

Extending JSON CRDT with Move Operations

Liangrun Da

Technical University of Munich
liangrun.da@tum.de

Martin Kleppmann

Technical University of Munich
martin@kleppmann.com

Abstract

Conflict-Free Replicated Data Types (CRDTs) for JSON allow users to concurrently update a JSON document without introducing any conflicts and automatically merge the updates into a consistent state. However, moving a subtree in a map or reordering elements in a list for a JSON CRDT is considered challenging as an algorithm that is not carefully designed may introduce unexpected results such as duplicates or cycles. In this paper, we show how to extend Automerge, a CRDT library, by incorporating move operations and evaluating the performance of our algorithm.

1 Introduction

Automerge [1] is a Conflict-Free Replicated Data Type (CRDT) [12], which allows concurrent changes to data on different devices to be merged automatically without requiring any central server. It is utilized in the development of local-first software [5], which includes applications such as collaborative drawing, text editing, and more.

Automerge uses a *document* as its data model, which can be viewed as a JSON data type on local devices. A document can be accessed locally through operations such as *get*, *put*, and *delete* provided by Automerge library, resembling how we manipulate JSON in programming languages such as JavaScript. The modifications made are then replicated to other devices. However, those operations have limitations when it comes to fulfilling the requirements of certain applications. Many applications require reordering elements in a list, moving elements in a map, or even moving elements between lists and maps. For instance, when an application utilizes a document to represent a file system, moving a file from one location to another is a very common operation.

Although move operations are straightforward for an unreplicated JSON object, as they only require deletion and reinsertion, they become challenging when dealing with CRDTs. If several replicas concurrently delete and reinsert the same object, the merged result contains duplicates of the moved

object [3]. Another issue arises with cycles, as illustrated in Figure 1. In this example, each device individually executes a move operation without creating a cycle. However, when merging these operations, a cycle may appear if the algorithm does not take care to prevent the cycle.

Previous research has demonstrated the possibility of moving elements within lists and maps [3, 4, 7, 8], but the existing implementations of move operations are minimal research prototypes that are not integrated with fully-featured CRDT libraries such as Automerge. To fully support move operation in a JSON CRDT, we extend the algorithm to enable users to move elements from one location to any other location, even between a list and a map. In this paper, we show how to implement a move operation with the desired semantics in Automerge. Through performance experiments, we demonstrate the practical feasibility of the move operation. Our move operation provides the following guarantees:

- **No cycles:** The document will never be in a cyclic state at any point.
- **No duplicates:** If two concurrent move operations attempt to move the same element, only one move operation will take effect, preventing the occurrence of duplicate elements.

2 The Core Mechanics of Automerge

In Automerge, although a document is presented in the API as a JSON object, its internal representation uses a set of *operations* called *OpSet*. The set of operations can be exchanged between different peers, and two sets can be merged without generating any conflicts by taking their union.

When modifying a document, a new operation is added to the OpSet. Operations are never removed from the OpSet. Each operation has the following fields:

ID: Every operation in Automerge is assigned a unique ID, which is implemented as a Lamport Clock [6]. The total

