Extending JSON CRDTs with move operations

Liangrun Da, Martin Kleppmann

April, 2024

Contents

1. Introduction

2. Algorithm

3. Evaluation

Introduction

Why move?

It is everywhere!

- In distributed file systems, we move a directory from one place to another
- In collaborative to-do lists, we reorder tasks
- In collaborative drawing tools, we move layers up and down
- ...

Why is it hard?

1. Concurrent move operations might cause duplicates and cycles
2. Interactions between concurrent move and non-move operations might cause unexpected result and even inconsistency.

Duplicate

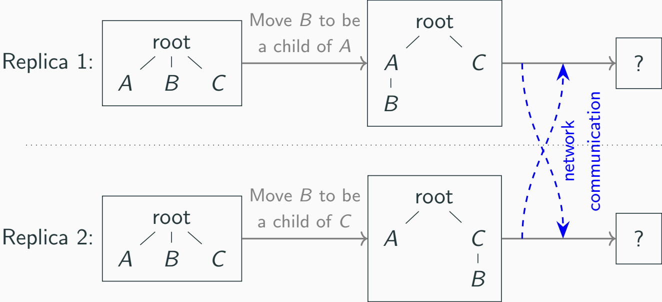
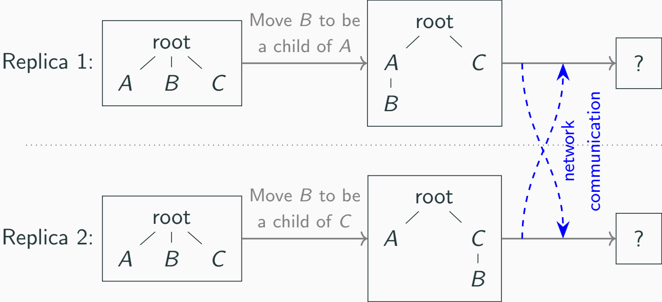


Figure 1: Concurrent moving the same element might cause duplicate nodes.

Duplicate



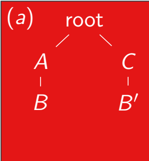
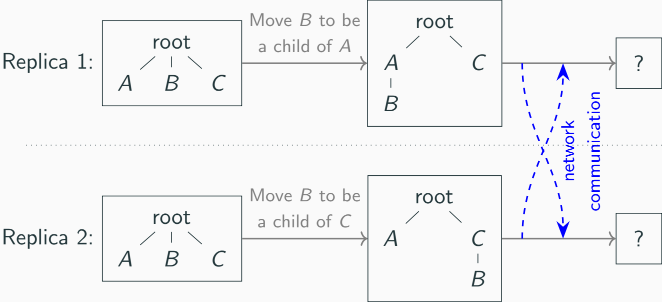


Figure 1: Concurrent moving the same element might cause duplicate nodes.

Duplicate



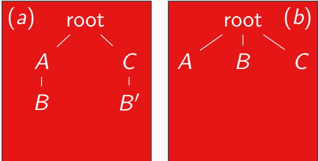
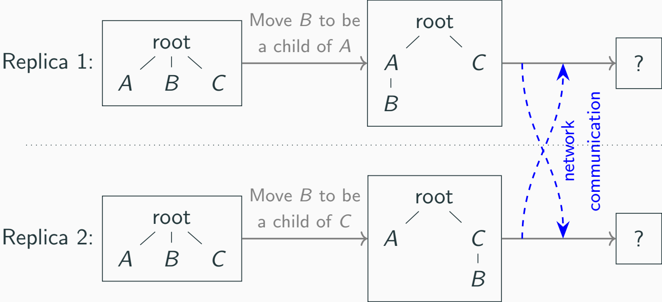


Figure 1: Concurrent moving the same element might cause duplicate nodes.

Duplicate



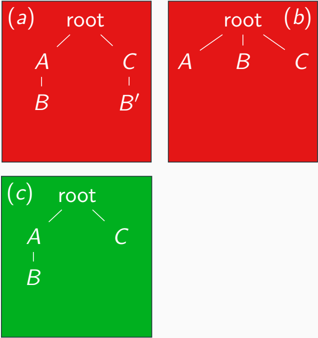
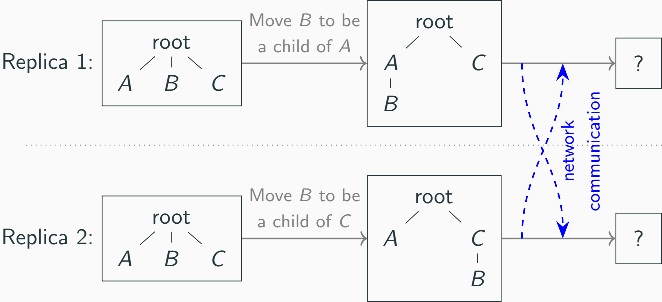


Figure 1: Concurrent moving the same element might cause duplicate nodes.

