Improving Replicated Sequences Performances

Matthieu Nicolas, Gérald Oster, and Olivier Perrin

Université de Lorraine, CNRS, Inria, LORIA, F-54500, France

Abstract

To achieve high availability, large-scale distributed systems have to replicate data and to minimise coordination between nodes. The literature and industry increasingly adopt Conflict-free Replicated Data Types (CRDTs) to design such systems. CRDTs are data types which behave as traditional ones, e.g. the Set or the Sequence. However, compared to traditional data types, they are designed to support natively concurrent modifications. To this end, they embed in their specification a conflict-resolution mechanism.

To resolve conflicts in a deterministic manner, CRDTs usually attach identifiers to elements stored in the data structure. Identifiers have to comply with several constraints such as uniqueness or being densely ordered according to the kind of CRDT. These constraints may prevent the identifiers' size from being bounded. As the number of the updates increases, the size of identifiers grows. This leads to performance issues, since the efficiency of the replicated data structure decreases over time.

To address this issue, we propose a new CRDT for Sequence which embeds a renaming mechanism. It enables nodes to reassign shorter identifiers to elements in an uncoordinated manner. Obtained experiment results demonstrate that this mechanism decreases the overhead of the replicated data structure and eventually limits it.

1 Introduction

2 Replicated Sequences

2.1 Sequence

The *Sequence* is an Abstract Data Type (ADT) which allows to represent a list of ordered values. Sequences are widely used in algorithms to represent collections of values where the order of the values is relevant such as strings, messages from a discussion or events from a log. Traditionally, implementations provided by programming languages support the following specification:

Specification 1 (Sequence)**.**

$\forall V : \mathsf{Value}\ \forall S : \mathsf{Sequence}\langle V\rangle \cdot \langle S, constructor, queries, commands\rangle$

$S \stackrel{\text{def}}{=} \{v_i \in V\}_{i\in\mathbb{N}}$

$constructor : () \to S$: Generates and returns an empty sequence

$queries = \{length, get\}$

$commands = \{insert, remove\}$

$length : S \to \mathbb{N}$: Returns the number of values contained in the sequence

$get : \{s \in S\} \times \{n \in \mathbb{N} \mid n < length(s)\} \to V$: Returns the value from the sequence s at the index n

$insert : \{s \in S\} \times \{n \in \mathbb{N} \mid n < length(s)\} \times V \to S$: Inserts the given value into the sequence s at the index n and ...

$remove : \{s \in S\} \times \{n \in \mathbb{N} \mid n < length(s)\} \to S$: Removes the value from the sequence s at the index n and returns ...

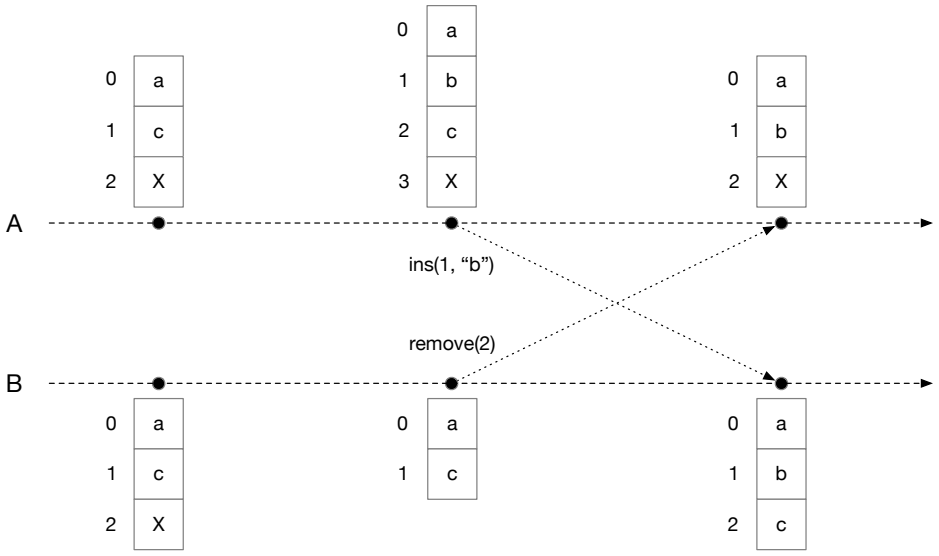

Figure 1: Example of concurrent operations on an index-based sequence resulting into an inconsistency

TODO: Fixer la mise en page pour que les descriptions des fonctions ne soient pas tronquées – Matthieu

However, this specification has been designed for a sequential execution. Using naively this ADT in a distributed system to replicate a sequence among nodes would result in inconsistencies, as illustrated in Figure 1. In this example, two nodes A and B own initially a copy of the same sequence. Without coordinating, both of them perform an update and broadcast it to the other node. However, applying both updates does not yield the same final state on each node.

This issue is a well-know problem in the domain of collaborative editing and has been an area of research for many years. *TODO: Ajouter des références à des papiers sur OT – Matthieu* These works eventually led to new specifications of the *Sequence* belonging to a new family of data types: Conflict-free Replicated Data Types (CRDTs). *TODO: Ajouter référence à WOOT – Matthieu*

2.2 Conflict-free Replicated Data Types (CRDTs)

Conflict-free Replicated Data Types (CRDTs) [1, 2] are new specifications of ADTs, such as the *Set* or the *Sequence*. Contrary to traditional specifications, CRDTs are designed to support natively concurrent updates. To this end, these data types embed directly into their specification a conflict resolution mechanism. These specifications can be followed to implement optimistically replicated data structures which ensure Strong Eventual Consistency (SEC) [2].

Definition 1 (Strong Eventual Consistency)**.** Strong Eventual Consistency (SEC) is a consistency model which guarantees that any two nodes of the distributed system observing the same set of updates reach equivalent states, without requiring any further communications than the ones needed to broadcast the updates.

These data structures are particularly suited to build highly-available large-scale distributed systems in which nodes share and update data without any coordination.

For a given ADT, several specifications of CRDTs can be proposed. They can be classified into three categories: State-based CRDTs, Operation-based CRDTs and Delta-based CRDTs. State-based CRDTs are often more complex data structures than their Operation-based counterparts, but make no assumptions on the reliability of the message-passing layer. Operation-based CRDTs are thus simpler but usually rely on a message-passing layer ensuring the exactly-once causally-ordered delivery of updates. The third category, the Delta-based CRDTs one, was more recently proposed and draws out the best from both worlds.

To solve conflicts deterministically and ensure the convergence of all nodes, CRDTs relies on additional metadata. In the context of Sequence CRDTs, two different approaches were proposed, each trying to

minimize the overhead introduced. The first one affixes constant-sized identifiers to each value in the sequence and uses them to represent the sequence as a linked list. The downside of this approach is an evergrowing overhead, as it needs to keep removed values to deal with potential concurrent updates, effectively turning them into tombstones. The second one avoids the need of tombstones by instead attaching densely-ordered identifiers to values. It is then able to order values into the sequence by comparing their respective identifiers. However this approach also suffers from an increasing overhead, as the size of such densely-ordered identifiers is variable and grows over time.

In this paper, we focus on Densely-identified Operation-based Sequence CRDTs and propose a renaming mechanism to reduce the metadata overhead introduced by this approach.

2.3 Logoot

Logoot [3] is an Element-wise Densely-identified Operation-based Sequence CRDT. Its key insight is to replace the use of mutable *indexes* to refer to values into the sequence with immutable *positions*. As illustrated previously in Figure 1, in the case of traditional sequences, operations update the sequence and shift values, resulting in inconsistencies when applying several concurrent operations. By using immutable *positions* to refer to values, Logoot is able to define a sequence with commutative operations, suited for usages in distributed settings.

When inserting a value into the sequence, the node generates a fitting *position* and associates it to the value. These *positions* fulfill several roles:

Note 1. A *position* identifies uniquely a value.

Note 2. A *position* embodies the intended order relation between the value and other values from the sequence.

To perform these roles, *positions* have to comply to several constraints:

Property 1. (Global Unicity) Nodes should not able to compute the same *position* concurrently.

Property 2. (Timeless Unicity) Nodes should not be able to associate the same *position* to different values during the lifetime of the sequence.

Property 3. (Total Order) A total order relation must exist over *positions* so nodes can order two values given their respective *positions*.

