root* that can not be moved) is assigned as the previous parent. So one of the nodes in the cycle is always moved back to a node which is not in the sub-tree of the node to moved. As a result of this the new parent of the node to be moved will no longer be in its sub-tree. So there is no ancestor-descendant relation between the node to be moved and the new parent and the operation is safe to be applied.

Having explained the lemmas and theorem, we now explain the main theorem.

Theorem 5 (Convergence): Replicas that have seen and applied the same set of operations will converge to the same tree.

Proofsketch. Say two replicas r_1 and r_2 have seen the same set of operations. They will have the same parent for each node as the operation with the latest timestamp taken as the parent.

From Lemmas 1, 2, 3, and Theorem 4, we get that the tree structure is maintained and there will be no cycles in the tree. Next, assume that replicas r_1 and r_2 have seen the same set of updates but have different parents for a key (k). Suppose the replica r_1 for k has timestamp ts_1 and parent p_1. The replica r_2 for k has timestamp ts_2 and parent p_2. We know $ts_1 \neq ts_2$ since we are using globally unique timestamps (ties are broken by replica id), and if they were equal, then parents would have been the same. This implies that either $ts_1 < ts_2$ or $ts_1 > ts_2$. This means that one of the replicas has not applied the latest timestamp. However, according to correctness of our algorithm, it was supposed to do that. Hence this is not possible. It means the initial assumption was wrong that the replicas have different parents for the same key or have seen the same set of updates.

TABLE I: Network latency (ms) between different replicas

	US East	West Europe	Southeast Asia
US East	0	41	111
West Europe	41	0	79
Southeast Asia	111	79	0

VI. Performance Evaluation

This section presents the implementation details (§VI-A) and performance comparison (§VI-B) of the proposed approach with the state-of-the-art approach by Kleppmann et al. [13] in a geo-replicated setting at three different continents to demonstrate the usefulness of the proposed approach.

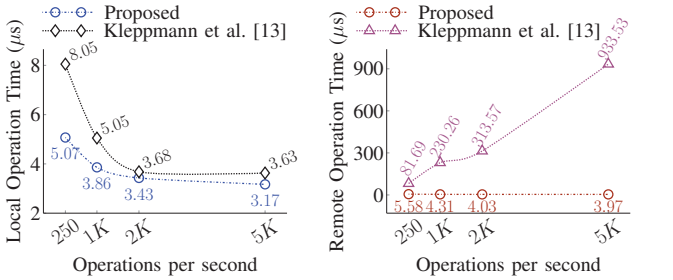
A. Implementation

We have implemented the proposed algorithm in Golang [27] and wrapped it in gRPC [28] 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 temporary storage running Ubuntu 20.04 operating system and Intel Xeon Platinum 8272CL processor.

Table I shows the network latency's between different geo-locations chosen for the experiments. The network latency from US East to Southeast Asia is 111 ms, the maximum, and West Europe to US East is 41 ms which is the minimum. However, it is not considered in the final results because the *time to apply* a move operation (local or remote) is computed at each replica.

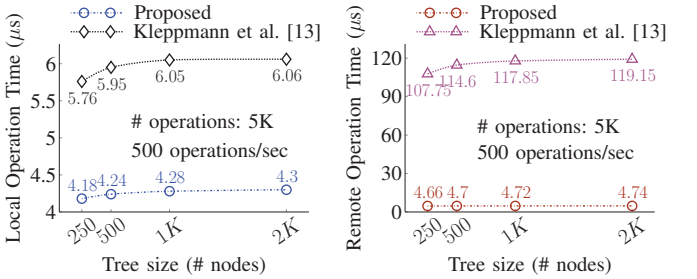
We ran the experiments seven times for each data point in the experiments. The first two runs were considered warm-up runs for the system to stabilize and hence ignored. Thus each data point in the plot was averaged over the remaining five times and across the different replicas. The synthetic workload used for the experiments consisted of insert, delete, and move operations. Initially, the tree was empty and based on the experiment, the number of nodes varied in the tree. The node is identified by *key* or *node id* which is an integer. Lamport's clock [17] was used to timestamp each operation.

