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

Two main approaches were proposed to design efficient Sequence Conflict-free Replicated Data Types (CRDTs). Both approaches aim to ensure the Strong Eventual Consistency (SEC) property while minimising the required overhead. To do so, these approaches assign identifiers to elements stored in the sequence. Nonetheless, they differ by the design and purpose of these identifiers.

The first approach [1–3] is the fixed size identifiers approach. This approach use identifiers when inserting new elements to specify their predecessor. However this approach requires nodes to keep tombstones for removed elements in order to deal with operations concurrent to the *remove* ones. The metadata overhead of such Sequences thus monotonically increases with each element inserted.

The second approach [4–6] is the variable size identifiers approach. This approach use identifiers to define a dense total order on elements. It enables nodes to delete safely elements and their identifier as soon as they are removed. However, it requires variable size identifiers to be able to generate new identifiers between two existing ones. The metadata overhead of such Sequences thus increases, as the size of such dense totally ordered identifiers grows over time.

In the context of this paper, we focus on the later approach.

2.1 RenamableLogootSplit

- State of the art of the variable size identifiers approach
- An extension of LogootSplit
- Assign identifiers made of one or several tuples of the following form : $position_{offset}^{node\ id\ node\ seq}$ to order elements relatively to each others
- Group elements with *contiguous* identifiers into blocks to store them effectively
- Aims to reduce LogootSplit's evergrowing metadata overhead periodically
- To do so, propose a *rename* operation
- Enables nodes to reassign shorter and contiguous identifiers to elements
- Resulting identifiers are designed in such way to enable to aggregate elements into one only block, minimising the overhead
- To deal with concurrent *insert* and *remove* operations, propose to use Operational Transformation (OT) techniques to transform these operations against the effect of the *rename* ones.

2.2 Limits

- Two limits
- First, nodes have to store the state on which the *rename* operation took place, called *former state*
- Is required to transform concurrent *insert* and *remove* operations against the effect of the *rename* operation
- Can be safely garbage-collected once *rename* operation is causally stable and not more concurrent operations can be safely issued by nodes *NOTE: Aborder ce point dans Limits ou comme on ne s'attaque pas à cet aspect dans cette nouvelle version de RLS, ça ne sert pas à grand chose d'en parler ici ? – Matthieu*
- Second, designed *Rename* operation is not commutative with itself
- Encounters a conflict if nodes generates concurrent *rename* operations
- Assume that nodes synchronise to issue *rename* operations, for example using a consensus protocol
- Actually not suitable to large scale distributed systems as consensus protocols do not scale

3 Overview

3.1 Proposed approach

- Propose a new version of RenamableLogootSplit
- Introduce additional mechanisms to enable nodes to resolve conflicts in case of concurrent *rename* operations
- Rework in consequence algorithms used to rename identifiers and to define as garbage-collectable *former states*
- Describe in details these additions in section 4

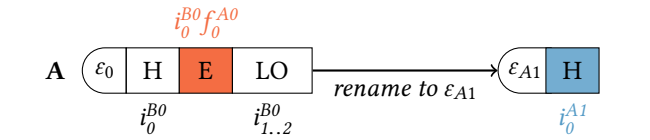
3.2 System Model

TODO: Reprendre la description du system model *de PaPoC, juste préciser que maintenant le renommage peut se faire sans coordination – Matthieu*

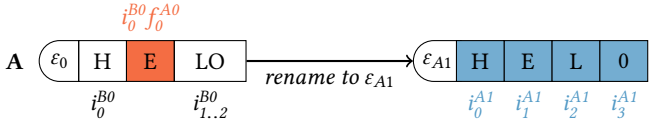
4 RenamableLogootSplit

4.1 *rename* operation

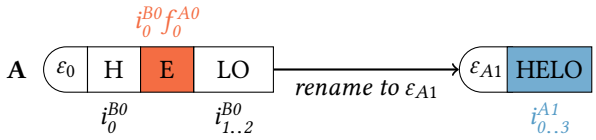
To enable nodes to reassign shorter identifiers to elements and to aggregate them into fewer blocks, RenamableLogootSplit has a *rename* operation. We present in Figure 1 an example of its behavior.



(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*

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 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 [7].