Property 4. (Dense Set) Nodes should always be able to generate new *positions* between two others.

To define *positions* meeting these properties, Logoot first introduces *LogootTuples* which are specified as in Specification 2. *LogootTuples* are triples made of the following elements:

- *priority*: sets the order of this tuple relatively to others, arbitrary picked by the node upon generation
- id_{site}: refers to the node's identifier, assumed to be unique
- seq_{site}: refers to the node's logical clock, which increases monotonically with local updates

Specification 2 (LogootTuple)**.**

$T : \mathsf{LogootTuple} \cdot \langle T, constructor, queries, commands \rangle$

$T \stackrel{\text{def}}{=} \langle \mathbb{N}, \mathbb{I}, \mathbb{N} \rangle$

$constructor : \mathbb{N} \times \mathbb{I} \times \mathbb{N} \to T$: Returns a LogootTuple made of the given *priority*, id_{site} and seq_{site}

$queries = \{priority, peer, seq\}$

$commands = \{\}$

$priority : T \to \mathbb{N}$: Returns the *priority* of the given tuple

$peer : T \to \mathbb{I}$: Returns the id_{site} of the given tuple

$seq : T \to \mathbb{N}$: Returns the seq_{site} of the given tuple

Based on this building block, Logoot defines positions as sequences of *LogootTuples*, as shown in Specification 3.

Specification 3 (LogootPos)**.**

$P : \mathsf{LogootPos} \cdot \langle P, constructor, queries, commands \rangle$
$T : \mathsf{LogootTuple}$
$P \stackrel{\text{def}}{=} \{t_i \in T\}_{i \in \mathbb{N}}$
$constructor : \{t_i \in T\}_{i \in \mathbb{N}} \to P$: Returns a LogootPos made of the given tuples
$queries = \{length, get, lastTuple, peer, seq, uid\}$
$commands = \{\}$
$length : P \to \mathbb{N}$: Returns the number of tuples composing the position
$get : \{p \in P\} \times \{n \in \mathbb{N} \mid n < length(p)\} \to T$: Returns the n-th tuple of the given position
$lastTuple : P \to T$: Returns the last tuple of the given position
$peer : P \to \mathbb{I}$: Returns the id_{site} of the last tuple of the given position
$seq : P \to \mathbb{N}$: Returns the seq_{site} of the last tuple of the given position
$uid : P \to \langle \mathbb{I}, \mathbb{N} \rangle$: Returns the unique id of the given position

It allows positions to meet all the required constraints:

Note 3. Given a position p, the couple $\langle peer(p), seq(p) \rangle$ is globally and timelessly unique as:

- No other node can generate a position using the same id_{site} as it is unique.
- No other position can be generated by the same node using the same seq_{site} as it is increasing monotonically with local updates.

Note 4. A dense total order can be created over positions by:

- Comparing theirs tuples using the lexicographical order.
- Defining a special tuple, $minTuple$ such that $\forall t \in LogootTuple \cdot minTuple < t$. Given two positions $p1, p2$ such as $p1 < p2$, it allows any node to generate a new position $p3$ such as $p1 < p3 < p2$ by reusing the tuples of $p1$, appending $minTuple$ as many times as required and finally appending a tuple of its own creation.

Relying on these positions, Logoot proposes a new specification corresponding to a replicable sequence, described in Specification 4.

Specification 4 (LogootSeq)**.**

$\forall V : \mathsf{Value}\ \forall S : \mathsf{LogootSeq}\langle V \rangle \cdot \langle S, constructor, queries, commands \rangle$
$P : LogootPos$
$S \stackrel{\text{def}}{=} \{\langle p \in \mathsf{P}, v \in \mathsf{V} \rangle_i\}_{i \in \mathbb{N}}$
$constructor : () \to S$: Generates and returns an empty Logoot sequence
$queries = \{length, getPos, generatePos\}$
$commands = \{insert, remove\}$
$length : S \to \mathbb{N}$: Returns the number of values contained in the sequence
$getPos : \{s \in S\} \times \{n \in \mathbb{N} \mid n < length(S)\} \to P$: Returns ...
$generatePos : \mathbb{I} \times \mathbb{N} \times \{s \in S\} \times \{n \in \mathbb{N} \mid n < length(S)\} \to P$: Returns ...
$insert : S \times P \times V \to S$: Inserts the given value into the sequence using its position and returns...
$remove : S \times P \to S$: Removes the value from the sequence attached to the given position and returns...

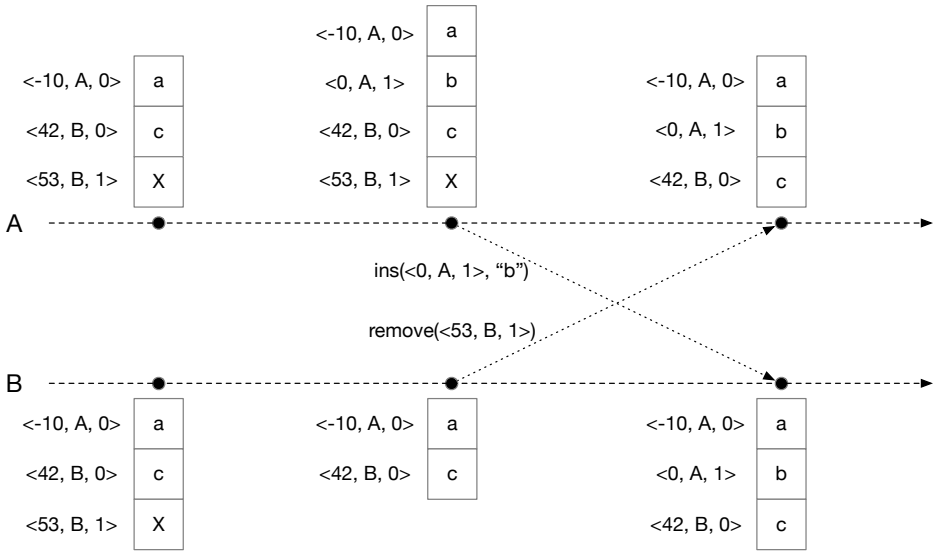

Figure 2: The previous scenario fixed using Logoot positions instead of indexes

Using this data type, we can replay the previous scenario while this time ensuring the correctness and convergence of the final states as illustrated in Figure 2.

In this scenario, node A wants to insert the value "b" between the values "a" and "c". To this end, it generates and attaches to "b" a position greater than the position of "a", but lesser than the position of "c". As there is plenty of room between these two positions, node A is able to generate a position of the same size embodying the intended order. It that was not the case, node A would have to generate a position by reusing the tuple of "a" and appending to it its own tuple, resulting in a longer position.

Meanwhile, node B wants to remove the value "x" from the sequence. To do so, it uses the position attached to this value which identifies it uniquely.

TODO: Trouver comment conclure cet exemple – Matthieu

TODO: Changer la figure pour utiliser des naturels comme priority, histoire d'être cohérent avec la spécification – Matthieu

2.4 LogootSplit

LogootSplit [4] is a Block-wise Densely-identified Operation-based Sequence CRDT. Proposed by André et al. [4], its goal is to improve further the efficiency of the replicated sequence.

Indeed, it is expensive to generate and associate a new position to each value of the sequence. To reduce the metadata overhead, the authors propose to aggregate dynamically values into blocks. By regrouping values into blocks, LogootSplit can assign logically a position to each value, while effectively storing only the position of the first value of each block. This shifts the cause of metadata growth from the number of values to the number of blocks. As a block can contains an arbitrary number of values, it can lead to a significant increase of the efficiency of the data structure.

To achieve this, LogootSplit adds a new component to the tuples composing its positions, the $offset$. This component allows to specify the offset of a value into a block. According to this change, LogootSplit redefines the tuples that it uses as well as the positions, as shown respectively in Specification 5 and Specification 6.

