

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

Matthieu Nicolas

December 14, 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

##### Description

Associates an identifier  $id$  to each element  $elt$  of the sequence

- Unique (an identifier cannot 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  $[(id_1, elt_1), \dots, (id_3, elt_3), \dots, (id_2, elt_2)]$  we always have  $id_1 < \dots < id_3 < \dots < id_2$ .

## Details

An identifier is actually composed of a list of tuples. Each tuple is of the following form:

$$< pos, id_{site}, clock_{site} >$$

where

- $pos : Int$ , allows to determine the position of this identifier compared to other ones.
- $id_{site} : Int$ , refers to the site's identifier, assumed to be unique.
- $clock_{site} : Int$ , refers to the site's logical clock, which increases monotonically with local operations.

We note the  $id_{site}$  and the  $clock_{site}$  of the last tuple of  $id$  as  $id.id_{site}$  and  $id.clock_{site}$  respectively.

## Generation

To generate a new identifier  $id_3$  between two others  $id_1 = [tuple_{1,1}, tuple_{1,2}, \dots, tuple_{1,n}]$  and  $id_2 = [tuple_{2,1}, tuple_{2,2}, \dots, tuple_{2,n}]$ , we use the algorithm 1:

We compare the identifiers' tuples in a pairwise manner. As soon as we are able to generate a new tuple  $tuple_3$  such as  $tuple_1 < tuple_3 < tuple_2$ , we add it to  $id_3$  and return the later.

If we cannot generate such a tuple, we add instead  $tuple_1$  to  $id_3$  and move to the next pair.

**Note:** If the identifiers  $id_1$  and  $id_2$  have different sizes, we use some default tuples to "fill" the shorter of the two:

- $minTuple$  if it is  $id_1$
- $maxTuple$  otherwise

## Comparison

To compare two identifiers, we use the algorithm 2:

When comparing two identifiers, we compare their tuples in a pairwise manner. As soon as we find one element which is different from its pair, we can determine the order between the two identifiers.

---

**Algorithm 1** Identifier generation algorithm (simplified)

---

**function** GENERATEIDENTIFIER( $id_1 : Id, id_2 : Id, id_{site} : Int, clock_{site} : Int$ ):  $Id$

**Require:**  $id_1 < id_2$

**Ensure:**  $id_1 < id_3 < id_2$

```
 $id_3 \leftarrow []$ 
 $continue \leftarrow true$ 
 $i \leftarrow 0$ 
while  $continue$  do
   $tuple_1 \leftarrow id_1[i]$ 
   $tuple_2 \leftarrow id_2[i]$ 
  if  $tuple_2.pos - tuple_1.pos > 2$  then
     $newPos \leftarrow randomBetween(tuple_1.pos, tuple_2.pos)$ 
     $id_3 \leftarrow id_3 :: < newPos, id_{site}, clock_{site} >$ 
     $continue \leftarrow false$ 
  else
     $id_3 \leftarrow id_3 :: tuple_1$ 
  end if
   $i \leftarrow i + 1$ 
end while
return  $id_3$ 
end function
```

---

### 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 users' requests 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  $localOp(S, data) = (S', metadata)$ , we have  $remoteOp(S, metadata) = S'$
- **Note:** given an *local* operation  $localOp$ , there may be several equivalent *remote* operations  $remoteOp, remoteOp', remoteOp'' \dots$ , but only one is picked.

We note the identifier of the element targeted by an *remote* operation as  $remoteOp.id$ .

### 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

---

**Algorithm 2** Identifier comparison algorithm

---

```
function COMPAREIDENTIFIERS( $id_1 : Id, id_2 : Id$ ):  $LESS \mid EQUALS \mid GREATER$ 
  for  $i \leftarrow 0, \min(id_1.length, id_2.length)$  do
     $tuple_1 \leftarrow id_1[i]$ 
     $tuple_2 \leftarrow id_2[i]$ 
    if  $tuple_1.pos < tuple_2.pos$  then
      return  $LESS$ 
    else if  $tuple_1.pos > tuple_2.pos$  then
      return  $GREATER$ 
    else if  $tuple_1.id_{site} < tuple_2.id_{site}$  then
      return  $LESS$ 
    else if  $tuple_1.id_{site} > tuple_2.id_{site}$  then
      return  $GREATER$ 
    else if  $tuple_1.clock_{site} < tuple_2.clock_{site}$  then
      return  $LESS$ 
    else if  $tuple_1.clock_{site} > tuple_2.clock_{site}$  then
      return  $GREATER$ 
    end if
  end for
  if  $id_1.length < id_2.length$  then
    return  $LESS$ 
  else if  $id_1.length > id_2.length$  then
    return  $GREATER$ 
  end if
  return  $EQUALS$ 
end function
```