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

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
```

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*.

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.

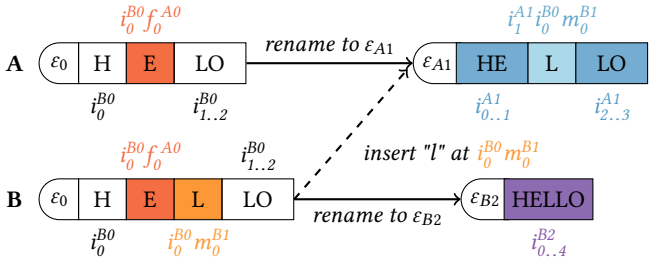
To this end, node A uses Algorithm 2 to retrieve fitting counterparts for every identifiers of its current state. For identifiers of the form i^{A1}_{offset}, it simply uses their offset to retrieve the original identifiers, as offsets correspond to the identifier indexes in *renamedIds*. For other identifiers 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

Algorithm 2 Revert rename identifier

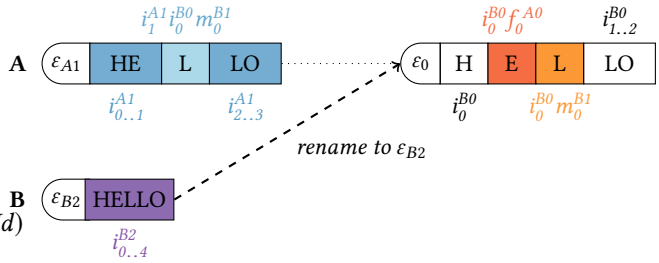
```
function revRenId(id, renamedIds, nId, nSeq)
▷ id is the identifier to reverse rename
▷ renamedIds is the former state shared by the rename op
▷ nId is node id of the node which issued the rename op
▷ nSeq is node seq of the node which issued the rename op

