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. Identifiers hence tend to grow as the system progresses, which increases the overhead of CRDTs over time and leads to performance issues. To address this issue, 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 trade-off 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 defines 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.

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 exhibit 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 ever-growing metadata decreases the memory efficiency of these data structures, it also increases their bandwith consumption and computational overhead. Some previous work proposed renaming mechanisms [18], [19] to reduce periodically 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 periodically minimise the memory overhead induced by metadata. 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 Transfor-*

• *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.

mations (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 ever-growing 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 explains how to garbage collect metadata introduced by the renaming mechanism itself. Section 7 presents the experimental evaluation of RenamableLogootSplit. Section 8 discusses several trade-offs that RenamableLogootSplit implementations offer. Section 9 compares our approach to related work. Finally Section 10 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. A 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.

Despite several comparisons [15], [25], [31], literature has provided no decisive evidence of the superiority of one approach on the other, but simply trade-offs. The tombstone approach offers a smaller network footprint since its identifiers are of fixed size, while the variable-size identifiers approach offers a better latency since updates can be delivered out of order. In this paper, we focus on the variable-size identifiers approach and more specifically on the LogootSplit algorithm. The general idea of the presented work is not specific to any technical aspects of this approach. As so, it might be applied on the other approach with some additional research work.

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: (i) a *position*, which embodies the intended position of the element (ii) a *node identifier* (iii) a *node sequence number* and (iv) 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.

Instead of storing an identifier for each element of the sequence, the main insight of LogootSplit is to dynamically aggregate elements into blocks. Grouping elements into blocks enables LogootSplit to logically assign 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 Figure 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 Figure 1b.

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

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

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 significantly reduce the memory overhead of the data structure.

2.2 Limits

As previously stated, 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 cannot 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 system progresses. This growth reduces 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

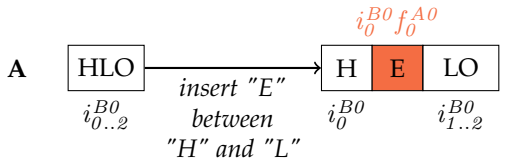


Fig. 2: Insertion leading to longer identifiers

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 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 7, 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 dynamically join and leave 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 asynchronously broadcast 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. No other constraints exist on the delivery order of operations. 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 without altering 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 eventually reaches 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. (Uniqueness 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.

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 achieve Property 4.

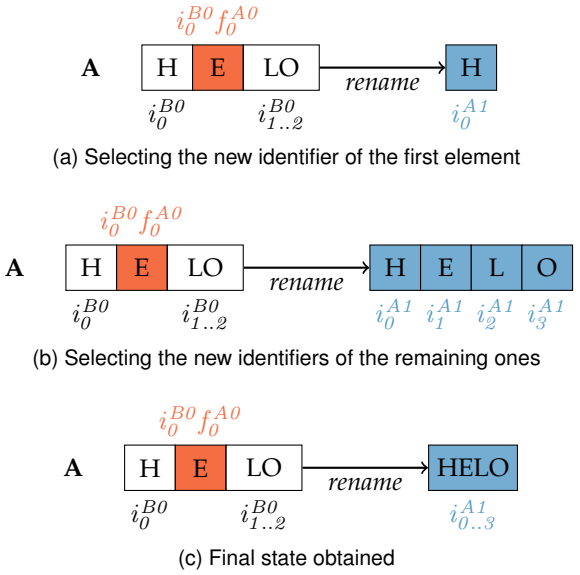
For the sake of simplicity, the presentation of the *rename* operation is divided into two parts. In Section 4, we present the proposed *rename* operation under the assumption that no concurrent *rename* operations may be issued. This assumption enables us to focus on the inner working of the

rename operation and on how to deal with concurrent *insert* and *remove* operations. Then, in Section 5, we remove this assumption and present our approach to handle scenarios with concurrent *rename* operations.

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.



(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

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 Figure 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 Figure 3b. As we assign contiguous identifiers to all elements of the sequence, we eventually group them into one block as illustrated in Figure 3c. It allows nodes to benefit the most from the block feature and to minimise the overhead of the resulting state. Proceeding this way also enables us to assign an unique triplet $\langle node\ id,\ seq, offset\rangle$ to every identifier, thus ensuring their uniqueness as required by Property 3.1.