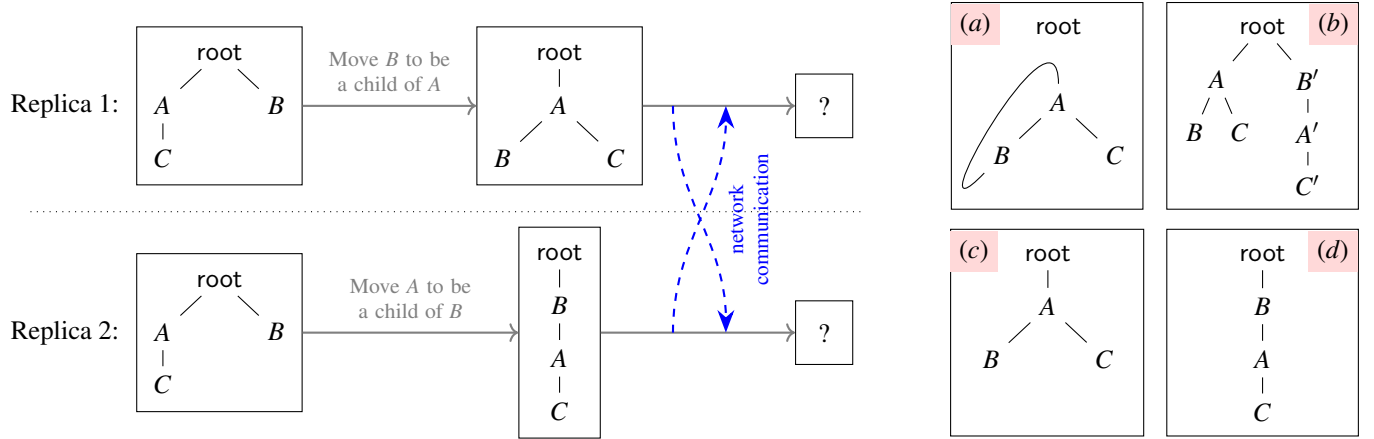


Figure 1: Initially, nodes *A* and *B* are siblings. Replica 1 moves *B* to be a child of *A*, while concurrently replica 2 moves *A* to be a child of *B*. Boxes (a) to (d) show possible outcomes after the replicas have communicated and merged their states. Figure from [4]

order of operation IDs is consistent with the causal order of operations.

Type: Every operation in Automerge has a type that signifies the action it performs. Although Automerge supports multiple operation types, this paper will focus on four key types: *make*, *put*, *delete*, and *move*. These four types provide the necessary functionality to implement a JSON document with standard operations as well as the ability to perform move operations.

Object ID: Every operation has an object ID that identifies the list or map object within the JSON document which the operation modifies. In Automerge, a list or map object is created using a *make* operation. The ID of this *make* operation subsequently serves as the unique identifier for the object it created.

Key: Every operation has a *key* that indicates the position within the object at which the operation operates. In a map object, the key represents the property name associated with the operation. In a list object, the key represents the ID of the preceding element.

Value: This field serves to hold the value associated with the operation. For a *make* operation, it contains the specific type (map or list type), whereas, for a *put* operation, it stores the value that the operation puts.

Predecessors: If an element is deleted, overwritten, or moved by a newly generated operation, the operation associated with that element becomes a predecessor of the new operation, and the operation becomes a successor of the overwritten operation. An operation is invisible if it has one or more successors. The value of an invisible operation is not returned when querying the current state of the document, but it might be returned by queries to a past version of the document.

Figure 2 illustrates the relationship between a JSON document and an OpSet. A user’s query for the document is translated into a query for the operations within the OpSet.

As an example, the query for “document.A” is translated into a search for the operation where the *Key* is equal to *A*, and the *ObjectID* is equal to $\langle 0, 0 \rangle$, the ID of the root object.

3 The Moving Algorithm

The moving algorithm consists of two components. Firstly, it involves generating a move operation to move the target element. Secondly, it identifies an operation as invalid if it violates the guarantees of “no cycle” or “no duplicates”.

3.1 Generating Move Operations

When the user moves an element from one location to another, a move operation is generated and added to the OpSet. This move operation identifies the element being moved and where it is being moved to. The destination of the move is determined by the *ObjectID* and *Key* fields of the move operation. The ID of a prior *put* operation or *make* operation identifies the element being moved. A move operation has a *MoveID* field where this ID is stored.

Figure 3 shows an example of moving the scalar value *d* to be an element of list *B*. The operation $\langle 7, 0 \rangle$ is not visible anymore, as it has a successor move operation.

From the user’s perspective, the move operation is generated by invoking a function named *move*:

```
move(sourceObject, destinationObject,
     sourceKey, destinationKey)
```

The function searches the operation being moved by the source object and key. If the source is a multi-value register and does have multiple concurrent values, the function will take the value with the greatest ID as the element to be moved. The other values will be deleted from the document.

