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:

$\forall V : \mathsf{Value}\ \forall S : \mathsf{Sequence}\langle V \rangle$
$S = \{v_i : V\}_{i \in \mathbb{N}}$
$constructor : () \to S$: Generates and returns an empty sequence
$length : S \to \mathbb{N}$: 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: Traditional specification of a sequence

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 2. 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].

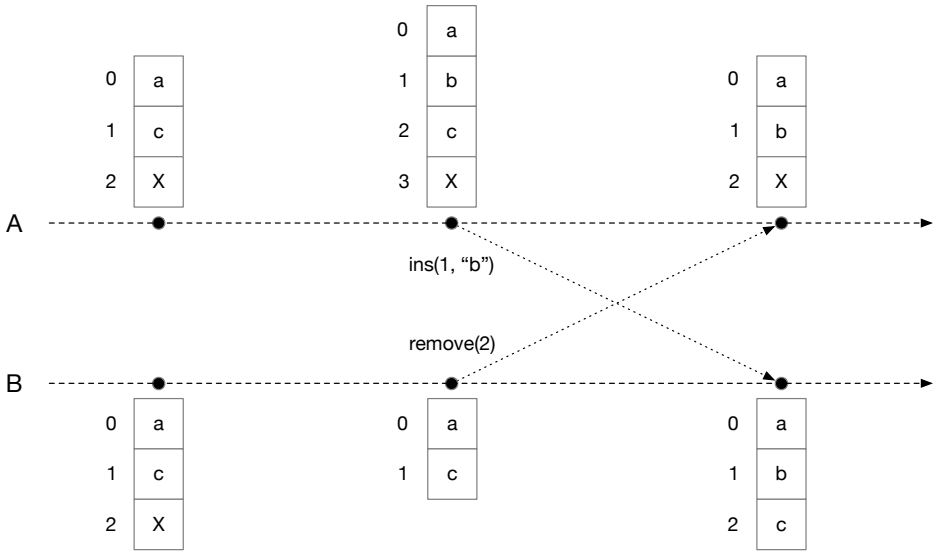

Figure 2: Example of concurrent operations on an index-based sequence resulting into an inconsistency

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 dense 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 dense 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 a 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 2, 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 Figure 3. *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

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

Figure 3: Specification of a Logoot tuple

Based on this building block, Logoot defines positions as sequences of *LogootTuples*, as shown in Figure 4.

$\forall T$: LogootTuple $\forall P$: LogootPos
$P = \{t_i : T\}_{i \in \mathbb{N}}$
$constructor : \{t_i : T\}_{i \in \mathbb{N}} \to P$: Returns a LogootPos made of the given tuples
$get : \{p \in P\} \times \{n \in \mathbb{N} \mid n < length(p)\} \to T$: 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

Figure 4: Specification of a Logoot 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 Figure 5.

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

Figure 5: Specification of a Logoot sequence

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 6. Thanks to positions, each node is able to insert the value at the correct place and to remove the intended value.

2.4 LogootSplit [4]

- Block-wise Sequence CRDTs
- Goal is to improve further the performances of Logoot
- It is expensive to generate and associate a position to each element of the sequence
- André et al. [4] proposes to aggregate dynamically elements into blocks
- It allows to reduce the metadata per element as we only need to store a *position interval* for a block, independently of how many elements it contains
- To do so, add a new component to positions tuples: the $offset$ *NOTE: Préciser que $offset$ doit appartenir à un ensemble énumérable, de façon à ce que l'on puisse déterminer son prédecesseur, successeur ou si une valeur est manquante ? – Matthieu*
- LogootSplit *positions* are thus composed of one or several of the following tuples:

