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

Matthieu Nicolas

December 19, 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 $[(id_1, elt_1), ..., (id_3, elt_3), ..., (id_2, elt_2)]$ we always have $id_1 < ... < id_3 < ... < id_2$.

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 $id_1 = [tuple_{1,1}, tuple_{1,2}, ..., tuple_{1,n}]$ and $id_2 = [tuple_{2,1}, tuple_{2,2}, ..., tuple_{2,n}]$, 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 ← [ ]
    continue ← true
    i ← 0
    while continue do
        tuple_1 ← id_1[i]
        tuple_2 ← id_2[i]
        if tuple_2.pos − tuple_1.pos > 1 then
            newPos ← randomBetween(tuple_1.pos, tuple_2.pos)
            id_3 ← id_3 ::< newPos, id_site, clock_site >
            continue ← false
        else
            id_3 ← id_3 :: tuple_1
        end if
        i ← 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*"..., 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 | EQUALS | GREATER
    for i ← 0, min(id_1.length, id_2.length) do
        tuple_1 ← id_1[i]
        tuple_2 ← 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"... 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 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

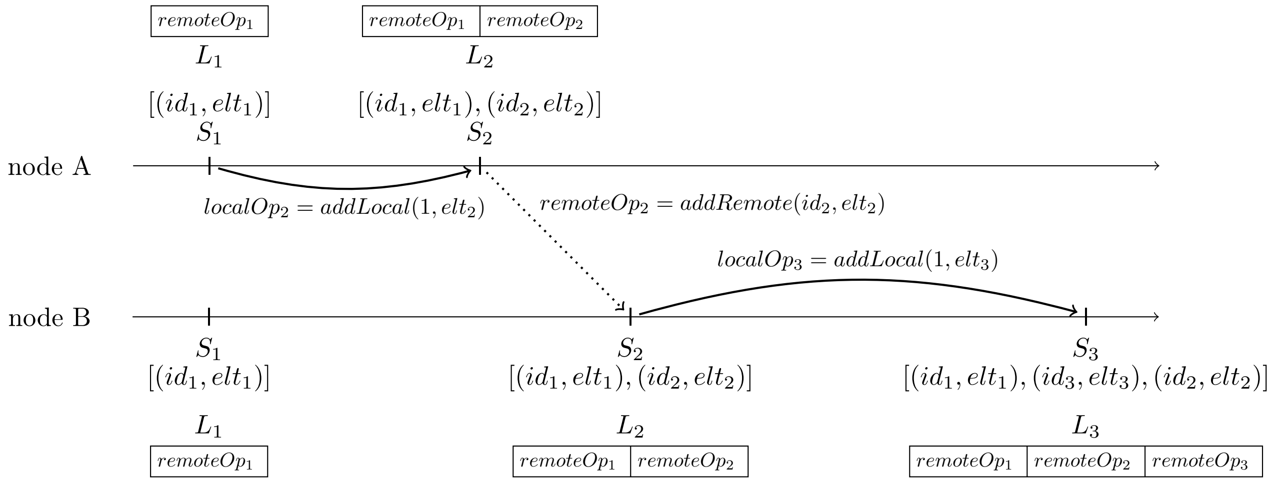

Figure 1: Insertion of elements in the replicated sequence

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' ← mapIds
    for all remoteOp ∈ concurrentOps(cc_S', L) do  ▷ Get concurrent operations to the renaming
        id ← remoteOp.id
        if id ∉ mapIds' then
            prevId ← prev(id, mapIds)                 ▷ Get the predecessor of id in mapIds
            prevId' ← mapIds.get(prevId)
            id' ← prevId' :: id     ▷ Generate the new identifier by concatenating prevId' and id
            mapIds'.set(id, id')
        end if
    end for
    for all (id, elt) ∈ S do
        id ← mapIds'.get(id)
    end for
end procedure
```

In this algorithm, we want to replace all identifiers from the current state S by the new ones using the renaming map $mapIds$ provided by the node which performed the renaming.

Beforehand we have to complete $mapIds$ to also rename the identifiers of elements concurrently inserted to the renaming. Thus we have to generate a new identifier id' for each id concurrently inserted and add the mapping $id \rightarrow id'$ to $mapIds$. To do so, we browse each concurrent operation, retrieve the id of the targeted element and generate its new identifier id'.