To converge, other nodes have to rename their states identically. However, they cannot 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 (with regard to Property 2 and Property 4), nodes have to process the *rename* operation themselves. However, the applied renaming depends on the state[1] – called *former state* – of the node when the *rename* operation was generated. Other nodes need this information to compute the new renamed identifiers. The *rename* operation thus embeds the *former state* to propagate it.

1. More precisely, the intervals of identifiers composing the sequence.

4.2 Dealing with concurrent updates

After applying *rename* operations on their local states, 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, naively applying concurrent updates would result in inconsistencies and conflict with Property 2. 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 deterministically rename unknown identifiers at the time of the *rename* operation generation using their predecessor. An example is provided in Figure 5. This figure depicts the same scenario as in Figure 4, except that node A uses RenameID 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

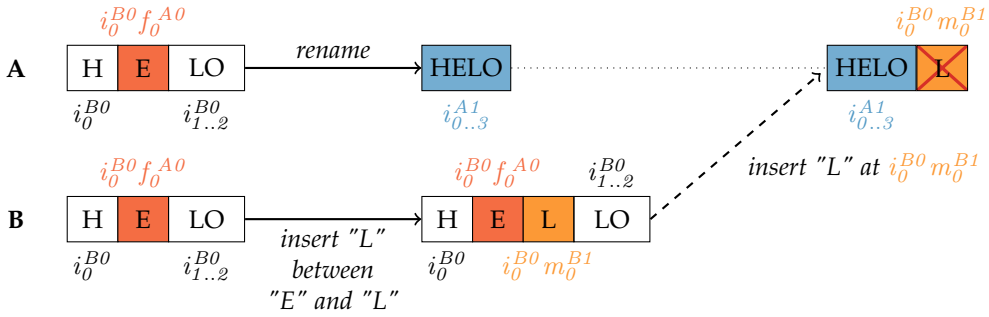


Fig. 4: Concurrent update leading to inconsistency

A is able to insert it in its state while preserving the 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 ∈ renamedIds 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: Main functions to rename an identifier

RENAMEID also enables nodes to handle the opposite case : to integrate remote *rename* operations on their local states 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 RENAMEID to every identifier of its state would enable node B to reach an equivalent state to node A's one. RENAMEID thus makes *rename* operations commutative with *insert* and *remove* ones.

Note that nodes rely on the *former state* – called here *renamedIds* – to apply RENAMEID. As such, nodes have to store it to be able to transform concurrent operations. The identifiers composing the *former state* are hence tombstones. But unlike those of the tombstone approach, these tombstones are not kept directly in the sequence but in the node of the *epoch chain*. They have thus no impact on integration times of future operations. Additionally, we present a mechanism to garbage collect them in Section 6.

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 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 B.

The algorithm we present here enables nodes to rename their states identifier by identifier. A possible extension is to design RENAMEBLOCK, an improved version that renames the state block by block. RENAMEBLOCK would reduce the integration times of *rename* operations, since its time complexity would no longer depend on the number of identifiers (i.e. the number of elements) but on the number of blocks. Additionally, its execution would reduce the integration time of the next *rename* operation since the renaming mechanism merges existing blocks into one.

5 RenamableLogootSplit with concurrent *rename* operations

5.1 Concurrent *rename* operations

We now consider scenarios with concurrent *rename* operations. Figure 6 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.

As concurrent epochs are generated, epochs now form the *epoch tree*. We represent in Figure 7 the *epoch tree* that nodes obtain once they eventually synchronise. Epochs are displayed as nodes of the tree and the *parent-child* relation between them displayed as black arrows.

At the end of the scenario described in Figure 6, nodes A and B are at epochs ε_{A1} and ε_{B2} respectively. In order to converge, every node should eventually reach the same epoch. However, the function RENAMEID described in Algorithm 1 only enables nodes to move from a *parent* epoch to one of its *children* epoch. Node A (resp. B) is then unable to progress towards the epoch of node B (resp. A). It is therefore necessary to extend our renaming mechanism to break this deadlock and to make *rename* operations commutative.