J'aurai bien aimé avoir une abstraction unique pour les Tuples et Positions de Logoot et LogootSplit plutôt que de les redéfinir ici, mais je n'ai pas réussi à la formaliser pour le moment. – Matthieu

Specification 5 (LogootSplitTuple)**.**

$T : \mathsf{LogootSplitTuple} \cdot \langle T, constructor, queries, commands \rangle$

$T \stackrel{\text{def}}{=} \langle \mathbb{N}, \mathbb{I}, \mathbb{N}, \mathbb{N} \rangle$

$constructor : \mathbb{N} \times \mathbb{I} \times \mathbb{N} \times \mathbb{N} \to T$: Returns a LogootSplitTuple made of the given $priority$, id_{site}, seq_{site} and $offset$

$queries = \{priority, peer, seq, offset\}$

$commands = \{\}$

$priority : T \to \mathbb{N}$: Returns the $priority$ of the given tuple

$peer : T \to \mathbb{I}$: Returns the id_{site} of the given tuple

$seq : T \to \mathbb{N}$: Returns the seq_{site} of the given tuple

$offset : T \to \mathbb{N}$: Returns the $offset$ of the given tuple

Specification 6 (LogootSplitPos)**.**

$P : \mathsf{LogootSplitPos} \cdot \langle P, constructor, queries, commands \rangle$

$T : \mathsf{LogootSplitTuple}$

$P \stackrel{\text{def}}{=} \{t_i \in T\}_{i \in \mathbb{N}}$

$constructor : \{t_i \in T\}_{i \in \mathbb{N}} \to P$: Returns a LogootSplitPos made of the given tuples

$queries = \{length, get, lastTuple, peer, seq, offset, uid\}$

$commands = \{fromBase, concat, truncate\}$

$length : P \to \mathbb{N}$: Returns the number of tuples composing the position

$get : \{p \in P\} \times \{n \in \mathbb{N} \mid n < length(p)\} \to T$: Returns the n-th tuple of the given position

$lastTuple : P \to T$: Returns the last tuple of the given position

$peer : P \to \mathbb{I}$: Returns the id_{site} of the last tuple of the given position

$seq : P \to \mathbb{N}$: Returns the seq_{site} of the last tuple of the given position

$offset : P \to \mathbb{N}$: Returns the $offset$ of the last tuple of the given position

$uid : P \to \langle \mathbb{I}, \mathbb{N}, \mathbb{N} \rangle$: Returns the unique id of the given position

$fromBase : P \times \mathbb{N} \to P$: Returns a new position made by copying the given position and replacing its offset...

$concat : P \times P \to P$: Returns a new position made by concatenating tuples from both given positions

$truncate : P \times \mathbb{N} \to P \times P$: Returns a new position made by concatenating tuples from both given positions

Based on its specification of positions, LogootSplit defines the aggregability of positions as follows:

Definition 2 (Base of a Position)**.** The base of a position corresponds to the position deprived of the offset of its last tuple.

Definition 3 (Positions Aggregability)**.** Two positions are aggregable into a block if they share the same base and if their respective offsets are consecutives. It implies that a given block can contain only values inserted by the same node, as the positions have to share the same base thus the same peer.

FIXME: je ne savais pas où insérer l'highlight sur le fait que les valeurs d'un bloc doivent avoir été insérées par le même noeud donc je l'ai mis dans la définition. À bouger? – Matthieu

This rule to aggregate positions into blocks results into their following specification. A block is composed of a position corresponding to the position of its first value, and of the offset of its last value. LogootSplit can then compute the position of each value of the block by replacing the offset of the first position of the block by the offset of the value. When inserting a new value into the sequence, LogootSplit first seeks to append or to prepend it respectively to the preceding block or to the succeeding one. If it is not possible, LogootSplit generates and inserts a new block at the intended place. To ensure that the positions comply with Property 2, LogootSplit keeps track of the used offsets per block to not reassign the same position to different values.

Specification 7 (LogootSplitBlock)**.**

$B : \mathsf{LogootSplitBlock} \cdot \langle B, constructor, queries, commands \rangle$
$P : \mathsf{LogootSplitPos}$
$B \stackrel{\text{def}}{=} \langle \{p \in P\}, \{n \in \mathbb{N} \mid offset(p) \leq n\} \rangle$
$constructor : P \times \mathbb{N} \to I$: Returns a LogootSplitBlock...
$queries = \{length, posBegin, posEnd, begin, end, peer, seq, uid\}$
$commands = \{\}$
$length : B \to \mathbb{N}$: Returns the number of values of the block
$posBegin : B \to P$: Returns the first position of the block
$posEnd : B \to P$: Returns the last position of the block
$begin : B \to \mathbb{N}$: Returns the offset of posBegin
$end : B \to \mathbb{N}$: Returns the offset of posEnd
$peer : B \to \mathbb{I}$: Returns the peer of the positions of the given block
$seq : B \to \mathbb{N}$: Returns the sequence number of the positions of the given block
$uid : B \to \langle \mathbb{I}, \mathbb{N} \rangle$: Returns the unique id of the given block

It is interesting to notice that, in the context of LogootSplit:

- Given a position p, the triple $\langle peer(p), seq(p), offset(p) \rangle$ identifies uniquely this position.
- Given a block b, the couple $\langle peer(b), seq(b) \rangle$ identifies uniquely the base of its positions.
- Given a block b, the quadruple $\langle peer(b), seq(b), begin(b), end(b) \rangle$ identifies uniquely this block.

LogootSplit proposes a new specification of the replicated sequence illustrated in Specification 8, supporting string-wise operations. An example of its behavior is shown in Figure 3.

Specification 8 (LogootSplitSeq)**.**

$\forall V : \mathsf{Value}\ \forall S : \mathsf{LogootSplitSeq}\langle V \rangle \cdot \langle S, constructor, queries, commands \rangle$
$P : LogootSplitPos$
$B : LogootSplitBlock$
$S \stackrel{\text{def}}{=} \left\{ \langle b \in \mathsf{B}, \{v_j \in \mathsf{V}\}_{j \in \mathbb{N}} \rangle_i \right\}_{i \in \mathbb{N}}$
$constructor : () \to S$: Generates and returns an empty LogootSplit sequence
$queries = \{length, getBlocks, generatePos\}$
$commands = \{insert, remove\}$
$length : S \to \mathbb{N}$: Returns the number of values contained in the sequence
$getBlocks : \{s \in S\} \times \{n_1 \in \mathbb{N} \mid n_1 < length(S)\} \times \{n_2 \in \mathbb{N} \mid n_1 \leq n_2 < length(S))\} \to \{b_i \in B\}_{i \in \mathbb{N}}$: ...
$generatePos : \mathbb{I} \times \mathbb{N} \times \{s \in S\} \times \{n \in \mathbb{N} \mid n < length(S)\} \to P$: Returns ...
$insert : S \times P \times \{v_i \in V\}_{i \in \mathbb{N}} \to S$: Inserts the given values into the sequence using its position and returns...
$remove : S \times \{b_i \in B\}_{i \in \mathbb{N}} \to S$: Removes the values from the sequence attached to the given position...

TODO: Finalement, j'aurai tendance à fusionner les sections sur Logoot et LogootSplit pour ne garder que LogootSplit. Dans un premier temps, je peux introduire la notion de position en ne tenant pas compte de offset (ça m'embête juste de mettre l'exemple Figure 2, réadapté pour LogootSplit, sans avoir mis la spécification de la séquence au préalable). Puis je peux motiver le fait de regrouper les valeurs par blocs pour réduire l'overhead, expliquer le rôle de offset et remettre l'exemple Figure 3. – Matthieu

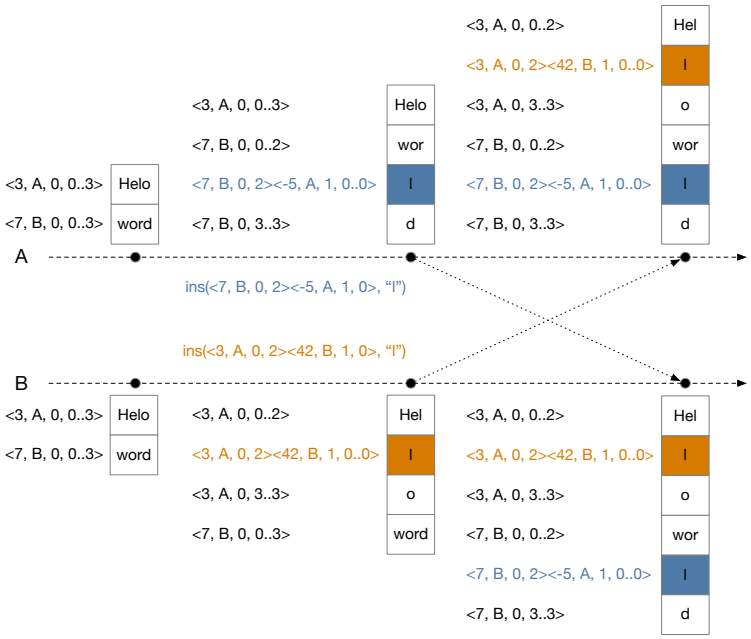

Figure 3: An example of replicated sequence using LogootSplit

2.5 Limits

As shown previously, the size of positions in Densely-identified Sequence CRDTs is not bounded in order to comply with the dense set constraint (Property 4). As more values are added to the sequence, the size of new positions grows to be able represent the intended order between values. However, this growth impacts negatively the performances of the data structure on several aspects. Since positions attached to values become longer, the memory overhead of the data structure increases accordingly. This also results in an increase of the bandwidth consumption as nodes have to broadcast positions to others.

