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

TODO: Reprendre la description du system model *de PaPoC – Matthieu*

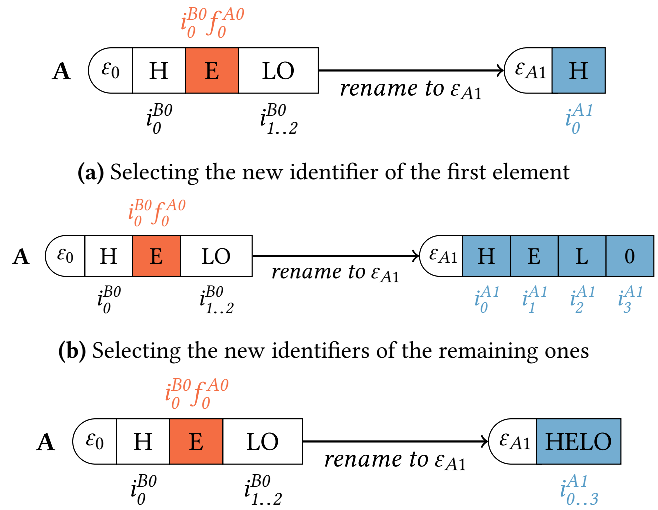
4 RenamableLogootSplit

4.1 *rename* operation

To enable nodes to reassign shorter identifiers to elements and to aggregate them into fewer blocks, RenamableLogootSplit has a *rename* operation. We present in Figure 1 an example of its behavior.

In this example, node A performs a renaming on its current state. As illustrated in Figure 1a, Node A first proceeds to assign a new identifier to the first element of the sequence (H). It generates this new identifier (i_0^{A1}) by reusing the position of the current first identifier (i) and using its own node id (**A**) and current sequence number (1). The offset of this new identifier is set to 0.

Then node A derives identifiers for all remaining elements by successively incrementing the offset (i_1^{A1}, i_2^{A1} and i_3^{A1}), as shown in Figure 1b. Since resulting identifiers are consecutive, elements can be aggregated into one block as illustrated in Figure 1c. It enables the *rename* operation to effectively



(a) Selecting the new identifier of the first element

(b) Selecting the new identifiers of the remaining ones

(c) Final state obtained

Figure 1. Renaming the sequence on node A

minimize the memory overhead of the resulting replicated sequence.

Identifiers are used to position elements relatively to each others. Since the *rename* operation enables to reassign identifiers to elements, this operation can be considered as a change of reference frames. To denotes different frames of reference, we use *epochs*. Initally, the replicated sequence starts at the *origin* epoch noted ε_0. Each *rename* operations introduces a new epoch and enables nodes to advance their states to it from the previous epoch. The generated epoch is characterized using the node id and its current sequence number upon the generation of the *rename* operation. For example, the *rename* operation described in Figure 1 enables nodes to advance their states from ε_0 to ε_{A1}.

Nodes tag every operation with their current epoch at the time the operation is generated. Upon the reception of *insert* and *remove* operations from former epochs, nodes can not naively apply them to their states at it would lead to inconsistencies. Beforehand, nodes have to transform the operations against the effect of concurrently applied *rename* operations.

To this end, nodes use Algorithm 1. This algorithm maps identifiers from the previous epoch to corresponding ones in the new epoch. To do so, this algorithm relies on the *former* *state*, the element identifiers of the state upon which the *rename* operation was generated. More details about the transformation of concurrent *insert* and *remove* operations against *rename* operations can be found in [1].

Nodes also use Algorithm 1 to apply *rename* operations issued by others. It thus requires to propagate *former states* by embedding them into their respective *rename* operations.

Furthermore, as no coordination is enforced between nodes, several of them may also 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 they eventually converge, nodes 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

We define *priority*, a total order relation between epochs. This relation enables nodes to designate the *primary rename* operation from a set of concurrent ones, but also the current epoch from any set of *rename* operations.

To define the *priority* relation, we may actually choose different strategies. In this work, we use the lexicographical order on the epoch identifiers as the *priority* relation. An epoch identifier is obtained by concatening the *node id* and *node sequence number* of the author of the *rename* operation to the current epoch identifier at the time.

Other strategies could be proposed to define the *priority* relation. For example, *priority* could rely on metrics embedded in *rename* operations representing the accumulated work on the document. This topic will be further discussed in subsection 6.2.