Each replica generates and applies local operations, subsequently asynchronously propagating and receiving the operations to/from the other two replicas. Replica generates the move operation by selecting tree nodes uniformly at random from the tree size. Each replica generates $(\frac{1}{3})^{rd}$ of the total number of operations applied and receives $(\frac{2}{3})^{rd}$ of the operations from the other two replicas. When a replica receives a remote operation, it applies, and in case of any undo operation due to cycle, identifies the appropriate previous parent (for the node with the higher timestamp in the cycle). Once the appropriate parent is identified, the node is moved as a child and the undo operation is sent to other replicas as per the protocol. Note that our experimental workload is



(a) Local move operation apply time (b) Remote move operation apply time

Fig. 4: Average time to apply a move operation with varying number of operations.



(a) Local move operation apply time (b) Remote move operation apply time

Fig. 5: Average time to apply a move operation with varying tree size.

more conservative and contains more conflicts than the real-time workload. Further, move operations conflict only with other move or delete operations; they do not conflict with other operations (such as updating a value at a node in the tree or inserting a new subtree).

B. Results and Analysis

We performed two kinds of experiments. In Figure 4, we show the variation in performance when the number of tree size (# nodes) is fixed to 500 while the number of operations that a single replica is issuing per second is varying. The x-axis is the *operations per second* varied from 250 to 5000 while the y-axis is the average time taken to apply an operation (local or remote) at a replica. It should be noted that this does not represent the total time required by all operations, but rather the average time required to apply an operation. In Figure 5 and Figure 6, the experiment is a function of nodes in the tree to conflicting operations when operations are fixed to 15K (5K operations per replica) and 500 operations were issued per second on the three different geo-locations. Figure 5 shows the impact of tree size on the move operation performance. In contrast, Figure 6 shows the number of undo and redo operations in the proposed approach at different replicas.

Having explained the high-level overview of the experiments, we now go into the details. Figure 4 depicts the average time to apply a local and remote move operation. As illustrated in Figure 4(a), the average time to apply a local move operation at a replica is not significant in both the approaches as the number of operations increases. Interestingly, the average apply time for a local move operation drops as the number of operations increases. We believe this is due to compiler optimizations that sets in when functions are called repeatedly. Kleppmann et al. [13] also observed similar trends for local operations. However, the average 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 as shown in Figure 4(b). There is a significant performance gap for the average remote move operation apply time in both the approaches; this is because the number of compensation operations (undo/redo) by Kleppmann's approach is $\approx$200 undo and redo operations for every remote operation a replica receives while in our case it is only 1, only when there is a cycle (conflict).

As shown in Figure 4(b), Kleppmann's approach attains a maximum average 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 14.63$\times$ higher than the maximum time of 5.58μs at 250 operations by our proposed approach. It can be seen that the proposed approach achieves, on average, a speedup of 1.34$\times$ for a local move operation, while 14.6$\times$ to 68.19$\times$ speedup for a remote move operation over Kleppmann's approach. Hence, the proposed approach is much faster in applying remote operations, and the difference in time only increases with an increase in the rate of operations per second.

In Figure 5 and Figure 6, we fixed the number of local operations to 5K per replica (total 15K operations) and the operation interval to 10 milliseconds while varying the number of nodes in the tree from 250 to 2K. With growing size of the tree one might assume that the time required to complete a move operation must decrease as the number of conflicts decreases (as shown in Figure 6). However, it increases the search time; hence we can see the divergent tendencies. When comparing both techniques, the performance patterns in Figure 5 diverge from those in Figure 4.

Figure 5(a) and Figure 5(b) show that an average time to apply a move operation (local or remote) at a replica increases with increasing tree size due to increased search time. The proposed method improves the performance of local move operations by almost 1.4$\times$ over Kleppmann's approach. While the performance difference for the remote move operation for both the approaches as in Figure 5(b) is quite significant 24.43$\times$. This is due to the performance benefit of the proposed approach for the remote move operation.

In Kleppmann's approach, as the operation generation rate increases, the number of operations in flight also increases. As a result, the compensation cost will be substantial, requiring a large number of undo and redo operations. In contrast, even if the rate increases in the proposed approach, the compensation cost remains unchanged. The rationale for this performance improvement is explained next.