Additionally, as the lifetime of the replicated sequence increases, the number of blocks composing it grows as well. Because of the constraints on the generation of positions and their aggregability, it is not always possible to append or prepend new values to existing blocks. This results in the addition of new blocks to the sequence. Since no mechanism to merge blocks a posteriori is provided, the resulting sequence ends up being fragmented into many blocks. The efficiency of the data structure decreases as each block introduces its own metadata overhead.

TODO: Illustrer l'augmentation de l'overhead avec un graphe. – Matthieu

TODO: Ajouter aussi des métriques sur l'évolution de la taille des identifiants, du nombre de blocs ? – Matthieu

TODO: Ajouter une phrase sur le fait qu'on se retrouve avec une séquence répliquée jusqu'à 100x plus lourde que la séquence équivalente non-répliquée. Cette mesure pose cependant un problème : elle repose sur notre implémentation actuelle de LogootSplit qui ne réutilise pas la même instance d'un tuple partagé par plusieurs identifiants. L'utilisation d'un pattern Multiton résoudrait ce problème et limiterait le poids de la structure de données et donc le résultat des mesures (mais ajouterait une complexité supplémentaire qui serait de traquer quand on peut GC un tuple). – Matthieu

It is thus necessary to either propose a more efficient specification of the replicated list, with a reduced growth of the overhead or to provide a mechanism allowing to reset the overhead of the data structure at times. In this paper, we present a work corresponding to the later approach.

3 Overview

We propose a new Densely-identified Operation-based Sequence CRDT: *RenamableLogootSplit*.

To address the limitations of LogootSplit, we embed in this data structure a renaming mechanism. The purpose of this mechanism is to reassign shorter positions to values in such a manner that we are then able to aggregate them into one unique block. This allows to reduce the bandwidth used to broadcast future updates as well as the metadata of the whole sequence. However, as the goal is to reduce LogootSplit's evergrowing memory overhead and bandwidth consumption, we have to design the renaming mechanism while minimizing its own footprint.

3.1 System Model

The system is composed of a dynamic set of nodes, as nodes join and leave dynamically the collaboration during its lifetime. Each node has a unique identifier from a set $\mathbb{I}$. The nodes collaborate to build and maintain a sequence using RenamableLogootSplit. Each node owns a copy of the sequence and edit it without any kind of coordination with others.

However, the network is unreliable. Messages can be lost, re-ordered or delivered multiple times. The network is also vulnerable to partitions, which split nodes into disjoined subgroups. To overcome the failures of the network, nodes rely on a message-passing layer. As RenamableLogootSplit is built on top of LogootSplit, it shares the same requirements for the operation delivery. This layer is thus used to deliver messages to the application exactly-once. The layer also ensures that *remove* operations are delivered after corresponding *insert* operations. Nodes use an anti-entropy mechanism to synchronise in a pairwise manner, by detecting and re-exchanging lost operations.

3.2 Specification

We introduce a new operation, the *rename* one. Nodes use this operation to define and share a common context between them. Each node use this context to map positions of their current sequence to new ones. This mapping is perform using a new function introduced : $renamePos$.

However, the mapping must be done while preserving the safety properties of the system. These properties are the following:

Property 5. (Well-formed Sequence) The sequence must be well-formed. We define a sequence as well-formed if it meets the following conditions:

> **Property 5.1.** (Unicity Preservation) Each position must be unique. Thus, for a given *rename* operation, each position should be mapped to a distinct new position.

> **Property 5.2.** (Order Preservation) The sequence must be sorted with regards to the positions. Therefore, the existing order between initial positions must be preserved by the mapping.

Property 6. (Strong Eventual Consistency (SEC)) All nodes which received the same set of operations must converge without any further coordination. To ensure this property, operations must be designed to comply with the following constraints:

> **Property 6.1.** (Deterministic Operations) Operations are applied by each node without any coordination. To ensure that each node reaches eventually the same state, operations must always produce the same output.

> **Property 6.2.** (Commutative Concurrent Operations) Concurrent operations may be delivered in different orders at each node. In order to ensure the convergence of nodes, the order of application of a set of concurrent operations should not have any impact on the resulting state.

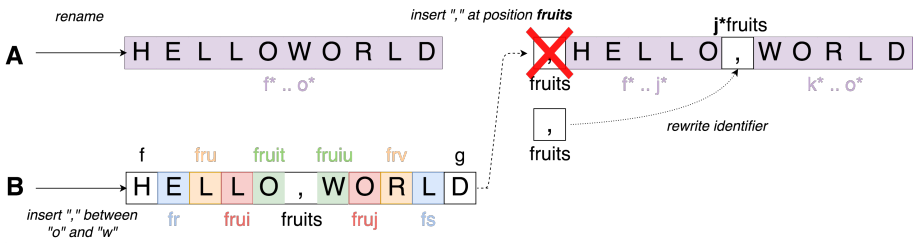

Figure 4: Concurrent *rename* and *insert* operations

3.3 Proposition

TODO: Introduire l'exemple – Matthieu

In this example, two nodes, A and B, own a copy of the same replicated sequence. Initially, their respective states are equivalent. Node A issues a *rename* operation: new positions of minimal size are reassigned to each value of the sequence. These new positions are picked in such a way that make it possible to aggregate all values into one block. Concurrently, node B inserts a new value into the sequence. To this end, the node computes and attaches a position to the value. Both operations are then broadcasted.

Upon reception of the *insert* operation, node A does not apply it as it is. Indeed, as positions of other values have been modified, inserting the new value at its given position would result in an inconsistency: the breach of the intended order. Therefore, it is necessary to rename this position and thus to transform the operation before applying it. Once a fitting new position has been computed, the value can be safely inserted into the sequence. When node B received the *rename* operation, it applies the renaming on each position of its current sequence to obtain the new state.

TODO: Reprendre Figure 4 et unifier son formalisme et style avec le reste des figures. – Matthieu In the next sections, we describe how we designed and implemented such renaming mechanism. For simplicity purposes, we present first RenamableLogootSplit under the assumption that no concurrent renamings can take place. This allows us to illustrate the process of renaming the sequence and how to design it to be commutative with *insert* and *remove* operations. We then present the complete version of RenamableLogootSplit, with the additional components required to deal with concurrent renamings.

4 Renaming without any concurrent *rename* operation

4.1 *Epoch-based* mechanism

A renaming can be seen as a change of frame of reference, applied to positions. Positions come from different frames according to if *rename* operations occurred between their generation. As illustrated in subsection 3.3, comparing positions from different frames and taking decision on this result would lead to inconsistencies. It is thus necessary to define the frame of reference in which a given position is valid and to add information to the system to model it.

To this end, we introduce the notion of *epochs*. The sequence is first assigned a default epoch : the *origin* one. As the default epoch *origin* does not belong to any actual node, we define and use a specific id_{site} noted as $\perp_{\mathbb{I}}$ for this particular case. Upon the delivery of a *rename* operation, nodes update the current epoch of their sequence with a newly generated one. We generate this new epoch using id_{site} and seq_{site} of the node which issued the operation to make it unique. We will then have to embed this data into the *rename* operation.