Duplicate



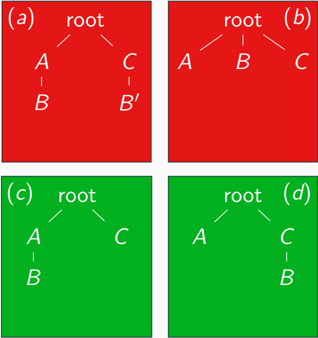


Figure 1: Concurrent moving the same element might cause duplicate nodes.

Cycle

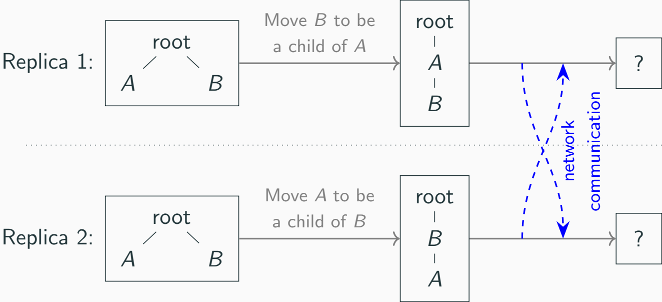
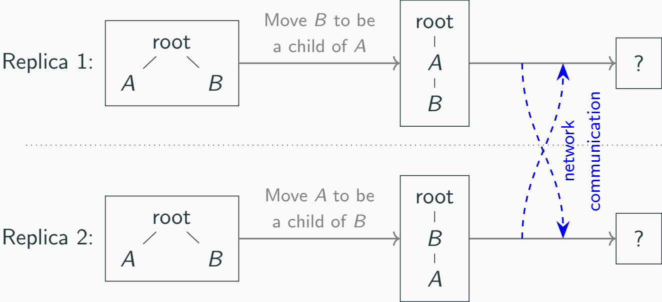


Figure 2: Concurrent moves might cause cycles.

Cycle



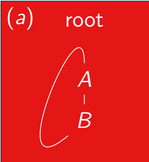
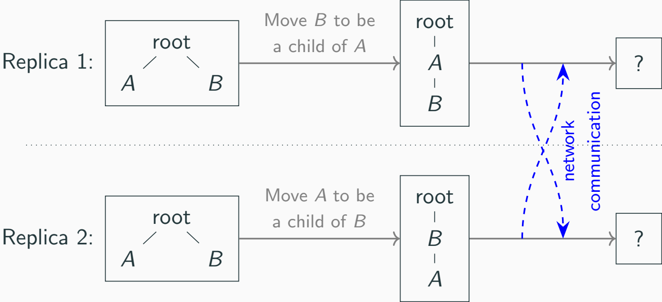


Figure 2: Concurrent moves might cause cycles.

Cycle



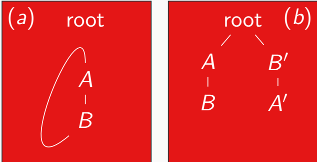
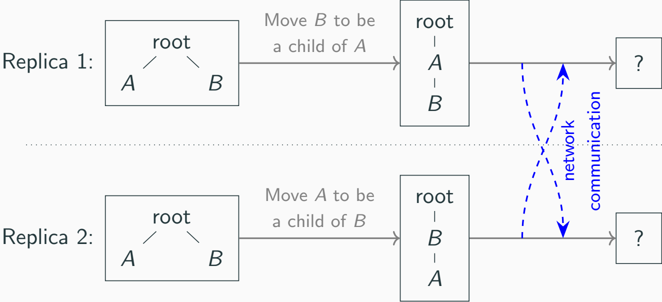


Figure 2: Concurrent moves might cause cycles.