    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
```

mechanism, *MIN_TUPLE* and *MAX_TUPLE*, to generate an identifier complying with the intended order.

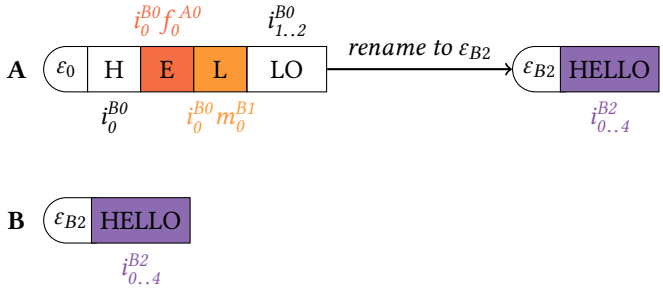
Once node A reverted its state to the LCA epoch (ε_0) using Algorithm 2, it can successively apply *rename* operations leading to the new *primary* epoch (ε_{B2}) using Algorithm 1, as illustrated in Figure 2c.



(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*

4.5 Garbage collection of *former states*

As explained in [7] 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. 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.*

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.*

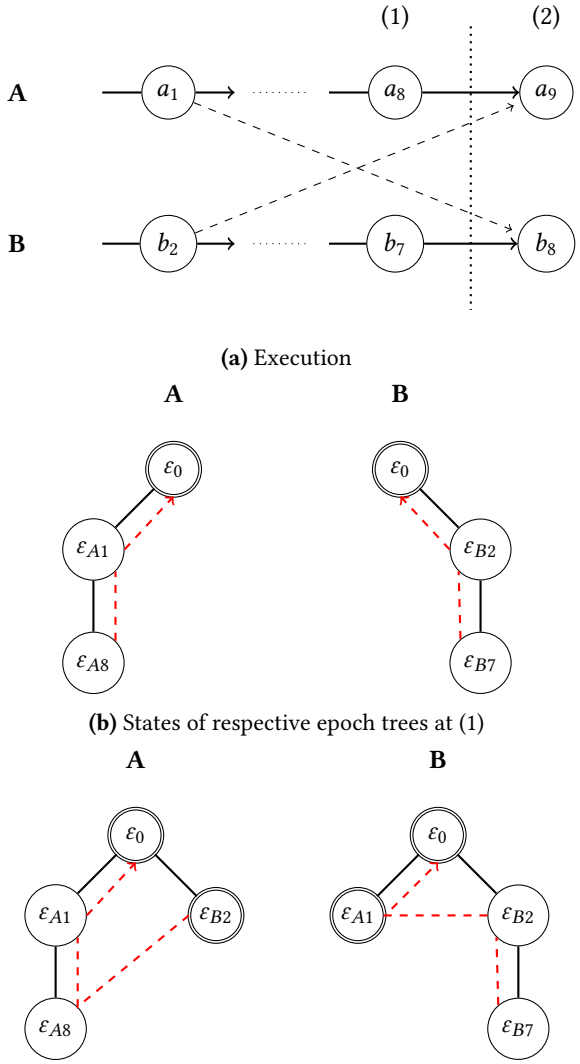
Figure 3 illustrates a use case of Rule 1 and Rule 2. In Figure 3a, we represent an execution in which two nodes A and B respectively issue two *rename* operation before eventually synchronising. In Figure 3b, we represent the states of their respective *epoch trees* once they each generate their *rename*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

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



(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*

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 used to run our benchmarks. These simulations mimic the following scenario.

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 an arbitrary given critical length, collaborators move on to the second phase of the simulation. During this 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 broadcasts it immediately to other nodes using a Peer-to-Peer (P2P) full mesh network. During the first phase, the probability of issuing

insert (resp. *remove*) operations is of 80% (resp. 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, 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 converged without any communication other than operations, thus satisfying the 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.

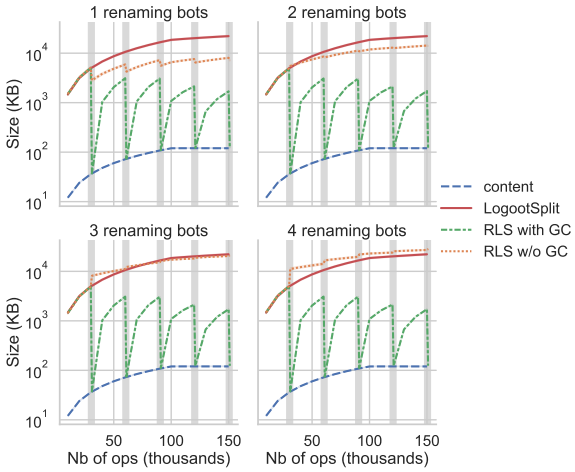


Figure 4. Evolution of the size of the document

Memory overhead We then proceeded to measure the evolution of the document's memory consumption throughout the simulations, according to the CRDT used and the number of *renaming bots*. We present the obtained results in Figure 4.

For each plot displayed in Figure 4, we represent 4 different data. The blue dashed line illustrates the size of the actual content of the document, i.e. the text, while the red solid line corresponds to the size of the whole LogootSplit document.

The green dashed-dotted line represents the size of the RenamableLogootSplit document in the best case scenario. In this scenario, nodes assume that *rename* operations are garbage-collectable as soon as they receive them. Nodes are thus able to benefit the effects of the renaming mechanism while removing its own metadata, such as *former states* and epochs. In doing so, nodes are able to minimise periodically the metadata overhead of the data structure, independently of the number of *renaming bots* and concurrent *rename* operations issued.

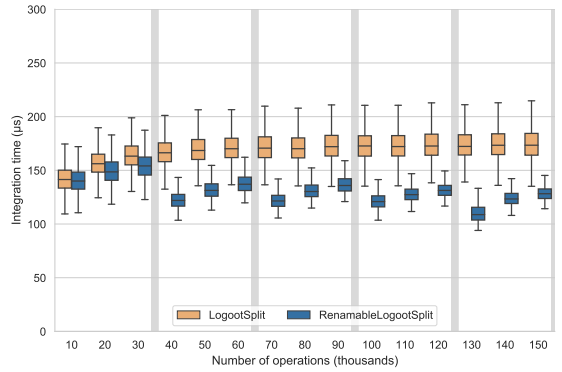
On the other hand, the orange dotted line represents the size of the RenamableLogootSplit document in the worst case scenario. In this scenario, nodes assume that *rename* operations never become causally stable and can thus never be garbage-collected. Nodes have to permanently store the metadata introduced by the renaming mechanism. The performances of RenamableLogootSplit thus decrease as the number of *renaming bots* and *rename* operations issued increases. Nonetheless, we observe that RenamableLogootSplit can outperform LogootSplit even in this worst case scenario while the number of *renaming bots* remains low (1 or 2). This result is explained by the fact that the renaming mechanism enables nodes to scrap the overhead of the internal data structure used to represent the document.

To summarise the results presented, the renaming mechanism introduces a temporary metadata overhead which increases with each *rename* operations. But the overhead will eventually subsides once the system becomes quiescent and *rename* operations become causally stable. In subsection 6.1, we discuss that *former states* may be offloaded until causal stability is achieved to address the temporary memory overhead.

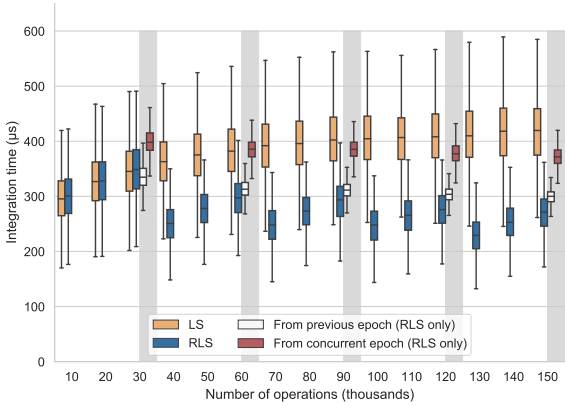
Integration times of standard operations Next, we compared the evolution of integration times of respectively local and remote operations on LogootSplit and RenamableLogootSplit documents. Figure 5 displays the results.

In these figures, orange boxplots correspond to integration times on LogootSplit documents while blue ones correspond to times on RenamableLogootSplit documents. While both are initially equivalent, integration times on RenamableLogootSplit documents are then reduced when compared to LogootSplit ones once *rename* operations have been applied. This improvement is explained by the fact that *rename* operations optimise the internal representation of the sequence.

