Research report : renaming in Identifier-based Sequence Conflict-free Replicated Data Types (CRDTs)

Matthieu Nicolas

December 8, 2017

1 Context

1.1 System model

- Distributed large-scale system
- Asynchronous network
- Partition-tolerant
- Replicated sequence among nodes
- Eventual consistency
- Use a Identifier-based Sequence CRDT as the conflict resolution mechanism
- Intention preserving

1.2 Identifier-based Sequence Conflict-free Replicated Data Types (CRDTs)

1.2.1 State

Has a state S which represents the replicated sequence (use additional metadata to do so)

- Noted as $[(id, elt)]$ in the following figures
- The function $view(S)$ allows to retrieve the sequence represented by the state S
- **Example:** $view([(id_1, elt_1), (id_2, elt_2)]) = [elt_1, elt_2]$

1.2.2 Identifiers

Associates an identifier id to each element elt of the sequence

- Unique (an identifier can not be generated twice)
- Order relation (so that we can compare two identifiers)
  - Allows to determine the order of elements of the sequence using their identifiers

- Belong to a dense set
  - Always able to add a new element (and thus a new identifier) between two other elements

The elements in the sequence are always ordered according to their identifiers : in a sequence $[(id_1, elt_1), ..., (id_3, elt_3), ..., (id_2, elt_2)]$ we always have $id_1 < ... < id_3 < ... < id_2$.

1.2.3 Operations

For each operation to update the data structure, has two forms of it: the *local* form and the *remote* one

- The *local* operation is triggered by the node (by user request for example)
- Performing a *local* operation on a given state S returns the new state S' and the metadata needed to build an equivalent *remote* operation
- The *remote* operation is propagated to other nodes so they can also update their own state
- Given a state S and an operation $local(S, data) = (S', metadata)$, we have $remote(S, metadata) = S'$
- **Note:** given an *local* operation *localOp*, there may be several equivalent *remote* operations *remoteOp*, $remoteOp'$, *remoteOp*"...

1.2.4 *add*

The operation *add* allows to insert an element into the sequence :

- $addLocal(S, index, elt) = (S', (id, elt))$
  - Update state S by adding an element *elt* at the position *index* in the sequence
  - Return the resulting state S' as well as the identifier *id* generated for this element
  - The identifier *id* will be generated according to the identifiers of the elements previously at the positions $index - 1$ and *index*
    - **Example:** $addLocal([(id_1, elt_1), (id_2, elt_2)], 1, elt_3)$ will return id_3 such as $id_1 < id_3 < id_2$
  - This identifier *id* will be used (and especially its order relation with other identifiers) to update correctly other nodes' state
  - **Note:** When generating a new identifier between id_1 and id_2, there may be several identifiers id_3, id_3', id_3"... such as $id_1 < id_3 < id_3' < id_3" < id_2$. The returned identifier is chosen in a undeterministic manner.
- $addRemote(S, id, elt) = (S', (index, elt))$
  - Update state S by adding an element *elt* in the sequence
  - The position of insertion of this element will be determined using its *id*
  - Return the resulting state S' as well as the current index of the element in the sequence
- Given a state S, to one *addLocal* operation on S, many *addRemote* correspond (since the resulting *id* is generated in an undeterministic manner)
- Given a state S, to one *addRemote* operation on S, only one *addLocal* corresponds

1.2.5 *del*

The operation *del* allows to remove an element from the sequence :

- $delLocal(S, index) = (S', id)$
  - Update state S by removing the element at the position $index$ in the sequence
  - Return the resulting state S' as well as the identifier id of the deleted element
- $delRemote(S, id) = (S', index)$ allowing to remove the element identified by id
  - Update state S by removing the element identified by id
  - Return the resulting state S' as well as the position $index$ of the deleted element in the sequence
- Given a state S, to one *delLocal* operation, only one *delRemote* corresponds
- Given a state S, to one *delRemote* operation, only one *delLocal* corresponds