First, nodes have to agree on a common epoch from the *epoch tree* as the target epoch. In order to avoid performance issues due to coordination, nodes should select this epoch

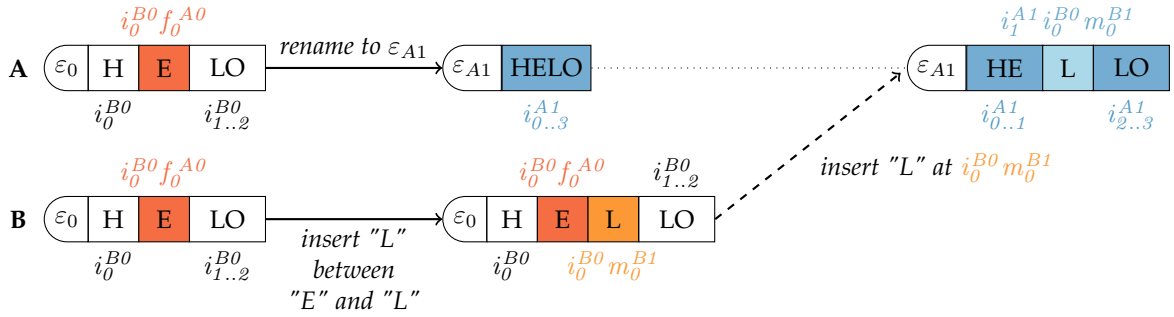


Fig. 5: Renaming concurrent update using RENAMEID before applying it to maintain intended order

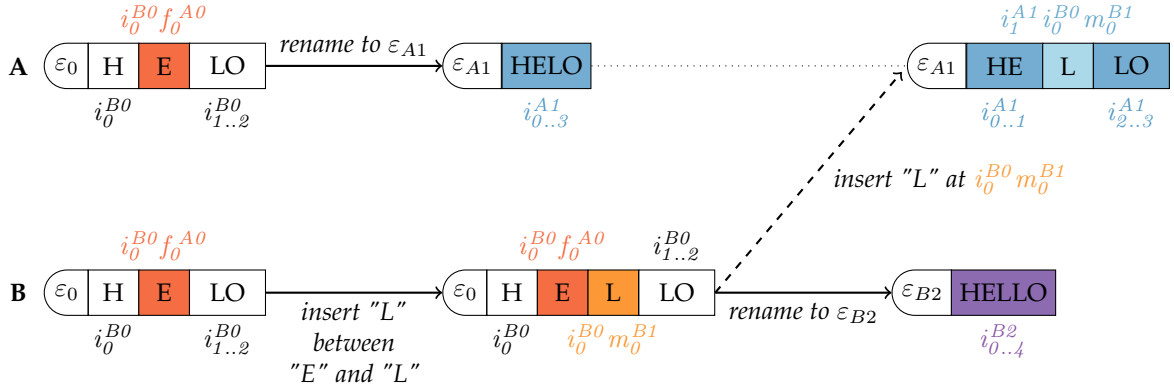


Fig. 6: Concurrent *rename* operations leading to divergent states

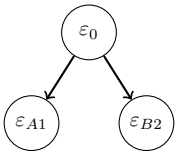


Fig. 7: The *epoch tree* corresponding to the scenario of Figure 6

in a coordination-free manner, i.e. solely using data from the *epoch tree*. We propose such a mechanism in Section 5.2.

Second, nodes have to move through the *epoch tree* to reach the target epoch. The function RENAMEID already enables nodes to move down the tree. Remaining cases to be handled are the ones in which nodes currently are on another branch of the *epoch tree*. In these cases, nodes have to be able to move up the *epoch tree* to return to the Lowest Common Ancestor (LCA) of the current epoch and the target one. This move is actually akin to reverting the effect of previously applied *rename* operations. We propose an algorithm that fulfills this purpose in Section 5.3.

5.2 Breaking tie between concurrent *rename* operations

To enable every node to select the same target epoch in a coordination-free manner, we define the *priority* relation.

Definition 1 (Priority relation). The *priority* relation is a strict total order on the set of epochs. It enables nodes to compare any pair of epochs.

Using the *priority* relation, we define the target epoch as follows:

Definition 2 (Target epoch). The epoch from the set of epochs towards which nodes should progress. Nodes select the maximum epoch according to the *priority* relation as the target epoch.