Thus, we specify epochs as follows:

Specification 9 (Epoch)**.**

$E : \mathsf{Epoch} \cdot \langle E, constructor, queries, commands \rangle$

$E \stackrel{\text{def}}{=} \{origin, \langle n, \mathbb{I} \rangle \mid n \in \mathbb{N}^*\}$

$origin = \langle 0, \perp_{\mathbb{I}} \rangle$

$constructor : \mathbb{I} \times \mathbb{N} \to E$: Returns an Epoch made of the given id_{site} and seq_{site}

$queries = \{peer, seq, epochId\}$

$commands = \{\}$

$peer : E \to \mathbb{I}$: TODO

$seq : E \to \mathbb{N}$: TODO

$epochId : E \to \langle \mathbb{I}, \mathbb{N} \rangle$: TODO

We tag operations with the current epoch at their time of generation to encapsulate their scope of validity. Upon reception of an operation, nodes first compare the current epoch of their sequence to the epoch embedded in the operation. If the two epochs match, the operation can be applied at it is. Otherwise, nodes need to transform the operation beforehand.

As operations are now tied to the concept of epochs, we have to update the existing constraints on their order of delivery. Delivering operations originating from an epoch unknown to a node would cause issues, as the node would have no clue on how to deal with them. Thus, we define the new consistency model used by the message-passing layer as follows:

Specification 10. ("Epoch-based + Causal Remove" Consistency Model) The message-passing layer delivers operations according to the following constraints:

1. Operations are delivered causally to *rename* operations which introduced their respective epochs.
2. remove operations are delivered after corresponding *insert* operations.

4.2 Generating *rename* operations

In order to ensure SEC, nodes have to apply each operation in a coordination-free manner. In our case, nodes have to rename any position the same way without coordinating. However, nodes may observe operations in different orders. They may not share the same state when applying the *rename* operation. Then we can not rely on their respective state to take any decision on how to rename a position. We thus design the *rename* operation to provide and set a common context between nodes. This common context will be used to rename positions in a deterministic fashion without requiring further communications.

We thus specify *rename* operations as follows:

Specification 11 (Rename Operation)**.**

$R : \mathsf{RenameOp} \cdot \langle R, constructor, queries, commands \rangle$

$R \stackrel{\text{def}}{=} \langle E, \mathbb{I}, \mathbb{N}, \{b_i \in B\}_{i \in \mathbb{N}} \rangle$

$constructor : E \times \mathbb{I} \times \mathbb{N} \times \{b_i \in B\}_{i \in \mathbb{N}} \to R$: Returns a *rename* operation made of the given *epoch*, id_{site}, seq_{site} and *renam*

$queries = \{epoch, peer, seq, renamedBlocks\}$

$commands = \{\}$

$epoch : R \to E$: TODO

$peer : R \to \mathbb{I}$: TODO

$seq : R \to \mathbb{N}$: TODO

$renamedBlocks : R \to \{b_i \in B\}_{i \in \mathbb{N}}$: TODO

However, the metadata overhead per block is not bounded, as blocks rely on positions which are themselves not bounded. Moreover, the number of blocks grows as the collaboration progresses until a *rename* operation is applied. Broadcasting *rename* operations may then prove to be expensive.

To adress this issue, we present a mechanism to compress the *rename* operation in subsection 8.4. This mechanism allows to reduce to a fixed amount the metadata per block at the price of additional computations. Furthermore, we configure nodes to issue a *rename* operation once the number of blocks in the sequence reaches a given treshold. This effectively limits the number of blocks that *renamedBlocks* may contain.

Combining these two actions thus allow us to bound the size of *rename* operations.

4.3 Renaming positions

Thanks to the data embedded in the *rename* operations, nodes are now able to rename positions in a convergent way. To this end, we define the new function $renameForwardPos$, which rename a given position *pos* into a new position based on the provided $renamedBlocks$. Its behavior, which is detailed in algorithm 1, can be described as follows.

This algorithm distinguishes two cases: if the given position *pos* is part of the known positions when the *rename* operation was issued, or not. In the first case, we map the position to one of the position of the resulting block. In the other case, we generate a new unique position respecting the intended order.

We start by checking in which case we find ourselves. To this end, we use the function $hasBeenRenamed$. This function browses $renameBlocks$ to determine if *pos* is contained in one of the blocks.

If that is the case, we use the function $findIndex$ to retrieve the corresponding index of *pos* in the sequence. Using this index, we generate and return the new corresponding position.

Otherwise, it means that *pos* has been inserted concurrently to the *rename* operation. *NOTE: On peut aussi rencontrer cette situation si pos a été supprimée avant (dans le sens happen-before) que l'opération de renommage ne soit générée, mais que le modèle de cohérence utilisé par l'application autorise la livraison du renommage avant celle de la suppression (ce qui va être notre cas). Rentrer à ce niveau de détail? – Matthieu* We have to transform it to preserve the existing order. According to the value of *pos*, we apply different strategies to generate its image: $newPos$.

We define $firstPos$ the first position of the first block of $renamedBlocks$ and $lastPos$ the last position of the last block. We call $newFirstPos$ and $newLastPos$ their respective image regarding $renameForwardPos$.

The first strategy is applied to positions which are included in the interval $]firstPos, lastPos[$. In that case, we first search $predecessor$, the predecessor of *pos* in $renamedBlocks$, using the function $findPredecessor$. Then, we retrieve the index of $predecessor$ thanks to $findIndex$. Using its index, we can compute the new position corresponding to $predecessor$: $newPredecessor$. Finally, we generate $newPos$ by concatenating *pos* to $newPredecessor$.

The second case is for positions such as we have $lastPos < pos < newLastPos$. To preserve the intended order between positions, we have to generate $newPos$ such as $newLastPos < newPos$. We achieve it by concatenating *pos* to $newLastPos$ to obtain $newPos$.

The third case is for positions such as we have $newFirstPos < pos < firstPos$. In that scenario, we have to generate $newPos$ such as $newPos < newFirstPos$. To this end, we pick the predecessor of $newFirstPos$, $predecessorOfNewFirstPos$, by decreasing its offset by one. We then generate $newPos$ by concatenating *pos* to $predecessorOfNewFirstPos$.

Finally, in all other cases, we return *pos* unchanged. Indeed, positions from previous cases are mapped to positions belonging to the interval $]predecessorOfNewFirstPos, successorOfNewLastPos[$. As all remaining positions are either smaller than $predecessorOfNewFirstPos$ or greater than $successorOfNewLastPos$, we can safely return it without any risk of breaking the existing order.

4.4 Applying *rename* operation

Upon the delivery of a *rename* operations, nodes execute the following steps to apply it.

First, they retrieve the newly generated epoch using the id_{site} and seq_{site} embedded in the operation. They update the current epoch of their state. Their future operations will then be tagged using this new epoch onward.

Second, using the $renamedBlocks$ contained in the operation and the function $renameForwardPos$, nodes rename each position of their current state. Resulting positions are then aggregated into blocks and a new sequence is built from them, replacing the previous one.

Algorithm 1 Rename position

```
1:  function renameForwardPos(pos : P, newId : 𝕀, newSeq : ℕ, renamedBlocks : {b_i ∈ B}_{i∈ℕ}): P
2:      firstPos ← posBegin(renamedBlocks[0])
3:      lastPos ← posEnd(renamedBlocks[renamedBlocks.length − 1]
4:
5:      newPriority ← priority(firstPos)
6:      newFirstPos ← new P(newPriority, newId, newSeq, 0)
7:      newLastPos ← new P(newPriority, newId, newSeq, length)
8:
9:      if hasBeenRenamed(pos, renamedBlocks) then
10:         index ← findIndex(pos, renamedBlocks)
11:         return new P(newPriority, newId, newSeq, index)
12:     else if firstPos < pos and pos < lastPos then
13:         predecessor ← findPredecessor(pos, renamedBlocks)
14:         indexOfPredecessor ← findIndex(predecessor, renamedBlocks)
15:         newPredecessor ← new P(newPriority, newId, newSeq, indexOfPredecessor)
16:         return concat(newPredecessor, pos)
17:     else if lastPos < pos and pos < newLastPos then
18:         return concat(newLastPos, pos)
19:     else if newFirstPos < pos and pos < firstPos then
20:         predecessorOfNewFirstPos ← new P(newPriority, newId, newSeq, −1)
21:         return concat(predecessorOfNewFirstPos, pos)
22:     else
23:         return pos          ▷ Return the position unchanged as it does not interfere with the renaming
24:     end if
25: end function
```

