Efficient Renaming in Sequence CRDTs

Matthieu Nicolas, Gérald Oster and Olivier Perrin

Abstract—To achieve high availability, large-scale distributed systems have to replicate data and to minimise coordination between nodes. For these purposes, literature and industry increasingly adopt Conflict-free Replicated Data Types (CRDTs) to design such systems. CRDTs are new specifications of existing data types, e.g. Set or Sequence. While CRDTs have the same behaviour as previous specifications in sequential executions, they actually shine in distributed settings as they natively support concurrent updates. To this end, CRDTs embed in their specification conflict resolution mechanisms. These mechanisms usually rely on identifiers attached to elements of the data structure to resolve conflicts in a deterministic and coordination-free manner. Identifiers have to comply with several constraints, such as being unique or belonging to a dense total order. These constraints may hinder the identifier size from being bounded. As a result, identifiers tend to grow as the system progresses, which deepens the overhead of CRDTs over time and leads to performance issues. To address this issue in the context of real-time collaborative editing, we propose a novel Sequence CRDT which embeds a renaming mechanism. It enables nodes to reassign shorter identifiers to elements in an uncoordinated manner. Experimental results demonstrate that this mechanism decreases the overhead of the replicated data structure and eventually minimises it.

Index Terms—CRDTs, replication, real-time collaborative editing, eventual consistency, memory-wise optimisation, performance.

1 Introduction

When creating distributed systems, designers have to make a tradeoff between *consistency* and *latency* [1]. Many systems choose to favour latency and thus adopt *optimistic replication* techniques [2]. This approach ensures the high *availability* of the system, even in case of network partitions. To this end, it relaxes consistency constraints and minimises coordination between nodes. In this approach, every node owns a copy of the data, can modify it and then propagate updates to others. Replicas are thus allowed to temporarily diverge. To ensure that they eventually reach equivalent states despite concurrently generated updates, a conflict resolution mechanism is required.

Several approaches were introduced to design efficient conflict resolution mechanisms. We propose to use Conflict-free Replicated Data Types (CRDTs) [3]. CRDTs are new specifications of abstract data types, e.g. Set or Sequence. From users' perspective, CRDTs share the same semantics and interfaces as non-replicated specifications. However, the particularity of CRDTs is that they are designed to natively support concurrent modifications. CRDTs thus have the same behavior as previous specifications in sequential executions, but also define additional semantics for scenarios that may occur in distributed executions.

CRDTs embed a conflict resolution mechanism directly in their specification. It enables them to respect the Strong Eventual Consistency (SEC) model [3]. This consistency model states that replicas reach equivalent states as soon as they observe the same set of updates, without any further communications required and in spite of possible different reception orders. This property makes CRDTs particularly suitable for the design of highly-available large-scale distributed systems.

• *The authors are with the Université de Lorraine, CNRS, Inria, LORIA, F-54500, Nancy, France.*
E-mail: {matthieu.nicolas, gerald.oster, olivier.perrin}@loria.fr.

CRDTs have been widely adopted by literature and industry since their conceptualisation. CRDTs corresponding to more powerful data types were designed and made available as libraries to developers [4], [5], [6], [7]; distributed data stores relying on CRDTs were released [8], [9], [10], [11]; and a new paradigm of applications using CRDTs as their keystone technology has been specified: Local-First Softwares [12], [13].

Additionally, CRDTs have become an important research topic in the domain of real-time collaborative editing. Real-time collaborative editors allow users to share and edit text documents, often represented as Sequences. The design of such applications faces many challenges. It requires the definition of correct and efficient conflict resolution mechanisms that preserve users' intention [14]. Collaborative systems must guarantee privacy and be resilient to censorship. Recent work [15], [16], [17] has demonstrated the relevance of Sequence CRDTs to address these issues, notably thanks to their compatibility with peer-to-peer approaches.

Still, Sequence CRDTs exhibits some limitations. In particular, Sequence CRDTs suffer from the accumulation of large amount of metadata over time due to their internal conflict resolution mechanisms. While the evergrowing metadata decreases the efficiency of these data structures memory-wise, it also increases their bandwith consumption and computational overhead. Some previous work proposed renaming mechanisms [18], [19] to reduce punctually this overhead. However, these mechanisms require synchronous coordination.

In this work, we propose a new Sequence CRDT: RenamableLogootSplit (RLS). This data structure allows nodes to insert or remove elements into a replicated sequence. It embeds a renaming mechanism to minimise the memory overhead induced by metadata punctually. To avoid costly and blocking consensus algorithms, we instead adopt an optimistic approach : nodes perform renaming without any coordination. This new feature is designed

as an additional type of operation: the *rename* one. Since this operation is not intrinsically commutative with others, conflicts may arise. Therefore, we use *Operational Transformations* (OT) [14], [20], [21] to enable nodes to resolve them deterministically. In this paper, we present experimental results that assess that metadata overhead in RenamableLogootSplit is significantly lower than other state-of-the-art Sequence CRDTs.

This paper is organised as follows: Section 2 introduces in more details Sequence CRDTs and the elements leading to their evergrowing overhead. Section 3 provides an overview of the renaming mechanism and of the properties it must respect. Section 4 presents the inner working of the renaming mechanism and how it interacts with concurrent *insert* and *remove* updates. Then Section 5 describes how RenamableLogootSplit handles concurrent executions of the renaming mechanism. Section 6 presents the experimental evaluation of RenamableLogootSplit. Section 7 discusses several tradeoffs that RenamableLogootSplit implementations offer. Section 8 compares our approach to related work. Finally Section 9 summarises our work and introduces future work.

2 Background

To deterministically solve conflicts and ensure convergence of all nodes, CRDTs rely on metadata. In the context of Sequence CRDTs, two different approaches were proposed, both trying to minimise the overhead introduced. The first one [15], [22], [23], [24], [25] attaches fixed size identifiers to each element in the sequence and uses them to represent the sequence as a linked list. The downside of this approach is an ever growing overhead, as it needs to keep removed elements to deal with potential concurrent updates, effectively turning them into tombstones. The second one [26], [27], [28], [29], [30], [31] avoids the need of tombstones by attaching identifiers from a dense total order to elements. Elements are ordered into the sequence by comparing their respective identifiers. New element can always be inserted in between with a proper identifier according to the dense order. However this approach suffers from an ever-increasing overhead, as the size of such identifiers is unbounded and grows over time. In the context of this paper, we focus on the later approach.

2.1 LogootSplit

LogootSplit (LS) [29] is the state of the art of the variable-size identifiers approach of Sequence CRDT. As explained previously, it uses identifiers from a dense total order to position elements into the replicated sequence.

To this end, LogootSplit assigns identifiers made of a list of tuples to elements. These tuples have four components: 1) a *position*, which embodies the intended position of the element 2) a *node identifier*, 3) a *node sequence number* and 4) an *offset*, which are combined to make identifiers unique. By comparing identifiers using the lexicographical order, LogootSplit is able to determine the position of the element relatively to others. In this paper, we represent identifiers using the following notation: $position_{offset}^{node\_id\ node\_seq}$ where $position$ is a lowercase letter, $node\_id$ an uppercase one and both $node\_seq$ and $offset$ integers.

(a) Elements with their corresponding identifiers (b) Elements grouped into a block

Fig. 1. Representation of a LogootSplit sequence containing the elements "HLO"

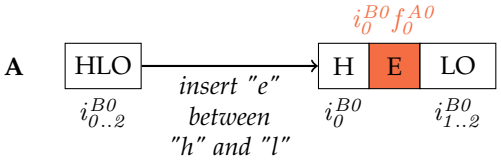


Fig. 2. Insertion leading to longer identifiers

Instead of storing an identifier for each element of the sequence, the main insight of LogootSplit is to aggregate dynamically elements into blocks. Grouping elements into blocks enables LogootSplit to assign logically an identifier to each element, using intervals of identifiers, while effectively storing only the block length and the identifier of its first element. LogootSplit gathers elements with *contiguous* identifiers into a block. We call *contiguous* two identifiers that are identical except for their last offset, and with both offsets being consecutive. We denote the interval of identifiers corresponding to a block using the following notation: $position_{begin..end}^{node\_id\ node\_seq}$ where $begin$ is the offset of the first identifier of the block and end the offset of its last identifier.