To generate id', we retrieve $prevId$, the predecessor of id in $mapIds$. Since every node observes the same id and $mapIds$ and thus $prevId$, we can use them to generate id' in a deterministic manner : we get it by concatenating $prevId'$ and id. This allows us to generate deterministically a new identifier id' and also to preserve the order relation with other identifiers.

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 Optimisation

3.1 Motivation

As stated previously, the presented mechanism's main flaw is the need to broadcast *mapIds*, a mapping between the old identifiers to the new ones. According to the size of the old identifiers, this map may be bandwidth-consuming.

3.2 Idea

The algorithm 3 is deterministic. Hence, as long as we provide to other nodes the required data (identifiers which are renamed, their index in the data structure, and the number of elements in the data structure), they will be able to compute the new identifiers.

All these data are actually contained in a sequence composed of the identifiers which are renamed. We can thus, instead of sending *mapIds*, send such a sequence.

Example

Instead of broadcasting $(id_1 \to id'_1, id_2 \to id'_2, id_3 \to id'_3)$, we only need to broadcast to other nodes $[id_1, id_2, id_3]$.

We were able to reduce the size of the data to broadcast, but can reduce it further. Indeed, each identifier is unique. But to refer to one identifier, we do not need all of its components : we only need two of them, $id.id_{site}$ and $id.clock_{site}$. Therefore, we are able to reduce again the data to broadcast to a fixed size per element by discarding every component of the identifier except $id.id_{site}$ and $id.clock_{site}$.

Example

Instead of broadcasting $[id_1, id_2, id_3]$, we only need to broadcast to other nodes $[(id_1.id_{site}, id_1.clock_{site}), (id_2.id_{site}, id_2.clock_{site}), (id_3.id_{site}, id_3.clock_{site})]$.

Algorithm 6 Map recomputation algorithm

function RECOMPUTEMAPIDS($S : State, L : Log, seq :< Int, Int > [], cc_{S'} : StateVector$): $Map < Id, Id >$
  $mapIds \leftarrow Map()$
  $n \leftarrow seq.length$
  $ids \leftarrow S.map((id, elt) \rightarrow id)$ ▷ Retrieve all identifiers from current state
  $ids \leftarrow ids :: L$ ▷ Retrieve all identifiers from concurrent operations
    $.filter(remoteOp \rightarrow remoteOp.isConcurrent(cc_{S'}))$
    $.map(remoteOp \rightarrow remoteOp.id)$
  for all $(id_{site}, clock_{site}, index) \in seq$ **do**
    $id \leftarrow ids.find(id \rightarrow id.id_{site} = id_{site}$ **and** $id.clock_{site} = clock_{site})$
    $id' \leftarrow renameIdentifier(id, index, n)$
    $mapIds.set(id, id')$
  end for
  return $mapIds$
end function

3.3 Algorithm

Given such a sequence, we can then recompute *mapIds* using the following algorithm 6:

To recompute the renaming map *mapIds* from the received sequence *seq*, we have to retrieve all the renamed identifiers given their id_{site} and $clock_{site}$.

First, we generate the list *ids* of all potentially renamed identifiers. This list contains all identifiers from the current state S but also the identifiers of concurrent operations to the renaming, since some renamed identifiers may have been deleted concurrently.

Then using this list, we can for each entry of *seq* find the original identifier *id*, compute its new one id' and add the mapping $id \rightarrow id'$ to *mapIds*.

After recomputing *mapIds*, we can resume the *renaming* using algorithm 5.

Notes:

- Able with this solution to get a fixed size per element in the data broadcasted (instead of broadcasting unbounded identifiers).
- Need to browse both the state and the concurrent operations
  - A renamed identifier may have been deleted concurrently
  - Thus it won't be in the state
  - But still need it to properly rename concurrently inserted elements
- Cannot factorise this algorithm with the generation of new identifiers for concurrent operations
  - We need the full *mapIds* to generate them

4 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)

5 Adaptation to blockwise CRDTs

5.1 Identifier

In blockwise CRDTs, identifiers have one more component in their tuples, the *offset*. The identifiers are thus of the following form:

$$[< pos, id_{site}, clock_{site}, offset >]$$

where

- $offset : Int$, represents the index of this identifier in an identifier interval

We note the $offset$ of the last tuple of id as $id.offset$.