4.3 Applying *primary rename* operations

Upon the reception of a *rename* operation, nodes first have to determine if its introduce the new *primary* epoch. To do so, they compare their current epoch to the new one using the *priority* relation. If the new epoch is indeed the new *primary* one, nodes have then to determine if they applied concurrent and conflicting *rename* operations previously. To this end, nodes use the *epoch tree*.

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
```

Epochs are introduced by *rename* operations, themselves issued from given epochs. We can thus establish the *parent* relation between epochs. Using this relation, nodes are each able to build the *epoch tree*.

Using the *epoch tree*, nodes are able to determine if their current epoch is the *parent* of the new *primary* one. If that is not the case, it means that they applied one or several concurrent *rename* operations.

In order to switch to the new *primary* epoch, nodes first revert the effects of these concurrent *rename* operations. To establish which operations to revert, nodes identify the Lowest Common Ancestor (LCA) of their current epoch and of the new *primary* one.

4.4 Reverting *rename* operations

Nodes have to revert *rename* operations applied since the LCA epoch between their current epoch and the new primary one, in the reverse order. To do so, nodes use Algorithm 2.

The goals of Algorithm 2 are the following : (i) To revert identifiers generated causally before or concurrently to the reverted *rename* operation to their former value (ii) To assign new ids complying with the intended order to elements inserted causally after the reverted *rename* operation. The Figure 2 illustrates an example of its usage.

This example describes the following scenario : in Figure 2a, node A and node B concurrently issue *rename* operations. In Figure 2b, node A receives node B's *rename* operation. Upon its reception, node A compares its current epoch (ε_{A1}) to the new epoch introduced (ε_{B2}). As $A < B$, node A deems ε_{B2} as the new *primary* one. To switch to this new *primary* epoch, node A has to revert the effect of its *rename* operation first.

To this end, node A uses Algorithm 2 to retrieve fitting counterparts for every identifiers of its current state. For identifiers of the form i^{A1}_{offset}, it simply uses their offset to retrieve the original identifiers, as offsets correspond to the identifier indexes in *renamedIds*. For other identifiers such as $i^{A1}_{1} i^{B0}_{0} m^{B1}_{0}$, Algorithm 2 removes its prefix (i^{A1}_{1}) to isolate its tail ($i^{B0}_{0} m^{B1}_{0}$). The algorithm returns the tail if it fits between the identifier of its predecessor ($i^{B0}_{0} f^{A0}_{0} < i^{B0}_{0} m^{B1}_{0}$) and the identifier of its successor ($i^{B0}_{0} m^{B1}_{0} < i^{B0}_{1}$). If it would not, Algorithm 2 would use exclusive tuples of the renaming mechanism, *MIN_TUPLE* and *MAX_TUPLE*, to generate an identifier complying with the intended order.

Once node A reverted its state to the LCA epoch (ε_0) using Algorithm 2, it can successively apply *rename* operations leading to the new *primary* epoch (ε_{B2}) using Algorithm 1, as illustrated in Figure 2c.

4.5 Garbage collection of *former states*

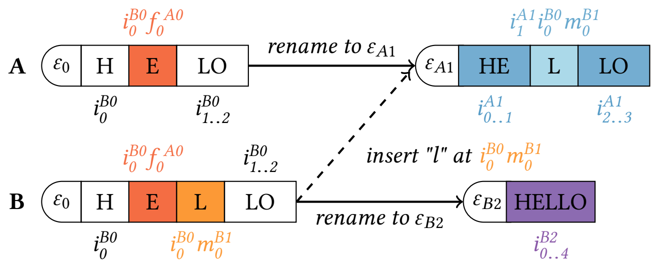
As explained in [1] and subsection 4.4, nodes have to store epochs and corresponding *former states* to transform operations from previous or concurrent epochs to the current one, or to rename their state to switch to a new *primary* epoch.

Algorithm 2 Revert rename identifier

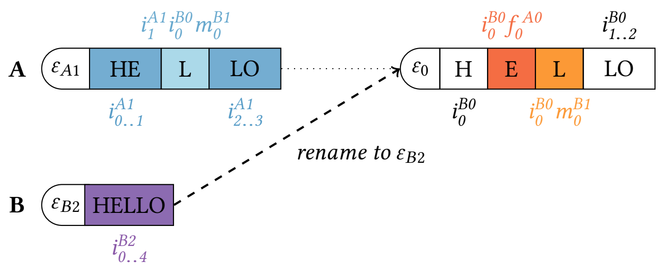
```
function revRenId(id, renamedIds, nId, nSeq)
▷ id is the identifier to reverse rename
▷ renamedIds is the former state shared by the rename op
▷ nId is node id of the node which issued the rename op
▷ nSeq is node seq of the node which issued the rename op