Figure 1 illustrates such a case: in 1a, the element identifiers i_0^{B0}, i_1^{B0}, i_2^{B0} form a chain of contiguous identifiers. LogootSplit is then able to group them into one block $i_{0..2}^{B0}$ to minimise the metadata stored, as shown in 1b.

This feature reduces the number of identifiers stored in the data structure, since identifiers are kept at the level of blocks rather than on every individual element. It enables to reduce significantly the memory overhead of the data structure.

2.2 Limits

As stated previously, the size of identifiers from a dense total order is variable. When nodes insert new elements between two others with the same *position* value, LogootSplit has no other option but to increase the size of the resulting identifiers. Figure 2 illustrates such cases. In this example, since node A inserts a new element between the contiguous identifiers i_0^{B0} and i_1^{B0}, LogootSplit can not generate a proper identifier of the same size. To comply with the intended order, LogootSplit generates a new identifier by appending a new tuple to the identifier of the predecessor : $i_0^{B0} f_0^{A0}$.

As a result, the size of identifiers tends to grow as the collaboration progresses. This growth impacts negatively the performance of the data structure on several aspects. Since identifiers attached to values become longer, the memory overhead of the data structure increases accordingly. This also increases the bandwidth consumption as nodes have to broadcast identifiers to others.

Additionally, during the lifetime of the replicated sequence, the number of blocks increases. Indeed, several constraints on identifier generation prevent nodes from adding new elements to existing blocks. For example, only the node that generated the block can append or prepend elements to it. These limitations cause the generation of new blocks. The sequence ends up fragmented into many blocks of only few characters each. However no mechanism to merge blocks a posteriori is provided. The efficiency of the data structure hence decreases as each block introduces its own overhead.

As shown later in section 6, we measured that content eventually represents less than 1% of the whole data structure size. Remaining 99% of the data structure hence correspond to metadata. It is thus necessary to address the previously highlighted issues.

3 Overview

We propose a new Sequence CRDT belonging to the variable-size identifiers approach: RenamableLogootSplit [32], [33]. This data structure allows nodes to insert or remove elements into a replicated sequence. We introduce a *rename* operation to reassign shorter identifiers to elements and to group them into blocks to minimise the memory overhead of the whole sequence.

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. Nodes collaborate to build and maintain a sequence using RenamableLogootSplit. Each node owns a copy of the sequence and edits it without any coordination. Nodes' updates take the form of operations that are immediately applied to nodes' replicas. Operations are then broadcast asynchronously to other nodes so that they also integrate updates.

Nodes communicate through a Peer-to-Peer (P2P) network, which 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 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. *TODO: Ajouter une petite phrase pour exprimer qu'on a pas d'autres contraintes sur la livraison des opérations – Matthieu* Nodes use an anti-entropy mechanism [34] to synchronise in a pairwise manner, by detecting and re-exchanging lost operations.

3.2 Definition of the *rename* operation

The purpose of the *rename* operation is to reassign new identifiers to elements of the replicated sequence, but it must not alter its content. Since identifiers are metadata used by the data structure solely for conflict resolution, users are unaware of their existence. *Rename* operations are thus system operations: they are issued and applied by nodes behind the scenes, without any user initiative.

In order to ensure the SEC property of the replicated sequence, we define several safety properties that the *rename* operation must respect. These properties are mainly inspired by those presented in [19].

Property 1. (Determinism) *Rename* operations are applied by each node without any coordination. To ensure that each node reaches eventually the same state, a given *rename* operation must always output the same new identifier from the current identifier.

Property 2. (User-intention Preservation) Although the *rename* operations itself has no user intention attached, it must not conflict with users actions. Notably, *rename* operations must not cancel or alter the outcome, from users' points of view, of *insert* and *remove* operations.

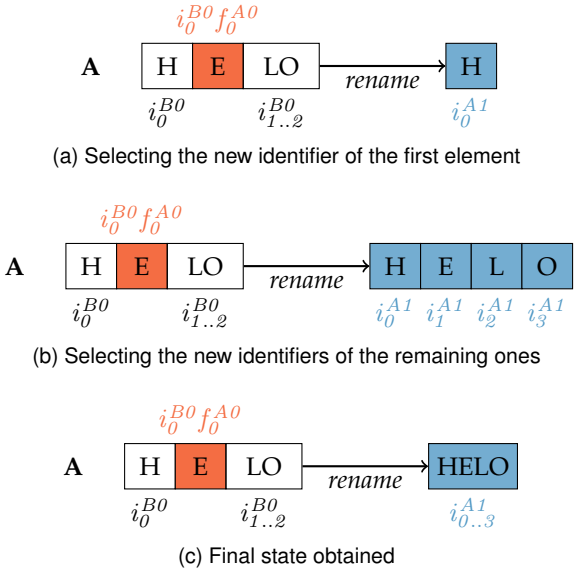
Property 3. (Well-formed Sequence) The replicated sequence must be well-formed. Applying a *rename* operation to a well-formed sequence must then output a well-formed sequence. A well-formed sequence ensures the following properties:

> ***Property 3.1.*** (Unicity Preservation) Each identifier must be unique. Thus, for a given *rename* operation, each identifier should be mapped to a distinct new identifier.
>
> ***Property 3.2.*** (Order Preservation) The elements of the sequence must be sorted according to their identifiers. Therefore, the existing order between initial identifiers must be preserved by the *rename* operation.

Property 4. (Commutativity with Concurrent Operations) Concurrent operations may be delivered in different orders to each node. To ensure convergence of replicas, the order of application of a set of concurrent operations should not have any impact on the resulting state. The *rename* operation must then be commutative with any other concurrent operation.

In the case of the *rename* operation, Property 4 is particularly difficult to achieve. This is due to the fact that *rename* operations modify identifiers assigned to elements. However, other operations such as *insert* and *remove* ones rely on these identifiers to specify where to insert elements or which ones to remove. *Rename* operations are thus intrinsically incompatible with concurrent *insert* and *remove* ones. Likewise, concurrent *rename* operations may reassign different identifiers to given elements. Concurrent *rename* operations are hence not commutative. Therefore, it is required to design and use conflict resolution strategies to apply operations concurrent to *rename* ones.

For the sake of simplicity, the presentation of the *rename* operation is divided into two parts. In section 4, we present the proposed *rename* operation under the assumption that no concurrent *rename* operations may be issued. This assumption enables us to focus on the inner working of the *rename* operation and on how to deal with concurrent *insert* and *remove* operations. Then, in section 5, we remove this assumption to allow concurrent *rename* operations. We present our approach to make the *rename* operation commutative with itself to deal with this scenario.



(a) Selecting the new identifier of the first element

(b) Selecting the new identifiers of the remaining ones

(c) Final state obtained

Fig. 3. Renaming the sequence on node *A*

4 RenamableLogootSplit without concurrent *rename* operations

4.1 Proposed *rename* operation

Our *rename* operation enables RenamableLogootSplit to reduce the overhead of nodes replica. To do so, it reassigns arbitrary identifiers to elements.

Its behaviour is illustrated in Figure 3. In this example, node A initiates a *rename* operation on its local state. First, node A reuses the id of the first element of the sequence (i_0^{B0}) but modifies it with its own node id (**A**) and current sequence number (*1*). Also the offset is set to 0. Node A reassigns the resulting id (i_0^{A1}) to the first element of the sequence as described in 3a. Then, node A derives contiguous identifiers for all remaining elements by successively incrementing the offset (i_1^{A1}, i_2^{A1} and i_3^{A1}), as shown in 3b. As we assign contiguous identifiers to all elements of the sequence, we eventually group them into one block as illustrated in 3c. It allows nodes to benefit the most from the block feature and to minimise the overhead of the resulting state.

