Efficient Renaming in Sequence CRDTs

Matthieu Nicolas
matthieu.nicolas@loria.fr
Université de Lorraine, CNRS, Inria, LORIA, F-54500
Nancy, France

Gérald Oster
gerald.oster@loria.fr
Université de Lorraine, CNRS, Inria, LORIA, F-54500
Nancy, France

Olivier Perrin
olivier.perrin@loria.fr
Université de Lorraine, CNRS, Inria, LORIA, F-54500
Nancy, France

Abstract

Keywords CRDTs, real-time collaborative editing, eventual consistency, memory-wise optimisation, performance

1 Introduction

2 Background

2.1 LogootSplit

2.2 Limits

3 Overview

3.1 Proposed approach

We propose a new Sequence Conflict-free Replicated Data Type (CRDT) belonging to the variable-size identifiers approach : *RenamableLogootSplit* (RLS).

This new CRDT associates to LogootSplit a renaming mechanism. The goal of this mechanism is to overcome LogootSplit evergrowing memory overhead. To this end, the mechanism reassigns shorter identifiers to elements and aggregates them into fewer blocks in a fully distributed manner.

We describe the behavior of the mechanism in section 4.

3.2 System Model

4 RenamableLogootSplit

4.1 *rename* operation

- May encounter concurrent *rename* operations in this setting
- The *rename* operation is not commutative with itself
- Deal with a conflict in case of concurrent *rename* operations
- No user intention attached to *rename* operations, operations behind the scene
- Can actually solve the conflict by arbitrary deciding with which operation to continue
  - Call the operation with which nodes continue the *winning* one *TODO: trouver un meilleur terme, "primary" ? – Matthieu*
  - Call the others the *losing* ones *TODO: trouver un meilleur terme, "secondary" ? – Matthieu*

TODO: décrire le comportement de l'opération de renommage – Matthieu

TODO: diriger le lecteur vers le papier pour PaPoC pour la gestion des opés "simples" concurrentes – Matthieu

As no coordination is enforced between nodes, several of them may concurrently rename their respective states. However, the proposed *rename* operation is not commutative with itself. Applying concurrent *rename* operations in different orders to nodes, according to the order in which they each receive the operations, would result in diverging states. Nodes therefore encounter a conflict when dealing with several *rename* operations concurrently issued.

To ensure that nodes eventually converge, they have to solve this conflict. Notice that *rename* operations are system operations : they have no impact on the content of the document and have no user intention attached. Nodes may thus solve the conflict by designating collegially one *rename* operation with which to proceed. We call the *rename* operation with which nodes continue the *primary* one. Others from the set of concurrent *rename* operations are called *secondary* ones.

In subsection 4.2, we present how nodes can select the *primary rename* operation from a set of *concurrent* ones in a coordination-free manner.

4.2 Breaking tie between concurrent *rename* operations

- Define *priority* between concurrent *rename* operations to make this decision
- A (total?) order relation between epochs *NOTE: le* total *dépend si on considère qu'on utilise* priority *seulement pour trancher entre epochs concurrentes ou si on l'utilise aussi pour des epochs causalement liées – Matthieu*
- May actually choose various strategies to define this relation
- In this work, use lexicographical order as a tiebreaker between conflicting operations
- But new strategies could be designed, for example based on metrics representing the accumulated work embodied by operations. This topic will be further discussed in subsection 6.2

4.3 Reverting *rename* operations

- Nodes may have applied losing *rename* operations
- Have to revert the effects of losing operations before applying the winning one to ensure convergence
- Designed a dedicated function : *reverseRenameId()*
- Its goals are the following:

Algorithm 1 Rename identifier

```
function renId(id, renamedIds, nId, nSeq)
▷ id is the identifier to rename
▷ renamedIds is the former state shared by the rename op
▷ nId is node id of the node which issued the rename op
▷ nSeq is node seq of the node which issued the rename op

    length ← renamedIds.length
    firstId ← renamedIds[0]
    lastId ← renamedIds[length − 1]
    pos ← getPosition(firstId)

    if id < firstId then
        newFirstId ← new Id(pos, nId, nSeq, 0)
        return renIdLessThanFirstId(id, firstId, newFirstId)
    else if id ∈ renameIds then
        index ← findIndex(id, renamedIds)
        return renIdFromIndex(pos, nId, nSeq, index)
    else if lastId < id then
        newLastId ← new Id(pos, nId, nSeq, length − 1)
        return renIdGreaterThanLastId(id, lastId, newLastId)
    else
        return renIdfromPredId(id, renamedIds)
    end if
end function

function renIdfromPredId(id, renamedIds)
    index ← findIndexOfPred(id, renamedIds)
    predId ← renamedIds[index]
    newPredId ← new Id(pos, nId, nSeq, index)

    if predId.length + 1 < id.length then
        prefix ← concat(predId, MIN_TUPLE)
        tail ← getTail(id, prefix.length)
        if isPrefix(prefix, id) and tail < predId then
            return concat(newPredId, tail)
        end if
    end if

    succId ← renamedIds[index + 1]
    if succId.length + 1 < id.length then
        offset ← getLastOffset(succId) − 1
        predOfSuccId ← createIdFromBase(succId, offset)
        prefix ← concat(predOfSuccId, MAX_TUPLE)
        tail ← getTail(id, prefix.length)
        if isPrefix(prefix, id) and succId < tail then
            return concat(newPredId, tail)
        end if
    end if

    return concat(newPredId, id)
end function
```

Algorithm 2 Reverse rename identifier

```
function revRenId(id, renamedIds, nId, nSeq)
▷ id is the identifier to reverse rename
▷ renamedIds is the former state shared by the rename op
▷ nId is node id of the node which issued the rename op
▷ nSeq is node seq of the node which issued the rename op

    length ← renamedIds.length
    firstId ← renamedIds[0]
    lastId ← renamedIds[length − 1]
    pos ← getPosition(firstId)

    predOfNewFirstId ← newId(pos, nId, nSeq, −1)
    newLastId ← newId(pos, nId, nSeq, length − 1)

    if id < newFirstId then
        return revRenIdLessThanNewFirstId(id, firstId, newFirstId)
    else if isRenamedId(id, pos, nId, nSeq, length) then
        index ← getFirstOffset(id)
        return renamedIds[index]
    else if newLastId < id then
        return revRenIdGreaterThanNewLastId(id, lastId)
    else
        index ← getFirstOffset(id)
        return revRenIdfromPredId(id, renamedIds, index)
    end if
end function

function revRenIdfromPredId(id, renamedIds, index)
    predId ← renamedIds[index]
    succId ← renamedIds[index + 1]
    tail ← getTail(id, 1)

    if tail < predId then
        return concat(predId, MIN_TUPLE, tail)
    else if succId < tail then
        offset ← getLastOffset(succId) − 1
        predOfSuccId ← createIdFromBase(succId, offset)
        return concat(predOfSuccId, MAX_TUPLE, tail)
    else
        return tail
    end if
end function
```

- Revert ids to their former value, for ids generated before or concurrently to the applied *rename* operation
- Generate new values complying with the intended order for ids generated after the applied *rename* operation

4.4 Garbage collection of *former states*

- Nodes have to store epochs and corresponding *former states* to transform operations from concurrent or previous epochs to the current one
- Epochs and *former states* can thus be garbage collected once they are not needed anymore
- An epoch can be safely garbage collected once
  1. The given epoch e is a leaf and a concurrent and primary epoch e' is causally stable
  2. The given epoch e is the root of the epoch tree, has only one child e' and e' is causally stable

5 Evaluation

5.1 Simulations and benchmarks

5.2 Results

Convergence

- Verified that nodes reach the same final state
- Did not spot any divergence in our results
- While it is an empirical result, not a proof...
- ... it provides some confidence in our algorithms

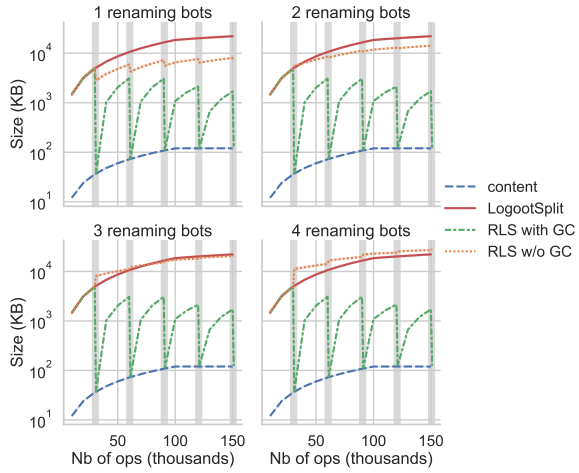


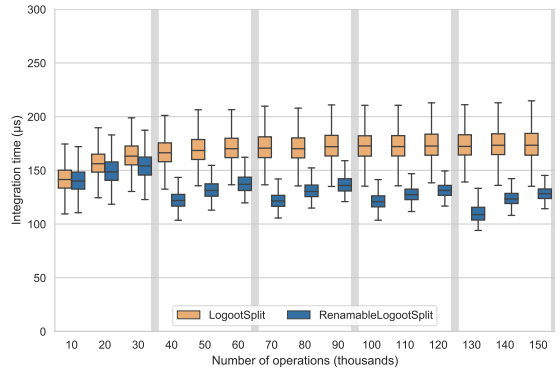
Figure 1. Evolution of the size of the document