5.2 IdentifierInterval

It represents an interval of several contiguous identifiers. It allows to factorise their representation. An identifier interval is of the following form:

$$(idBegin, end)$$

where

- $idBegin : Id$, is the first identifier of this interval.
- $end : Int$, is the offset of the last identifier of this interval.

We define:

- $idInterval.get(offset)$, the function which returns the identifier id from the interval $idInterval$ such as $offset = id.offset$.
- $idInterval.begin$, an alias for $idInterval.idBegin.offset$

Example

The identifier interval $idInterval$

$$([< pos, id_{site}, clock_{site}, 0 >], 5)$$

represents the following identifiers:

- $[< pos, id_{site}, clock_{site}, 0 >]$
- $[< pos, id_{site}, clock_{site}, 1 >]$
- $[< pos, id_{site}, clock_{site}, 2 >]$
- $[< pos, id_{site}, clock_{site}, 3 >]$
- $[< pos, id_{site}, clock_{site}, 4 >]$
- $[< pos, id_{site}, clock_{site}, 5 >]$

5.3 renameLocal

As previously, *renameLocal* allows us to reduce the size of the identifiers of the data structure. But in blockwise CRDTs, we can also take advantage of the renaming to merge existing blocks into one.

Algorithm

The following algorithm 7 describes how to perform the renaming of the current state :

We generate a new identifier interval $idInterval'$. We attached to this interval the whole content of the current state S, $view(S)$. The resulting block is then added to a blank state S'.

Finally, we replace the current state S by S'.

Algorithm 7 Local renaming algorithm

procedure RENAMELOCAL($S : State, id_{site} : Int, clock_{site} : Int$)
  $id' \leftarrow [< 0, id_{site}, clock_{site}, 0 >]$
  $idInterval' \leftarrow (id', S.length)$
  $S' \leftarrow State()$
  $S'.addBlock(idInterval', view(S))$
  $S \leftarrow S'$
end procedure

5.4 generateRenamingSequence

As previously, we need to share with other nodes a mapping between the existing identifier intervals to the new one. And, since the renaming is again deterministic, we are able to optimise the amount of data to broadcast.

However, since in this case we are using identifier intervals, we have to share additional data : the bounds of the interval *begin* and *end*.

Thus we will broadcast to other nodes a sequence composed of entries with the following form :

$$< id_{site}, clock_{site}, begin, end >$$

Example

Instead of broadcasting

$$[idInterval_1, idInterval_2, idInterval_3],$$

we only need to broadcast to other nodes

$$\begin{aligned}
&[(idInterval_1.id_{site}, idInterval_1.clock_{site}, idInterval_1.begin, idInterval_1.end),\\
&(idInterval_2.id_{site}, idInterval_2.clock_{site}, idInterval_2.begin, idInterval_2.end),\\
&(idInterval_3.id_{site}, idInterval_3.clock_{site}, idInterval_3.begin, idInterval_3.end)].
\end{aligned}$$

Algorithm

The following algorithm 8 describes how to generate such a sequence :

Algorithm 8 Blocks mapping algorithm

function GENERATERENAMINGSEQUENCE($S : State$):$< Int, Int, Int, Int > []$
  $seq \leftarrow []$
  for all $(idInterval, elt) \in S$ **do**
    $id \leftarrow idInterval.idBegin$
    $begin \leftarrow idInterval.begin$
    $end \leftarrow idInterval.end$
    $seq \leftarrow seq :: < id.id_{site}, id.clock_{site}, begin, end >$
  end for
end function

5.5 recomputeMapBlocks

When receiving the *renaming* operation, other nodes have first to recompute the blocks renaming map.