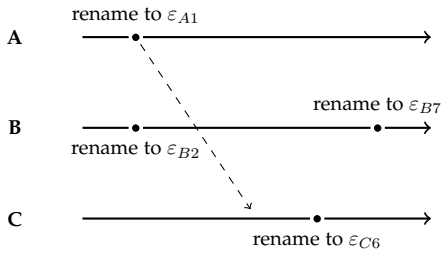
To define the *priority* relation, we can actually select different strategies. In this work, we use the lexicographical order on the path of epochs in the *epoch tree*. Figure 8 provides an example.

Figure 8a describes an execution in which the three nodes A, B and C issue several operations before eventually synchronising. As solely *rename* operations are relevant to the problem at hand, only them are represented in this figure. Initially, node A issues a *rename* operation introducing the epoch ε_{A1}. This operation is delivered to node C, which later issues its own *rename* operation creating ε_{C6}. Concurrently to these operations, node B issues two *rename* operations, generating ε_{B2} then ε_{B7}.

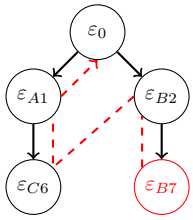
Once nodes synchronised, they obtain the *epoch tree* represented in Figure 8b. In this figure, the red dashed arrow represents the order between epochs according to the *priority* relation while the selected target epoch is displayed as a red node.

To determine the target epoch, nodes rely on the *priority* relation. According to the lexicographical order on the path of epochs in the *epoch tree*, $\varepsilon_0 < \varepsilon_0\varepsilon_{A1} < \varepsilon_0\varepsilon_{A1}\varepsilon_{C6} < \varepsilon_0\varepsilon_{B2} < \varepsilon_0\varepsilon_{B2}\varepsilon_{B7}$. Therefore every node selects ε_{B7} as the target epoch in a coordination-free manner.

Other strategies could be proposed to define the *priority* relation. For example, *priority* could rely on metrics em-



(a) Example of execution with concurrent *rename* operations



(b) Corresponding epoch tree with *priority* relation shown

Fig. 8: Selecting target epoch from execution with concurrent *rename* operations

bedded in *rename* operations representing the accumulated work on the document. It would enable favouring the branch from the *epoch tree* with the more and most active collaborators to minimise the overall amount of computations performed by nodes of the system.

5.3 Reverting *rename* operations

We now extend the scenario described in Figure 6. In Figure 9, node A receives node B's *rename* operation, which is concurrent to the *rename* operation node A previously applied. According to the proposed *priority* relation, node A determines the introduced epoch ε_{B2} as the target one. Therefore, it should revert its state to an equivalent one at ε_0, the LCA of its current epoch ε_{A1} and the target epoch ε_{B2}. Hence, we need to consider a mechanism enabling nodes to revert a previously applied *rename* operation.

This is precisely the objective of REVERTRENAMEID, which maps identifiers from the *child* epoch to equivalent ones in its *parent* epoch. The function is described in Alg 2.

The goals of REVERTRENAMEID are the following: (i) To revert identifiers generated causally before the reverted *rename* operation to their former value (ii) To revert identifiers generated concurrently to the reverted *rename* operation to their former value (iii) To assign new identifiers complying with the intended order to elements inserted causally after the reverted *rename* operation.

Case (i) is the most trivial one. To retrieve the value of id, REVERTRENAMEID simply uses the offset of $newId$[2] since it corresponds to the index of id in the *former state* (i.e. $renamedIds[offset] = id$).

Cases (ii) and (iii) are handled given the following strategies. The generic pattern for the identifier $newId$ is of the form $newPredId\ tail$. Associated with this pattern are two invariants. According to Property 3.2, we have:

$$newId \in]newPredId, newSuccId[$$

2. We name $newX$ identifiers in the resulting epoch of the *rename* operation, while X refer to their counterpart in the preceding epoch.