Memory overhead

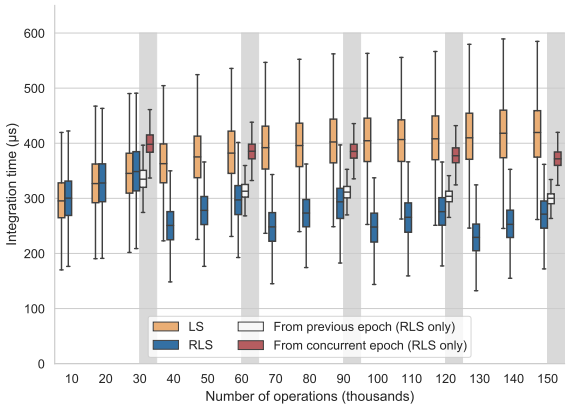
- Display in Figure 1 the evolution of the size of the document throughout its lifetime
- Compare the obtained results according to the number of *renaming bots*, i.e. the number of bots authorized to issue *rename* operations
- For each diagram, present 4 different data
- Blue dashed line represents the size of the content
- Red line represents the size of the LogootSplit document
- Green dashed-dotted line represents RenamableLogootSplit best-case scenario. In this scenario, nodes assume that *rename* operations become causally stable as soon as nodes received them. Nodes are able to garbage collect metadata introduced by the renaming mechanism, such as the *former states*, instantaneously
- Orange dotted line represents RenamableLogootSplit worst-case scenario. In this scenario, nodes assume that *rename* operations never become causally stable. Nodes have thus to store renaming mechanism metadata indefinitely
- Observe that RenamableLogootSplit is able to dispose of its overhead eventually, since overhead is garbage collected as *rename* operations become causally stable. And this result is independent of the number of *renaming bots*.
- Observe that RenamableLogootSplit still outperforms LogootSplit in its worst-case scenario while the number *renaming bots* remains low (1 or 2). This result can be explained by the fact that the renaming mechanism enable us to scrap as well the overhead of the data structure used in LogootSplit to represents the sequence.
- But as the number of concurrent *rename* operations increases, the performances of RenamableLogootSplit decreases as the number of *former states* that nodes have to store to transform operations expand
- So a greater number of *renaming bots* may lead to a temporary expanded overhead, but which eventually subsides once causal stability is achieved.
- In subsection 6.1, we discuss that *former states* may be offloaded until causal stability is achieved to address the temporary memory overhead

Integration times of standard operations

- In Figure 2, compare the evolution of integration time of respectively local and remote operations on LogootSplit and RenamableLogootSplit documents
- Orange boxplots correspond to times on LogootSplit documents while blue ones correspond to times on RenamableLogootSplit documents
- Observe that integration times are faster on RenamableLogootSplit, as *rename* operations improve the internal representation of the sequence
- In Figure 2b, also measure the integration times of remote operations from previous epochs, displayed in white, and of operations from concurrent epochs, displayed in red
- Observe a negligible overhead for operations from previous epochs compared to remote operations from the same epochs, as nodes have to rename them beforehand. But still outperforms LogootSplit
- Observe an additional overhead for operations from concurrent epochs, as nodes have to reverse the effect of the concurrent epoch first. Achieve performances comparable to LogootSplit ones in this worst-case scenario