$$\langle priority, id_{site}, seq_{site}, offset\rangle$$

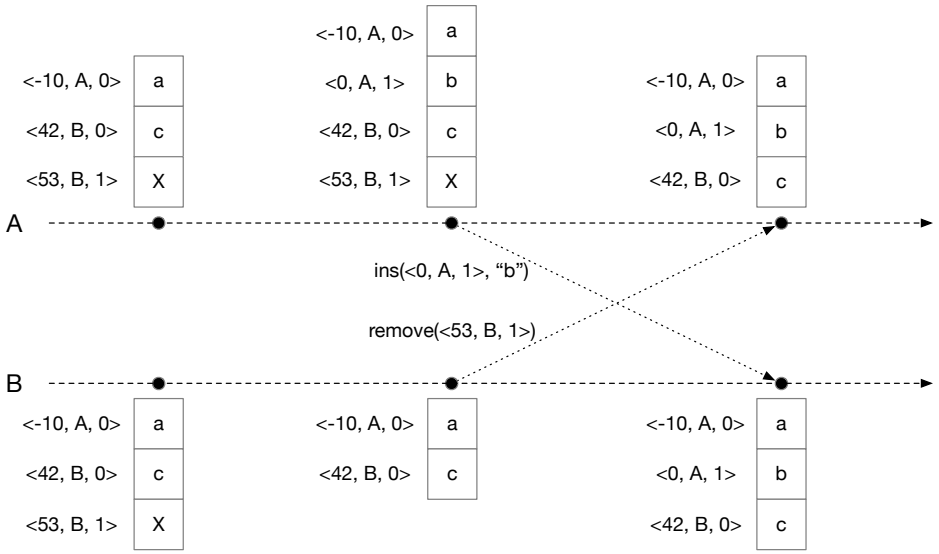

Figure 6: The previous scenario fixed using Logoot positions instead of indexes

- Define positions as aggregable if all components but the $offsets$ of their last tuple are identical and if their respective $offsets$ are consecutive
- Define *position interval* as the following couple:

$$\langle posBegin, end \rangle$$

  where:

  - $posBegin$ refers to the first position of the interval
  - end refers to the value of the offset of the last position of the interval

- Given these two values, we are able to recreate all positions from the interval if needed
- It is worth to notice that
  - The uniqueness of a LogootSplit position is ensured by the triple $\langle id_{site}, seq_{site}, offset \rangle$
  - As such, an $offset$ can not be used twice for the same block
  - It is then necessary to prevent such a case from happening when a node prepend or append a new element to an existing block
  - It can be achieved by keeping track of used $offset$ per block
    - To be more precise, only need to keep track of the minimum and maximum $offsets$ ever used for this block
- Updates the definition of the operations *insert* and *remove* to make them block-wise
  - $insert(S, pos, elts) = S'$ inserts into the sequence S the values $elts$ from the position pos. Returns the updated sequence S'
  - $remove(S, posIntervals) = S'$ removes from the sequence S the values with their position contained in one of the interval of $posIntervals$ and returns the updated sequence S'
- Also introduce the following function:
  - $getPosIntervals(S, index, length) = posIntervals$ retrieves the list of *position intervals* at the index $index$ in the current sequence S

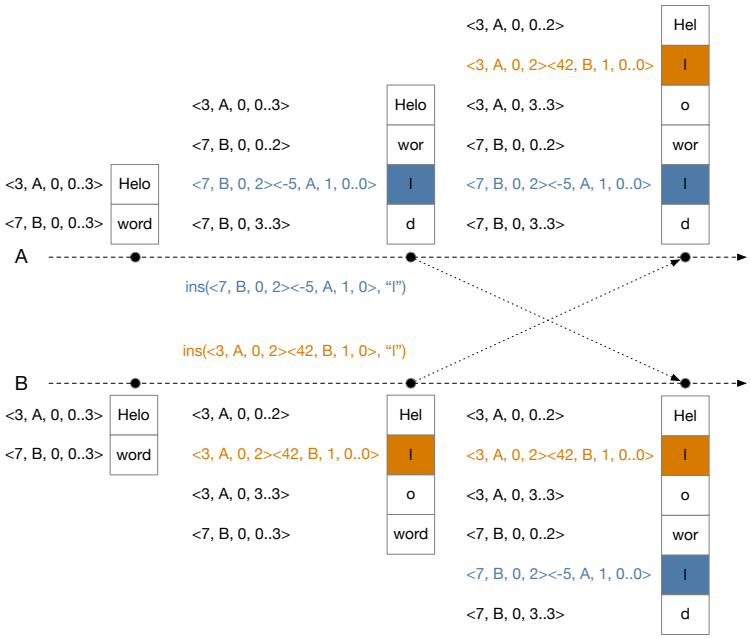

Figure 7: An example of replicated sequence using LogootSplit

2.5 Limits

- As shown previously, LogootSplit positions' size is not bounded in order to comply with the dense constraint.
- As more elements are added to the sequence, less positions of the minimum size are available to insert new elements while respecting the intended order.
- Thus the size of new positions tends to grow as the lifetime of the replicated sequence increases.
- However, the growth of the size of positions impacts negatively the performances of the application on several aspects.
  - As positions attached to elements become longer, the memory overhead of the data structure increases accordingly.
  - This also results in an increase of the bandwidth consumption, as positions have to be broadcast between nodes.
  - Also, operations needs to compare positions, either to look for the deleted positions or to find the place to insert the ones. The longer the positions are, the slower is the processing of operations.
- Additionally, the number of blocks composing the sequence tends to growth as well.
- It is not always possible to add newly inserted positions to existing blocks because of the rules on the generation of positions and on their aggregability.
- This results in the addition of new blocks to the sequence.
- As no mechanism to merge blocks a posteriori is provided, the resulting sequence ends up being fragmented into many blocks.
- This degrades as well the performances of the application, as the number of blocks increases the memory overhead of the data structure and the computation time required to browse the sequence.