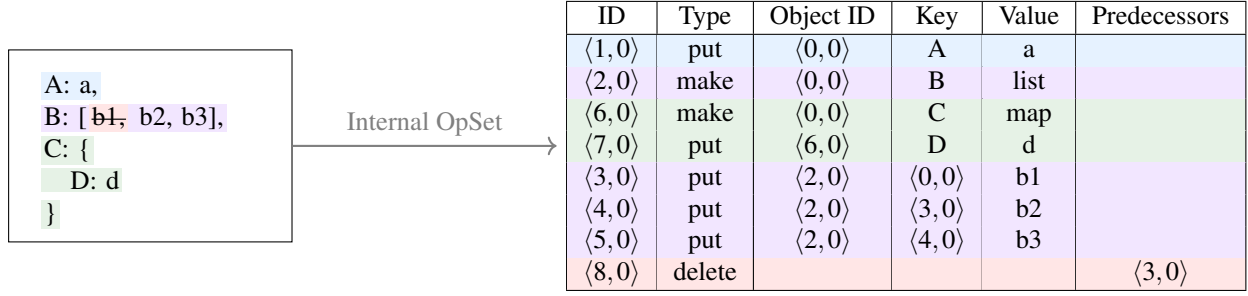


Figure 2: An example JSON document with its internal OpSet

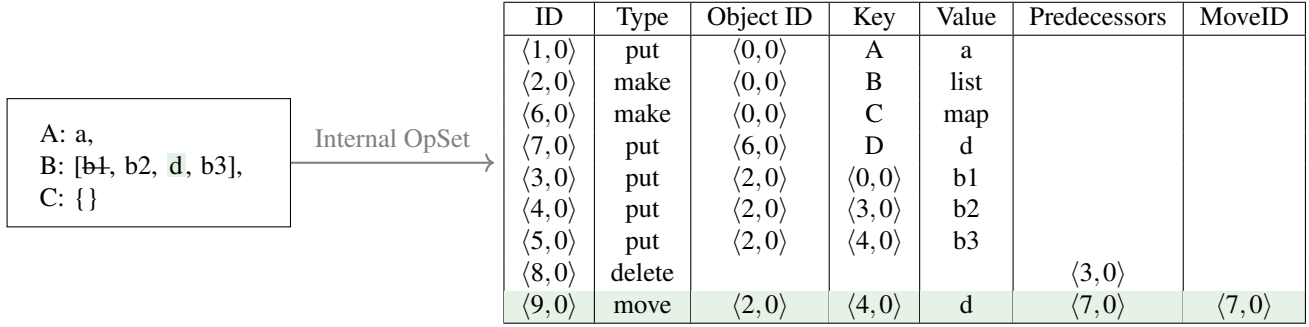


Figure 3: Moving d to be an element of list B

Consecutive move operations are possible, as the old and new move operations can share the same MoveID. If the user moves d again to a new location, the second move operation will have the same MoveID $\langle 7, 0 \rangle$.

To handle the situation where multiple concurrent move operations are moving the same element, we define the move operation with the largest ID as the winner among those operations. In other words, only the winner is considered as a successful move, while the others behave as if they were not performed. This rule prevents duplicating the moved element through concurrent moves.

3.2 Validity of Move Operations

We define a move operation to be valid if and only if there is no concurrent move operation with a greater ID that moves the same element and it does not introduce any cycles. If a move operation is invalid, the move operation itself is invisible and its predecessors will not treat the move operation as a successor.

The challenge is to ensure a consistent decision among replicas regarding the validity of the same operation. Consider Figure 1, where Replica 1 moves B to be a child of A and then receives the change from Replica 2, while Replica 2 moves A to be a child of B and then receives the change from Replica 1. If we check the validity at the time of applying operations, Replica 1 would consider the operation of moving

A to be a child of B as invalid, while Replica 2 would say the operation of moving B to be a child of A is invalid, resulting in inconsistent document states on two replicas.

To ensure consistent decisions on the validity of operations, we can apply operations in ascending ID order. Whenever an operation is inserted into OpSet, all the operations in the OpSet are reapplied, and the validity of operations is updated accordingly. We first show a simple but inefficient algorithm for computing validity and present an optimized algorithm in Section 3.3.

Algorithm 1 provides details about the process of computing operations' validity when a new operation is received from a peer. We maintain a map named tree to keep track of the parent-child relationship between objects. All objects within a list are considered children of the list object, and the same applies to map objects. When an object is deleted from the tree, we record this in the map by setting the deleted object's parent to *null*. A deletion can be thought of as similar to a move that moves the deleted object to a "trash" tree that is separate from the visible document tree.

Additionally, we maintain another map named winner, whose key is the ID of an element being moved, and the associated value is the greatest ID among operations that move this element.