Cycle



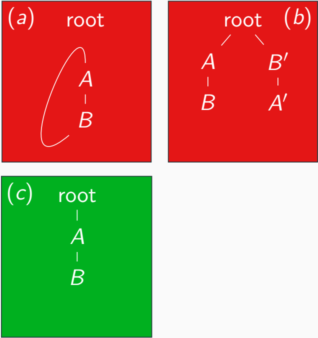
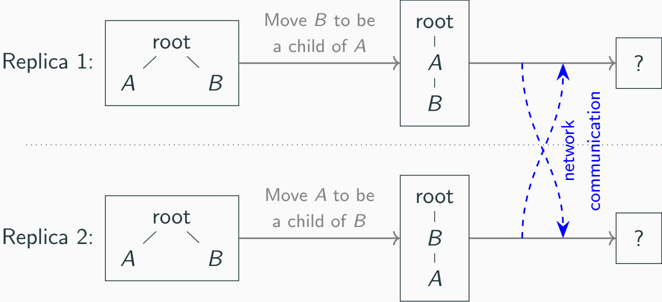


Figure 2: Concurrent moves might cause cycles.

Cycle



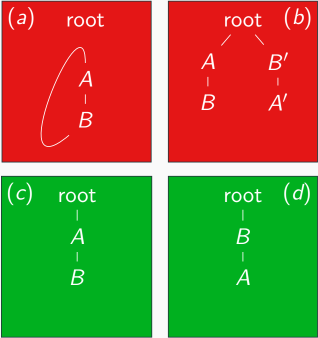


Figure 2: Concurrent moves might cause cycles.

Delete operation might cancel a cycle

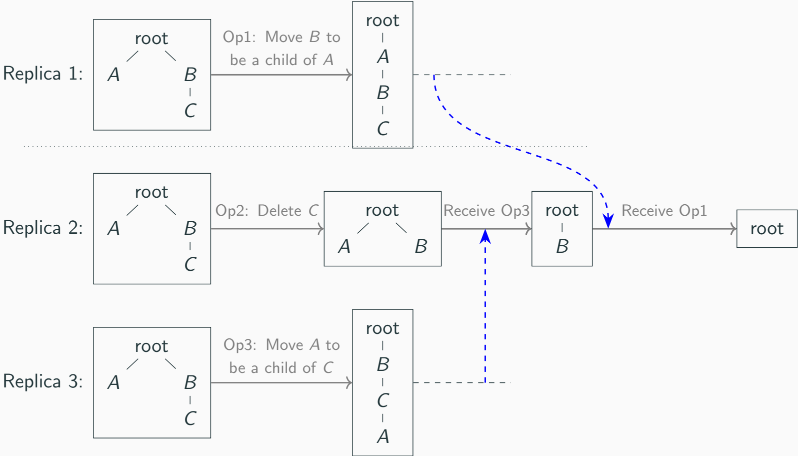


Figure 3: With a concurrent delete operation, two move operations no longer form a cycle and can be both executed.

Overwriting a key that is moved

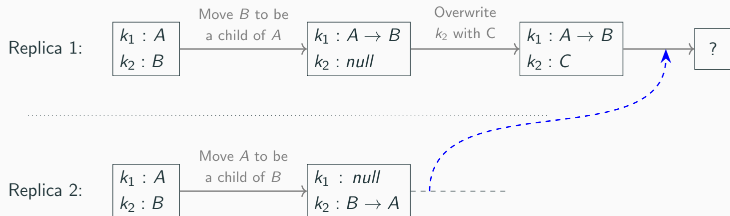


Figure 4: C overwrites k_2 where the object B is moved, it might lose value if the move operation is considered invalid later.

Overwriting a key that is moved

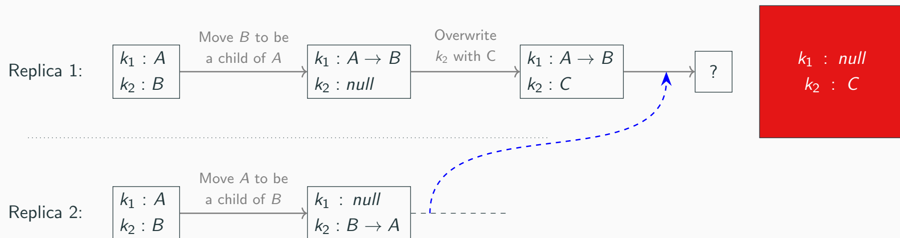


Figure 4: C overwrites k_2 where the object B is moved, it might lose value if the move operation is considered invalid later.