To converge, other nodes have to rename their state identically. However, they can not simply replace their current state with the new renamed one. Indeed, they may have performed concurrent updates on their states. In order not to discard these updates, nodes have to process the *rename* operation themselves. To this end, the node issuing the *rename* operation broadcasts its *former state* to others. Using this state, other nodes compute the new identifier of each renamed identifier. As for concurrently inserted identifiers, we explain in subsection 4.2 how nodes rename them in a deterministic way.

4.2 Dealing with concurrent updates

After applying *rename* operations on their local state, nodes may receive concurrent updates. Figure 4 illustrates such cases. In this example node B inserts the new element "L", assigns the id $i_0^{B0}m_0^{B1}$ to it and broadcasts its update, concurrently to the *rename* operation described in Figure 3. Upon reception of the *insert* operation, node A adds the inserted element into its sequence, using the element id to determine its position. However, since identifiers were modified by the concurrent *rename* operation, node A inserts the new element at the end of its sequence (since $i_3^{A1} < i_0^{B0}m_0^{B1}$) instead of at the intended position. As described by this example, applying naively concurrent updates would result in inconsistencies. It is thus necessary to handle concurrent operations to *rename* operations in a particular manner.

First, nodes have to detect concurrent operations to *rename* ones. To this end, we use an *epoch-based* system. Initally, the replicated sequence starts at the *origin* epoch noted ε_0. Each *rename* operation introduces a new epoch and enables nodes to advance their states to it from the previous epoch. The generated epoch is characterised using the node id and its current sequence number upon the generation of the *rename* operation. For example, the *rename* operation described in Figure 4 enables nodes to advance their states from ε_0 to ε_{A1}.

As they receive *rename* operations, nodes build and maintain the *epoch chain*, a data structure ordering epochs according to their *parent-child* relation. Additionally, nodes tag every operation with their current epoch at the time the operation is generated. Upon the reception of an operation, nodes compare the operation epoch to their current one. If they differ, nodes have to transform the operation before applying it. Nodes determine against which *rename* operations to transform the received operation by computing the path between the operation epoch and their current one using the *epoch chain*. For this purpose, it is required to add the following rule to existing constraints upon the delivery of operations: operations must now be delivered after the *rename* operation which introduced their epoch.

Nodes use the function $renameId$, described in Algorithm 1, to transform *insert* or *remove* operations against *rename* ones. This algorithm maps identifiers from a *parent* epoch to corresponding ones in the *child* epoch. The main idea of this algorithm is to rename unknown identifiers at the time of the *rename* operation generation using their predecessor. An example of its usage is illustrated in Figure 5. This figure depicts the same scenario as in Figure 4, except that this time node A uses Algorithm 1 to rename the concurrently generated id before inserting it in its state.

The algorithm proceeds as follows. First, node A retrieves the predecessor of the given id $i_0^{B0}m_0^{B1}$ in the former state: $i_0^{B0}f_0^{A0}$. Then it computes the counterpart of $i_0^{B0}f_0^{A0}$ in the renamed state: i_1^{A1}. Finally, node A prepends it to the given id to generate the renamed id: $i_1^{A1}i_0^{B0}m_0^{B1}$. By reassigning this id to the concurrently added element, node A is able to insert it in its state while preserving the intended order.

Algorithm 1 also enables nodes to handle the opposite case : to integrate remote *rename* operations on their local state while they have previously applied concurrent updates. This case corresponds to node B's one in Figure 5. Upon the delivery of node A's *rename* operation, applying Algorithm 1 to every identifiers of its state would enable node B to reach an equivalent state to node A's one.

Algorithm 1 features only the main case of $renameId$, i.e. when the identifier to rename is in the range of renamed identifiers ($firstId \leq id \leq lastId$). Functions to deal with

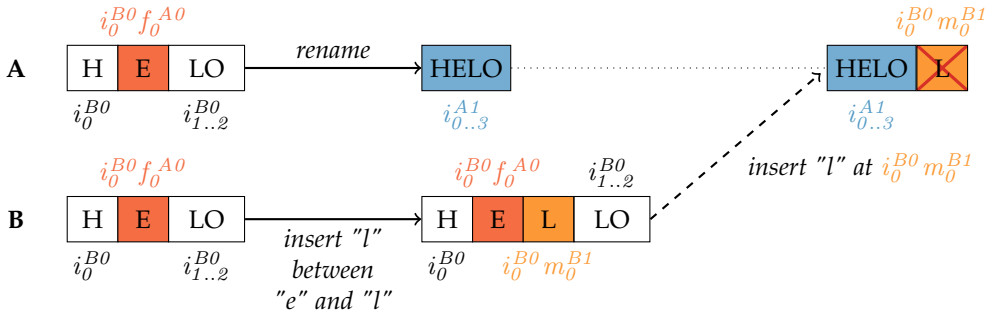


Fig. 4. Concurrent update leading to inconsistency

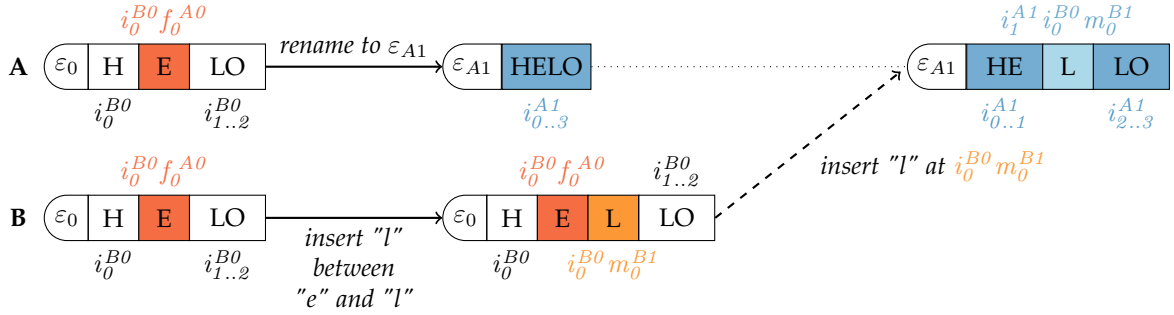


Fig. 5. Renaming concurrent update using Algorithm 1 before applying it to maintain intended order

```
function RENAMEID(id, renamedIds, nId, nSeq)
    length ← renamedIds.length
    firstId ← renamedIds[0]
    lastId ← renamedIds[length - 1]
    pos ← position(firstId)

    if id < firstId then
        newFirstId ← new Id(pos, nId, nSeq, 0)
        return renIdLessThanFirstId(id, newFirstId)
    else if id ∈ renameIds then
        index ← findIndex(id, renamedIds)
        return new Id(pos, nId, nSeq, index)
    else if lastId < id then
        newLastId ← new Id(pos, nId, nSeq, length - 1)
        return renIdGreaterThanLastId(id, newLastId)
    else
        return renIdFromPredId(id, renamedIds, pos, nId,
nSeq)
    end if
end function

function RENIDFROMPREDID(id, renamedIds, pos, nId,
nSeq)
    index ← findIndexOfPred(id, renamedIds)
    newPredId ← new Id(pos, nId, nSeq, index)

    return concat(newPredId, id)
end function
```

Alg 1. Rename concurrently generated identifier

other cases, i.e. when the identifier to rename is out of the range of renamed identifiers ($id < firstId$ or $lastId < id$), are presented in Appendix A.

4.3 Optimisations

TODO: Ajouter une phrase ou deux pour introduire cette sous-section – Matthieu

Garbage collection and offloading of former states Since nodes rely on the former state to transform concurrent operations to a *rename* one, they have to store it. Nodes need it until each of them can no longer issue concurrent operations to the corresponding *rename* operation. In other words, nodes can safely garbage collect the former state once the *rename* operation became causally stable [35]. To determine that a given *rename* operation is causally stable, nodes have to be aware of others and of their progress. A group membership protocol such as [36], [37] is thus required.

Causal stability may take some time to be achieved. Meanwhile, nodes can actually offload former states onto the disk since they are only required to handle concurrent operations to *rename* ones. We discuss this topic further in subsection 7.2.