By reapplying the operations, we update the tree to reflect any changes in the parent-child relationship in the document. However, when a put operation is encountered, the scalar

Algorithm 1 A naive approach for updating validity of operations

Input: ops - operations in the OpSet in ascending ID order**Output:** valid - a map from operation ID to its validity

```
1: procedure UPDATEVALIDITY(ops)
2:   tree  $\leftarrow \{\}$   $\triangleright$  a map from child ID to parent ID
3:   winners  $\leftarrow \{\}$   $\triangleright$  a map from object ID to winner ID
4:   valid  $\leftarrow \{\}$ 
5:   for op in ops do  $\triangleright$  in order of ascending op.ID
6:     for pred in op.Predecessors do
7:       if pred.Type == move and valid[pred.ID]
      then
8:         tree[pred.MoveID]  $\leftarrow$  null
9:         else if pred.Type == make then
10:          tree[pred.ID]  $\leftarrow$  null
11:          if op.Type == make then
12:            tree[op.ID]  $\leftarrow$  op.ObjectID
13:          else if op.Type == move then
14:            mid  $\leftarrow$  op.MoveID
15:            oid  $\leftarrow$  op.ObjectID
```

```
16:   if ISANCESTOR(tree, oid, mid) then
17:     valid[op.ID]  $\leftarrow$  false
18:     continue
19:   valid[op.ID]  $\leftarrow$  true
20:   tree[mid]  $\leftarrow$  oid
21:   prevWinner  $\leftarrow$  winners[mid]
22:   if prevWinner != null then
23:     valid[prevWinner]  $\leftarrow$  false
24:     winners[mid]  $\leftarrow$  op.ID
25:   return valid
26: function ISANCESTOR(tree, node, ancestor)
27:   while true do
28:     if node == ancestor then
29:       return true
30:     if node == null || node == root then
31:       return false
32:     node  $\leftarrow$  tree[node]
```

value is not added to the tree, as put operations cannot introduce cycles. Line 6 to line 10 shifts the objects that are deleted and overwritten from the document tree to the trash tree, by updating their parent to null. Furthermore, from line 11 to line 24 we update the validity by checking for cycles and concurrent moves.

3.3 Performance Optimization

Algorithm 1 can ensure the consistency of the move operations by applying them in the same order on all replicas. However, reapplying the entire set of operations in the OpSet is very inefficient, especially when the OpSet contains tens of thousands of operations.

To optimize the algorithm, we can avoid executing the parts that repeat the previous invocation of *UpdateValidity*. Specifically, during a particular invocation, we assume that the smallest operation ID in the OpSet is s , and the ID of the newly added operation is n . In Algorithm 1, the execution of the loop for operations within the ID range $[s, n)$ is the same as the previous invocation. In other words, the validity of operations with IDs in the range $[s, n)$ are not affected by the newly added operation.

During real-time collaboration among devices that are online, when an operation is generated on one device, it is propagated to other peers quickly. This implies that when a device receives a new operation from a peer, only a few operations in the OpSet with greater IDs are affected. Consequently, we only need to update the validity of those affected operations. Offline editing can lead to greater divergence, but for now, we concentrate on the case of low-latency collaboration.

Algorithm 2 describes an optimized version of Algorithm 1.

To insert a new operation with ID n into OpSet, our algorithm initiates by reverting the parent-child relationship to a prior state, specifically to a point where only operations with IDs lower than n are applied. Subsequently, we apply the new operation, followed by reapplying the operations with IDs greater than n . This sequence of steps is denoted as the *Restore-Apply-Reapply (RAR)* procedure.

To be able to restore the parent-child relationship to a past state, we store old parents of overwritten, deleted, and moved objects in parents, in which we maintain a map from object IDs to the old parent IDs for each operation.

Additionally, to restore the validity status of operations with a lower ID, we maintain a stack of move operations for every object that is moved. Operations in a stack are sorted by ID, leading to the top of the stack being the winner among the concurrent move operations of the same element. To restore to a past state, all the move operations with IDs greater than that of the past operation are popped.

Line 10 to line 17 in Algorithm 2 describes the process of restoring, in which we revert all the effects of the operations greater than that of the newly inserted operation on T . From lines 18 to 48, we update T to reflect any changes in the parent-child relationship by applying the newly inserted operation and reapplying the following operations. The validity is updated by detecting cycles and concurrent move operations during applying and reapplying operations similarly to Algorithm 1. Additionally, we store the old parent of objects, ensuring the ability to restore to any past state.