Overwriting a key that is moved

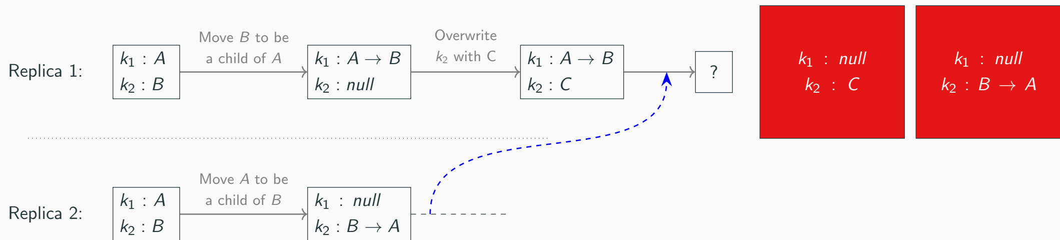


Figure 4: C overwrites k_2 where the object B is moved, it might lose value if the move operation is considered invalid later.

Overwriting a key that is moved

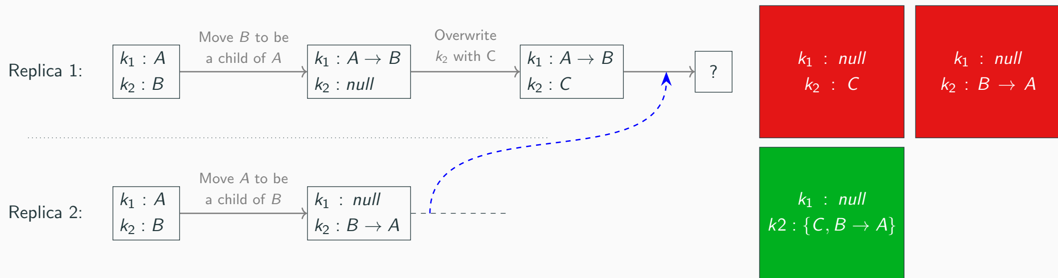


Figure 4: C overwrites k_2 where the object B is moved, it might lose value if the move operation is considered invalid later.

Overwriting a key that is moved

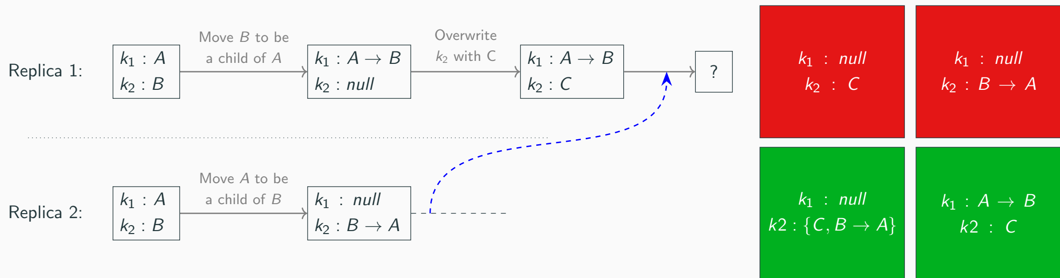
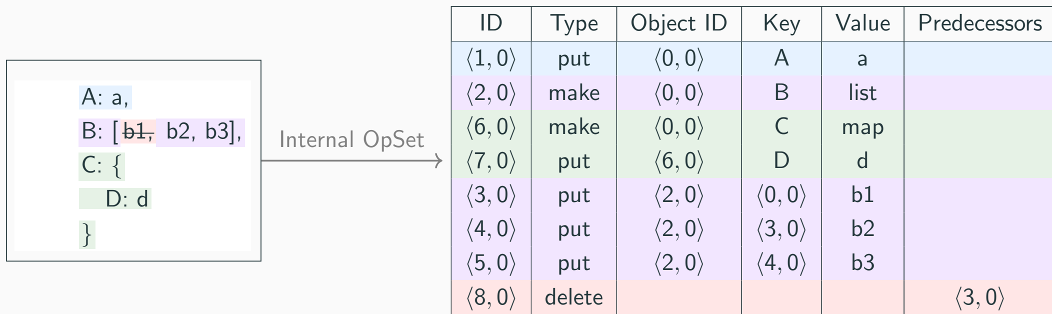


Figure 4: C overwrites k_2 where the object B is moved, it might lose value if the move operation is considered invalid later.