1.2.6 Log of operations

Associates to a state S a log L

- Is a sequence of entries $(remoteOp, localOp)$, a remote operation and its local counterpart
- The sequence of remote operations, performed in order from a blank state S_{blank}, allows to recreate state S
- Each entry represented as $\boxed{\begin{array}{c} remoteOp \\ localOp \end{array}}$ in the following figures

1.2.7 Causal context

Associates to a state S a causal context cc

- Represents all operations known at state S
- Can use a *version vector* for example as an implementation

An example of the lifecycle of such a replicated data structure is shown in figure 1

2 *rename* operations

2.1 Motivation

- Identifiers growing over time
- Performances of the data structure thus decreasing over time

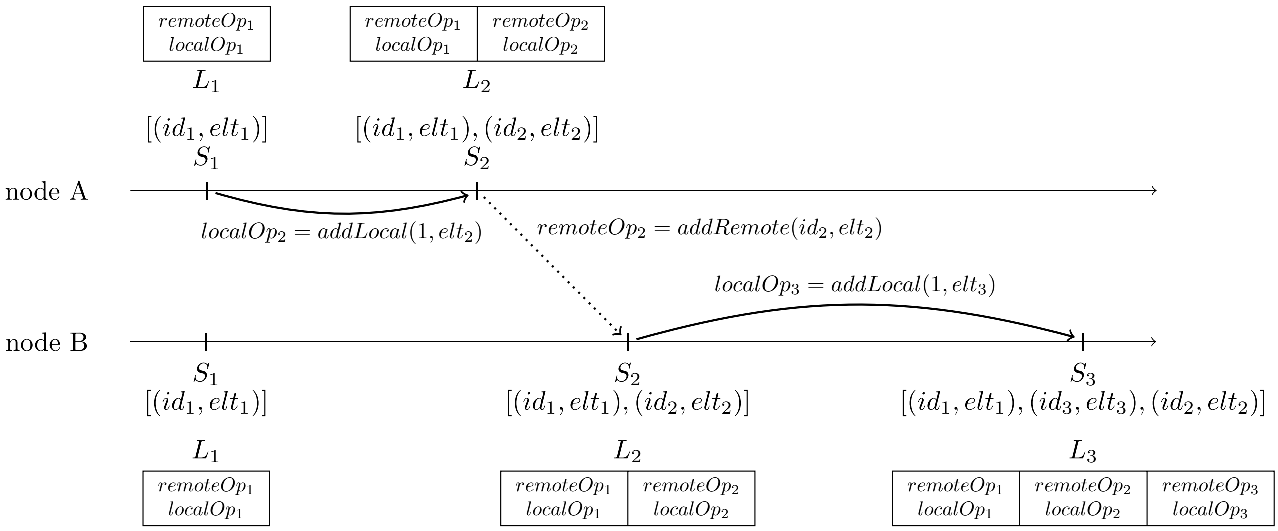

Figure 1: Insertion of elements in the replicated sequence

2.2 *renameLocal*

- Add an operation $renameLocal(S) = (S', mapIds, cc_S)$
  - Replace each identifier attached to elements of S with new ones
  - Return a map *mapIds* of the previous identifiers to the new ones
  - Also need to return the causal context cc_S of the state S to indicate on which state has been performed the renaming operation
  - $view(S) = view(S')$ where $(S', \_, \_) = renameLocal(S)$
  - Represented by figure 2