Algorithm 2 The Restore-Apply-Reapply approach for updating validity of operations

```
1: procedure UPDATEVALIDITY
2:   ops  $\leftarrow \square$  ▷ operations in ascending ID order
3:   tree  $\leftarrow \{\}$  ▷ a map from child ID to its parent ID
4:   moves  $\leftarrow \{\}$  ▷ a map from element ID to a stack of move operation IDs for moving this element
5:   parents  $\leftarrow \square$  ▷ a list, where each element is a map, from object ID to its parent ID
6:   valid  $\leftarrow \{\}$ 
7:   while whenever receives an operation o do
8:     Insert o into ops at index i such that all operations at indexes > i have an ID greater than o.ID, and all operations at
     indexes < i have an ID less than o.ID
9:     Insert a new map into parents at index i
10:    for k  $\leftarrow$  |ops| - 1 to i + 1 do ▷ Revert
11:      for (object, location) in parents[k] do
12:        tree[object] = location
13:      if ops[k].Type is move and valid[ops[k].ID] then
14:        moves[ops[k].MoveID].pop()
15:        prevMove  $\leftarrow$  moves[ops[k].MoveID].peek()
16:        if prevMove  $\neq$  null then:
17:          valid[prevMove]  $\leftarrow$  true
18:    for k  $\leftarrow$  i to |ops| - 1 do ▷ Apply and Reapply
19:      op  $\leftarrow$  ops[k]
20:      parent  $\leftarrow \{\}$ 
21:      prevParent  $\leftarrow$  null
22:      if op.Type is make then
23:        tree[op.ID]  $\leftarrow$  op.ObjectID
24:        parent[op.ID]  $\leftarrow$  null
25:      else if op.Type is move then
26:        if op.Value == null then ▷ if it moves a map or list object
27:          if ISANCESTOR(tree, op.ObjectID, op.MoveID) then
28:            valid[op.ID]  $\leftarrow$  false
29:            continue
30:          prevParent = tree[op.MoveID]
31:          tree[op.MoveID]  $\leftarrow$  op.ObjectID
32:          valid[op.ID]  $\leftarrow$  true
33:          if moves[op.MoveID] == null then
34:            moves[op.MoveID]  $\leftarrow$  new stack
35:          prevMove  $\leftarrow$  moves[op.MoveID].peek()
36:          if prevMove  $\neq$  null then:
37:            valid[prevMove]  $\leftarrow$  false
38:          moves[op.MoveID].push(op.ID)
39:      for pred in op.Predecessors do
40:        if pred.Type is move and valid[pred.ID] then
41:          parent[pred.MoveID]  $\leftarrow$  tree[pred.MoveID]
42:          tree[pred.MoveID]  $\leftarrow$  null
43:        else if pred.Type == make then
44:          parent[pred.ID]  $\leftarrow$  tree[pred.ID]
45:          tree[pred.ID]  $\leftarrow$  null
46:      if prevParent  $\neq$  null then
47:        parent[op.MoveID]  $\leftarrow$  prevParent
48:      parents[k]  $\leftarrow$  parent
```

3.4 Further Optimization

3.4.1 Bulk Updating

In Automerge, to mutate a document, users can start a *transaction* and perform a series of operations. Once a transaction is committed, the operations within the transaction are consolidated into a single change, which can then be sent to other devices. Instead of calling `UpdateValidity` for each operation, it is more efficient to collectively apply all operations within the change at once, and thereby amortise the cost of restoring and reapplying over many operations. Moreover, there exists the option to delay applying changes until users perform actual query operations. This allows for the aggregation of operations from multiple changes, facilitating a bulk updating of these operations all at once.

3.4.2 Lifecycle Tracking

In the process of Restore-Apply-Reapply, objects shift between the tree and the trash. As each operation's ID is a logical clock, we can create a sequence of IDs that trace the chronological progression of these shifts for each object. This sequence of IDs is referred to as the *LifecycleList* of an object.

A LifecycleList can be split into two sublists:

1. *PresentList*: a list of operation IDs in ascending ID order, where each operation creates or moves the object.
2. *TrashList*: a list of operation IDs in ascending ID order, where each operation overwrites or deletes the object.

Upon receiving an operation, the ID of the new operation is inserted into the TrashList of objects it deletes or overwrites, as well as the PresentList of objects it creates or moves.

By maintaining the lifecycle of each object, there is no longer a necessity to revert and reapply make, delete and put operations. These operations merely shift objects back and forth between the tree and the trash, without changing their locations within the tree. By utilizing a list detailing an object's lifecycle, we can deduce its existence - whether within the tree or in the trash - without the need to directly shift the objects between the two.

By adopting lifecycle tracking, we can ensure that the algorithm's complexity is only related to the number of move operations in the document, rather than the overall size of the OpSet. Furthermore, applications that do not use move operations incur only a constant overhead.

