Extending JSON CRDT with move operations

Liangrun Da, Martin Kleppmann

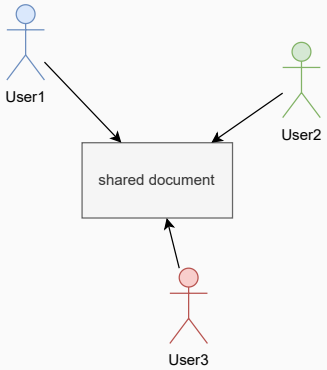
October, 2023

Contents

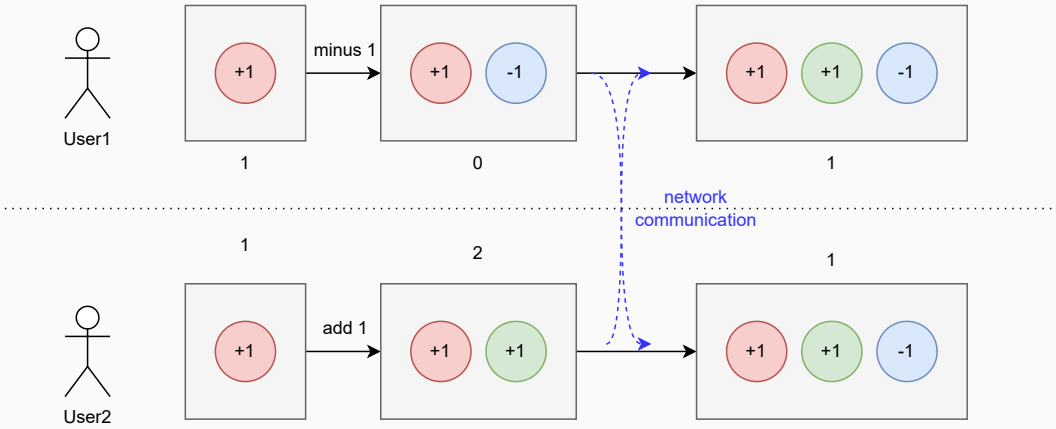
Introduction

Collaborative applications

- Nowadays we use a lot of collaborative applications
- Google Docs, Figma, Trello, ...
- Even our slides were done in a collaborative way (on sharelatex.tum.de)
- Each user has local replica of the data and can make changes to it concurrently
- All replicas converge to the same state automatically
- This can be achieved by using Conflict-free Replicated Data Types (CRDTs) [3]



A simple example of operation-based CRDT



Automerge: A CRDT for JSON documents

- Automerge [1] is a CRDT library that exposes a JSON document

Automerge: A CRDT for JSON documents

- Automerge [1] is a CRDT library that exposes a JSON document
- Created by Martin but is now a community-led project

Automerge: Operation IDs

- Pair of

$$\langle counter, actor\_id \rangle$$

[1] The operation id of the operation that created the object is used as its `object id`

Automerge: Operation IDs

- Pair of

$$\langle counter, actor\_id \rangle$$

- Define a total order (among operations)

$$\langle c_a, id_a \rangle < \langle c_b, id_b \rangle \iff c_a < c_b \vee (c_a = c_b \wedge id_a < id_b)$$

[1] The operation id of the operation that created the object is used as its `object id`

Automerge: Operation IDs

- Pair of

$$\langle counter, actor\_id \rangle$$

- Define a total order (among operations)

$$\langle c_a, id_a \rangle < \langle c_b, id_b \rangle \iff c_a < c_b \vee (c_a = c_b \wedge id_a < id_b)$$

- `counter` is set to the maximum of all known operations at a replica and incremented for each generated operation

[1] The operation id of the operation that created the object is used as its `object id`

Automerge: Operation IDs

- Pair of

$$\langle counter, actor\_id \rangle$$

- Define a total order (among operations)

$$\langle c_a, id_a \rangle < \langle c_b, id_b \rangle \iff c_a < c_b \vee (c_a = c_b \wedge id_a < id_b)$$

- `counter` is set to the maximum of all known operations at a replica and incremented for each generated operation
- Also used to refer to an object (list or map) in the JSON document[1]