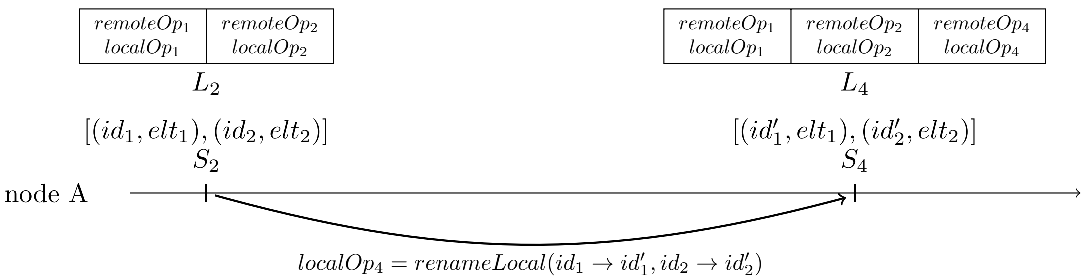

Figure 2: Local renaming of identifiers of the replicated sequence

2.3 *renameRemote*

- Add an operation $renameRemote(S, L, mapIds, cc_{S'}) = (S", L")$

- Replace current state S by equivalent state S" and current log L by equivalent log L"
- Rename all identifiers $id \in S \cdot id \in S'$ using $mapIds$
- Also have to rename all identifiers $id \in S \cdot id \notin S'$ to preserve the current order of elements
- **Precondition:** $S \geq S'$ (S has seen all the operations seen by S' but may have seen more)
- $view(S) = view(S")$ where $(S", \_) = renameRemote(S, L, mapIds, cc_{S'})$

2.4 Usage

Given an operation $renameRemote(S, L, mapIds, cc_{S'})$, resulting from the execution of $renameLocal(S')$ on another node, we have to perform the following steps to apply it:

1. Instantiate a blank state S" and its empty log L"
2. Generate a log L_{causal} made of all operations belonging to the causal context $cc_{S'}$
3. For each entry $(remoteOp, localOp)$ of L_{causal}
   (a) Update state S" by performing $remoteOp(S", metadata)$
   (b) Add entry $(remoteOp, localOp)$ to L"
4. Rename all identifiers of the data structure according to $mapIds$ (at this point, $S" = S'$)
5. Generate a log $L_{concurrent}$ made of all operations of L not included in L_{causal}
6. For each entry $(remoteOp, localOp)$ of $L_{concurrent}$
   (a) Update state S" by performing $localOp(S"_{prev}) = (S"_{new}, metadata)$
   (b) Build new remote operation $remoteOp'$ given $metadata$
   (c) Add entry $(remoteOp', localOp)$ to L"
   (d) Propagate $remoteOp'$

This algorithm is represented by figure 3

2.5 Limits

- Differents nodes, while performing the remote renaming operation, may replay at step 6a the same operation
- Since there is no coordination between them, in the case of a $addLocal$, they will end up generating two different remote operations $remoteOp'$ and $remoteOp$" during step 6b
- We will have to deliver them both to each node to actually converge (the states would differ otherwise)
- This will result in the duplication of the user's intention (since the inserted element will end up being added twice)
- An example is show in figure 4