3.4.3 Checkpoint

For offline applications, the states of documents could become significantly divergent. A device might receive severely delayed operations, which means numerous operations with IDs greater than that of the new operation are affected. A run

of Algorithm 2 is slow, as those affected operations need to be reverted and reapplied.

To address this issue, checkpoint files can be used to reduce the number of operations to be reverted. When applying operations, snapshots of T are created for every n operation and persisted in checkpoint files, where n could be hundreds or thousands, depending on the application. To restore to a particular past state, we can locate the nearest checkpoint and reach the state by either reapplying or reverting fewer than $n/2$ operations.

4 Evaluation

In this section, we demonstrate the feasibility of the algorithm through performance experiments. We study (1) the complexity and performance of convergence when two actors generate move operations concurrently; (2) the overhead caused by adding support for move operations; (3) the comparison with an unreplicated JSON object. Additionally, we provide a correctness evaluation of the algorithm.

We deployed the experiments on AWS c5.large instances, with each instance having 4vCPU and 2GB RAM. The first two experiments were conducted on two machines within a local area network, as network latency had no impact on the results. The third experiment was conducted on three AWS instances in different regions across the continents, namely California (us-west-1), Tokyo (ap-northeast-1) and London (eu-west-2).

We implemented the move algorithm in a standalone Automerge prototype, which is a simplified version of Automerge. We haven't integrated the move algorithm into the rust implementation of Automerge [1]. The source code is available on GitHub¹.

4.1 Convergence Complexity and Performance

We experimented to measure the time it takes for two divergent actors to converge. In this experiment, both actors start with identical documents containing 100 map objects. They generate N move operations concurrently, and then send the operations to each other. A move operation is generated by choosing a random object to be moved, and a random object as the destination. As N grows larger, the actors diverge further.

To apply a remote move operation, the actors must revert and reapply the operations with greater IDs. Without bulk updating mentioned in Section 3.4.1, each actor needs to reapply operations at most $N(N+1)/2$ times. It takes $O(d)$ time to reapply a move operation, where d is the depth of the destination object. Therefore, the time complexity for achieving convergence is $O(N^2d)$. However, with bulk updating, the UpdateValidity procedure is executed only once for all newly

¹<https://github.com/LiangrunDa/AutomergetWithMove>

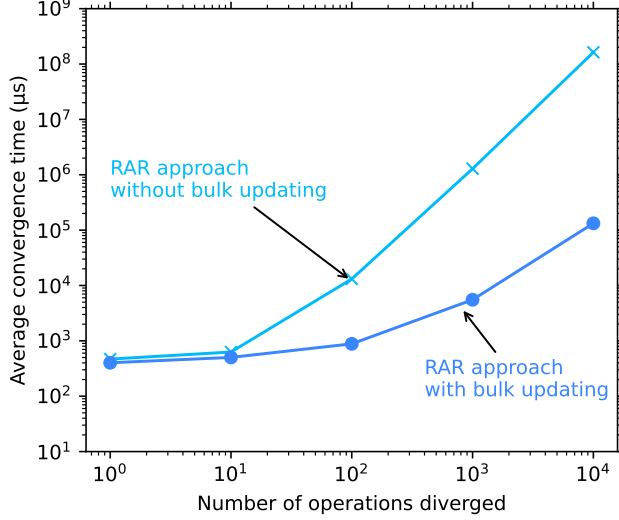


Figure 4: Average convergence time of two actors that diverge by move operations

added remote operations. Through this amortization of costs, the time complexity for adding a remote operation is reduced to $O(d)$, and the convergence complexity is subsequently reduced to $O(Nd)$.

Figure 4 shows the result of the experiment, which aligns well with our complexity analysis. With bulk updating, when $N = 100$, it takes about 1 ms to converge, which is acceptable for most real-time collaborative applications. Regarding the offline applications, the number of operations diverged is uncontrollable, as the actors can be offline for a long time. However linear convergence complexity is still acceptable, and it is the best we can achieve for adding N remote operations.

4.2 Overhead Caused by Move Support

Even when an application does not use any move operations, our algorithm needs to perform additional work on operations that modify the parent-child relationship of objects by creating, deleting or overwriting objects. To quantify this overhead, we measure the time to achieve convergence on our implementation with support for move operations enabled, and compare it to a version of the same implementation with support for move operations disabled. Initially, both actors have identical documents containing 100 map objects. They then concurrently generate and exchange a set of N make operations.