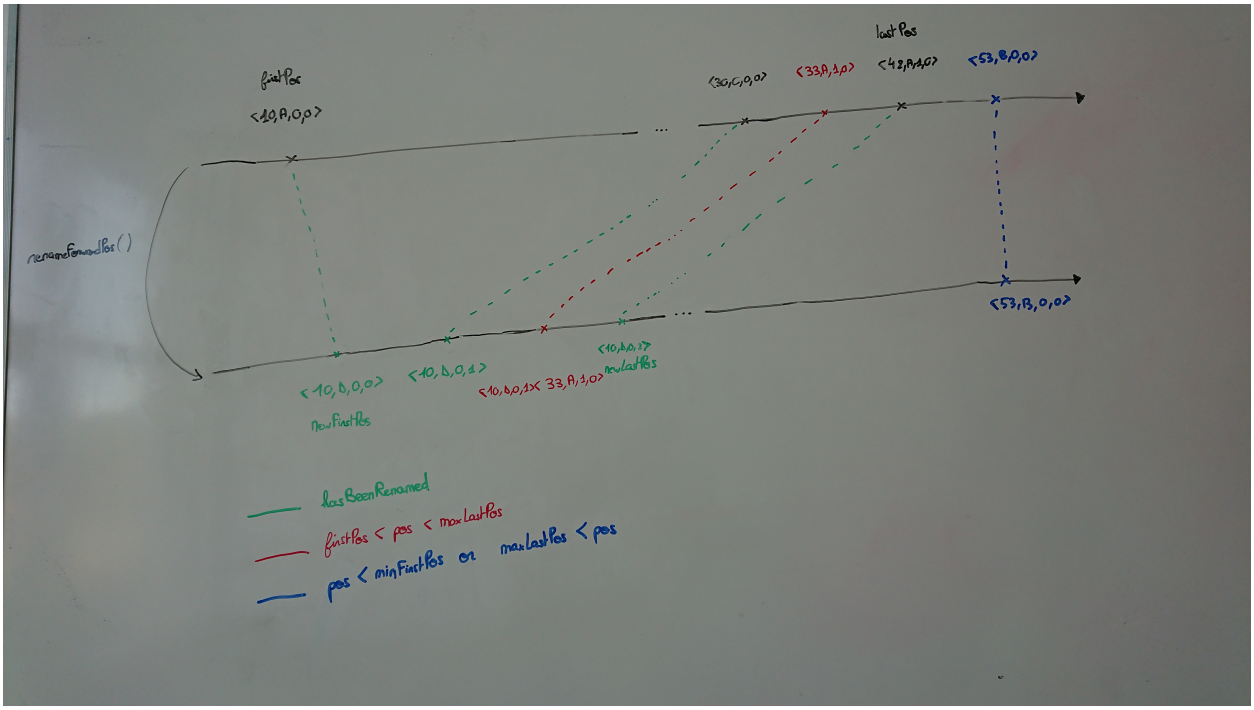

Figure 5: TODO: À refaire sur papier, là on voit rien...

However, nodes may still received operations from previous epochs. Some of these operations may even be outdated by several *rename* operations. To apply these operations, nodes will have to transform them beforehand. Nodes thus need to store the *renamedBlocks* and have to be able to compute the list of epochs separating two others.

To this end, we add a new data structure to the replicated sequence: the *Epoch Store*. Nodes register into the *Epoch Store* each epoch and the corresponding *renamedBlocks*. Newly inserted epochs are linked to the previous epoch by a parent-child relationship.

Specification 12 (Epoch Store)**.**

$ES : \mathsf{EpochStore} \cdot \langle ES, constructor, queries, commands \rangle$

$ES \stackrel{\text{def}}{=} \left\{ \langle E, E, \{b_i \in B\}_{i \in \mathbb{N}} \rangle \right\}$

$constructor : () \to ES$: Returns an empty Epoch Store

$queries = \{epoch, parentEpoch, renamedBlocks, getPathForwardBetweenEpochs\}$

$commands = \{add\}$

$epoch : ES \times \langle \mathbb{I}, \mathbb{N} \rangle \to E$: TODO

$parentEpoch : ES \times E \to E$: TODO

$renamedBlocks : ES \times E \to \{b_i \in B\}_{i \in \mathbb{N}}$: TODO

$getPathForwardBetweenEpochs : ES \times E \times E \to \{e_i \in E\}_{i \in \mathbb{N}}$: TODO

$add : ES \times \langle E, E, \{b_i \in B\}_{i \in \mathbb{N}} \rangle \to ES$: TODO

4.5 Applying *insert* and *remove* operations

Upon the reception of a *insert* or *remove* operation, nodes first compare their current epoch to the operation's one. If the two epochs match, then nodes can apply the operation as it is. Otherwise, it means that one or several *rename* operations were applied locally since this operation has been issued. Nodes have to transform the operation beforehand to preserve its semantic.

In that case, nodes retrieve the list of these renamings using the *Epoch Store*, the current epoch and the operation's one. They rename successively the positions of the operation to build the corresponding positions in the new epoch. Using them, nodes build the new operation and apply it to their state.

4.6 Garbage collection

- As stated previously, the renaming mechanism generates and stores additional metadata: the epochs and the renamingMaps used to transform concurrent operations against the renaming
- However, we do not need to keep this additional metadata forever
- Since they are used to handle concurrent operations to the renaming, they are not required anymore once no additional concurrent operation can be issued by a node
- i.e. we can safely garbage collect rewriting rules once the corresponding renaming operation is causally stable [5]
- Nodes need thus to keep track of the progress of others to detect when this condition is met
- This can be done in a coordination-free manner by exploiting the epochs attached to operations:
  - Each node stores a vector of epochs, with one entry for each node
  - Upon the reception of an operation, the node updates the entry of the sender with the epoch of the operation
  - As nodes collaborates, epochs in the vector will progress
  - By retrieving the minimum epoch from the vector, we can identify which epoch has been reached by all nodes
  - We can then safely garbage collect all previous epochs and corresponding renamingMaps

5 Evaluation

To validate the proposed renaming mechanism, we performed an experimental evaluation to measure its performances on several aspects: 1. the size of the data structure 2. the integration time of the *rename* operation 3. the integration time of *insert* and *remove* operations. In cases 1 and 3, we use LogootSplit as the baseline data structure to compare results.

Since we were not able to retrieve an existing dataset of traces of realtime collaborative editing sessions, we ran simulations to generate traces to evaluate our data structure. The simulations depict the following scenario: several authors collaborate in order to write an article. Initially, they prioritize adding content as everything remains to be done. Thus they mainly insert elements into the document during this first phase. A few *remove* operations are still issued to simulate spelling mistakes. Once the document approaches the critical length, the collaborators switch to the second phase. From this point, they stop adding new content and focus on revamping existing parts instead. This is simulated by balancing the ratio between *insert* and *remove* operations. Each bot has to perform a given number of operations and the collaboration ends once every bot received all operations. We take snapshots of the document at given steps of the collaboration to follow the evolution of the document.

We ran these 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 (version 13.1.0) simulating an author. The 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. During the first phase, the probabilities for each operation of being an *insert* or a *remove* are respectively of 80% and 20%. Once the document reaches 60k characters (around 15 pages), the probabilities are both set to 50%. The generated operation is then broadcast to others using a Peer-to-Peer (P2P) full mesh network. After issuing an operation, there are 5% of chances that the bot moves its cursor to another position in the document. Each bot performs 15k operations. Snapshots are taken every 10k operations. Additionally, in the case of RenamableLogootSplit, one bot is arbitrarily designated as the master. It performs *rename* operations every 30k operations.

The code of the simulations is available at the following address: `https://github.com/coast-team/mute-bot-random/`. This repository also contains the code corresponding to the benchmarks described in the next subsections as well as the results computed.

Meanwhile, our implementation of LogootSplit and RenamableLogootSplit are available at `https://github.com/coast-team/mute-structs`. Both implementations use an AVL Tree, a self-balancing binary search tree, to represent the state of the sequence. This data structure enables us to achieve *insert* and *remove* operations in logarithmic time. Additionally, while we described `renameForwarPos()` in Algorithm 1 as a function to rename one Position only, the corresponding function in our implementation enables to rename an entire Block to improve performances.