[1] The operation id of the operation that created the object is used as its `object id`

Automerge: Core Operations

- `make_list`: creates a new list object

Automerge: Core Operations

- `make_list`: creates a new list object
- `make_map`: creates a new map object

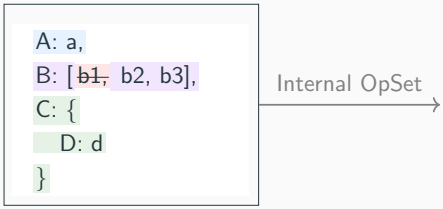
Automerge: Core Operations

- `make_list`: creates a new list object
- `make_map`: creates a new map object
- `put`: sets a scalar value

Automerge: Core Operations

- `make_list`: creates a new list object
- `make_map`: creates a new map object
- `put`: sets a scalar value
- `delete`: removes an object/scalar

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 1: An example JSON document with its internal OpSet

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?

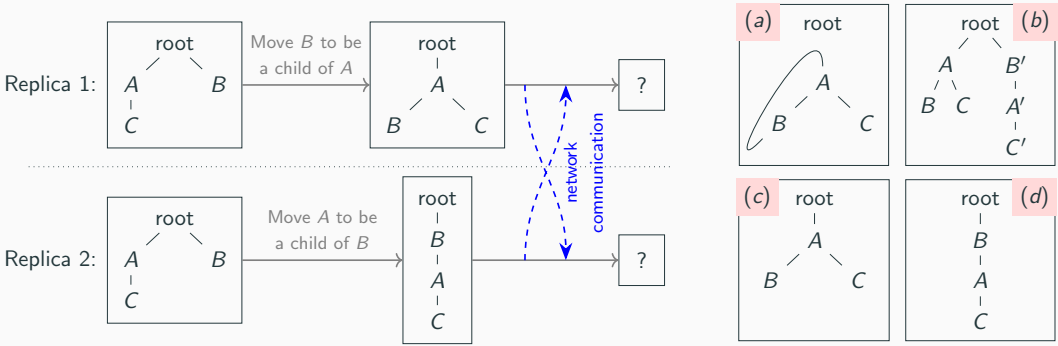
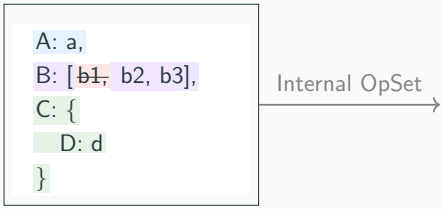


Figure 2: Concurrent moves might cause cycles or duplicate nodes. Figure from [2]

Generating Move Operations



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 3: An example JSON document with its internal OpSet

Algorithm

Generating Move Operations

A: a,
B: [~~b1~~, b2, d, b3],
C: {}

Internal OpSet →

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 4: Moving d to be an element of list B

Validity Check

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

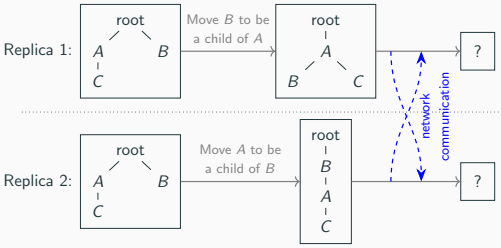
- It does not introduce any cycles.
- There is no concurrent move operation with a greater ID that moves the same element.

Validity Check: A wrong approach

Whenever a move operation is added

1. Check if the new operation is valid
2. If the new operation introduces a cycle or there is an operation with a greater ID moving the same element, it is invalid

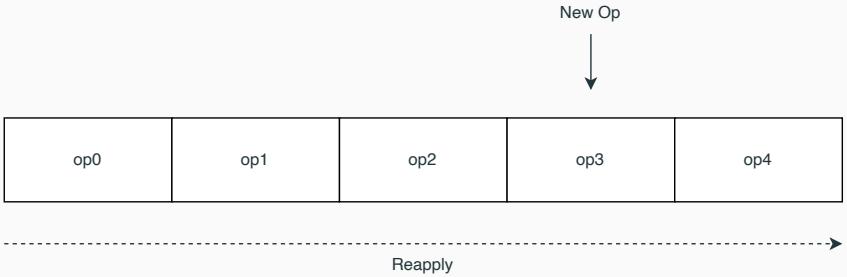
Result: inconsistent decisions on the validity of move operations across replicas