Algorithm

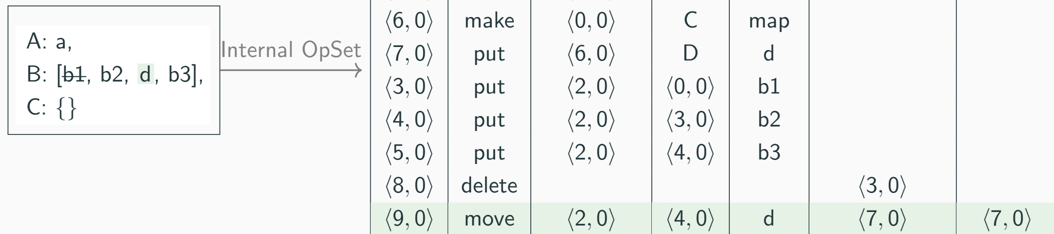
Automerge: Example JSON Document



ID	Type	Object ID	Key	Value	Predecessors
$\langle 1, 0\rangle$	put	$\langle 0, 0\rangle$	A	a	
$\langle 2, 0\rangle$	make	$\langle 0, 0\rangle$	B	list	
$\langle 6, 0\rangle$	make	$\langle 0, 0\rangle$	C	map	
$\langle 7, 0\rangle$	put	$\langle 6, 0\rangle$	D	d	
$\langle 3, 0\rangle$	put	$\langle 2, 0\rangle$	$\langle 0, 0\rangle$	b1	
$\langle 4, 0\rangle$	put	$\langle 2, 0\rangle$	$\langle 3, 0\rangle$	b2	
$\langle 5, 0\rangle$	put	$\langle 2, 0\rangle$	$\langle 4, 0\rangle$	b3	
$\langle 8, 0\rangle$	delete				$\langle 3, 0\rangle$

Figure 5: An example JSON document with its internal OpSet

Generating Move Operations



ID	Type	Object ID	Key	Value	Predecessors	MoveID
$\langle 1,0\rangle$	put	$\langle 0,0\rangle$	A	a		
$\langle 2,0\rangle$	make	$\langle 0,0\rangle$	B	list		
$\langle 6,0\rangle$	make	$\langle 0,0\rangle$	C	map		
$\langle 7,0\rangle$	put	$\langle 6,0\rangle$	D	d		
$\langle 3,0\rangle$	put	$\langle 2,0\rangle$	$\langle 0,0\rangle$	b1		
$\langle 4,0\rangle$	put	$\langle 2,0\rangle$	$\langle 3,0\rangle$	b2		
$\langle 5,0\rangle$	put	$\langle 2,0\rangle$	$\langle 4,0\rangle$	b3		
$\langle 8,0\rangle$	delete				$\langle 3,0\rangle$	
$\langle 9,0\rangle$	move	$\langle 2,0\rangle$	$\langle 4,0\rangle$	d	$\langle 7,0\rangle$	$\langle 7,0\rangle$

Figure 6: Moving d to be an element of list B

Validity Check

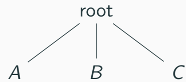
We define a move operation to be valid if and only if:

- It does not introduce any cycles.
- There is no concurrent move operation with a greater ID that moves the same element.

Validity Check: A naive approach

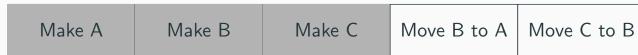
Whenever an operation is added:

1. Reapply all the operations in ascending ID order and check the validity of each operation
2. If an operation introduces a cycle, it is invalid
3. If an operation is valid, all the operations that move the same element with lower IDs are invalid

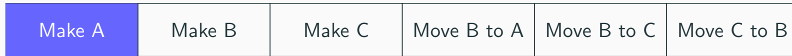


Replica 1:

Replica 2:



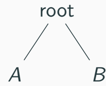
root

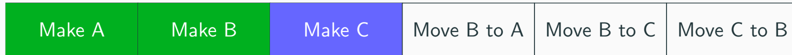


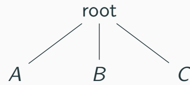
Ascending ID order

root

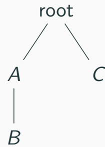
A


root
A
B

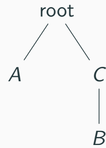

Make A
Make B
Make C
Move B to A
Move B to C
Move C to B


root
A
B
C

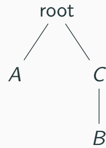
Make A
Make B
Make C
Move B to A
Move B to C
Move C to B