In Figure 6, a line chart depicts the average number of conflicts (undo and redo operations) at different replicas in the proposed approach. This experiment is performed to demonstrate the number of undos and redos operations performed by each replica and the system performance when the tree size is

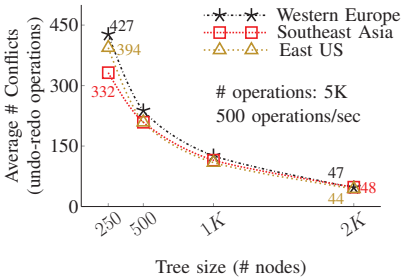


Fig. 6: Number of conflicts at different replicas in proposed approach with varying tree size.

changed. In Kleppmann's approach, the number of undos and redos per remote move operation is ≈200 [13], while the proposed approach needs just one compensation operation; hence we explicitly do not compare our approach with Kleppmann's approach in this plot. The maximum number of conflicts is ≈427 when the number of nodes in the tree is 250, but it drops to ≈47 when the number of nodes in the tree is 2K, as shown in the Figure 6. These numbers are much smaller than the number of undos and redos by Kleppmann's approach, which roughly equals 2M (million) for 10K remote move operations at a replica in the worst case. As a result, we can see that the proposed approach significantly improves the remote move operations apply time than Kleppmann's approach.

VII. Conclusion and Future Directions

We proposed a novel algorithm for efficient move operations on a replicated tree structure. The proposed technique ensures that replicas that have viewed and applied the same set of operations will eventually converge to the same state. Moreover, it does not require active cross-replica communication, making it highly accessible even during network partitions. We have followed a last write win scheme on globally unique timestamps. The proposed technique requires a single compensating operation to undo the effect of the cyclic operation. It achieves an average speedup of ≈68.19× over the state-of-the-art [13] approach. We have stored a constant number of the previous parents for every node. Identifying the optimal number of previous parents is left as future work. Implementing an efficient move operation on other replicated data structures could be an exciting area to explore. Also, performing operations on a range of elements in the list and tree CRDTs, or applying operations in a group from the same replica, are other potential directions to pursue.

References

[1] W. Vogels, "Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs? between consistency and availability," *Queue*, vol. 6, no. 6, p. 14–19, Oct. 2008.

[2] M. Roohitavafa, M. Demirbas, and S. Kulkarni, "Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks." in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS), pp. 184-193.*, ser. SRDS'17. IEEE, 2017.

[3] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[4] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond: How can applications be built on eventually consistent infrastructure given no guarantee of safety?" *Queue*, vol. 11, no. 3, p. 20–32, mar 2013.

[5] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, p. 40–44, jan 2009.

[6] S. Burckhardt, "Principles of eventual consistency," *Foundations and Trends in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, 2014. [Online]. Available: http://dx.doi.org/10.1561/2500000011

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

[8] 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.

[9] 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.

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

[11] 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.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[13] 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.

[14] 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.

[15] Figma, "Figma: the collaborative interface design tool," https://www.figma.com/, [Online; accessed 31-03-2022].

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

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[18] T. Jungnickel and T. Herb, "Simultaneous editing of json objects via operational transformation," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 812–815.

[19] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, p. 42–81, Mar. 2005.

[20] C. Baquero, P. S. Almeida, and A. Shoker, "Making operation-based crdts operation-based," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2014, pp. 126–140.

[21] S. Martin, P. Urso, and S. Weiss, "Scalable xml collaborative editing with undo," in *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*. Springer, 2010, pp. 507–514.

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

[23] M. Najafzadeh, "The analysis and co-design of weakly-consistent applications," Ph.D. dissertation, Université Pierre et Marie Curie-Paris VI, 2016.

[24] M. Najafzadeh, M. Shapiro, and P. Eugster, "Co-design and verification of an available file system," in *Verification, Model Checking, and Abstract Interpretation*. Cham: Springer International Publishing, 2018, pp. 358–381.

[25] V. Tao, M. Shapiro, and V. Rancurel, "Merging semantics for conflict updates in geo-distributed file systems," in *Proceedings of the 8th ACM International Systems and Storage Conference*, ser. SYSTOR '15. New York, NY, USA: ACM, 2015.

[26] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks." in *International Conference on Principles of Distributed Systems, pp. 17-32. Springer, Cham*, ser. OPODIS'14. Springer, Cham, 2014.

[27] "Go programming language," https://golang.org/, [Online; accessed 31-03-2022].

[28] "grpc: a high performance, open source universal rpc framework," https://grpc.io/, [Online; accessed 31-03-2022].