With move operations disabled, the time complexity of achieving convergence is $O(N)$, as we don't need to revert and reapply any operations. However, when the move support is enabled without lifecycle tracking and bulk updating, the time complexity of achieving convergence is $O(N^2)$, as the

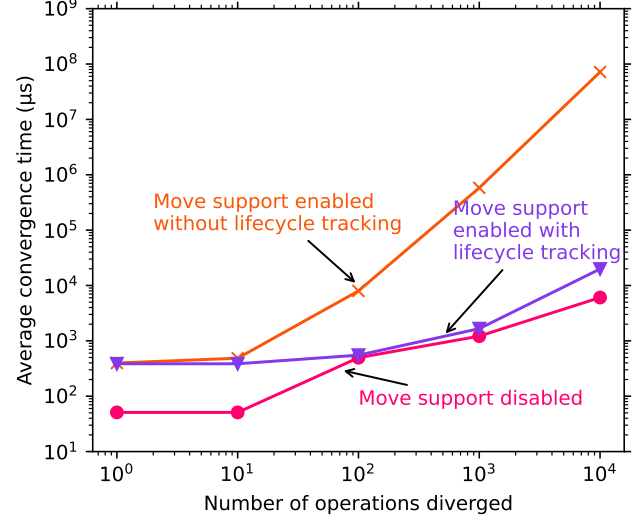


Figure 5: Average convergence time of two actors that diverge by non-move operations. The disabling of the move extension is achieved by turning off the UpdateValidity procedure

algorithm needs to reapply all the operations with greater IDs, even when there are no move operations necessitating validity updates. When lifecycle tracking is utilized, the time complexity of convergence is reduced back to $O(N)$, as the algorithm only needs to revert and reapply the move operations.

Figure 5 shows the results. Without lifecycle tracking, the impact of move support on system overhead becomes evident when $N > 1000$, and the overhead is proportional to N^2 . For applications that don't rely heavily on move operations, the overhead can't be overlooked. However, with lifecycle tracking, the overhead is reduced to a constant, as the difference between the curve of move support disabled and the curve of lifecycle tracking enabled is negligible.

Another noteworthy advantage of lifecycle tracking is its efficient handling of scenarios where two actors converge with unbalanced concurrent non-move operations. For instance, if one actor generates N non-move operations while the other actor generates only one non-move operation, the convergence complexity is $O(1)$ for the actor receiving only one remote operation from its peer.

4.3 Comparison to Unreplicated JSON Object

In this experiment, we compare the move operation performance of the JSON CRDT to an unreplicated JSON object.

To put the JSON CRDT to the test, we set up three AWS instances located in different regions across the continents: us-west-1, ap-northeast-1, and eu-west-2. These instances communicate with one another over the internet, with an average network latency of 169 ms between them. The three actors start with identical documents containing 100 map ob-

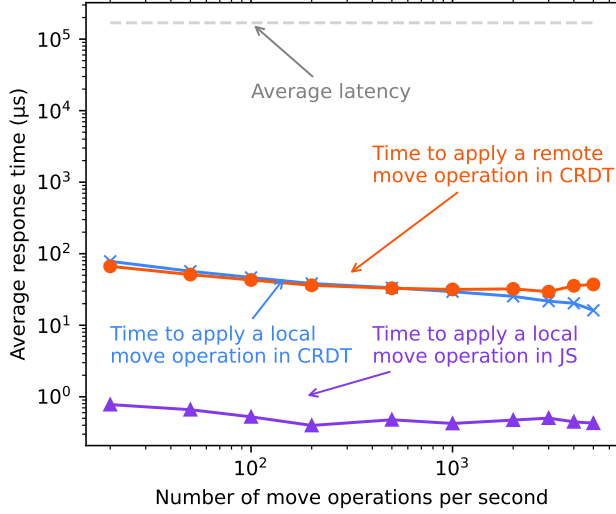


Figure 6: Average response time of move operation on JSON CRDT and unreplicated JSON object

jects. They then concurrently generate N move operations per second and send the operations to each other immediately. We measure the time it takes for the instances to apply both local and remote operations.

The unreplicated JSON object is implemented using a plain JavaScript object. It does not support concurrent operations so we only tested its performance with local operations. The actor starts with an object containing 100 map objects. It then randomly moves N objects from one location to another per second. We measure the time required to execute a local move in JavaScript.