    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
```

However, once nodes become aware that some epochs can not possibly be required anymore to apply future operations, they can garbage collect these epochs and their corresponding *rename* operations. We propose the two following rules to enable nodes to identify unnecessary epochs :

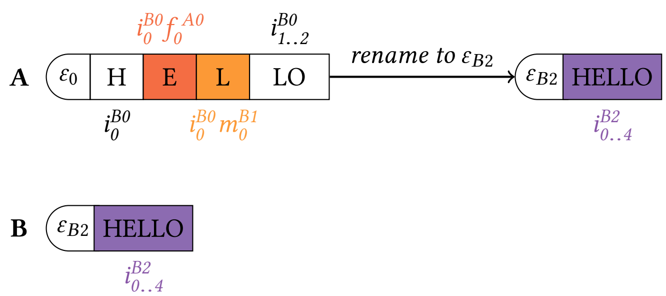
Rule 1. *An epoch ε can be garbage collected if ε is a leaf of the epoch tree and a concurrent primary epoch ε' is causally stable.*



(a) Node A and node B generating concurrent *rename* operations



(b) Node A reverting the effect of the *rename* operation it applied upon the reception of a new *primary* one



(c) Node A then applying the *primary rename* operation on its reverted state

Figure 2. Applying *primary rename* operation on node *A*

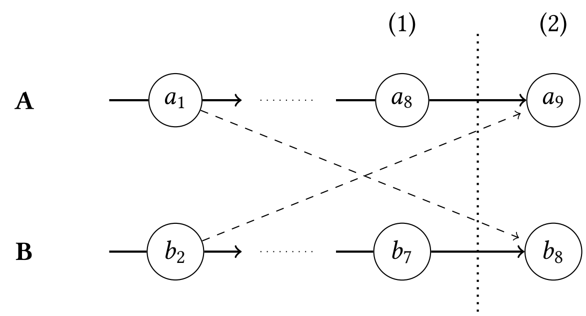
Rule 2. *An epoch ε can be garbage collected if ε is the root of the epoch tree, has only one child ε' and that ε' is causally stable.*

Figure 3 illustrates a use case of Rule 1 and Rule 2. In Figure 3a, we represent an execution in which two nodes A and B respectively issue two *rename* operation before eventually synchronising. In Figure 3b, we represent the states of their respective *epoch trees* once they each generate their *rename* operations. In Figure 3c, we represent the new states of their *epoch trees* once they each receive the first *rename* operation issued by each other.

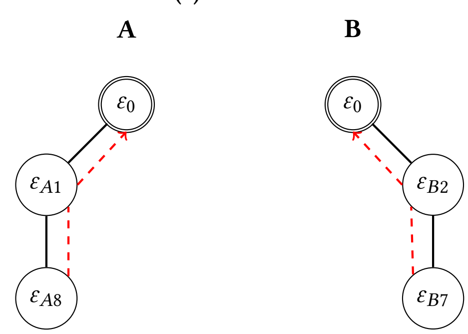
Upon the delivery of the *rename* operation introducing epoch ε_{B2} to node A, ε_{B2} becomes causally stable. From this point, node A knows that every node switched to this epoch at least. Therefore nodes can no longer issue operations from ε_{e0}, ε_{A1} or ε_{A8}. Thus these epochs and the *rename* operations enabling nodes to switch between them can now be garbage collected. Rule 1 enables node A to garbage collect epochs ε_{A8} then ε_{A1}. Then Rule 2 enables node A to garbage collect ε_0 and the *renaming* operation to switch to ε_{B2}.

On the other hand, node B can not garbage collect any epochs despite ε_{A1} being causally stable. Indeed, from its point of view, other nodes may still issue operations from epoch ε_{A1}. Since in that case node B would have to transform operations to apply them to ε_{B7}, node B has to retain all epochs forming the path between ε_{A1} and ε_{B7} and their corresponding *rename* operations.

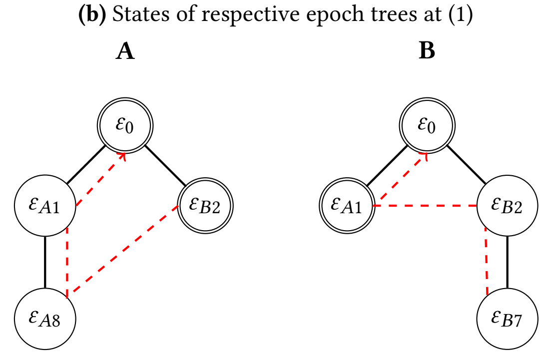
Eventually, once the system becomes idle, the current *primary* epoch will become causally stable. Nodes will then be able to garbage collect all other epochs using Rule 1 and Rule 2, effectively suppressing the overhead of the renaming mechanism.