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

    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
                    ▷ id has been inserted causally after the rename op
        return concat(predId, MIN_TUPLE, tail)
    else if succId < tail then
                    ▷ id has been inserted causally after the rename op
        offset ← getLastOffset(succId) - 1
        predOfSuccId ← createIdFromBase(succId, offset)
        return concat(predOfSuccId, MAX_TUPLE, tail)
    else
        return tail
    end if
end function
```

Alg 2: Main functions to revert an identifier renaming

and we must have:

$$id \in]predId, succId[$$

The first subcase is when we have $tail \in]predId, succId[$. In this context, $newId$ may be the result of a concurrent *insert* operation to the *rename* one (i.e. Case (ii)), and we have $newId \in]newPredId\ predId, newPredId\ succId[$. In such case, $newId$ has been obtained using RENIDFROMPREDID and we have $id = tail$. We thus note that by returning $tail$, REVERTRENAMEID meets the two constraints, i.e. preserving the intended order and restoring to its original value the identifier.

The second subcase is when $tail < predId$. $newId$ could only been inserted causally after the *rename* operation (i.e. Case (iii)), and we have $newId \in]newPredId, newPredId\ predId[$. Since $newId$ has been inserted after the *rename* operation, there is no existing contraint on the returned identifier except the Property 3.2. To handle this case, we introduce two new tuples exclusive to the renaming mechanism : MIN_TUPLE and MAX_TUPLE. They are respectively the minimal tuple and the maximal one available to generate identifiers. Thanks to MIN_TUPLE, REVERTRENAMEID is able to return id fitting the intended order (with $id = predId\ MIN\_TUPLE\ tail$).

The last subcase is the counterpart of the previous subcase and occurs when $succId < tail$. We have $newId \in$

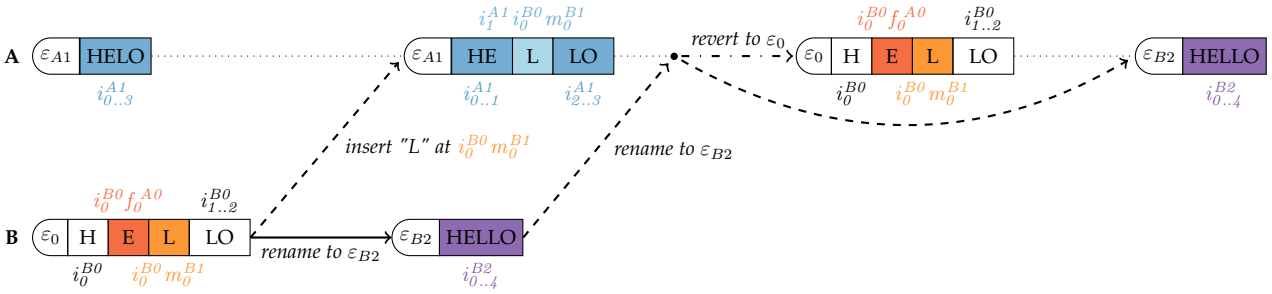


Fig. 9: Reverting a previously applied *rename* operation

$]newPredId\ succId, newSuccId[$. The strategy is the same and consists in adding a prefix to enforce the intended order. To generate this prefix, REVERTRENAMEID uses $predOfSuccId$ and MAX_TUPLE. $predOfSuccId$ is obtained by decrementing the last offset of $succId$. Hence, to preserve the intended order, REVERTRENAMEID returns id with $id = predOfSuccId\ MAX\_TUPLE\ tail$.

Once node A mapped its state to the equivalent one at ε_0 using REVERTRENAMEID, it can now apply RENAMEID to compute the corresponding state at ε_{B2}.

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 remaining cases are featured in Appendix C.

Note that RENAMEID and REVERTRENAMEID are not inverse functions. REVERTRENAMEID reverts to their original value identifiers inserted causally before or concurrently to the *rename* operation. On the other hand, RENAMEID does not do the same for identifiers inserted causally after the *rename* operation. Thus redoing a previously undone *rename* operation alters these identifiers. This modification can 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.

6 GARBAGE COLLECTION OF *former states*

Nodes store epochs and corresponding *former states* to transform identifiers through different epochs. But as the system progresses, some epochs and associated metadata become unnecessary since no more operations are issued from these epochs. Nodes can then garbage collect these epochs. In this section, we present a mechanism enabling nodes to identify obsolete epochs.

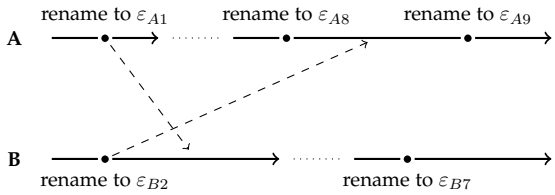