---

- 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  $id_3, id'_3, id_3'' \dots$  such as  $id_1 < id_3 < id'_3 < id_3'' < id_2$ . The returned identifier is chosen in an 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 the *remote* operations observed
- The sequence of remote operations, performed in order from a blank state  $S_{blank}$ , allows to recreate state  $S$
- Each entry is represented as  $\boxed{remoteOp}$  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

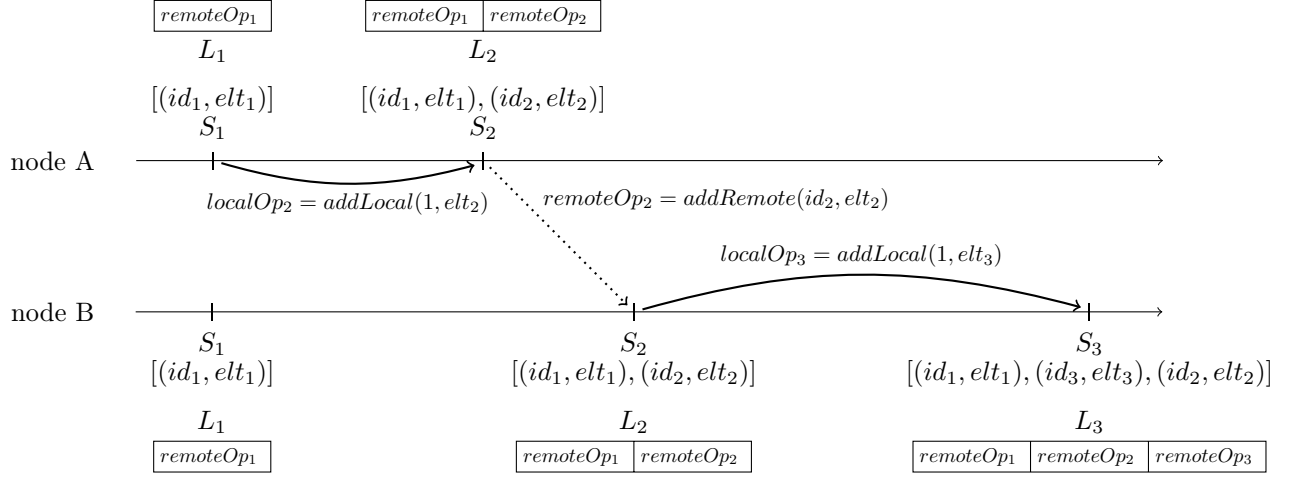


Figure 1: Insertion of elements in the replicated sequence

## 2 rename operations

### 2.1 Motivation

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

### 2.2 renameLocal

**Definition**

- 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  $(S', \_, \_) = renameLocal(S)$

## Algorithm

The renaming algorithm can be seen here as an algorithm to distribute evenly a number of values  $n$  over an interval *range*. Such an algorithm is described in 3:

---

**Algorithm 3** Identifier renaming algorithm

---

```
function RENAMEIDENTIFIER( $id : Id, index : Int, n : Int$ ):  $Id$ 
   $min \leftarrow MIN$  ▷ The minimum value of the range used
   $max \leftarrow MAX$  ▷ The maximum value of the range used
   $range \leftarrow max - min$ 
   $step \leftarrow range / (n + 1)$ 
   $newPos \leftarrow (index + 1) * step + min$ 
   $id' \leftarrow [< newPos, id.id_{site}, id.clock_{site} >]$ 
  return  $id'$ 
end function
```

---

This algorithm can then be used to rename each identifier for a given state  $S$  and generate a mapping between the previous identifiers and the new ones as shown in 4:

---

**Algorithm 4** Local renaming algorithm

---

```
function RENAMELOCAL( $S : State$ ):  $Map < Id, Id >$ 
   $mapIds \leftarrow Map()$ 
   $n \leftarrow S.length$ 
  for all  $(id, elt, index) \in S$  do
     $id' \leftarrow renameIdentifier(id, index, n)$ 
     $mapIds.set(id, id')$ 
     $id \leftarrow id'$ 
  end for
  return  $mapIds$ 
end function
```