Additionally, in the case of remote operations, we measured specific integration times for RenamableLogootSplit : integration times of remote operations from previous epochs and from concurrent epochs, respectively displayed as white and red boxplots in Figure 5b. Operations from previous



(a) Local operations



(b) Remote operations

Figure 5. Integration time of standard operations

epochs are operations generated concurrently to the *rename* operation but applied after it. Since the operation has to be transformed beforehand using Algorithm 1, we observe a computational overhead compared to other operations. But this overhead is actually compensated by the optimisation of the internal representation of the sequence performed by *rename* operations.

Regarding operations from concurrent epochs, we observe an additional overhead as nodes have first to reverse the effect of the concurrent *rename* operation using Algorithm 2. Because of this overhead, RenamableLogootSplit's performances for these operations are comparable to LogootSplit ones.

To summarise, transformation functions introduce an overhead with regard to integration times of concurrent operations to *rename* ones. Despite this overhead, RenamableLogootSplit achieves better performances than LogootSplit as long as the distance between the epoch of generation of the operation and the current epoch of the node remains limited. As the distance between both epochs increases, it leads to cases presenting worse performances than LogootSplit ones since the overhead is multiplied. Nonetheless, the renaming mechanism reduces the integration times of the majority of operations, i.e. the operations issued between two rounds of *rename* operations.

Parameters		Integration Time (ms)			
Type	Nb Ops (k)	Mean	Median	99^{th} Quant.	Std
Local	30	41.75	38.74	71.68	6.84
	60	78.32	78.16	81.42	1.24
	90	119.19	118.87	124.22	2.49
	120	143.75	143.57	148.59	2.16
	150	158.04	157.95	164.38	2.49
Remote	30	481.32	477.13	537.30	17.11
	60	981.62	978.24	1072.83	31.54
	90	1491.28	1481.83	1657.58	51.10
	120	1670.00	1663.85	1814.38	50.29
	150	1694.17	1675.95	1852.55	59.94
Prim. Remote	30	643.53	643.57	682.80	13.42
	60	1317.66	1316.39	1399.55	28.67
	90	1998.23	1994.08	2111.98	45.37
	120	2239.71	2233.22	2368.45	50.06
	150	2241.92	2233.61	2351.02	52.20
Sec. Remote	30	1.36	1.30	3.53	0.37
	60	2.82	2.69	4.85	0.45
	90	4.45	4.23	5.81	0.71
	120	5.33	5.10	8.78	0.90
	150	5.53	5.26	8.70	0.79

Table 1. Integration time of rename operations

Integration time of* rename *operation Finally, we measured the evolution of integration times of *rename* operation according to the number of operations since the last *rename* one. The results are displayed in Table 1.

The main outcome of these measures shows that *rename* operations are expensive when compared to others. Local *rename* operations take hundreds of milliseconds while remote ones and concurrent *primary* ones may last seconds if delayed for too long. It is thus necessary to take this result into account when designing strategies to trigger *rename* operations to prevent them from impacting negatively user experiences.

Another interesting result from this benchmark is that concurrent *secondary rename* operations are cheap to apply, as they only consist in storage of corresponding *former states*. Thus nodes can significantly reduce the overall computations of a set of concurrent *rename* operations by applying them in a specific order. We will discuss further this topic in subsection 6.3.

6 Discussion

6.1 Offloading on disk unused *former states*

As explained in subsection 4.4, nodes have to store *former states* corresponding to *rename* operations to transform operations from previous or concurrent epochs. Nodes may receive such operations given 2 different cases : (i) nodes have recently issued *rename* operations (ii) nodes logged

back in the collaboration. Between these specific events, *former states* are actually not needed to handle operations.

We can thus propose the following trade-off : to offload *former states* on the disk until their next use or until they can be garbage collected. It would enable nodes to mitigate the temporary memory overhead introduced by the renaming mechanism but increases integration times of operations requiring one of these *former states*. Nodes could adopt various strategies to deem *former states* offloadable and to retrieve them preemptively according to their constraints. The design of these strategies could be based on several heuristics : epochs of currently online nodes, number of nodes still able to issue concurrent operations, time elapsed since last use of the *former state*...

6.2 Designing a more effective *priority* relation

Although the *priority* relation proposed in subsection 4.2 is simple and ensures that nodes designate the same epoch as the *primary* one, it introduces a significant computational overhead in some cases. Notably it allows a single node, disjoined from the collaboration since a long time, to force every other nodes to revert *rename* operations they performed meanwhile because its own *rename* operation is deemed as the *primary* one.