Compression technique for* rename *operations To limit bandwidth consumption of *rename* operations, we propose the following compression technique. Node may broadcast only necessary components to uniquely identify blocks instead of whole identifiers. Indeed, an identifier can be uniquely identified from the *node identifier*, *node sequence number* and *offset* of its last tuple. A block can therefore be uniquely identified from these components and its length. This reduces the data to send to a fixed amount per block. To decompress the received operation, nodes browse their

current state and log of concurrent *remove* operations. This allows them to retrieve whole identifiers and to reconstruct the original *rename* operation. Additionally, we can set an upper-bound to the size of *rename* operations by issuing them as soon as the state reaches a given number of blocks.

5 RenamableLogootSplit with concurrent *rename* operations

5.1 Concurrent *rename* operations

We now consider scenarios with concurrent *rename* operations. Figure 6 depicts such one. This figure expands the scenario previously described in Figure 5.

After broadcasting its *insert* operation, node B performs a *rename* operation on its state. This operation reassigns new identifiers to every element based on the id of the first element of the sequence (i_0^{B0}), its node id (**B**) and current sequence number (2). This operation also introduces a new epoch: ε_{B2}. Since node A's *rename* operation was not yet delivered to node B at that moment, both *rename* operations are concurrent.

Node A and B then synchronise: each node applies the other one's *rename* operation. However, although they applied the same set of operations, their resulting states diverge. The identifiers attached to every element as well as the current epoch are now different. While the content of both sequences is still identical at that moment, the divergence will only widen. Subsequent operations will rely on these divergent identifiers to express at which position to insert new elements or which ones to remove. The same operations will thus produce different outputs on each copy.

The divergence depicted in Figure 6 occurs since the proposed *rename* operation is not commutative with itself. As concurrent operations may be delivered and applied in different orders to each nodes, conflicts may arise as a result. Nodes have to solve these conflicts to ensure the convergence of their state.

To this end, we propose the following approach: nodes effectively apply only one *rename* operation from a set of concurrent *rename* operations. The insight behind this approach is that *rename* operations are system operations: they have no effect on the document content and they do not carry any user intention. Therefore nodes can ignore several *rename* operations from a set of concurrent *rename* ones without introducing any anomalies. Thus, as long as every nodes apply one same *rename* operation from the set of concurrent *rename* ones, they will be able to benefit from its effects while avoiding conflicts.

We denote the one *rename* operation that nodes apply from a set of concurrent *rename* operations as the *primary* one. Other operations from the set are called *secondary* ones.

To make this approach feasible, we have to address two issues. First, nodes have to be able to select the *primary rename* operation to apply from a set of concurrent ones. In order to avoid performance issues due to coordination, nodes should select this operation in a coordination-free manner, i.e. solely using data from the set of concurrent *rename* operations. We propose such a mechanism in subsection 5.2. Second, nodes have to able to revert the effect of applied *rename* operations. Upon the reception of *rename* operations, nodes can only rely on their current knowledge to make decisions. However, nodes have only partial knowledge of the state of the system and of the operations issued at a given time. They may thus consider a received *rename* operation as the *primary* one and apply it, only to learn a few moments later that there is a concurrent *rename* operation with higher priority. In that case, nodes have to revert mistakenly applied *rename* operations to apply the correct *primary* one eventually. We present a corresponding algorithm in subsection 5.3.

5.2 Breaking tie between concurrent *rename* operations

TODO: Reformuler/Simplifier 1er paragraphe – Matthieu As stated in subsection 4.2, nodes build and maintain the *epoch chain* from received *rename* operations. This data structure orders epochs according to their *parent-child* relation. Using the *epoch chain*, nodes determine their current epoch as the latest one. However, in case of concurrent *rename* operations, this data structure is no longer suitable as several epochs may share the same parent epoch. Epochs now form the *epoch tree*. Several epochs may be considered as the latest ones. To converge, every node should eventually select the same epoch as the *primary* one.

To this end, we define *priority*, a total order relation between epochs. The goal of this relation is to enable nodes to designate the *primary rename* operation from a set of concurrent ones, but also the current epoch from the set of all issued *rename* operations, in a coordination-free manner.

To define the *priority* relation, we may actually select different strategies. In this work, we use the lexicographical order on the path of epochs in the *epoch tree*. Figure 7 provides an example of its use.

This figure displays the evolution of node A and B's *epoch trees* during the execution of Figure 6's scenario. Red dashed arrows represents the order between epochs according to the *priority* relation while current epochs are displayed as red nodes.

Initially, both nodes are only aware of their own *rename* operation. Node A's (resp. node B's) *epoch tree* is thus only composed of the epochs ε_0 and ε_{A1} (resp. ε_0 and ε_{B2}). Using the *priority* relation, both nodes order epochs to determine the current one. According to the lexicographical order on the path of epochs in the *epoch tree*, node A (resp. node B) establishes that $\varepsilon_0 < \varepsilon_0\varepsilon_{A1}$ (resp. $\varepsilon_0 < \varepsilon_0\varepsilon_{B2}$). Therefore node A selects ε_{A1} as its current epoch while node B picks ε_{B2}.

Then, both nodes synchronise and their *epoch trees* converge. Using the *priority* relation, they are able to decide which *rename* operation to deem as the *primary* one in a coordination-free manner. Nodes establish that $\varepsilon_0 < \varepsilon_0\varepsilon_{A1} < \varepsilon_0\varepsilon_{B2}$. Therefore, they choose ε_{B2} as the new *primary* and current epoch. As ε_{B2} was already node B's current epoch, node B does not rename its state upon the reception of the *rename* operation to ε_{A1}. On the other hand, node A has to *rename* its state from ε_{A1} to ε_{B2}. We explain how node A proceeds in subsection 5.3.

Other strategies could be proposed to define the *priority* relation. For example, *priority* could rely on metrics embedded in *rename* operations representing the accumulated work on the document. This topic will be further discussed in subsection 7.3.

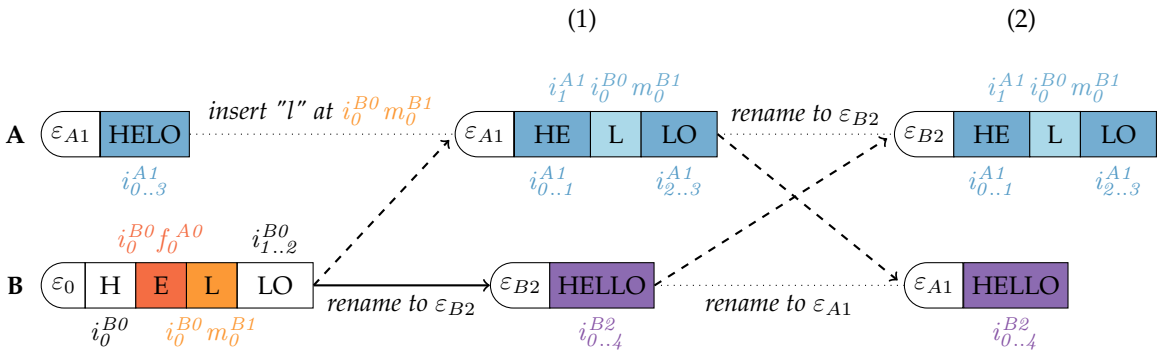
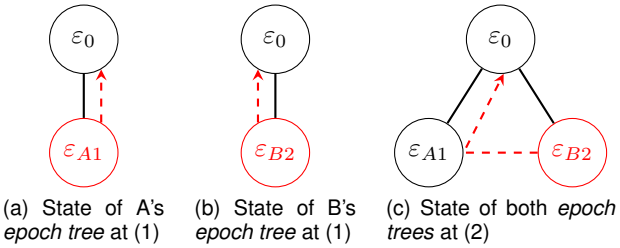


Fig. 6. Concurrent *rename* operations leading to divergent states



(a) State of A's *epoch tree* at (1) (b) State of B's *epoch tree* at (1) (c) State of both *epoch trees* at (2)

Fig. 7. Evolution of *epoch trees* during the scenario of Figure 6

5.3 Applying *primary rename* operations

As illustrated in subsection 5.2, nodes may have to rename their state from an epoch to a concurrent one. However, the proposed algorithm in Algorithm 1 is only designed to rename to an epoch from its parent one. To address this issue, we propose that nodes having applied concurrent *secondary rename* operations first revert the effect of these concurrently applied *rename* operations.