(a) Local operations



(b) Remote operations

Figure 2. Integration time of standard operations

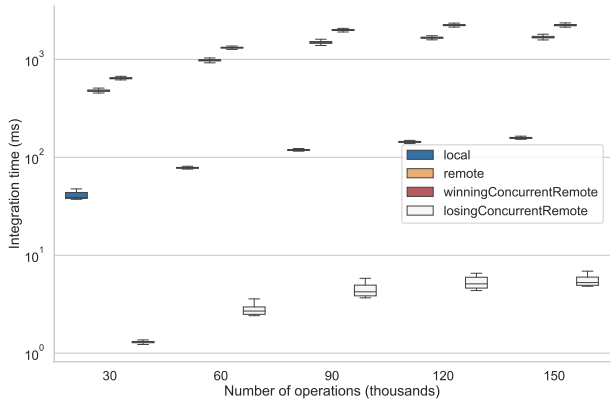


Figure 3. Integration time of rename operations

Integration time of* rename *operation

- In figure Figure 3, display integration times of the different kinds of *rename* operations
- Main result is that *rename* operations are generally expensive compared to other operations. Local *rename* operations, in blue, takes hundred of milliseconds while remote ones, in orange, may reach seconds if delayed for too long. Should design the strategy to trigger *rename* operations according to this result to prevent a negative impact on the user experience
- Another interesting result is that, while *winning rename* operations are expensive to integrate, *losing* ones are cheap. Can thus significantly reduce computations by integrating concurrent *rename* operations in correct order. Will discuss this topic in subsection 6.3

6 Discussion

6.1 Offloading on disk unused *former states*

- *Former states* are only needed to transform operations from previous or concurrent epochs
- May receive these kind of operations in 2 cases : *rename* operations are being issued or nodes (re)joined the collaboration
- Between these events, *former states* won't actually be needed
- Can offload *former states* on disk to reduce the memory overhead until causal stability is achieved, without impacting much performances

6.2 Designing more effective *priority* relation

- While simple and ensuring convergence, the *priority* relation designed and used in this paper introduces a significant computational overhead in some cases
- For example a single node, disjoined from the collaboration for a long time, may force every other nodes to revert *rename* operations they issued meanwhile because of its own primary *rename* operation
- Should define a *priority* which aims to reduce the global amount of computations of the system, while still ensuring convergence
- To this end, could integrate some metrics representing the work done beforehand in *rename* operations
- And build a new *priority* relation based on these metrics

6.3 Postponing transition to new epoch in case of high concurrency

- Primary remote *rename* operations are expensive to integrate as nodes have to browse and rename their whole current state in the process
- It can introduce a significant computational overhead in some cases
- For example a node may receive concurrent *rename* operations in the reverse order to the one set by the *priority* relation
- The node would then consider each operation as the primary one and rename its state in a successive manner
- On the other hand, secondary remote *rename* operations are cheap to integrate as nodes simply add to

their state a reference to the corresponding *former state*

- To reduce the likelihood and the negative impact of the scenario described previously, we can decompose the integration of *rename* operations into two parts in case of concurrency detection
- Nodes first process *rename* operations as secondary ones. It enables nodes to integrate remote *insert* and *remove* operations, even from concurrent epochs, by transforming them
- Then once nodes obtain a given amount of confidence that one *rename* operation is the primary one, proceed to the renaming of their states
- This strategy introduces a slight overhead for each *insert* or *remove* operation received during this period, but reduces the probability of erroneously integrating *rename* operations as primary ones

Algorithm 3 Remaining functions to rename identifier

```
function renIdFromIndex(pos, nId, nSeq, index)
    return newId(pos, nId, nSeq, index)
end function

function renIdLessThanFirstId(id, firstId, newFirstId)
    offset ← getLastOffset(firstId) − 1
    predOfFirstId ← createIdFromBase(firstId, offset)
    prefix ← concat(predOfFirstId, MAX_TUPLE)
    predNewFirstId ← createIdFromBase(newFirstId, −1)
    if isPrefix(prefix, id) then
        tail ← getTail(id, prefix.length)
        return concat(predNewFirstId, tail)
    else if id < newFirstId then
        return id
    else
        return concat(predNewFirstId, id)
    end if
end function

function renIdGreaterThanLastId(id, lastId, newLastId)
    prefix ← concat(lastId, MIN_TUPLE)
    if isPrefix(prefix, id) then
        tail ← getTail(id, prefix.length)
        return tail
    else if newLastId < id then
        return id
    else
        ▷ lastId < id < newLastId
        return concat(newLastId, id)
    end if
end function
```

Algorithm 4 Remaining functions to reverse rename identifier

```
function revRenIdLessThanNewFirstId(id, firstId, newFirstId)
    predNewFirstId ← createIdFromBase(newFirstId, −1)
    if predNewFirstId < id then
        tail ← getTail(id, 1)
        if tail < firstId then
            return tail
        else
            offset ← getLastOffset(firstId)
            predFirstId ← createIdFromBase(firstId, offset)
            return concat(predFirstId, MAX_TUPLE, tail)
        end if
    else
        return id
    end if
end function

function revRenIdGreaterThanNewLastId(id, lastId)
    if id < lastId then
        return concat(lastId, MIN_TUPLE, id)
    else
        return id
    end if
end function
```