The *priority* relation should thus be designed to ensure convergence, but also to minimise the overall amount of computations performed by nodes of the system. In order to design the *priority* relation, we could embed into *rename* operations metrics that represent the state of the system and the accumulated work on the document (number of nodes currently at the *parent* epoch, number of operations generated at the *parent epoch*, length of the document...). This way, we can favour the branch from the *epoch tree* with the more and most active collaborators and prevent isolated nodes from overthrowing the existing order.

6.3 Postponing transition to *primary* epoch in case of high concurrency

As shown in Table 1, integrating *primary rename* operations is expensive as nodes have to browse and rename their whole current state. This process can introduce a signification computational overhead in some cases. Especially, a node may receive concurrent *rename* operations in the reverse order to the one defined by the *priority* relation. In this scenario, the node would successively consider every *rename* operation as the *primary* one and would rename its state each time. On the other hand *secondary rename* operations are cheap to integrate, as nodes simply add to their state a reference to the corresponding *former state*.

To mitigate the negative impact of this scenario, we can decompose the integration of *rename* operations into two parts in case of concurrency detection. First, nodes process *rename* operations as secondary ones. It enables nodes to integrate remote *insert* and *remove* operations, even from concurrent epochs, by transforming them. Then each node keeps track of a level of confidence in the current *primary* operation, computed for example from the time elapsed since the node received a concurrent *rename* operation and the number of online nodes still known as using the *parent* epoch. Once the level of confidence reaches a given threshold, the node renames its state according to the operation.

This strategy introduces a slight computational overhead for each *insert* or *remove* operations received during the period of uncertainty, as nodes may issue operations from different epochs during that time. In return, the strategy reduces the probability of erroneously integrating *rename* operations as *primary* ones.

7 Related work

Several works were proposed to address our problem of growth of identifiers in variable-size identifiers Sequence CRDTs. We present in this section the most relevant ones.

7.1 The core-nebula approach

The *core-nebula* approach [8, 9] was proposed to reduce the size of identifiers in Treedoc[4], another variable-size identifiers Sequence CRDT.

In this work, authors introduce a *rebalance* operation enabling nodes to reassign shorter identifiers to elements of the document. However, this *rebalance* operation is not commutative with *insert* and *remove* operations nor with itself. To achieve Eventual Consistency (EC)[10], the *core-nebula* approach prevents concurrent *rebalance* operations by regulating them using a consensus protocol. Operations such as *insert* and *remove* can still be issued without coordination and can thus be concurrent to *rebalance* ones. To deal with this issue, authors propose a *catch-up* protocol to transform these concurrent operations against the effects of *rebalance* ones.

Since consensus protocols do not scale well, the *core-nebula* approach proposes to split nodes among two groups : the *core* and the *nebula*. The *core* is a small set of stable and highly connected nodes while the *nebula* is an unbounded set of dynamic nodes. Only nodes from the *core* participate in the execution of the consensus protocol. Nodes from the *nebula* can still contribute to the document by issuing *insert* and *remove* operations.

Our work can be seen as an extension of this work. It adapts the *rebalance* mechanism and the *catch-up* protocol to LogootSplit and takes advantage of its block feature. Furthermore, it integrates a mechanism to deal with concurrent *rename* operations, hence removing the requirement of a consensus protocol. It makes this approach usable in systems without existing authorities providing nodes to the *core*.

However, systems can actually adopt the *core-nebula* approach to simplify the implementation of RenamableLogootSplit. The use of a consensus protocol to regulate *rename* operations enables systems to discard all parts dedicated to the

handling of concurrent *rename* operations, i.e. the design of a *priority* relation and the implementation of Algorithm 2 and Rule 1.

7.2 The LSEQ approach

The LSEQ approach [11, 12] is another approach proposed to address the growth of identifiers in variable-size identifiers Sequence CRDT. Instead of reducing periodically the identifier metadata using an expensive renaming mechanism, Nédelec et al. define new identifier allocation strategies to reduce their growth rate.