5.1 Overhead of the data structure

Using the snapshots generated, we compare the evolution of the size of the data structure in collaborative editing session. The results are displayed in Figure 6. On this plot, the red line corresponds to the size of the whole LogootSplit data structure, content and metadata, while the blue one corresponds to the size of only its content. The yellow line represents the evolution of the size of the RenamableLogootSplit data structure, without garbage collecting states from previous epoches while the green one shows the same data with the garbage collection enabled.

The main result we observe is that RenamableLogootSplit, with garbage collection, allows us to bound the size of its data structure. In this example, as the document's size is reset by successive *rename* operations, it never exceeds 20 Mo. Whereas the LogootSplit document's size grows continuously and eventually reaches 100 Mo.

The second interesting result is that RenamableLogootSplit performed better memory-wise than LogootSplit, even without garbage collection. This result appears at unintuitive, as the *rename* process consists in starting a new equivalent sequence while keeping the former one's state to deal with concurrent updates. However, this can be explained by implementation details. We presented the state of LogootSplit document as a sequence of blocks and the corresponding content. But in pratice, additional metadata is incorporated

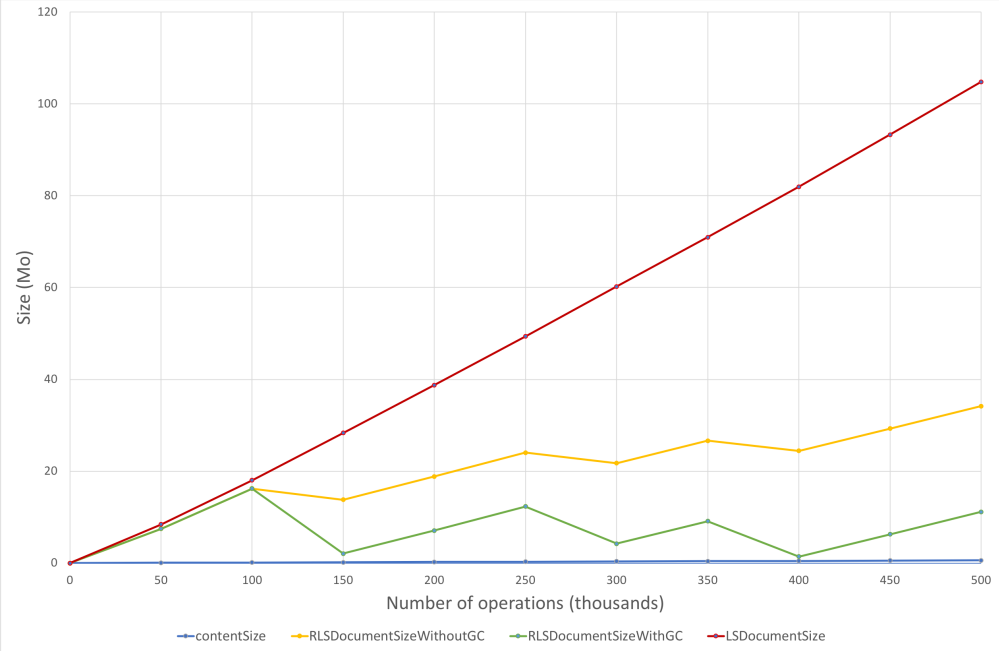

Figure 6: Evolution of the size of the document

to the state to browse the sequence more efficiently when performing updates. When a *rename* operation is applied, we only keep the sequence of blocks from the former state to be able to transform concurrent operations. Other metadata is scrapped, which results in this memory gain even without garbage collecting the rest of the former state.

TODO: Expliquer pourquoi la taille du document LogootSplit évolue de façon quasi-linéaire (paramètres des bots qui empêche la création de longs blocs, ce qui limite leur bénéfice) – Matthieu

TODO: refaire les mesures de la taille du document LogootSplit en supprimant au préalable le champ mapBaseToBlock qui, de mémoire, représente une partie non-négligeable des métadonnées additionnelles et dont on n'est plus très sûr de l'utilité actuellement. Permettra de comment se compare cette version de LogootSplit à RenamableLogootSplit sans GC – Matthieu

5.2 Integration time of *rename* operations

We set up benchmarks to evaluate the performances of the renaming mechanism. We ran the benchmarks on a workstation equipped of a Intel Xeon CPU E5-1620 (10MB Cache, 3.50 GHz) with 16GB of RAM running Fedora 31. The benchmark consists in a single mono-threaded Node.js (version 13.3.0) process measuring the integration times of *rename* operations on snapshots of different sizes. Times were obtained using `process.hrtime.bigint()`. We do not take into account the first hundreds measures to avoid abnormalities due to the Just-In-Time (JIT) compilation process. The results are presented in Figure 7.

The obtained results correspond to the theoritical ones. The time of integration of local *rename* operations grows linearly, as they browse the whole sequence to compute the *renamedIdIntervals*. The time of integration of remote ones follows a quasilinear progression, since they rename each block of the current sequence using `renameForwardPos()`, whose complexity is logarithmic.

An interesting feature of the *rename* operation is that it aggregates renamed elements into one new block, effectively resetting the number of blocks composing the sequence. This allows to speeds up the next *rename* operation, as its complexity depends on the number of blocks. We can then effectively limit the integration time of *rename* operations by using the number of blocks to decide when to issue *rename* operations. This allows the introduction of the proposed mechanism in real-time collaborative applications to improve the memory-wise performances without degrading the user experience eventually.

Furthermore, it is important to highlight that, while the current evaluated implementation is mono-

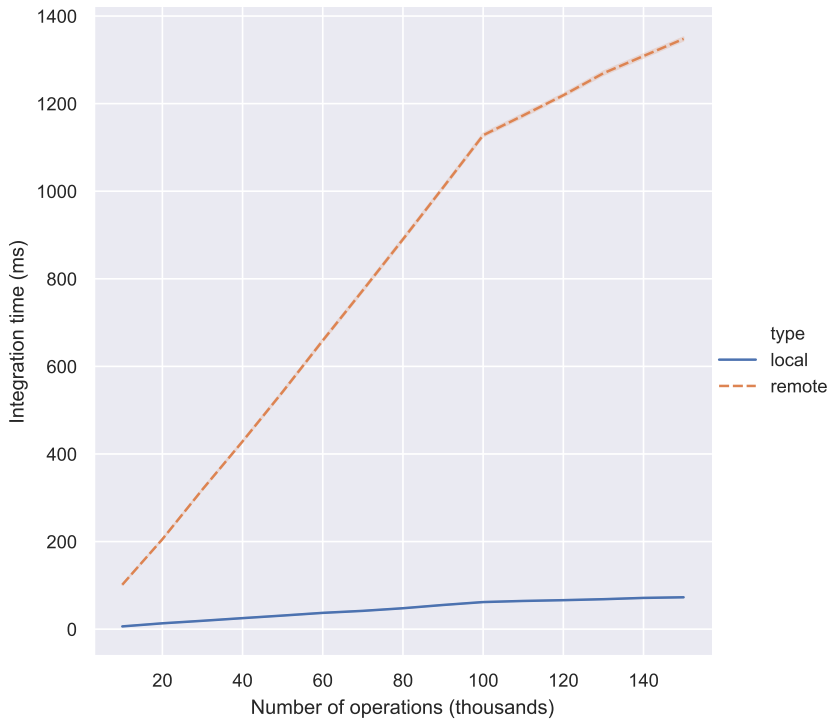

Figure 7: Evolution of the integration time of rename operations

threaded, it is possible to parallelise some computations in the case of the remote *rename* operation. The process of renaming each block using `renameForwardPos()` can be parallelised, as it is a pure function.

5.3 Integration time of *insert* and *remove* operations

TODO: Faire tourner le benchmark pour obtenir les courbes correspondantes pour l'opération remove – Matthieu

We setup another benchmark to measure the impact of the renaming mechanism on the integration times of *insert* and *remove* operations. The experimental settings used are the same as in subsection 5.2. The obtained results are presented in Figure 8.

In this benchmark, we use the integration times of local and remote *insert* and *remove* operations as references, displayed respectively in blue and orange. Both of them follow a logarithmic progression due to them browsing the AVL tree to insert, remove or update an existing block.