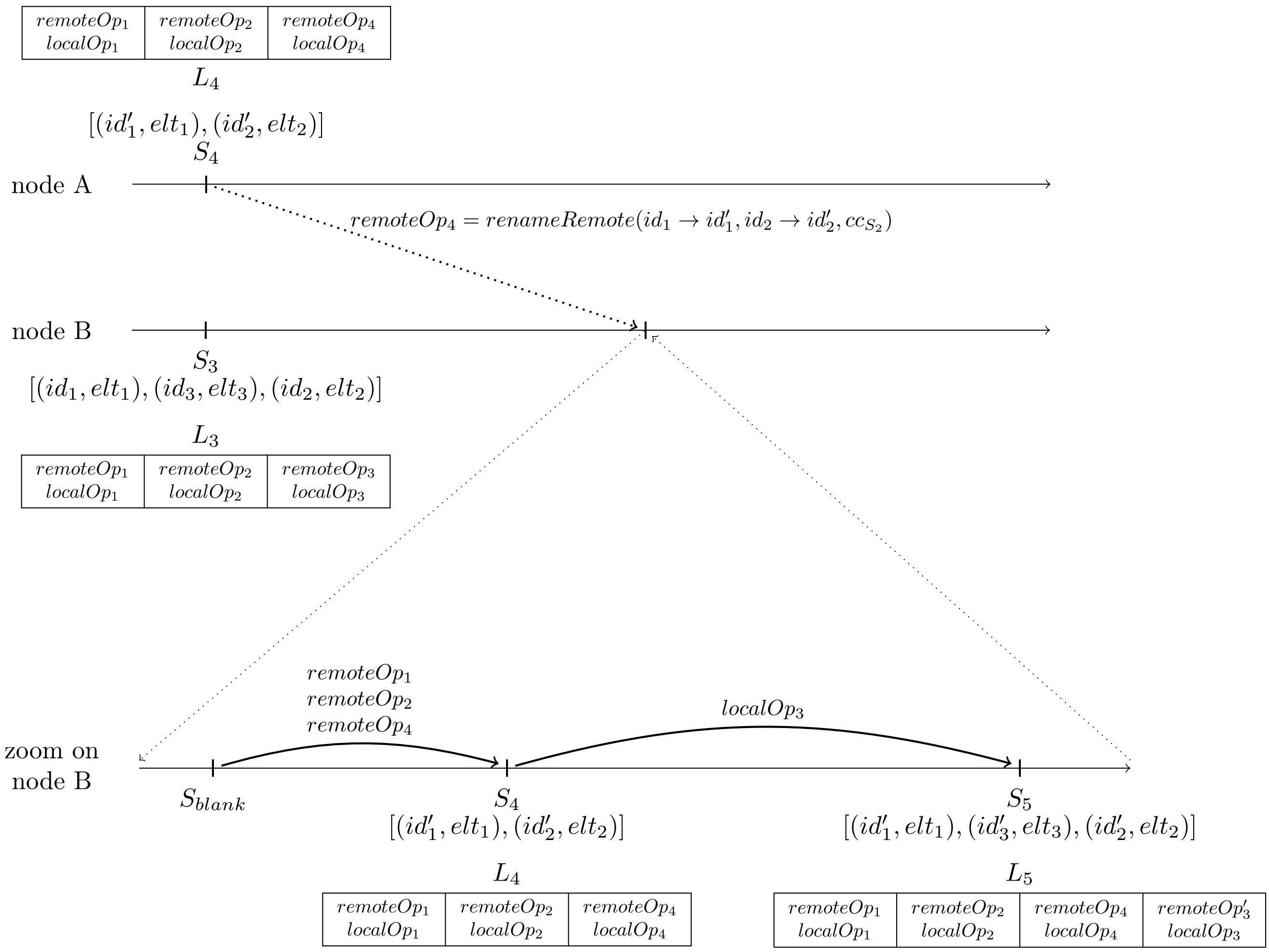

Figure 3: Renaming with concurrent operations

3 *addRedo* operation

3.1 Idea

- At step 6a, if we can generate deterministically the resulting id for a given previous log entry $(addRemote, addLocal)$, then we would not duplicate the user's intention
- Indeed, each node would thus generates the same operation $addRemote'$ at step 6b
- In that case, we would only need to deliver at least once $addRemote'$ to the nodes to converge (or exactly-once if the $addRemote$ is not idempotent)