root
A
C
B

Make A
Make B
Make C
Move B to A
Move B to C
Move C to B



Make A	Make B	Make C	Move B to A	Move B to C	Move C to B



Make A	Make B	Make C	Move B to A	Move B to C	Move C to B

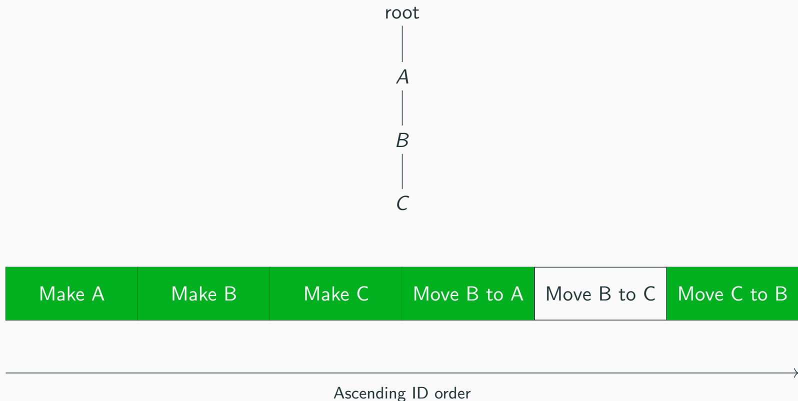
Validity Check: An optimized approach

Whenever adding a new operation, the process of reapplying operations with lower IDs remains the same.

1. Revert all the operations with greater IDs
2. Apply the new operation and check its validity
3. Reapply the reverted operations in ascending ID order and update the validity

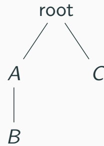
Before replica2 receives the new operation from replica1:

After replica2 receives the new operation from replica1:



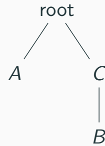
Revert:

Apply:

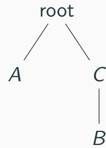


Make A	Make B	Make C	Move B to A	Move B to C	Move C to B

Reapply:



Make A	Make B	Make C	Move B to A	Move B to C	Move C to B



Make A	Make B	Make C	Move B to A	Move B to C	Move C to B

Validity Check: Further Optimization

- Batch Updating: Run revert-apply-reapply for a batch of new operations
- Lifecycle tracking: Avoid reverting and reapplying non-move operations as they are always valid

Batch Updating

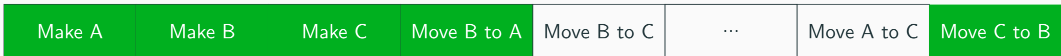
Replica 2 might receive multiple operations at once and we can do revert-apply-reapply once for all remote operations

root

A

B

C



Ascending ID order

Lifecycle tracking

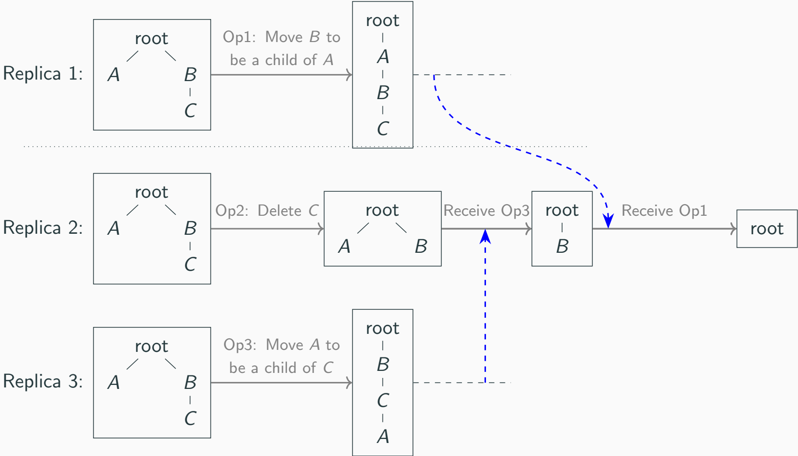
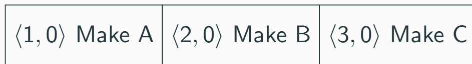


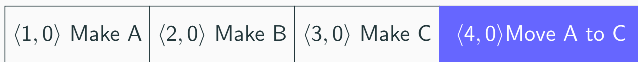
Figure 7: With a concurrent delete operation, two move operations no longer form a cycle and can be both executed.