(a) Execution



(b) States of respective epoch trees at (1)



(c) States of respective epoch trees at (2)

Figure 3. Garbage collecting epochs and corresponding *former states*

TODO: Marquer comme GC ε_{A1} et ε_{A8} via Rule 1 et ε_0 via Rule 2 de l'epoch tree de A dans la Figure 3c – Matthieu

5 Evaluation

5.1 Simulations and benchmarks

In order to validate the proposed approach, we proceed to an experimental evaluation. The aims of this evaluation are to measure (i) the memory overhead of the replicated sequence (ii) the computational overhead added to *insert* and *remove* operations by the renaming mechanism (iii) the cost of integrating *rename* operations.

Unfortunately, we were not able to retrieve an existing dataset of real-time collaborative editing sessions. We thus setup simulations to generate the dataset on which run our benchmarks. These simulations mimic the scenario below.

Several authors collaboratively write an article in real-time. First of all, the authors mainly specify the content of the article. Few *remove* operations are issued in order to simulate spelling mistakes. Once the document reaches a arbitrary given critical length, collaborators move on to the second phase of the simulation. During this second phase, authors stop adding new content but instead focus on revamping existing parts. This is simulated by balancing the ratio between *insert* and *remove* operations. Every author has to issue a given number of *insert* and *remove* operations. The simulation ends once every collaborators received all operations. During the simulation, we take snapshots of the replicas' state at given steps to follow their evolution.

We ran simulations with the following experimental settings: we deployed 10 bots as separate Docker containers on a single workstation. Each container corresponds to a single mono-threaded Node.js process simulating an author. Bots share and edit collaboratively the document using either LogootSplit or RenamableLogootSplit according to the session. In both cases, each bot performs an *insert* or a *remove* operation locally every 200 ± 50ms and broadcast it immediately to other nodes using a Peer-to-Peer (P2P) full mesh network. During the first phase, the probabilities of issuing *insert* and *remove* operations are respectively of 80% and 20%. Once the document reaches 60k characters (around 15 pages), bots switch to the second phase and set both probabilities to 50%. After each local operation, the bot may move its cursor to another random position in the document with a probability of 5%. Every bot generates 15k *insert* or *remove* operations and stops once it observed 150k operations. Snapshots of the state of bot are taken periodically, every 10k observed operations.