Algorithm

The following algorithm 9 describes how to do so:

Algorithm 9 Blocks mapping recomputation algorithm

function $\text{RECOMPUTEMAPBLOCKS}(S : State, seq :< Int, Int, Int, Int > [], id_{site} : Int, clock_{site} : Int)$: $Map < IdInterval, IdInterval >$
  $mapBlocks \leftarrow Map()$
  $ids \leftarrow S.map((idInterval, elts) \rightarrow idInterval.idBegin)$
  $ids \leftarrow ids :: L$
      $.filter(remoteOp \rightarrow remoteOp.isConcurrent(cc_{S'}))$
      $.map(remoteOp \rightarrow remoteOp.idInterval.idBegin)$
  $currentOffset \leftarrow 0$
  for all $(id_{site}, clock_{site}, begin, end) \in seq$ **do**
    $length \leftarrow end - begin + 1$
    $id \leftarrow ids.find(id \rightarrow id.id_{site} = id_{site}$ **and** $id.clock_{site} = clock_{site})$
    $id.offset \leftarrow begin$ ▷ The found occurrence may not have the correct offset, so we replace it
    $idInterval \leftarrow (id, end)$
    $id' \leftarrow [< 0, id_{site}, clock_{site}, currentOffset >]$
    $idInterval' \leftarrow (id', currentOffset + length - 1)$
    $mapBlocks.set(idInterval, idInterval')$
    $currentOffset \leftarrow currentOffset + length$
  end for
  return $mapBlocks$
end function

To recompute the blocks renaming map *mapBlocks* from the received sequence *seq*, we have to retrieve all the renamed identifier intervals given their id_{site}, $clock_{site}$, *begin* and *end*.

First, we generate the list *ids* of all potentially renamed identifiers. This list contains all identifiers from the current state S but also the identifiers of concurrent operations to the renaming, since some renamed identifiers may have been deleted concurrently.

Then using this list, we can for each entry of *seq* find the original identifier *id* and compute its new one id'. We can then recreate the original interval *idInterval* using *id*, *begin* and *end* and generate its image $idInterval'$.

Finally, we add the mapping $idInterval \rightarrow idInterval'$ to *mapBlocks*.

5.6 renameRemote

Once *mapBlocks* is recomputed, we can proceed to the renaming of the current state S.

As previously, some differences may exist between the current state and the state S' on which the renaming operation has been performed since some concurrent operations may have been applied to current state.

Thus we need to be able to also rename deterministically the concurrently inserted identifiers intervals.

Algorithm

The following algorithm 10 describes how to perform the renaming:

Algorithm 10 Remote renaming algorithm

---

procedure RENAMEREMOTE(S : $State, mapBlocks$: $Map < IdInterval, IdInterval >, cc_{S'}$: $StateVector$)
  $S' \leftarrow State()$
  $mapBlocks' \leftarrow mapBlocks$
  for all $remoteOp \in concurrentOps(cc_{S'}, L)$ **do** ▷ Get concurrent operations to the renaming
    $idInterval \leftarrow remoteOp.idInterval$
    if $idInterval \not\subset mapBlocks'$ **then**
      $id \leftarrow idInterval.idBegin$
      $prevIdInterval \leftarrow prev(id, mapBlocks)$ ▷ Get interval containing predecessor of id
      $prevId \leftarrow prev(id, prevIdInterval)$ ▷ Get predecessor of id in $prevIdInterval$
      $prevIdInterval' \leftarrow mapBlocks.get(prevIdInterval)$
      $offset \leftarrow prevId.offset - prevIdInterval.begin$
      $offset' \leftarrow prevIdInterval'.begin + offset$
      $prevId' \leftarrow prevIdInterval'.get(offset')$
      $id' \leftarrow prevId' :: id$ ▷ Generate new identifier by concatenating $prevId'$ and id
      $idInterval' \leftarrow (id', idInterval.end)$
      $mapBlocks'.set(idInterval, idInterval')$
    end if
  end for
  for all $(idInterval, elts) \in S$ **do**
    $idInterval' \leftarrow mapBlocks.get(idInterval)$
    $S'.addBlock(idInterval', elts)$
  end for
  $S \leftarrow S'$
end procedure

We start by completing $mapBlocks'$ with the identifier intervals concurrently inserted to the renaming. To do so, we look for identifier intervals which are not already present in $mapBlocks'$. For each interval, we compute id', the new identifier corresponding to $idInterval.idBegin$.

To generate id', we retrieve $prevId$, the predecessor of id in $mapBlocks$. Since every node observes the same id and $mapBlocks$ and thus $prevId$, we can use them to generate id' in a deterministic manner : we get it by concatenating $prevId'$ and id. This allows us to generate deterministically a new identifier id' and also to preserve the order relation with other identifiers. Once we generated id', we can generate $idInterval'$ by generating the interval from id' to $idInterval.end$.

Once $mapBlocks'$ is completed, we browse the current state S. For each block, we insert the corresponding block in S'. Once all elements have been added to S', we replace the current state S by the new one S'.

6 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
- 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
- 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