To propose such a mechanism, we rely on the notion of *causal stability of operations* [35]. An operation is causally stable once it was delivered to every node. In the context of a *rename* operation, it implies that every node progressed to the epoch introduced by this operation or to a greater one according to the *priority* relation. Using this knowledge, nodes determine the *potential current epochs*:

Definition 3 (Potential current epochs). The set of epochs on which nodes may currently be and from which they could issue operations, according to a node's own knowledge. It is a subset of the set of epochs, made of the maximum epoch introduced by a causally stable *rename* operation and of every epoch that is greater according to the *priority* relation.

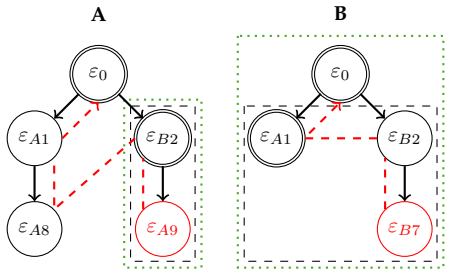
To handle next operations, nodes have to maintain the paths between every epoch of *potential current epochs*. We denote as *required epochs* the corresponding set of epochs:

Definition 4 (Required epochs). The set of epochs that a node has to maintain to handle potential next operations. It is the set of epochs that form the paths between every *potential current epochs* and their LCA.

It follows that any epoch that does not belong to the set of *required epochs* can be garbage collected by nodes. Figure 10 illustrates a use case of the proposed garbage collection mechanism.



(a) Execution



(b) States of respective epoch trees with *potential current epochs* and *required epochs* displayed

Fig. 10: Garbage collecting epochs and corresponding *former states*

In Figure 10a, we represent an execution in which two nodes A and B respectively issue several *rename* operations. In Figure 10b, we represent nodes' respective *epoch trees*. Epochs introduced by causally stable *rename* operations are displayed using double circles. The set of *potential current*

epochs is shown as a dashed black rectangle while the set of *required epochs* is represented as a dotted green rectangle.

Node A first generates a *rename* operation to ε_{A1} and later a *rename* operation to ε_{A8}. It thereafter receives from node B a *rename* operation that introduces ε_{B2}. Since ε_{B2} is greater than its current epoch ($\varepsilon_0\varepsilon_{A1}\varepsilon_{A8} < \varepsilon_0\varepsilon_{B2}$), node A selects it as its new current epoch and proceeds to rename its state accordingly. Finally, node A issues a third *rename* operation to ε_{A9}.

Concurrently, node B issues the *rename* operation to ε_{B2}. It then receives from node A the *rename* operation to ε_{A1}. However, node B keeps ε_{B2} as its current epoch (since $\varepsilon_0\varepsilon_{A1} < \varepsilon_0\varepsilon_{B2}$). Afterwards, node B issues another *rename* operation to ε_{B7}.

Upon the delivery of the *rename* operation introducing epoch ε_{B2} to node A, this operation becomes causally stable. From this point, node A knows that every node progressed to this epoch or a greater one according to the *priority* relation. Epochs ε_{B2} and ε_{A9} form the set of *potential current epochs* and nodes can now only issue operations from these epochs or one of their not yet known descendants. Node A then proceeds to compute *required epochs*. To this end, it determines the LCA of *potential current epochs* : ε_{B2}. It then generates *required epochs* by adding every epoch forming the paths between ε_{B2} and *potential current epochs*. Epochs ε_{B2} and ε_{A9} thus form the set of *required epochs*. Node A infers that epochs ε_0, ε_{A1} and ε_{A8} can safely be garbage collected.

On the other hand, the delivery of the *rename* operation introducing ε_{A1} to node B does not enable it to garbage collect any data. From its knowledge, node B computes that ε_{A1}, ε_{B2} and ε_{B7} form the *potential current epochs*. From this, nodes B derives that these epochs and their LCA ε_0 constitute *required epochs*. Every known epoch thus belong to the set of *required epochs*, preventing their garbage collection.