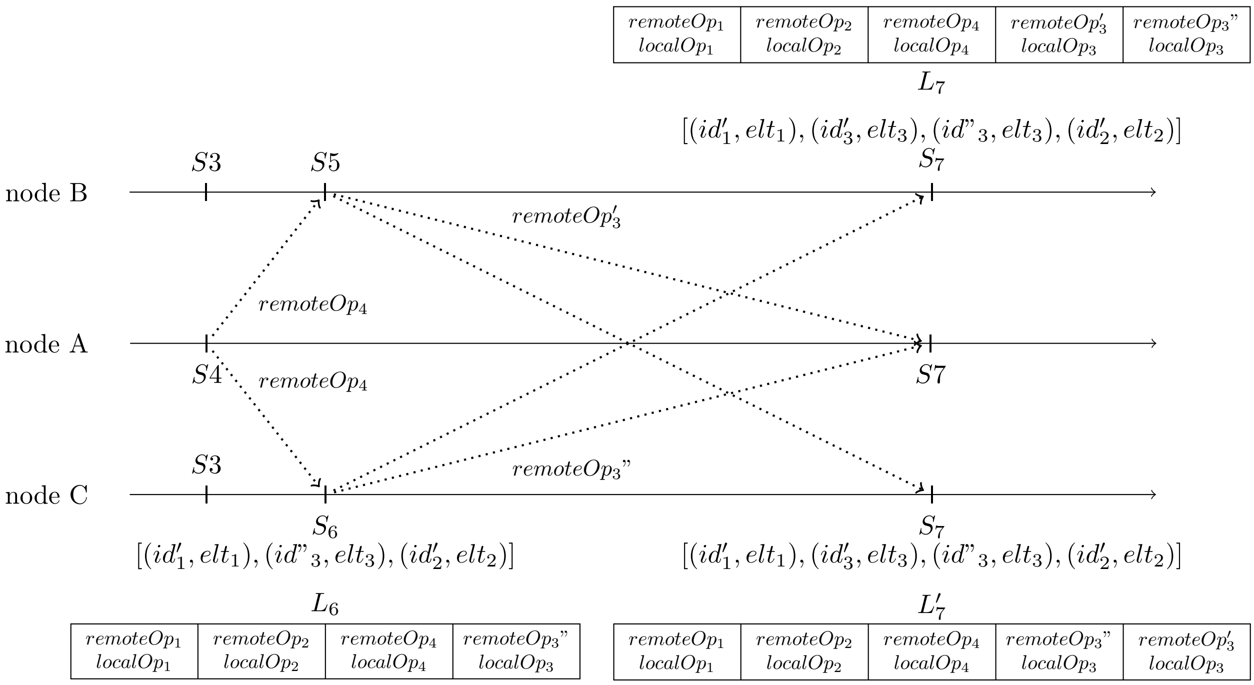

Figure 4: Duplication of the intention of $localOp_3$

3.2 Research question

Can we define the following operation $addRedo(S, (addRemote, addLocal)) = (S', (id', elt))$ such as :

- id' is generated deterministically
- $view(S) = view(S')$ where $(S, id) = addLocal(S", index, elt)$ and $(S', \_) = addRedo(S", (addRemote(S", id, elt), addLocal(S", index, elt))$

This operation would be used at step 6a instead of simply using *addLocal* and would solve the duplication effect.

4 Discussion

- Need to keep the log of operations (both *remote* and *local*)
- Performances of a *renameRemote* depend on the number of operations in the log and the number of concurrent operations
  - Have to replay all operations from the causal context of the *renameRemote* operation
  - Have to regenerate concurrent operations and propagate them
- Can propose mechanism to reduce the size of the log
  - By pruning causally stable entries and using snapshots

- New identifiers generated by *addRedo* operations may be larger than the initial ones according to the chosen strategy
  - Can argue that they will shrink with the next *rename*
- Solving concurrent *rename* looks difficult
  - For now, can assume that only one node can perform such operations

5 Counter-example

Found a counter-example which invalidate the algorithm proposed in section 2.4. We replay the same operations as in previous examples but in this case:

- $localOp_2$ and $localOp_3$ are concurrent
- The generated identifiers id_2 and id_3 are in this order: $id_2 < id_3$

In this scenario, when replaying $localOp_3$ at step 6a, we will swap the position of the elements elt_2 and elt_3 compared to the previously observed state. This result in a incoherence of the system and may result in the violation of the intention of following operations based on this previously observed state. This scenario is represented by figure 5.