- *TODO: Illustrer ces baisses d'efficacité à l'aide de mesures :*
  - Un graphe sur l'évolution du poids de la séquence par rapport au nombre d'éléments contenus en fonction du nombre d'opérations jouées préalablement
  - Un graphe sur l'évolution du temps de traitement d'une opération par rapport au nombre d'éléments contenus en fonction du nombre d'opérations jouées préalablement – 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.
- The work introduced in this paper corresponds to the later approach.

3 Overview

- We build up on top of LogootSplit
- To address its limitations, we introduce a renaming mechanism
- The purpose of this mechanism is to reassign shorter positions to each element such as all of them can be aggregated into one unique block
- This allows to reduce the metadata per element, the computation time required to apply next updates and also the bandwidth used to broadcast future updates
- *TODO: Expliquer que toutefois le mécanisme de renommage ajoute un coût et qu'on cherche à le minimiser. Mais j'ai du mal à voir comment introduire la notion de cet overhead sans rentrer dans les détails qui en sont la source (règles de réécriture, arborescence des epochs, transformation des opérations concurrentes), que je préférerais introduire un par un dans la suite du papier – Matthieu*
- For simplicity purposes, will first present the renaming mechanism in the context of a centralised system, with only one node able to trigger the renaming mechanism
  - Will describe the functioning of this initial version and discuss its limitations
- Will then present a more elaborated version of the renaming mechanism, overcoming the limitations of the centralised version and allowing its use in a distributed setting

4 Renaming in a centralised setting

4.1 System Model

NOTE: Ça me paraît correct mais confusant de parler d'un système centralisé pour un système P2P où seulement une fonctionnalité (le renommage) n'est disponible que pour un noeud particulier. Voir pour mieux présenter cet aspect – Matthieu

- Peer-to-peer network
  - Nodes join and leave dynamically
- Nodes build and maintain a sequence collaboratively using LogootSplit
  - Each node owns a copy of the sequence and can edit it without any kind of coordination with others
- The network is unreliable
  - Messages can be lost, re-ordered and delivered multiple times

- To overcome the faults of the network, a message-passing layer is used to deliver messages to the application exactly-once, in the correct order
  - At each node, the insertion of a position happens before its removal
- An anti-entropy mechanism is used by nodes to synchronise in a pairwise manner, by detecting and re-exchanging lost messages
- One node is arbitrarily designed as the leader
- This node only is able to trigger renamings

4.2 Specification

- Introduce a new operation *rename*.
- This operation allows nodes to map each *position* of their current sequence to a new one.
- Must be defined for all *positions* of the current sequence of the node triggering the renaming...
- ... but also for all *positions* which can be added concurrently by other nodes.
- The *rename* operation must preserve the safety properties of the system, which are:
  - The resulting sequences must be valid
  - The nodes must converge
- We define a sequence valid if
  - Each position is unique. Thus a *rename* operation should not associate several positions to the same resulting one. To put it in a more formal way, the *rename* operation must be designed such as:

$$\nexists p1, p2 \in Pos \cdot rename(p1) = rename(p2)$$

  - The sequence must be sorted with regards to the positions. The existing order between initial positions must then be preserved by the *rename* operation. It results in the following property :

$$\forall p1, p2 \in Pos \wedge p1 < p2 \cdot rename(p1) < rename(p2)$$

- To ensure the Strong Eventual Consistency of the system, the *rename* must be:
  - Determinist : an operation being applied without any kind of coordination by nodes, it must always produce the same output so that nodes reach the same state.
  - Commutative with the *insert* and *remove* operations : as operations can be delivered in different orders at each node, for nodes to reach the same state in a coordination-free manner, the order of application of a set of operations must not have any impact on the resulting output.
    - However, as only one node is able to trigger the renaming mechanism, it is impossible for concurrent *rename* operations to exist. It is thus not required to design the *rename* operation such as it is commutative with itself in this system model.