In this work, authors observe that the identifier allocation strategy proposed in Logoot[5] is suited to a single editing pattern : from left to right, top to bottom. If insertions are made according to other patterns, generated identifiers quickly saturate the space of possible identifiers for a given size. Following insertions therefore trigger an increase of the identifier size. As a result, Logoot identifiers grow linearly with the number of insertions instead of the expected logarithmic progression.

LSEQ thus defines several identifier allocation strategy fitted to different editing pattern. Nodes pick randomly one of these strategies for each identifier size. Additionally LSEQ adopts an exponential tree model for identifiers : the range of possible identifiers doubles as the identifier size increases. *TODO: Voir comment mieux formuler ce passage sur le nombre de bits alloués à* position *qui double à chaque "profondeur" d'identifiant. – Matthieu* It enables LSEQ to fine-tune the size of identifiers according to needs. By combining the different allocation strategies to the exponential tree model, LSEQ achieves a polylogarithmic growth of identifiers according to the number of insertions.

While the LSEQ approach reduces the growth rate of identifiers in variable-size identifier Sequence CRDT, the sequence's overhead is still proportional to its number of elements. On the other hand, RenamableLogootSplit's renaming mechanism enables to reduce metadata to a fixed amount, independently of the number of elements.

These two approaches are actually orthogonal and can, as in the previous approach, be combined. The resulting system would reset the sequence's metadata periodically using *rename* operations while LSEQ's identifier allocation strategies would reduce their growth in-between. This would also enable to reduce the frequency of *rename* operations, decreasing the system computations overall.

8 Conclusions and future work

In this paper, we proposed an evolution of the variable-size identifier Sequence CRDT : RenamableLogootSplit. This new version adds algorithms to handle concurrent *rename* operations. Thanks to these algorithms, RenamableLogootSplit no longer needs to rely on consensus protocols to regulate the generation of *rename* operations. We then proceeded to the empirical validation of this new version through experiments. Experiment results showed that RenamableLogootSplit reduces by hundreds times the metadata overhead of the data structure in the best-case scenario, independently of the number of concurrent *rename* operations issued. On the other hand, in the worst-case scenario, the metadata introduced by the renaming mechanism increases with the number of *rename* operations and may result in poorer performances. However, this problem can be addressed by offloading the renaming mechanism metadata to disk, as this data is only needed to handle specific operations.

As future work, we will now focus on the design of strategies to regulate the generation of *rename* operations. These strategies have to meet two different goals. First, they have to minimise the impact of *rename* operations on the user experience. User behaviour studies, inspired by [13, 14], could be led in the context of real-time collaborative writing to set an acceptable upper-bound to their integration times. Strategies should then issue *rename* operations accordingly to keep integration times below the defined threshold. Secondly, strategies have to minimise the likelihood of issuing concurrent *rename* operations without relying on consensus protocols. Strategies should then use the node's local knowledge to determine whether the node should issue a *rename* operation or trust another to do so.

Additionally, we would like to present a formal proof of the correctness of RenamableLogootSplit's algorithms. Another research trail to explore is to study data types to determine which ones could benefit from a combination of OT techniques and CRDT ones for the design or improvement of their replicated counterpart.

A Algorithms

References

[1] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data Consistency for P2P Collaborative Editing. In *ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, pages 259 – 268, Banff, Alberta, Canada, November 2006. ACM Press. URL https://hal.inria.fr/inria-00108523.

[2] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2010.12.006. URL http://www.sciencedirect.com/science/article/pii/S0743731510002716.

[3] Loïck Briot, Pascal Urso, and Marc Shapiro. High Responsiveness for Group Editing CRDTs. In *ACM International Conference on Supporting Group Work*, Sanibel Island, FL, United States, November 2016. doi: 10.1145/2957276.2957300. URL https://hal.inria.fr/hal-01343941.

[4] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, June 2009. doi: 10.1109/ICDCS.2009.20.

[5] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot : A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, pages 404–412, Montreal, QC, Canada,

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
```


