Improving Replicated Sequences Performances

Matthieu Nicolas, Gérald Oster, and Olivier Perrin

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

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 ...

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) [1, 2]

Conflict-free Replicated Data Types (CRDTs) 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].

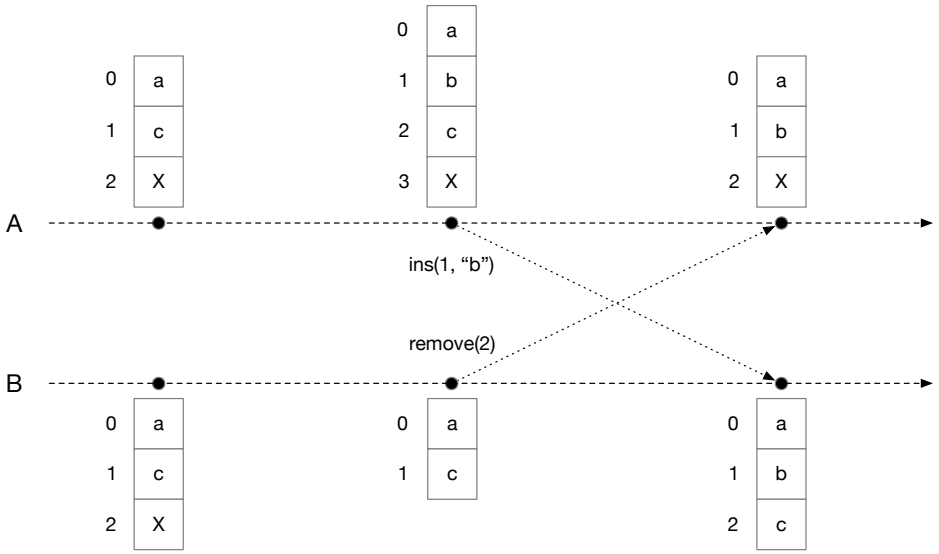

Figure 1: Example of concurrent operations on an index-based sequence resulting into an inconsistency

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 [3]

Logoot 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...

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 [4]

LogootSplit 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

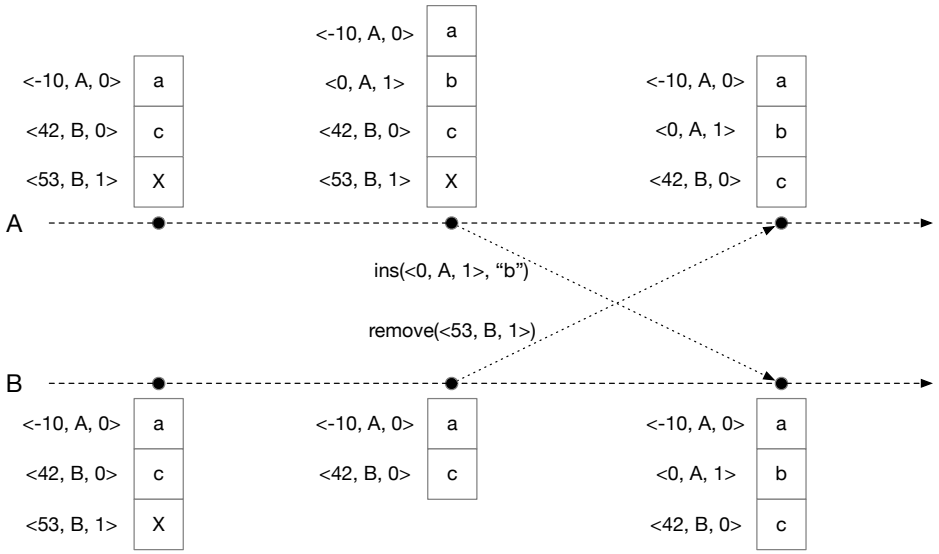

Figure 2: The previous scenario fixed using Logoot positions instead of indexes

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 follow:

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

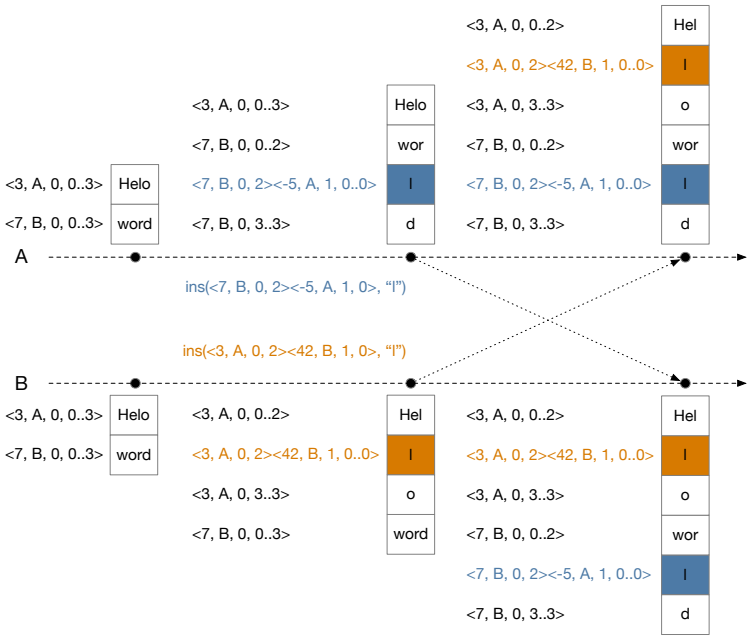

Figure 3: An example of replicated sequence using LogootSplit

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.

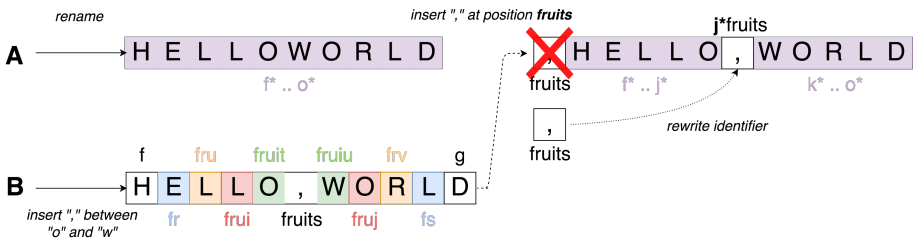

Figure 4: Concurrent *rename* and *insert* operations

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. When a node issues a *rename* operation, it updates the current epoch of its sequence with a newly generated one. To generate the new epoch, the node uses its id_{site} and seq_{site}. 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 case.

We thus specify epochs as follow:

Specification 9 (Epoch)**.**

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

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

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

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

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

$commands = \{\}$

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

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

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

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.

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 follow:

Specification 10 (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*

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

$commands = \{\}$

$epoch : R \to E$: ???

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

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

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

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 7.4. 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 Proposition

- Decompose the *rename* operation into the following components and steps

4.3.1 Rename positions

- To compute the new position corresponding to a given position in the new epoch, define the following function $renameForwardPos(renamingMap, pos) = pos'$
- Its behavior, which is detailed in algorithm 1, can be described as follow:

Algorithm 1 Rename position

function RENAMEFORWARDPOS($renamingMap, pos$)
  if $renamingMap.has(pos)$ **then**
    return $renamingMap.get(pos)$
  end if

  $renamedPositions \leftarrow keysOf(renamingMap)$
  $firstPos \leftarrow renamedPositions[0]$
  $lastPos \leftarrow renamedPositions[renamedPositions.length - 1]$
  $newFirstPos \leftarrow renamingMap.get(firstPos)$
  $newLastPos \leftarrow renamingMap.get(lastPos)$
  $minFirstPos \leftarrow min(firstPos, newFirstPos)$
  $maxLastPos \leftarrow max(lastPos, newLastPos)$

  if $pos < minFirstPos$ **or** $maxLastPos < pos$ **then**
    return pos ▷ Return the position unchanged as it does not conflict with the renaming
  end if

  if $newFirstPos < pos < firstPos$ **then**
    $< priority, id, seq, offset > \leftarrow newFirstPos$ ▷ Retrieve all components of $newFirstPos$
    $predecessorOfNewFirstPos \leftarrow \langle priority, id, seq, offset - 1 \rangle$
    return $concat(predecessorOfNewFirstPos, pos)$
  end if

  $predecessorOfPos \leftarrow findPredecessor(renamedPositions, pos)$
  $newPredecessorOfPos \leftarrow renamingMap.get(predecessorPos)$
  return $concat(newPredecessorOfPos, pos)$
end function

  - If pos was one of the positions belonging to the state when the $renamingMap$ was computed, then its renaming has already been decided and its new value is stored in the $renamingMap$. We just return it.
  - If that is not the case, we need to compute the new position corresponding to it in the new frame of reference
  - To be able to preserve the existing order between pos and other positions through the renaming, we use different strategies according to the position's value:
    * We define respectively as $firstPos$ and $lastPos$ the first and last positions contained in the entries of the $renamingMap$, and $newFirstPos$ and $newLastPos$ their images in the new frame of reference.
    * If pos is actually outside of the range impacted by the renaming, i.e $pos < min(firstPos, newFirstPos)$ or $max(lastPos, newLastPos) < pos$, we can return it unchanged as it will not conflict with other renamed positions
    * If we have $newFirstPos < pos < firstPos$, we rename pos to shift it just before $newFirstPos$ by concatenating pos to the predecessor of $newFirstPos$. The predecessor of $newFirstPos$ is obtained by subtracting 1 to its $offset$ part.