Validity Check: A naive approach

Whenever an operation is added:

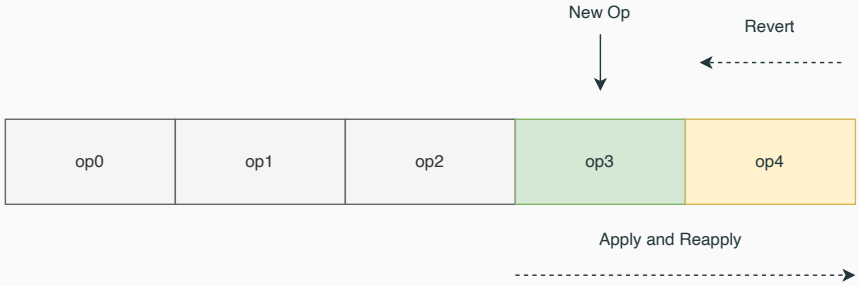
1. Reapply all the operations in ascending ID order and check the validity of each operation
2. If an operation introduces a cycle, it is invalid
3. If an operation is valid, all the operations that move the same element with lower IDs are invalid



Validity Check: An optimized approach

Since the validity of operations with lower IDs is not affected by the new operation:

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



Validity Check: Further Optimization

- Bulk Updating: Run revert-apply-reapply for a batch of new operations
- Checkpoint: Run revert-apply-reapply since the nearest checkpoint
- Avoid reverting and reapplying non-move operations as they are always valid

Evaluation

Evaluation

- Average convergence time with validity check enabled
- Comparison to unreplicated JSON object

Experiment setup:

1. Three replicas: c5.large (2 vCPU, 4 GB RAM)
2. Average latency between replicas: 169 ms
3. Randomly generate move operations at different rates and exchange with each other

Linear complexity of convergence

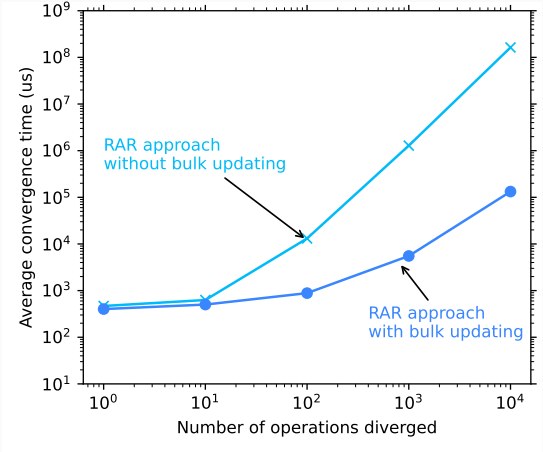


Figure 5: Average convergence time of two actors that diverge by move operation

Microseconds cost for replicated move operations!

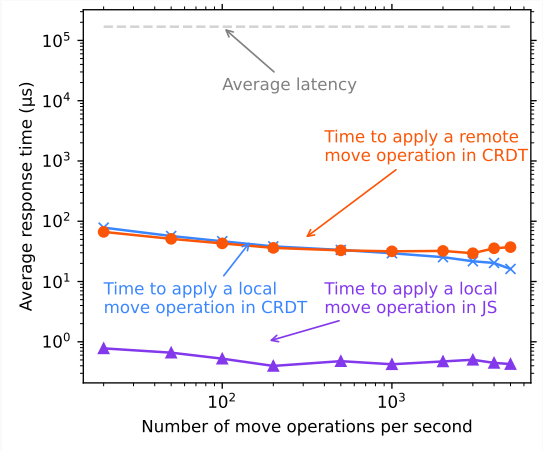


Figure 6: Average response time of move operation on JSON CRDT and unreplicated JSON object

Summary

- Exteding move operations is feasible in a collaborative setting without any major performance cost
- Move operations take care of potential duplicates and cycles

References

[1] Automerge CRDT. `https://automerge.org`.

[2] 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.

[3] 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.