Additionally, in the case of RenamableLogootSplit, from 1 to 4 bots are arbitrarily designated as *renaming bots* according to the session. *Renaming bots* issue *rename* operations every time they observe 30k operations overall. These *rename* operations are generated in a way ensuring that they are concurrent.

Code, benchmarks and results are available at: https://github.com/coast-team/mute-bot-random/.

5.2 Results

Convergence We first proceeded to verify the convergence of nodes states at the end of simulations. For each simulation, we compared the final state of every nodes using their respective snapshots. We were able to confirm that nodes eventually converged without any communication other than operations, thus satisfying the Strong Eventual Consistency (SEC) consistency model.

This result sets a first milestone in the validation of the correctness of RenamableLogootSplit. It is however only empirical. Further work to formally prove its correctness should be undertaken.

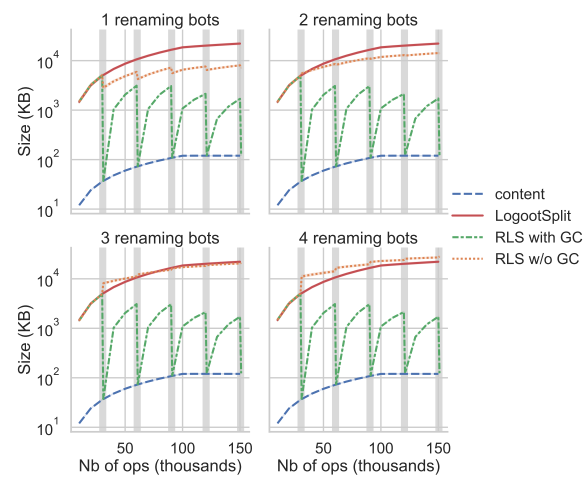


Figure 4. Evolution of the size of the document

Memory overhead

- Display in Figure 4 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

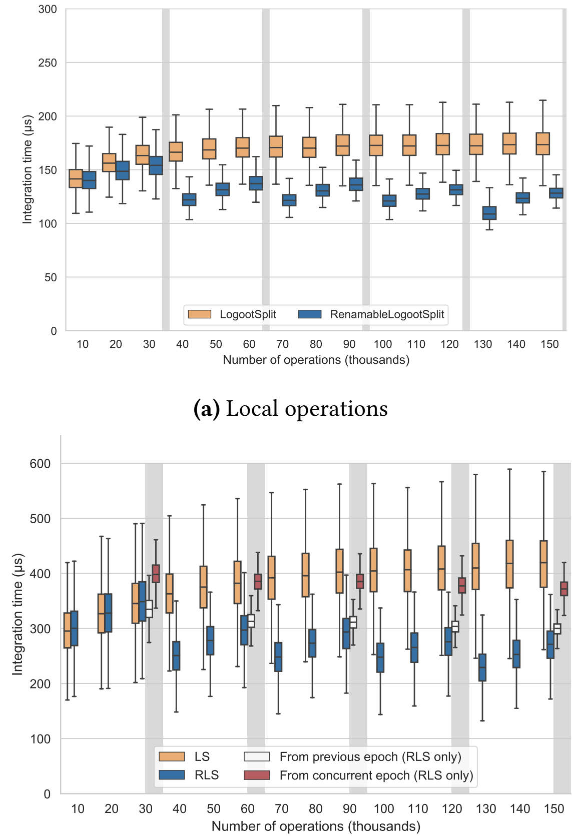
- Observe that RenamableLogootSplit is able to dispose of its overhead eventually, since overhead is garbage collected as *rename* operations become causally stable. And this result is independent of the number of *renaming bots*.
- Observe that RenamableLogootSplit still outperforms LogootSplit in its 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 5, 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 5b, 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

Integration time of rename *operation*

- In figure Figure 6, 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



(a) Local operations

(b) Remote operations

Figure 5. Integration time of standard operations

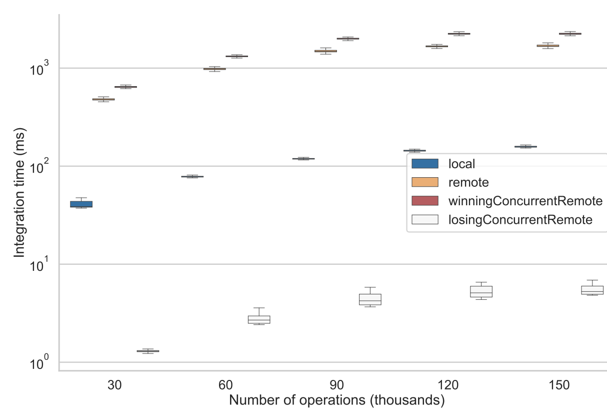


Figure 6. Integration time of rename operations

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

7 Related work

7.1 The core-nebula approach

7.2 The LSEQ approach

8 Conclusions and future work

A Algorithms

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 revert identifier renaming

```
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
```

References

[1] Matthieu Nicolas, Gérald Oster, and Olivier Perrin. Efficient Renaming in Sequence CRDTs. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'20)*, Heraklion, Greece, April 2020. URL https://hal.inria.fr/hal-02526724.