We thus introduce the function *revertRenameId*, described in Algorithm 2. The goals of *revertRenameId* are the following: (i) To revert identifiers generated causally before or concurrently to the reverted *rename* operation to their former value (ii) To assign new ids complying with the intended order to elements inserted causally after the reverted *rename* operation.

TODO: Ajouter commentaire dans l'algo pour préciser le cas où on rename un id inséré de façon concurrente ou causale au rename – Matthieu

Upon the reception of a *rename* operation introducing a concurrent but primary epoch to their current one, nodes have first to determine which *rename* operations to revert. To this end, nodes use the *epoch tree*. By identifying the Lowest Common Ancestor (LCA) of their current epoch and of the new *primary* one, nodes can determine which operations to revert. Nodes have then to revert *rename* operations applied since the LCA epoch between their current epoch and the new primary one in the reverse order, using Algorithm 2. The Figure 8 illustrates such a scenario.

This figure describes the same scenario as Figure 6, except that nodes now use the *priority* to determine how to process the concurrent *rename* operation received.

Upon the reception of both *rename* operations, nodes compares introduced epochs to determine the *primary* one.

```
function REVERTRENAMEID(id, renamedIds, nId, nSeq)
    length ← renamedIds.length
    firstId ← renamedIds[0]
    lastId ← renamedIds[length - 1]
    pos ← getPosition(firstId)

    predOfNewFirstId ← new Id(pos, nId, nSeq, -1)
    newFirstId ← new Id(pos, nId, nSeq, 0)
    newLastId ← new Id(pos, nId, nSeq, length - 1)

    if id < newFirstId then
        return revRenIdLessThanNewFirstId(id, firstId, newFirstId)
    else if isRenamedId(id, pos, nId, nSeq, length) then
        index ← getFirstOffset(id)
        return renamedIds[index]
    else if newLastId < id then
        return revRenIdGreaterThanNewLastId(id, lastId)
    else
        index ← getFirstOffset(id)
        return revRenIdfromPredId(id, renamedIds, index)
    end if
end function

function REVRENIDFROMPREDID(id, renamedIds, index)
    predId ← renamedIds[index]
    succId ← renamedIds[index + 1]
    tail ← getTail(id, 1)

    if tail < predId then
        return concat(predId, MIN_TUPLE, tail)
    else if succId < tail then
        offset ← getLastOffset(succId) - 1
        predOfSuccId ← createIdFromBase(succId, offset)
        return concat(predOfSuccId, MAX_TUPLE, tail)
    else
        return tail
    end if
end function
```

Alg 2. Revert rename identifier

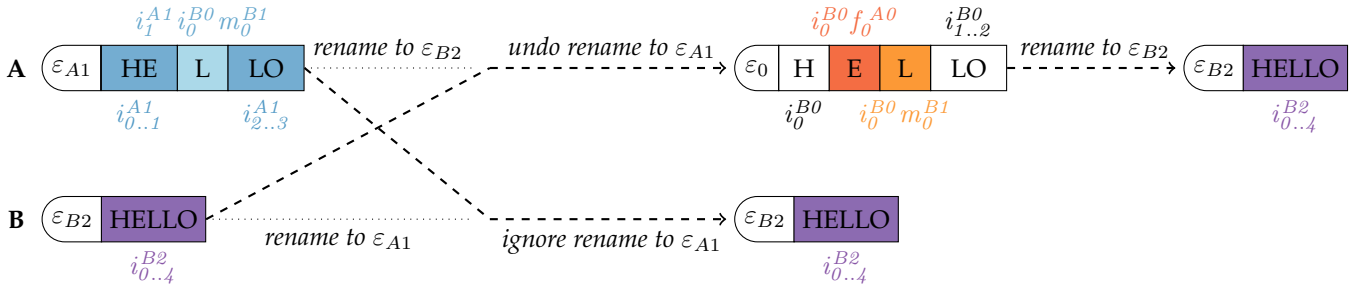


Fig. 8. Handling concurrent *rename* operations

According to the proposed *priority* relation, nodes deems ε_{B2} as *primary* one since $\varepsilon_0 < \varepsilon_0\varepsilon_{A1} < \varepsilon_0\varepsilon_{B2}$. Since node B's current epoch is already ε_{B2}, node B can simply ignore the received *rename* operation. Meanwhile, node A has to rename its state from ε_{A1} to ε_{B2}.

Node A first determines the LCA between ε_{A1} and ε_{B2}: ε_0. It then undoes each *rename* operations that separates its current epoch from ε_0, i.e. the *rename* operation from ε_0 to ε_{A1}. To this end, node A uses Algorithm 2.

Algorithm 2 enables node A to retrieve fitting counterparts for every identifiers of its current state. For identifiers of the form i^{A1}_{offset}, it simply uses their offset to retrieve the original identifiers, as offsets correspond to the identifier indexes in *renamedIds*. For other identifiers such as $i^{A1}_1 i^{B0}_0 m^{B1}_0$, Algorithm 2 removes its prefix (i^{A1}_1) to isolate its tail ($i^{B0}_0 m^{B1}_0$). The algorithm returns the tail if it fits between the identifier of its predecessor ($i^{B0}_0 f^{A0}_0 < i^{B0}_0 m^{B1}_0$) and the identifier of its successor ($i^{B0}_0 m^{B1}_0 < i^{B0}_1$). If it would not, Algorithm 2 would use exclusive tuples of the renaming mechanism, MIN_TUPLE and MAX_TUPLE, to generate an identifier complying with the intended order.

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

As with Algorithm 1, Algorithm 2 only features the main case of $revertRenameId$. It corresponds to the case where the identifier to revert is in the range of renamed identifiers ($newFirstId \leq id \leq newLastId$). Functions to handle the rest of cases are featured in Appendix B.

Note that Algorithm 1 and Algorithm 2 are not inverse functions. Algorithm 2 reverts to their original value identifiers inserted causally before or concurrently to the *rename* operation. But on the other hand, Algorithm 1 does not do the same for identifiers inserted causally after the *rename* operation. Thus redoing a previously undone *rename* operation alter these identifiers. This alteration may cause a divergence between nodes, as the same element will be referred to using different identifiers.

This issue is however prevented in our system by the proposed *priority* relation. Since the *priority* relation is defined using the lexicographical order on the path of epochs in the *epoch tree*, nodes only move towards the rightmost epoch of the *epoch tree* when switching epochs. Nodes thus avoid going back and forth between different epochs and undoing then redoing corresponding *rename* operations.

Nonetheless, the requirement to prevent nodes from redoing *rename* operation limits our possibilities to design new *priority* relations. It would be interesting to remove this constraint to propose more effective *priority* relations, as discussed in subsection 7.3. It can be achieved by designing a set of inverse functions for Algorithm 1 and Algorithm 2. Another solution is to propose an alternative implementation of the renaming mechanism that use original identifiers instead of transformed ones, for example by relying on an *operation log*.