June 2009. IEEE Computer Society. doi: 10.1109/ICDCS.2009.75. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.75.

[6] Luc André, Stéphane Martin, Gérald Oster, and Claudia-Lavinia Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2013*, pages 50–59, Austin, TX, USA, October 2013. IEEE Computer Society. doi: 10.4108/icst.collaboratecom.2013.254123.

[7] 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.

[8] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. In *LADIS 2009 - 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, volume 44 of *Operating Systems Review*, pages 29–34, Big Sky, MT, United States, October 2009. Assoc. for Computing Machinery. doi: 10.1145/1773912.1773921. URL https://hal.inria.fr/hal-01248270.

[9] Marek Zawirski, Marc Shapiro, and Nuno Preguiça. Asynchronous rebalancing of a replicated tree. In *Conférence Française en Systèmes d'Exploitation (CFSE)*, page 12, Saint-Malo, France, May 2011. URL https://hal.inria.fr/hal-01248197.

[10] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, December 1995. ISSN 0163-5980. doi: 10.1145/224057.224070. URL https://doi.org/10.1145/224057.224070.

[11] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng 2013, pages 37–46, September 2013. doi: 10.1145/2494266.2494278.

[12] Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. A scalable sequence encoding for collaborative editing. *Concurrency and Computation: Practice and Experience*, page e4108. doi: 10.1002/cpe.4108. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4108.

[13] Claudia-Lavinia Ignat, Gérald Oster, Meagan Newman, Valerie Shalin, and François Charoy. Studying the Effect of Delay on Group Performance in Collaborative Editing. In *Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, Springer 2014 Lecture Notes in Computer Science*, Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, pages 191 – 198, Seattle, WA, United States, September 2014. doi: 10.1007/978-3-319-10831-5_29. URL https://hal.archives-ouvertes.fr/hal-01088815.

[14] Claudia-Lavinia Ignat, Gérald Oster, Olivia Fox, François Charoy, and Valerie Shalin. How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking. In Nina Boulus-Rodje, Gunnar Ellingsen, Tone Bratteteig, Margunn Aanestad, and Pernille Bjorn, editors, *European Conference on Computer Supported Cooperative Work 2015*, Proceedings of the 14th European Conference on Computer Supported Cooperative Work, pages 223–242, Oslo, Norway, September 2015. Springer International Publishing. doi: 10.1007/978-3-319-20499-4_12. URL

https://hal.inria.fr/hal-01238831.