    * In the remaining case, it means that we have $firstPos < pos < maxLastPos$. In this case, we look for $predecessorOfPos$, the predecessor of pos among the keys of $renamingMap$. Then, we retrieve the image of this position in the new state, $newPredecessorOfPos$. By concatenating pos to $newPredecessorOfPos$, we are able to generate a new position while preserving the existing order.

- The main idea of this approach is to preserve the existing order between positions by concatenating the former positions to given prefixes to form the new positions.
- The greater the original position, the greater the prefix used to compose the new position.
- So, with $p1$ and $p2$ two positions such as $p1 < p2$ and $p1'$ and $p2'$ their respective renamed versions, we have
  - either $p1'$ and $p2'$ sharing the same prefix. In that case, comparing $p1'$ and $p2'$ effectively comes down to comparing $p1$ and $p2$.
  - either the prefix of $p2'$ is greater than the prefix of $p1'$.
- In both cases, we have $p1' < p2'$.
- *NOTE: Ce que je dis au-dessus concerne le cas où on a $firstPos < p1 < p2 < maxLastPos$, mais est vrai aussi pour le cas où on a $newFirstPos < p1 < firstPos < p2$ de façon moins évidente.*
  Reste à montrer que l'ordre est conservé dans les cas limites ($p1 < newFirstPos < p2 < firstPos$; $p1 < maxLastPos < p2$).
  Renvoyer à la section validation. – Matthieu

4.3.2 Putting it all together

TODO: Expliquer qu'on propose un nouveau CRDT, <insérer un nom ici> *(RenamableLogootSplit?).*
Cette structure de données encapsule une liste répliquée en utilisant LogootSplit, mais maintient aussi l'epoch courante, une map des epochs et les renamingMaps permettant d'avancer d'une epoch à l'autre.
Dispose des fonctions présentées précédemment, generateRenamingMap() et renameForwardPos().
Propose la fonction rename() qui retourne la séquence renommée en appliquant renameForwardPos à chaque position de l'état courant.
Surcharge le traitement des opérations insert et delete pour:
- Déléguer le traitement de l'opération à l'instance de LogootSplit si l'epoch de génération correspond à l'epoch courante
- Ou transformer l'opération au préalable à l'aide de renameForwardPos() (potentiellement à travers plusieurs epochs) si les epochs ne correspondent pas.
Indiquer qu'on a une contrainte supplémentaire pour la livraison des opérations : elles doivent être livrées dans l'ordre causal par rapport à la dernière opération de renommage observée (doit être à la même epoch que celle de génération de l'opération ou à une epoch suivante pour pouvoir appliquer une opération). – Matthieu

4.4 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

4.5 Limits

4.5.1 Size of concurrently generated positions

TODO: Ajouter quelques lignes sur le fait que le renommage a pour effet d'augmenter la taille des positions insérées en concurrence. Tempérer ce problème en argumentant que ce nombre de positions concurrentes devrait s'avérer faible par rapport au nombre total de positions contenues dans la séquence et qu'elles seront de toute façon réduites au cours du renommage suivant. – Matthieu

4.5.2 Fault-tolerance

- The system is vulnerable to failures, as only one particular node is able to trigger renamings
  - A failure of this node would prevent the renaming mechanism from being triggered ever again
  - But other nodes would still be able to continue their collaboration in such scenario
  - The failure of the renaming mechanism does not impede the liveness of the system
- To address this fault-tolerance issue, can set up a consensus-based system
  - Require nodes to perform a consensus to trigger a renaming
  - But consensus algorithms are expensive and not suited for dynamic systems
  - Can adapt the idea introduced in [6]
  - In this paper, authors propose to divide a distributed system into two tiers: the *Core*, a small set of controlled and stable nodes, and the *Nebula*, an uncontrolled set of nodes
    - Only nodes from the *Core* would participate in the consensus leading to a renaming
  - Provide a trade-off between the cost of performing a renaming and the resilience of the system
- But this approach is not suited for all kind of applications
- In fully distributed systems, there is no central authority to provide a set of stable nodes acting as the *Core*

5 Renaming in a fully distributed setting

5.1 System Model

5.2 Intuition

5.3 Strategy to determine leading epoch in case of concurrency

5.4 Transitioning from a losing epoch to the leading one

5.5 Garbage collection

6 Evaluation

7 Discussion

7.1 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

7.2 Alternative strategy to determine leading epoch

7.3 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

7.4 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

7.5 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

8 Related Works

8.1 Specification and Complexity of Collaborative Text Editing

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

8.2 LSEQ

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

8.3 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

9 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.

[6] 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*. Vol. 44. Operating Systems Review 2. Big Sky, MT, United States: Assoc. for Computing Machinery, Oct. 2009, pp. 29–34. DOI: 10.1145/1773912.1773921. URL: https://hal.inria.fr/hal-01248270.