Eventually, once the system becomes idle, nodes reach the same epoch and the corresponding *rename* operation becomes causally stable. Nodes can then garbage collect all other epochs and associated metadata, effectively suppressing the overhead of the renaming mechanism.

Note that the garbage collection mechanism can be simplified in systems preventing concurrent *rename* operations. Since epochs form a chain in such systems, the latest epoch introduced by a causally stable *rename* operation becomes the LCA of *potential current epochs*. It follows that this epoch and its descendants constitute the *required epochs*, and that its ancestors may be garbage collected. Nodes thus only need to track causally stable *rename* operations to determine which epochs to garbage collect in systems without concurrent *rename* operations.

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 Section 8.2.

7 Evaluation

7.1 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.

By means of simulations, we generated 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 collaboratively edit the document using either LogootSplit or RenamableLogootSplit according to the session. In both cases, each bot locally performs an *insert* or a *remove* operation every 200 $\pm$ 50ms and immediately broadcasts it 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/.

7.2 Results

Using generated snapshots, we performed several benchmarks. These benchmarks evaluate RenamableLogootSplit's performance and compare it 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 node 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 11.

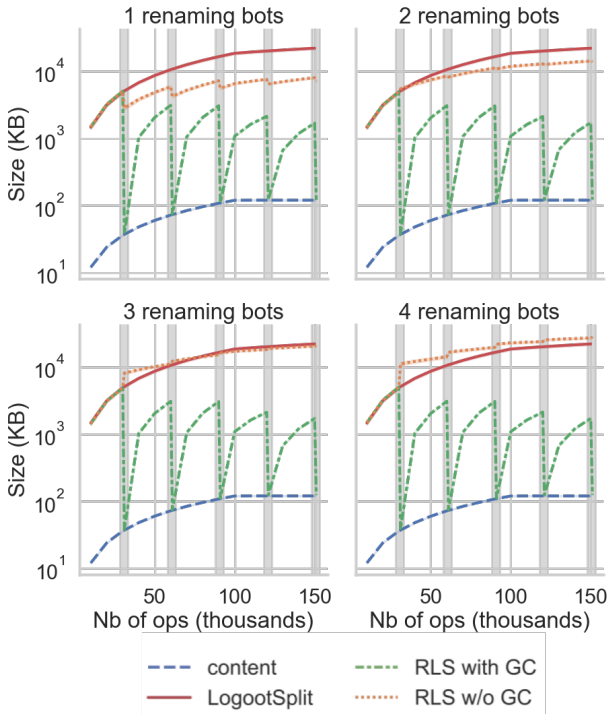


Fig. 11: Evolution of the size of the document

For each plot displayed in Figure 11, 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 periodically minimise 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 performance of RenamableLogootSplit thus decreases 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 Section 8.2, 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 12 displays the results.

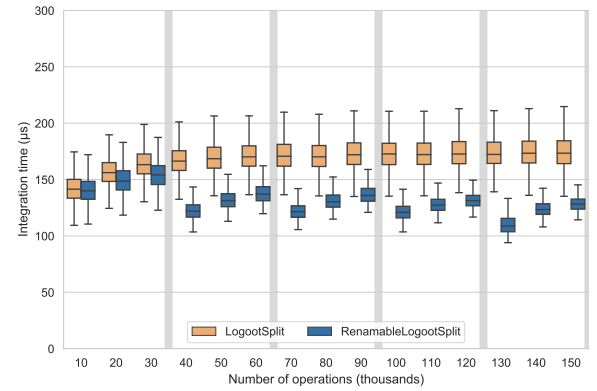
In these figures, orange boxplots correspond to integration times on LogootSplit documents while blue ones correspond to times on RenamableLogootSplit documents. While both are initially equivalent, integration times on RenamableLogootSplit documents are then reduced when compared to LogootSplit ones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

Additionally, in the case of remote operations, we measured specific integration times for RenamableLogootSplit: integration times of remote operations from previous epochs and from concurrent epochs, respectively displayed as white and red boxplots in Figure 12b.

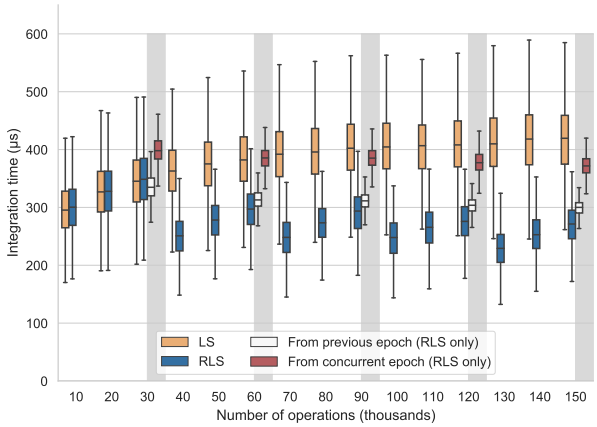
Operations from previous epochs are operations generated concurrently to the *rename* operation but applied after it. Since the operation has to be transformed beforehand using RENAMEID, we observe a computational overhead compared to other operations. But this overhead is actually compensated by the 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 REVERTRENAMEID. Because of this overhead, RenamableLogootSplit's performance for these operations is comparable to LogootSplit ones.

To summarise, transformation functions introduce an overhead to integration times of concurrent operations to *rename* ones. Despite this overhead, RenamableLogootSplit achieves better performance than LogootSplit as long as the



(a) Local updates



(b) Remote updates

Fig. 12: Integration time of standard operations