Lifecycle tracking



Ascending ID order

Lifecycle tracking



Ascending ID order

Lifecycle tracking

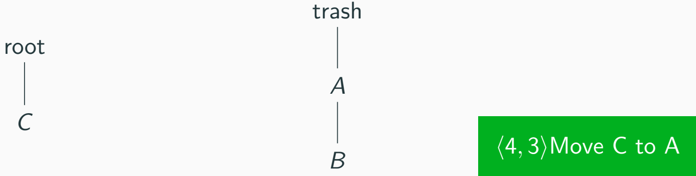
root
C

trash
A
B

$\langle 1, 0 \rangle$ Make A	$\langle 2, 0 \rangle$ Make B	$\langle 3, 0 \rangle$ Make C	$\langle 4, 0 \rangle$Move A to C	$\langle 4, 2 \rangle$Delete A

Ascending ID order

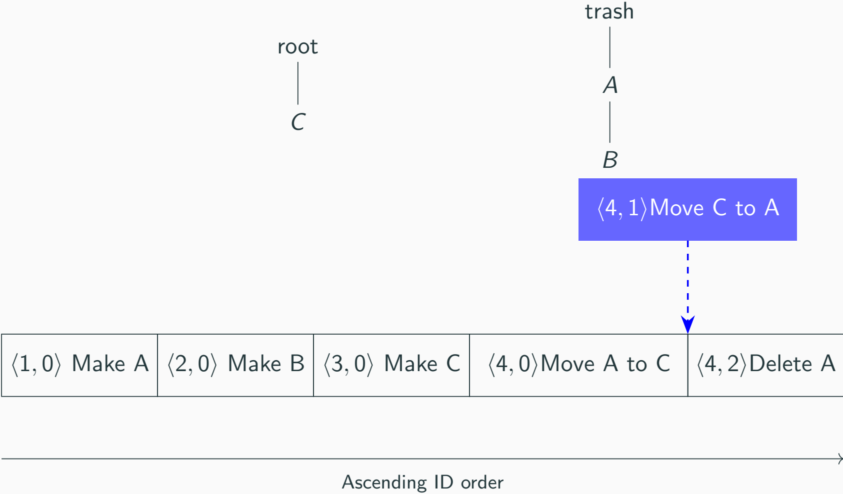
Lifecycle tracking



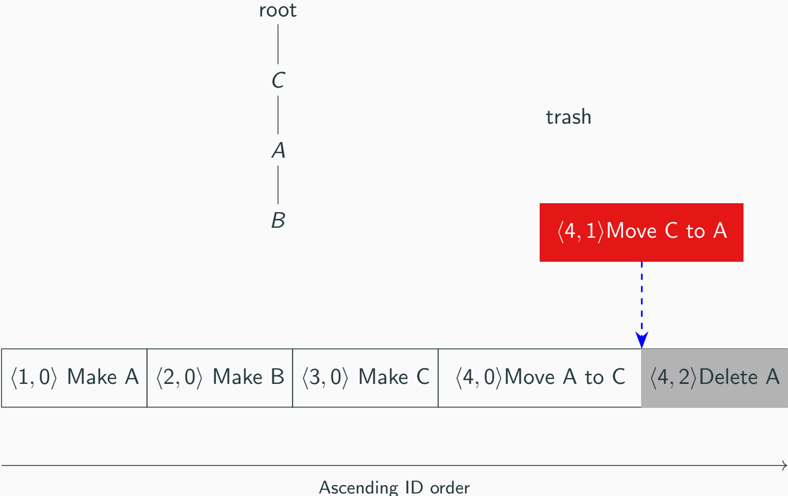
$\langle 1, 0 \rangle$ Make A	$\langle 2, 0 \rangle$ Make B	$\langle 3, 0 \rangle$ Make C	$\langle 4, 0 \rangle$Move A to C	$\langle 4, 2 \rangle$Delete A