4.3 Proposition

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

4.3.1 renamingMap

- The first step is to generate the $renamingMap$.
- This map is computed by the node which triggers the *rename* operation
- It associates each existing position of its current sequence to a new one
- It will be used by every nodes to apply the renaming on their state
- *TODO: Montrer qu'en se basant sur la renamingMap pour effectuer le renommage, on se découple de l'état courant du noeud au moment où il l'applique. Les noeuds prennent donc leurs décisions sur les mêmes entrées, ce qui garantit les mêmes résultats. – Matthieu*
- As it needs to be broadcast to other nodes, the size of the $renamingMap$ impacts the efficiency of the proposed solution
- A mechanism to compress the $renamingMap$ is introduced in section 7.4
- To compute this map, the node uses the algorithm 1

Algorithm 1 Generate renamingMap

function GENERATERENAMINGMAP(S, id', seq')
  $renamingMap \leftarrow Map()$

  $firstPos \leftarrow S.getFirstPos()$
  $firstTuple \leftarrow firstPos.getFirstTuple()$
  $priority \leftarrow firstTuple.priority$ ▷ Retrieve the priority of the first tuple of the first position

  for all $(pos, index) \in S$ **do**
    $pos' \leftarrow new\ Pos(\langle priority, id', seq', index\rangle)$ ▷ Generate the new corresponding position
    $renamingMap.set(pos, pos')$
  end for

  return $renamingMap$
end function

- This algorithm has the following behavior:
  - Consist in generating a Map associating each position of the sequence S to a new position
  - To optimise the resulting sequence, the new positions should be aggregable into one block
  - i.e. the new positions must form a *position interval*
  - First, have to pick a position as the beginning of the interval
  - This choice is arbitrary
  - In our case, pick the following position pos':

$$pos' = \langle priority, id', seq', 0\rangle$$

    where:
    - $priority$ is the same value as the *priority* of the first tuple of the first element of the sequence. We explain this choice in section 4.3.3.
    - id' is the id_{site} of the node performing the *rename* operation
    - seq' is the current seq_{site} of the node performing the *rename* operation
  - The first position pos of the current sequence will be map to pos'

- Then, all former positions are mapped to the consecutive ones of the new position according to the existing order
- All positions of the current sequence will be mapped to a set of positions which effectively form the following position interval $posInterval'$:

$$posInterval' = \langle pos', length - 1 \rangle$$

where:

* $length$ is the length of the sequence

- *TODO: Ajouter une phrase expliquant qu'on perd de l'information avec le renommage. Par exemple, on remplace le id_{site} contenu dans la position qui identifie l'auteur de cet élément par id' qui est l'identifiant de l'auteur du renommage – Matthieu*

NOTE: Le mappage des anciennes positions aux nouvelles se fait de manière séquentielle et ne dépend pas en finalité de la valeur de la position initiale.
On peut tout à fait remplacer la $renamingMap$ par:
- la liste des positions renommées
- et une fonction qui calcule la nouvelle position à partir de l'index de la position renommée, de priority de la 1ère position renommée, de id du noeud qui a déclenché le renommage et sa valeur de seq à ce moment là.
Ceci permet de limiter les métadonnées conservées qu'à l'ancien état, et non plus l'ancien état + n nouvelles positions de longueur 1 (1 nouvelle position pour chaque position renommée).
Retravailler cette partie en conséquence ? – Matthieu

4.3.2 *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 a renaming occurred between their generation
- We should not compare positions from different frames as it is meaningless
- It is thus necessary
  - to define the frame of reference in which a given position is valid
  - add information to the system to model this frame
- To achieve this, we introduce the notion of *epochs*
- The sequence has a default epoch : the *origin* one
- When a renaming is performed, a new epoch is generated and replaces the current epoch of the sequence
- The set of these epochs forms a chain in the case of the centralised setting
- The renamingMaps presented previously allow nodes to move their sequence from one epoch to the next one
- By tagging operations with the current epoch at the time of generation, we can encapsulate its scope of validity
- Upon reception of an operation, a node is able to determine by comparing its current epoch to the epoch embedded in the operation if it can be applied or if additional steps are required beforehand
- To represent epochs, we use the following data structure:

$$\langle \langle epochNumber, id_{site} \rangle, parentId \rangle$$

where:

- $epochNumber$ is the successor of the $epochNumber$ of the previous epoch
- id_{site} is the identifier of the node which generated the epoch
- $\langle epochNumber, id_{site} \rangle$ form the identifier of the epoch
- $parentId$ refers to the previous epoch by its identifier

4.3.3 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 2, can be described as follow:

Algorithm 2 Rename position

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

    renamedPositions ← keysOf(renamingMap)
    firstPos ← renamedPositions[0]
    lastPos ← renamedPositions[renamedPositions.length − 1]
    newFirstPos ← renamingMap.get(firstPos)
    newLastPos ← renamingMap.get(lastPos)
    minFirstPos ← min(firstPos, newFirstPos)
    maxLastPos ← 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 > ← newFirstPos          ▷ Retrieve all components of newFirstPos
        predecessorOfNewFirstPos ← ⟨priority, id, seq, offset − 1⟩
        return concat(predecessorOfNewFirstPos, pos)
    end if

    predecessorOfPos ← findPredecessor(renamedPositions, pos)
    newPredecessorOfPos ← 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

4.3.4 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 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.