---

## Limits

- Work only if  $n < range$ 
  - Can assume it is generally the case

## 2.3 *renameRemote*

### Definition

- 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  $id \in S \cap S'$  using  $mapIds$

- Also have to rename all identifiers  $id \in S \cdot id \notin S'$  to preserve the current order of elements
- **Precondition:**  $cc'_S \subset cc_S$  ( $S$  has seen all the operations seen by  $S'$  but may have seen more)
- $view(S) = view(S'')$  where  $(S'', \_) = renameRemote(S, L, mapIds, cc_{S'})$

## Algorithm

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 algorithm 5 to apply it:

---

### Algorithm 5 Remote renaming algorithm

---

```

procedure RENAMEREMOTE( $S : State, L : Log, mapIds : Map < Id, Id >, cc_{S'} : StateVector$ )
   $mapIds' \leftarrow mapIds$ 
  for all  $remoteOp \in concurrentOps(cc_{S'}, L)$  do  $\triangleright$  Get concurrent operations to the renaming
     $id \leftarrow remoteOp.id$ 
    if  $id \notin mapIds'$  then
       $prevId \leftarrow prev(id, mapIds)$   $\triangleright$  Get the predecessor of  $id$  in  $mapIds$ 
       $prevId' \leftarrow mapIds.get(prevId)$ 
       $id' \leftarrow prevId' :: id$   $\triangleright$  Generate the new identifier by concatenating  $prevId'$  and  $id$ 
       $mapIds'.set(id, id')$ 
       $remoteOp' \leftarrow generateRemoteOp(remoteOp, id')$ 
       $broadcast(remoteOp')$ 
    end if
  end for
  for all  $(id, elt) \in S$  do
     $id \leftarrow mapIds'.get(id)$ 
  end for
end procedure

```

---

## Limits

- Need a causal delivery of the *rename* operation
  - Actually not necessary
  - But would have to be able to transform operations from its causal context using *mapIds*
  - Thus require to keep a reference to *mapIds*
  - Would be possible to receive an operation which is outdated of several *renaming*
  - Would have to go through all its transformations
- Do not handle concurrent *rename* operations
  - For now, can assume that only one node can perform such operations
- *renameRemote* operation can be bandwidth-consuming (need to send old and new identifiers)
  - Can reduce its size but will require more computations



- Using the causal context of the operation, we can regenerate original state (replay the log)
- Since *renameLocal* is deterministic, can re-compute *mapIds* locally

### 3 Discussion

- The size of identifiers from concurrent operations to the *renaming* operation will increase
  - Can argue that they will shrink at next renaming
- Can use a mechanism of *epoch*
  - Each *rename* increase the *epoch* counter
  - Each operation is labelled with its *epoch* of generation
  - Allow us to reject obsoletes operations
- Do not require to add other causality information on all operations, only on the *renaming* one
  - The *epoch* mechanism and the *renaming*'s causal context are sufficient to determine the concurrency of operations to the *renaming* operation
  - If the *epoch* is the same as the one before the *renaming* and if this operation does not belong to the *renaming*'s causal context, this operation is a concurrent one
- Performances of the *renaming* operations depend on the number of elements of the data structure, the number of elements of the map and the number of concurrent operations
- Should be able to adapt the algorithms 4 and 5 to blockwise Identifier-based Sequence CRDTs
  - Here, each element is manipulated one by one (with *add* and *delete*, but also during search)
  - In some algorithms like *LogootSplit*, we actually group elements using blocks
  - It allows us to:
    - \* Factorise the identifiers of contiguous elements
    - \* Reduce the size of the collection by storing the blocks instead of elements directly (thus speed up search)
  - We could adapt the algorithms for these data structures
  - The *renaming* operations would thus help us to reduce the number of blocks too (could regroup all elements in one new block)

### 4 Questions

- How to deal with concurrent operations to *renaming* one when you already applied the *renaming*?
  - Can reject it and wait to receive its modified version
    - \* The node which sent us the original version should be able to send us its modified one

- \* But induces some delay
  - Can use *mapIds* to compute its transformation
    - \* Need to retrieve it or to compute it again
- When to trigger the *renaming*?
  - According to the size of the longer identifier?
  - According to the number of elements (in blockwise CRDTs)?
    - \* What would be the thresholds in these cases?
  - According to the state of the collaboration?
    - \* If the system is idle for example
- Which version(s) of the operations to store in the log?
  - The original one?
  - The modified one?
  - A mix ?
  - Actually depends on the answer to the previous question