Ascending ID order

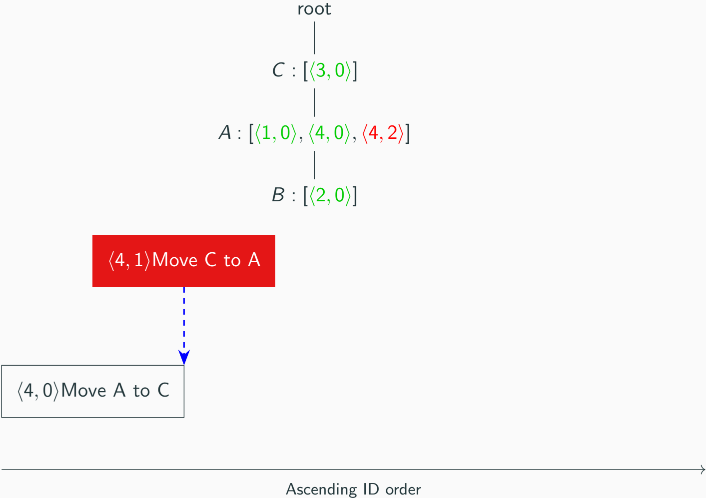
Lifecycle tracking



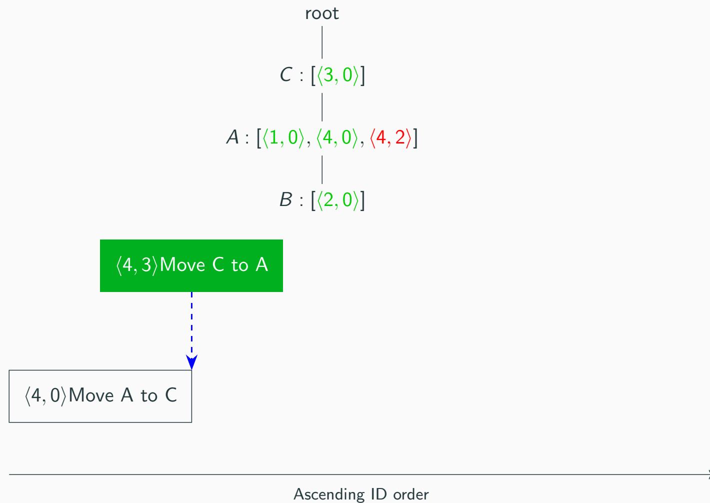
Lifecycle tracking



Lifecycle tracking



Lifecycle tracking



Evaluation

Miliseconds to converge two replicas with 10k operations diverged!

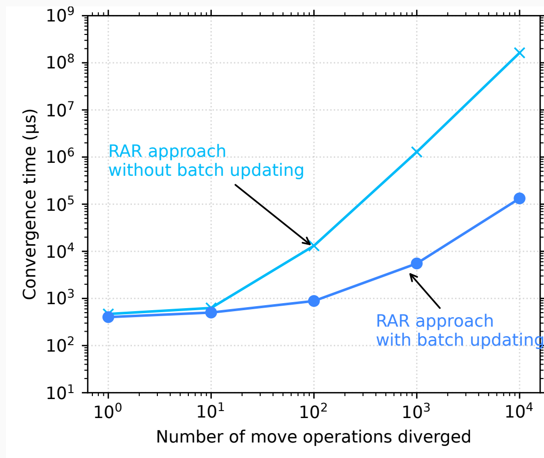


Figure 8: Convergence time of two actors that diverge by move operations

Overhead caused by move support is acceptable

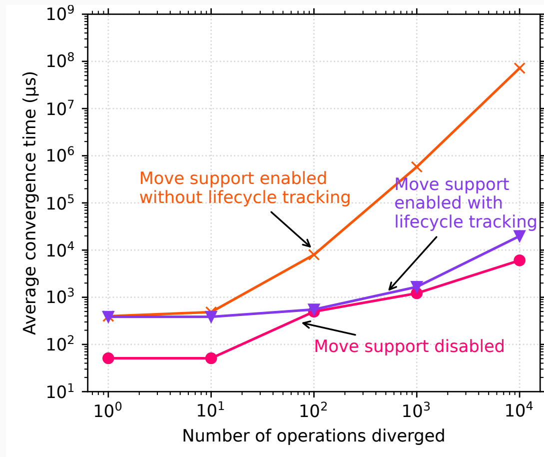


Figure 9: Convergence time of two actors that diverge by non-move operations

Random correctness testing

There were plenty of corner cases to consider:

1. Randomly generate move operations and apply them to a JSON object
2. Exchange the operations between replicas
3. Check if the JSON object is the same across all replicas

The test discovered a few bugs in the implementation, which were fixed.

Summary

- Extending move operations is feasible in a collaborative setting without major performance cost
- The move algorithm take care of potential duplicates and cycles and other corner cases