5.4 Garbage collection of *former states*

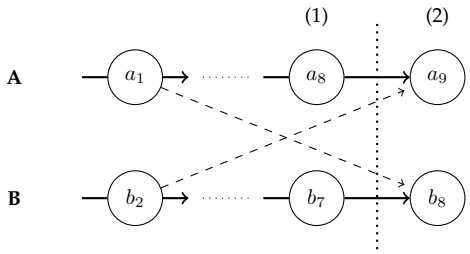
As explained in subsection 4.2 and subsection 5.3, nodes have to store epochs and corresponding *former states* to transform operations from previous or concurrent epochs to the current one, or to rename their state from their current epoch to the new *primary* one. However, once nodes become aware that some epochs can not possibly be required anymore to apply future operations, they can garbage collect these epochs and their corresponding *rename* operations. But, in systems which allow the generation of concurrent *rename* operations, nodes can not solely rely on causal stability of *rename* operations to determine which operations can be garbage collected. Therefore, we propose the two following rules to enable nodes to identify unnecessary epochs:

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

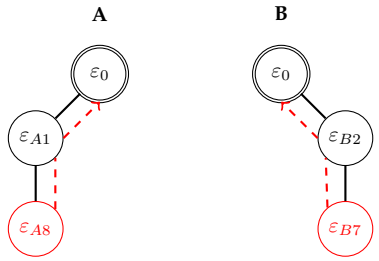
Rule 2. An epoch ε can be garbage collected if ε is the root of the epoch tree, has only one child ε' and that ε' is causally stable.

Figure 9 illustrates a use case of Rule 1 and Rule 2. In 9a, we represent an execution in which two nodes A and B respectively issue two *rename* operation before eventually synchronising. The operations shown in 9a are solely *rename* operations, as only these operations are relevant to the problem at hand. In 9b, we represent the states of their respective *epoch trees* once they each generate their *rename* operations. In 9c, we represent the new states of their *epoch trees* once they each receive the first *rename* operation issued by each other. Epochs introduced by causally stable *rename* operations are represented in *epoch trees* using double circles. The *origin* epoch ε_0 is considered as causally stable from the beginning by design. Epochs that are no longer necessary by applying Rule 1 and Rule 2 are respectively displayed as blue dashed nodes and green dotted nodes.

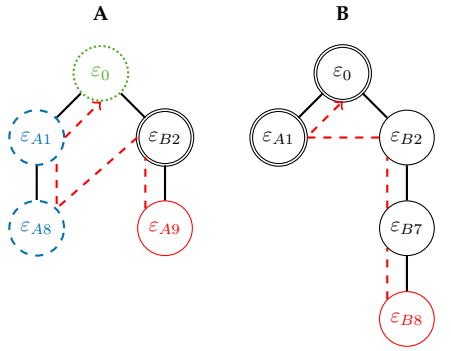
Upon the delivery of the *rename* operation introducing epoch ε_{B2} to node A, ε_{B2} becomes causally stable. From this



(a) Execution



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



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

Fig. 9. Garbage collecting epochs and corresponding *former states*

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

On the other hand, upon the reception of the *rename* operation introducing ε_{A1}, node B can not garbage collect any epochs despite ε_{A1} being causally stable. Indeed, from its point of view, other nodes may still issue operations from epoch ε_{A1}. Since in that case node B would have to transform operations to apply them to ε_{B8}, node B has to retain all epochs forming the path between ε_{A1} and ε_{B8} and their corresponding *rename* operations.

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

6 Evaluation

6.1 Simulations and benchmarks

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

TODO: Reformuler paragraphe pour juste dire qu'on a utilisé des simulations pour notre évaluation (plutôt que d'insister sur le fait qu'on a pas réussi à avoir de traces réelles) – Matthieu
Unfortunately, we were not able to retrieve an existing dataset of real-time collaborative editing sessions. We thus setup simulations to generate the dataset used to run our benchmarks. These simulations mimic the following scenario.

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

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

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

For the purpose of reproducibility, we make the code, benchmarks and results available at: https://github.com/coast-team/mute-bot-random/.

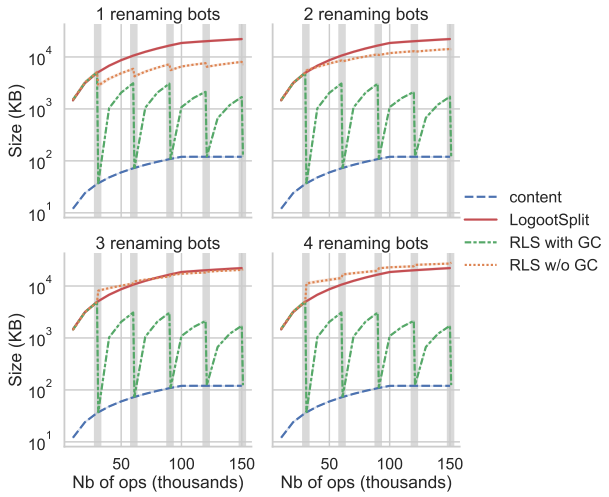


Fig. 10. Evolution of the size of the document

6.2 Results

Using generated snapshots, we performed several benchmarks. These benchmarks evaluate RenamableLogootSplit's performances and compare them to LogootSplit's ones. Results are presented and analysed below.

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

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

Memory overhead We then proceeded to measure the evolution of the document's memory consumption throughout the simulations, according to the CRDT used and the number of *renaming bots*. We present the obtained results in Figure 10.

For each plot displayed in Figure 10, we represent 4 different data. The blue dashed line illustrates the size of the actual content of the document, i.e. the text, while the red solid line corresponds to the size of the whole LogootSplit document.

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

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

To summarise the results presented, the renaming mechanism introduces a temporary metadata overhead which increases with each *rename* operations. But the overhead will eventually subsides once the system becomes quiescent and *rename* operations become causally stable. In subsection 7.2, we discuss that *former states* may be offloaded until causal stability is achieved to address the temporary memory overhead.

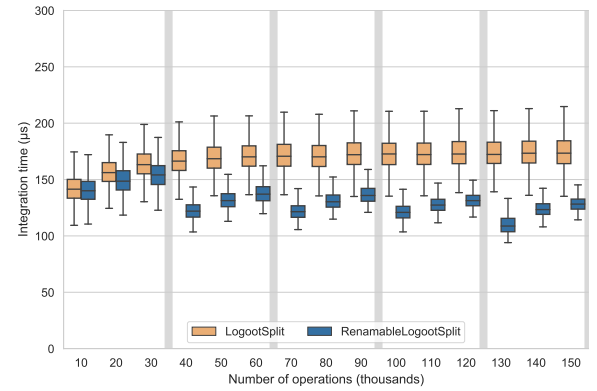
Integration times of standard operations Next, we compared the evolution of integration times of standard operations, i.e. *insert* and *remove* operations, on LogootSplit and RenamableLogootSplit documents. Since both types of operation share the same time complexity, we used solely *insert* ones in our benchmarks. We do however distinguish *local* and *remote* updates. Conceptually, local updates can be decomposed as presented in [38] in the two following steps: (i) the generation of the corresponding operation (ii) the application of the resulting operation on the local replica. However, for performance reasons, we merged these two steps in our implementation. We thus make a distinction between *local* and *remote* updates in our benchmarks. Figure 11 displays the results.

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

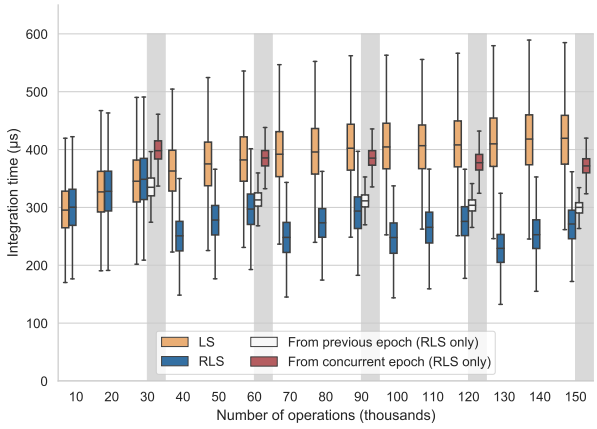
Additionally, in the case of remote operations, we measured specific integration times for RenamableLogootSplit: integration times of remote operations from previous epochs and from concurrent epochs, respectively displayed as white and red boxplots in 11b. Operations from previous epochs are operations generated concurrently to the *rename* operation but applied after it. Since the operation has to be transformed beforehand using Algorithm 1, we observe a computational overhead compared to other operations. But this overhead is actually compensated by the optimisation of the internal representation of the sequence performed by *rename* operations.