First, we measure the integration time of local operations and remote ones issued immediately after applying a *rename* operation, respectively in red and purple. We note that the integration times of these operations is actually constant despite the evolution of the number of operations previously applied to the sequence. This result can be explained easily: since the *rename* operation aggregates all elements into one block, the resulting AVL tree contains only one node, independently of the number of elements of the sequence or the number of preceding operations. The *rename* operation has thus for effect to reset the data structure and speeds up integration times of later *insert* and *remove* operations.

In this benchmark, we also measure the integration time of a remote operation issued concurrently to a *rename* operation, but applied afterward on our copy. As illustrated previously, *insert* and *remove* operations have to be first transformed before being applied in this case. This transformation introduces a computing overhead that corresponds to a call to `renameForwardPos()`. The resulting times are displayed as the green curve on the figure. We note that remote operations concurrent to *rename* operations are actually faster to integrate than remote operations in the standard scenario. This result can be explained by the different data structures used. While both cases require to browse the list of blocks corresponding to the former state, the

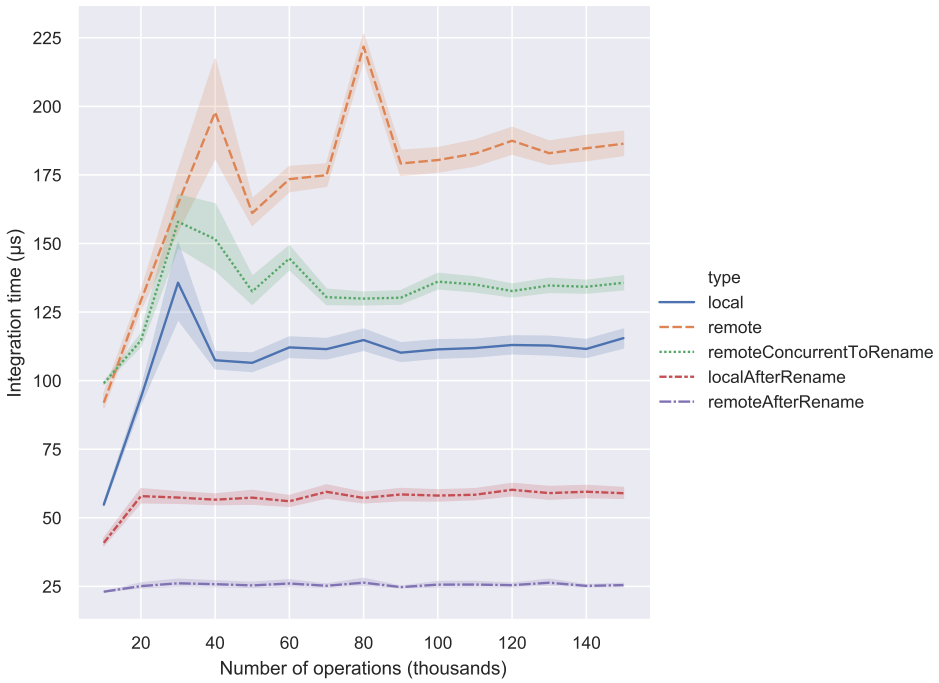

Figure 8: Evolution of the integration time of standard operations

first case requires to browse the state as *renamedIdIntervals*, which is an Array, while the later requires to browse the state as an AVL Tree. To walk through the AVL tree, the program has to follow the pointers between nodes while the Array allows our algorithm to benefit from the locality of reference.

TODO: confirmer/sourcer ces affirmations – Matthieu TODO: voir comment justifier les anomalies mesurées à 40k et 80k – Matthieu

6 Renaming in a fully distributed setting

6.1 System Model

6.2 Intuition

6.3 Strategy to determine leading epoch in case of concurrency

6.4 Transitioning from a losing epoch to the leading one

6.5 Garbage collection

7 Evaluation

8 Discussion

8.1 Offloading on disk unused renaming rules

- As stated previously, nodes have to keep renamingMaps as long as another nodes may issue operations which would require to be transformed to be applied
- Thus nodes need to keep track of the progress of others to determine if such operations can still be issued or if it is safe to garbage collect the renaming rules

- In a fully distributed setting, this requirement is difficult to reach as nodes may join the collaboration, perform some operations and then disconnect
- Other nodes, from their point of view, are not able to determine if they disconnected temporarily or if it left definitely the collaboration
- However, as the disconnected nodes stopped progressing, they hold back the whole system and keep the current active nodes from garbage collecting old renaming rules
- To limit the impact of stale nodes on active ones, we propose that nodes offload unused renamingMaps by storing them on disk

TODO: Présenter une méthode pour déterminer les règles de renommage non-utilisées (conserver uniquement les règles utilisées pour traiter les x dernières opérations ?) – Matthieu

8.2 Alternative strategy to determine leading epoch

8.3 Postponing transition between epochs in case of instability

- May reach a situation in which several nodes keep generating concurrent renaming operations on different epoch branches
- In such case, switching repeatedly between these concurrent branches may prove wasteful
- However, as long as nodes possess the required renamingMaps, they are able to rewrite operations from the other side and to integrate them into their copy, even if they are not on the latest epoch of their branch
  - At the cost of an overhead per operation
- Thus not moving to the new current epoch does not impede the liveness of the system
- Nodes can wait until one branch arise as the leading one then move to this epoch
- To speed up the emergence of such a branch, communications can be increased between nodes in such case to ease synchronisation

8.4 Compressing the renaming operation

TODO: Retravailler pour y ajouter la notion d'offset. Par contre, faire remarquer qu'on a pas besoin de l'offset pour identifier de manière unique "la base" d'une position (toute la position sauf l'offset) – Matthieu

- Propagating the renaming operation consists in broadcasting the list of blocks on which the renaming was performed, so that other nodes are able to compute the same rewriting rules
- This could prove costly, as the state before renaming can be composed of many blocks, each using long positions
- We propose an approach to compress this operation to reduce its bandwidth consumption at the cost of additional computations to process it
- Despite the variable length of positions, the parts required to identify an position uniquely are fixed
  - We only need the $siteId$ and the seq of the last tuple of the position to do so
- Instead of broadcasting the list of whole positions, the node which performs the renaming can just broadcast the list of tuples $< siteId, seq >$
- On reception of a compressed renaming operation, a node needs first to regenerate the list of renamed blocks to be able to apply it

- To achieve so, it can browse its current state looking for positions with corresponding tuples $< siteId, seq >$
- If some positions are missing from the state, it means that they were deleted concurrently
- The node can thus browse the concurrent remove operations to the renaming one to find the missing blocks
- Once all positions has been retrieved and the list of blocks computed, the renaming operation can be processed normally

8.5 Operational Transformation

NOTE: Ajouter une section sur OT pour expliquer que gérer les opérations concurrentes aux renommages consiste en finalité à transformer ces opérations, mais qu'on a décidé de ne pas présenter et formaliser l'approche comme étant de l'OT dans ce papier pour des raisons de simplicité ? – Matthieu

9 Related Works

9.1 Specification and Complexity of Collaborative Text Editing

TODO: voir comment on échappe à leur spécification, en quoi on diffère. – Matthieu

9.2 LSEQ

TODO: Présenter LSEQ et expliquer qu'on peut tout à fait combiner au mécanisme de renommage – Matthieu

9.3 Core and Nebula

TODO: Re-présenter Core et Nebula et expliquer qu'on peut l'utiliser dans le cadre du mécanisme de renommage pour limiter les risques de renommages concurrents – Matthieu

10 Conclusion

References

[1] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[2] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. SSS 2011. 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[3] 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*. Montreal, QC, Canada: IEEE Computer Society, June 2009, pp. 404–412. DOI: 10.1109/ICDCS.2009.75. URL: http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.75.

[4] Luc André et al. "Supporting Adaptable Granularity of Changes for Massive-Scale Collaborative Editing". In: *International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2013*. Austin, TX, USA: IEEE Computer Society, Oct. 2013, pp. 50–59. DOI: 10.4108/icst.collaboratecom.2013.254123.

[5] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. "Making Operation-Based CRDTs Operation-Based". In: *Distributed Applications and Interoperable Systems*. Ed. by Kostas Magoutis and Peter Pietzuch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 126–140.