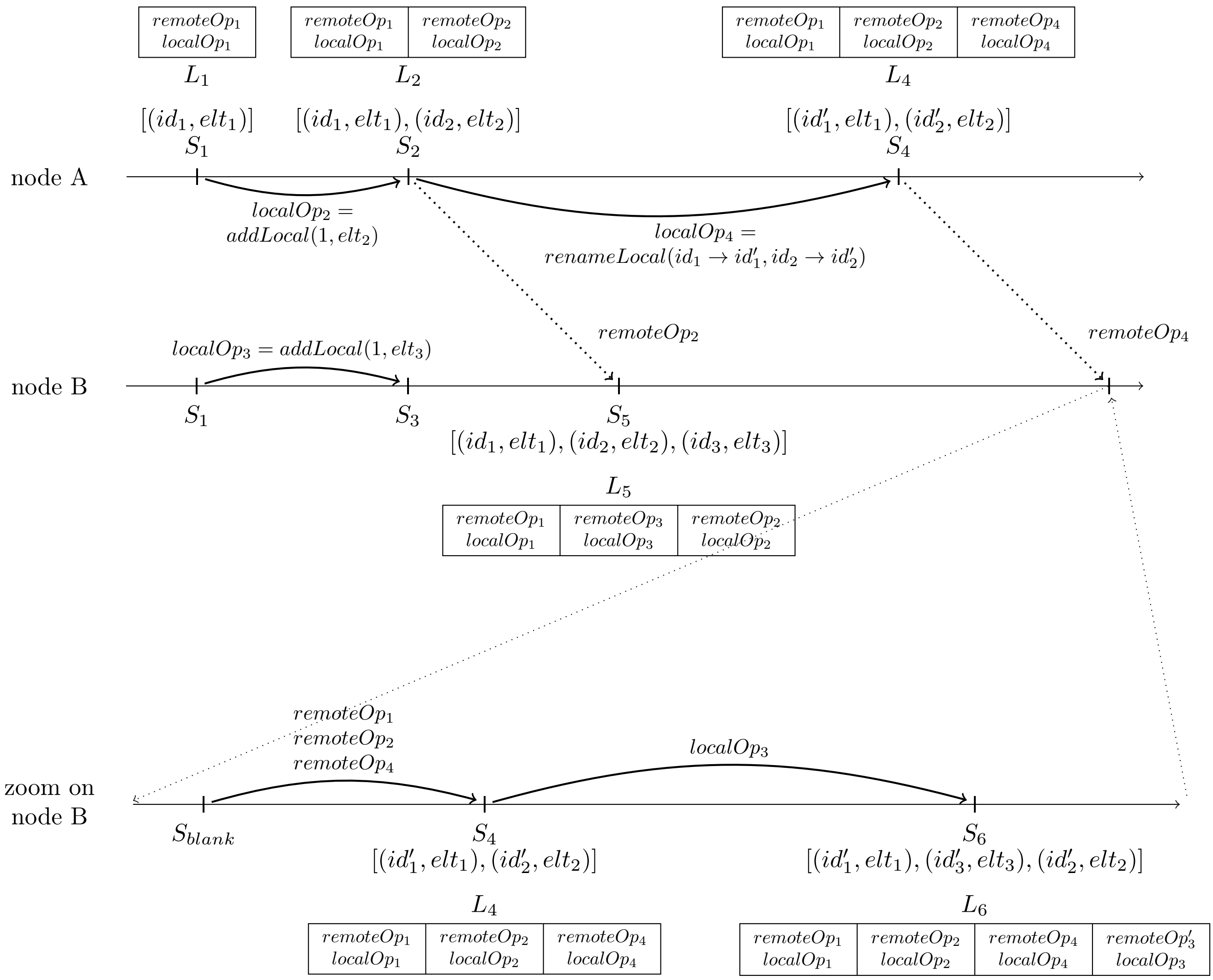

Figure 5: Incoherence occuring by replaying local operations during renaming process