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

To summarise, transformation functions introduce an overhead with regard to integration times of concurrent operations to *rename* ones. Despite this overhead, Renamable-



(a) Local updates



(b) Remote updates

Fig. 11. Integration time of standard operations

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

Integration time of* rename *operation Finally, we measured the evolution of integration times of *rename* operation according to the number of operations since the last *rename* one. As before, we distinguish performances of *local* and *remote* updates. The results are displayed in Table 1.

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

Another interesting result from this benchmark is that concurrent *secondary rename* operations are cheap to apply, as they only consist in storage of corresponding *former states*. Thus nodes can significantly reduce the overall computations of a set of concurrent *rename* operations by applying them in a specific order. We will discuss further this topic in subsection 7.4.

TABLE 1
Integration time of rename operations

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

7 Discussion

7.1 Issuing *rename* operations

As stated in subsection 3.2, *rename* operations are system operations. It is thus up to the designers of the system to determine when nodes should issue *rename* operations and to define a corresponding strategy. However, there is no silver bullet since each system has its own constraints.

Several aspects should be taken into account to define the strategy. The first one is the size of the data structure. As displayed in Figure 10, metadata progressively increases to represent 99% of the data structure. Using *rename* operations, nodes can discard metadata and reduce the size of the data structure to an acceptable amount. To determine when to issue *rename* operations, nodes may monitor the number of operations performed since the last *rename* one, the number of blocks composing the sequence or the length of identifiers.

A second aspect to take into account is the integration time of *rename* operations. As reported in Table 1, integrating remote *rename* operations takes up to seconds if delayed for too long. Although *rename* operations work behind the scenes, they can still impact negatively the user experience. Indeed, nodes can not integrate operations from others while they are processing *rename* operations From users' points of view, *rename* operations can thus be perceived as latency peaks. In the domain of real-time collaborative editing, user experiments have shown that delay degrades the quality of collaborations [39], [40]. It thus important to issue *rename* operations frequently to keep their integration times below the perceptible limit.

Finally, the last aspect to consider is the amount of concurrent *rename* operations. Figure 10 shows that concurrent *rename* operations decrease RenamableLogootSplit's

performances. The proposed strategy must then aim to minimise the number of concurrent *rename* operations issued. However, it should avoid to rely on synchronous coordination between nodes to do so. To reduce the likelihood of issuing concurrent *rename* operations, several techniques can be proposed. For example, nodes can monitor to which other nodes they are currently connected and delegate to the one with the highest *node identifier* the duty to issue *rename* operations.

7.2 Offloading on disk unused *former states*

As explained in subsection 4.2 and subsection 5.3, nodes have to store *former states* corresponding to *rename* operations to transform operations from previous or concurrent epochs. Nodes may receive such operations given 2 different cases: (i) nodes have recently issued *rename* operations (ii) nodes logged back in the collaboration. Between these specific events, *former states* are actually not needed to handle operations.

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

7.3 Designing a more effective *priority* relation

Although the *priority* relation proposed in subsection 5.2 is simple and ensures that nodes designate the same epoch as the *primary* one, it introduces a significant computational overhead in some cases. Notably it allows a single node, disjoined from the collaboration since a long time, to force every other nodes to revert *rename* operations they performed meanwhile because its own *rename* operation is deemed as the *primary* one.

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

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

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

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

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

8 Related work

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

8.1 The core-nebula approach

The *core-nebula* approach [18], [19] was proposed to reduce the size of identifiers in Treedoc [26], another variable-size identifiers Sequence CRDT.

In this work, authors introduce a *rebalance* operation enabling nodes to reassign shorter identifiers to elements of the document. However, this *rebalance* operation is not commutative with *insert* and *remove* operations nor with itself. To achieve Eventual Consistency (EC) [41], the *core-nebula* approach prevents concurrent *rebalance* operations by regulating them using a consensus protocol. Operations such as *insert* and *remove* can still be issued without coordination and can thus be concurrent to *rebalance* ones. To deal with this issue, authors propose a *catch-up* protocol to transform these concurrent operations against the effects of *rebalance* ones.

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

Our work can be seen as an extension of this work. It adapts the *rebalance* mechanism and the *catch-up* protocol

to LogootSplit and takes advantage of its block feature. Furthermore, it integrates a mechanism to deal with concurrent *rename* operations, hence removing the requirement of a consensus protocol. It makes this approach usable in systems without existing authorities providing nodes to the *core*.

However, systems can actually adopt the *core-nebula* approach to simplify the implementation of RenamableLogootSplit. The use of a consensus protocol to regulate *rename* operations enables systems to discard all parts dedicated to the handling of concurrent *rename* operations, i.e. the design of a *priority* relation and the implementation of Algorithm 2 and Rule 1.

8.2 The LSEQ approach

The LSEQ approach [30], [31] is another approach proposed to address the growth of identifiers in variable-size identifiers Sequence CRDT. Instead of reducing periodically the identifier metadata using an expensive renaming mechanism, the authors define new identifier allocation strategies to reduce their growth rate.

In this work, authors observe that the identifier allocation strategy proposed in Logoot [27] is suited to a single editing pattern: from left to right, top to bottom. If insertions are made according to other patterns, generated identifiers quickly saturate the space of possible identifiers for a given size. Following insertions therefore trigger an increase of the identifier size. As a result, Logoot identifiers grow linearly with the number of insertions instead of the expected logarithmic progression.

LSEQ thus defines several identifier allocation strategy fitted to different editing pattern. Nodes pick randomly one of these strategies for each identifier size. Additionally LSEQ adopts an exponential tree model for identifiers: the range of possible identifiers doubles as the identifier size increases. It enables LSEQ to fine-tune the size of identifiers according to needs. By combining the different allocation strategies to the exponential tree model, LSEQ achieves a polylogarithmic growth of identifiers according to the number of insertions.

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

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

9 Conclusions and future work

In this paper, we introduced a novel Sequence CRDT belonging to the variable-size identifiers approach: RenamableLogootSplit. This new data structure embeds a renaming mechanism in its specification. This mechanism enables nodes to reassign shorter identifiers to elements and to group them into one block to minimise metadata. The renaming mechanism takes the form of *rename* operations, system operations that are triggered automatically by nodes when deemed necessary. However, the proposed *rename* operation is not commutative with *insert* and *remove* operations, nor with itself. Conflicts hence occur when operations are issued concurrently to *rename* ones. We thus described how RenamableLogootSplit adopts and uses Operational Transformation (OT) techniques to solve resulting conflicts.

We then presented the results of our empirical validation of RenamableLogootSplit through experiments. Experiment results show that the renaming mechanism itself may temporary increase the metadata of the data structure, notably in case of concurrent *rename* operations. However the renaming mechanism enables nodes to garbage collect its own metadata eventually. Nodes are thus able to reduce the size of their data structure by several hundred times compared to previous work eventually.

As future work, we will now focus on the design of strategies to manage the generation of *rename* operations. These strategies have to meet two different goals. First, they have to minimise the impact of *rename* operations on the user experience. User behaviour studies, inspired by [39], [40], could be led in the context of real-time collaborative writing to set an acceptable upper-bound to their integration times. Strategies should then issue *rename* operations accordingly to keep integration times below the defined threshold. Secondly, strategies have to minimise the likelihood of issuing concurrent *rename* operations without relying on consensus protocols. Strategies should then use nodes' local knowledge to determine whether a node should issue a *rename* operation or trust another node to do so.

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

Appendix A
Algorithms for centralised settings

Appendix B
Algorithms for distributed settings

Acknowledgments

The authors would like to thank...