The results, shown in Figure 6, indicate that it takes less than $1 \mu s$ for the unreplicated JSON object to apply a local move operation, whereas the JSON CRDT takes approximately $100 \mu s$ to perform both local and remote move operations. Although there is a noticeable cost associated with using the JSON CRDT, it remains within an acceptable range in the order of microseconds. As a result, the users pay a $100 \mu s$ overhead for replicating the JSON object and gain the benefits of real-time and offline collaboration.

4.4 Correctness Testing

We run the system in a real environment, and the correctness of the system is checked by randomly generating operations on different actors to see if the actors can converge to the same state. This approach is inspired by Jepsen [2], a framework for distributed systems verification.

With this approach, we found several bugs during the design of the move algorithm. Those bugs were caused by not taking care of the corner cases with combinations of inputs that are rarely encountered during normal execution. There-

fore, even if this approach cannot prove the correctness of the system, it can uncover some subtle corner cases and provide confidence in correctness.

5 Related Work

5.1 CRDTs for Trees

Many CRDTs designed for trees have been put forth, and the following represent a selection that includes support for move operations.

- Previous work by Nair et al. [7] proposed a conflict-free replicated tree type with move operations. They categorized these moves into up-moves (towards the root) and down-moves (away from the root). The study formally verifies the safety of concurrent up-moves while resolving cycle-causing problems arising from other move operations through the selection of a definitive winner among concurrent moves.
- Najafzadeh et al. [8] proposed a fully asynchronous file system that supports concurrent move operations. They replace the move operation with copy-delete operations if a cycle occurs, leading to duplication of directories. They also proposed a mostly asynchronous file system that uses locks to coordinate concurrent move operations but this approach is not available under network partition.
- Kleppmann et al. [4] proposed an undo-do-redo algorithm to resolve the cycles introduced by concurrent move operations for replicated trees. It is similar to our approach used for detecting cycles in CRDTs for JSON, but we change the name to Revert-Apply-Reapply to avoid confusion with a user-facing undo feature.

The previous tree CRDTs focus on managing the parent-child relationship, without integrating the map or list CRDT data types that occur in JSON trees like Automerge. In contrast, our algorithm handles more complex features, such as overwriting in map objects, moving multi-value registers within map and list objects, and reordering elements within list objects. These features make the algorithm significantly more complicated. Furthermore, we introduce the concept of lifecycle tracking to reduce the overhead of the algorithm, which is not considered in previous work.

5.2 CRDTs for Lists

There are many CRDTs for lists, such as Treedoc [11], WOOT [10], Logoot [13], and LSEQ [9]. However, none of them support move operations.

Kleppmann introduced an algorithm to extend existing List CRDTs with move operations [3]. The algorithm uses an LWW register for each element to track the location of the

element. We incorporate the algorithm into the Revert-Apply-Reapply procedure by tracking concurrent move operations that move the same element and selecting the move operation with the greatest ID as the winner.

6 Conclusions

In this paper, we present an implementation of move operations in a JSON CRDT. We introduce the Revert-Apply-Reapply procedure, to handle move operations efficiently and guarantee correctness. The procedure is optimized by utilizing further optimizations namely bulk updating, and lifecycle tracking.

Performance experiments demonstrate the practical feasibility of the move operation, even in scenarios with high levels of concurrent operations. The algorithm’s overhead is shown to be within an acceptable range for real-time collaborative applications. Additionally, the paper provides a correctness evaluation through random testing, ensuring that the algorithm converges to the same state across different actors.

Overall, we provide a comprehensive overview of implementing move operations in JSON CRDTs, addressing both theoretical and practical aspects, and demonstrating its feasibility.

References

- [1] Automerge CRDT. <https://automerge.org>. Accessed: 2023-07-26.
- [2] Distributed systems safety research. <https://jepesen.io>. Accessed: 2023-09-24.
- [3] Martin Kleppmann. Moving elements in list CRDTs. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–6, 2020.
- [4] Martin Kleppmann, Dominic P Mulligan, Victor BF Gomes, and Alastair R Beresford. A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1711–1724, 2021.
- [5] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software(Onward!)*, pages 154–178, 2019.
- [6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.
- [7] Sreeja Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, and Marc Shapiro. A coordination-free, convergent, and safe replicated tree. *arXiv preprint arXiv:2103.04828*, 2021.
- [8] Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster. Co-design and verification of an available file system. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 358–381. Springer, 2017.
- [9] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. Lseq: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM symposium on Document engineering*, pages 37–46, 2013.
- [10] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 259–268, 2006.
- [11] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403. IEEE, 2009.
- [12] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.
- [13] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 404–412. IEEE, 2009.