References

[1] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[2] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, p. 42–81, Mar. 2005. [Online]. Available: https://doi.org/10.1145/1057977.1057980

[3] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS 2011, 2011, pp. 386–400.

```
function RENIDLESSTHANFIRSTID(id, newFirstId)
    if id < newFirstId then
        return id
    else
        pos ← position(newFirstId)
        nId ← nodeId(newFirstId)
        nSeq ← nodeSeq(newFirstId)
        predNewFirstId ← new Id(pos, nId, nSeq, -1)

        return concat(predNewFirstId, id)
    end if
end function

function RENIDGREATERTHANLASTID(id, newLastId)
    if id < newLastId then
        return concat(newLastId, id)
    else
        return id
    end if
end function
```

Alg 3. Remaining algorithms to rename an identifier

```
function REVRENIDLESSTHANNEWFIRSTID(id, firstId, newFirstId)
    predNewFirstId ← createIdFromBase(newFirstId, -1)
    if predNewFirstId < id then
        tail ← getTail(id, 1)
        if tail < firstId then
            return tail
        else
            offset ← getLastOffset(firstId)
            predFirstId ← createIdFromBase(firstId, offset)
            return concat(predFirstId, MAX_TUPLE, tail)
        end if
    else
        return id
    end if
end function

function REVRENIDGREATERTHANNEWLASTID(id, lastId)
    if id < lastId then
        return concat(lastId, MIN_TUPLE, id)
    else
        return id
    end if
end function
```

Alg 4. Remaining functions to revert an identifier renaming

[4] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near real-time peer-to-peer shared editing on extensible data types," in *19th International Conference on Supporting Group Work*, ser. GROUP 2016. ACM, Nov. 2016, pp. 39–49.

[5] Yjs, "Yjs: A CRDT framework with a powerful abstraction of shared data." [Online]. Available: https://github.com/yjs/yjs

[6] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, p. 2733–2746, Oct 2017. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2017.2697382

[7] Automerge, "Automerge: data structures for building collaborative applications in Javascript." [Online]. Available: https://github.com/automerge/automerge

[8] Riak, "Riak KV." [Online]. Available: http://riak.com/

[9] T. S. Consortium, "AntidoteDB: A planet scale, highly available, transactional database." [Online]. Available: http://antidoteDB.eu/

[10] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein, "Anna: A kvs for any scale," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 2, pp. 344–358, 2021.

[11] Concordant, "Concordant." [Online]. Available: http://www.concordant.io/

[12] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: You own your data, in spite of the cloud," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 154–178. [Online]. Available: https://doi.org/10.1145/3359591.3359737

[13] P. van Hardenberg and M. Kleppmann, "PushPin: Towards production-quality peer-to-peer collaboration," in *7th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC 2020. ACM, Apr. 2020.

[14] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 59–68. [Online]. Available: https://doi.org/10.1145/289444.289469

[15] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for Real-time Document Editing," in *11th ACM Symposium on Document Engineering*, ACM, Ed., Mountain View, California, United States, Sep. 2011, pp. 103–112. [Online]. Available: https://hal.inria.fr/inria-00629503

[16] B. Nédelec, P. Molli, and A. Mostefaoui, "CRATE: Writing stories together with our browsers," in *25th International World Wide Web Conference*, ser. WWW 2016. ACM, Apr. 2016, pp. 231–234.

[17] M. Nicolas, V. Elvinger, G. Oster, C.-L. Ignat, and F. Charoy, "MUTE: A Peer-to-Peer Web-based Real-time Collaborative Editor," in *ECSCW 2017 - 15th European Conference on Computer-Supported Cooperative Work*, ser. Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels, Posters and Demos, vol. 1, no. 3. Sheffield, United Kingdom: EUSSET, Aug. 2017, pp. 1–4. [Online]. Available: https://hal.inria.fr/hal-01655438

[18] M. Letia, N. Preguiça, and M. Shapiro, "Consistency without concurrency control in large, dynamic systems," in *LADIS 2009 - 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, ser. Operating Systems Review, vol. 44, no. 2. Big Sky, MT, United States: Assoc. for Computing Machinery, Oct. 2009, pp. 29–34. [Online]. Available: https://hal.inria.fr/hal-01248270

[19] M. Zawirski, M. Shapiro, and N. Preguiça, "Asynchronous rebalancing of a replicated tree," in *Conférence Française en Systèmes d'Exploitation (CFSE)*, Saint-Malo, France, May 2011, p. 12. [Online]. Available: https://hal.inria.fr/hal-01248197

[20] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 399–407. [Online]. Available: https://doi.org/10.1145/67544.66963

[21] D. Sun and C. Sun, "Context-based operational transformation in distributed collaborative editing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, pp. 1454 – 1470, 11 2009.

[22] G. Oster, P. Urso, P. Molli, and A. Imine, "Data Consistency for P2P Collaborative Editing," in *ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, ser. Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work. Banff, Alberta, Canada: ACM Press, Nov. 2006, pp. 259 – 268. [Online]. Available: https://hal.inria.fr/inria-00108523

[23] S. Weiss, P. Urso, and P. Molli, "Wooki: a p2p wiki-based collaborative writing tool," vol. 4831, 12 2007.

[24] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354 – 368, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731510002716

[25] L. Briot, P. Urso, and M. Shapiro, "High Responsiveness for Group Editing CRDTs," in *ACM International Conference on Supporting Group Work*, Sanibel Island, FL, United States, Nov. 2016. [Online]. Available: https://hal.inria.fr/hal-01343941

[26] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *2009 29th IEEE International Conference on Distributed Computing Systems*, June 2009, pp. 395–403.

[27] S. Weiss, P. Urso, and P. 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, Jun. 2009, pp. 404–412. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.75

[28] ——, "Logoot-Undo: Distributed Collaborative Editing System on P2P Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1162–1174, Aug. 2010. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00450416

[29] L. André, S. Martin, G. Oster, and C.-L. Ignat, "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.

[30] B. Nédelec, P. Molli, A. Mostéfaoui, and E. Desmontils, "LSEQ: an adaptive structure for sequences in distributed collaborative editing," in *Proceedings of the 2013 ACM Symposium on Document Engineering*, ser. DocEng 2013, Sep. 2013, pp. 37–46.

[31] B. Nédelec, P. Molli, and A. Mostéfaoui, "A scalable sequence encoding for collaborative editing," *Concurrency and Computation: Practice and Experience*, p. e4108. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4108

[32] M. Nicolas, "Efficient renaming in CRDTs," in *Middleware 2018 - 19th ACM/IFIP International Middleware Conference (Doctoral Symposium)*, Rennes, France, Dec. 2018. [Online]. Available: https://hal.inria.fr/hal-01932552

[33] M. Nicolas, G. Oster, and O. Perrin, "Efficient Renaming in Sequence CRDTs," in *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'20)*, Heraklion, Greece, Apr. 2020. [Online]. Available: https://hal.inria.fr/hal-02526724

[34] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, p. 240–247, May 1983. [Online]. Available: https://doi.org/10.1109/TSE.1983.236733

[35] C. Baquero, P. S. Almeida, and A. Shoker, "Making operation-based crdts operation-based," in *Distributed Applications and Interoperable Systems*, K. Magoutis and P. Pietzuch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 126–140.

[36] A. Das, I. Gupta, and A. Motivala, "Swim: scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 303–312.

[37] A. Dadgar, J. Phillips, and J. Currey, "Lifeguard: Local health awareness for more accurate failure detection," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 22–25.

[38] C. Baquero, P. S. Almeida, and A. Shoker, "Pure operation-based replicated data types," 2017.

[39] C.-L. Ignat, G. Oster, M. Newman, V. Shalin, and F. Charoy, "Studying the Effect of Delay on Group Performance in Collaborative Editing," in *Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, Springer 2014 Lecture Notes in Computer Science*, ser. Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, Seattle, WA, United States, Sep. 2014, pp. 191 – 198. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01088815

[40] C.-L. Ignat, G. Oster, O. Fox, F. Charoy, and V. Shalin, "How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking," in *European Conference on Computer Supported Cooperative Work 2015*, ser. Proceedings of the 14th European Conference on Computer Supported Cooperative Work, N. Boulus-Rodje, G. Ellingsen, T. Bratteteig, M. Aanestad, and P. Bjorn, Eds. Oslo, Norway: Springer International Publishing, Sep. 2015, pp. 223–242. [Online]. Available: https://hal.inria.fr/hal-01238831

[41] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, p. 172–182, Dec. 1995. [Online]. Available: https://doi.org/10.1145/224057.224070

PLACE PHOTO HERE

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.