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 performance 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 Afterward, we measured the evolution of integration times of *rename* operation according to the number of operations, which relates to the number of elements in the sequence. As before, we distinguish performance of *local* and *remote* updates. The case of *remote rename* operations is subdivided into three categories. *Direct remote* denotes a remote *rename* operation that introduces a new epoch child of the current node epoch. *Concurrent introducing greater* (resp. *lesser*) *epoch* designates a remote *rename* operation that introduces a new epoch sibling of the current node epoch. According to the *priority* relation, the introduced epoch is greater (resp. lesser) than the current node epoch. The results are displayed in Table 1.

The main outcome of these measures shows that *rename* operations are generally expensive when compared to others, since nodes have to browse and rename their whole current state. Local *rename* operations take hundreds of milliseconds while *direct remote* ones and *concurrent introducing greater epoch* 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 negatively impacting user experiences. In parallel, several improvements may be made to reduce these integration times :

- Instead of using RENAMEID, which renames the state identifier by identifier, one could design and use RENAMEBLOCK. This function would rename the state block per block, achieving a better time complexity. Additionally, since *rename* operations merge existing blocks into one, this would set up a virtuous circle, where each *rename* operation reduces the cost of the next one.
- Since each call to RENAMEID or REVERTRENAMEID is independant from others, these functions are well-suited for parallel programming. Instead of renaming identifiers (or blocks) sequentially, one could slice the sequence into chunks and rename them in parallel.

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
Direct 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
Cc. int. greater epoch	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
Cc. int. lesser epoch	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

Another interesting result from this benchmark is that *concurrent introducing lesser epoch* operations are cheap to apply. Since these operations introduce epochs that are not selected as the new target epoch, nodes do not actually proceed to rename their states. The application of *concurrent introducing lesser epoch* operations hence solely consists of adding the introduced epoch and the corresponding *former state* into the epoch tree. Thus nodes can significantly reduce the overall computations of a set of concurrent *rename* operations by applying them in the most suitable order given the context.

Time to replay operation log In order to compare the performance of RenamableLogootSplit to LogootSplit on a whole, we measured the time needed by both to replay a log of operations. Obtained results are shown in Figure 13.

We observe that gains on integration times of *insert* and *remove* operations thanks to the renaming mechanism initially help to counterbalance the integration times of *rename* operations. But as the collaboration progresses, integration times of *rename* operations sightly increase since more elements are involved. This tendency is even more pronounced in scenarios with concurrent *rename* operations. Fortunately, in real-world applications, such a scenario (i.e., replaying

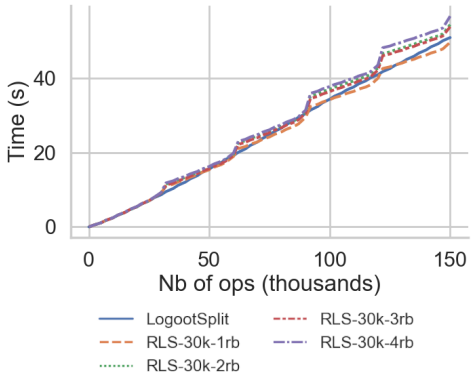


Fig. 13: Evolution of time needed to replay operation log

the entire log) will not be the usual case and could be mitigated, for example, by using snapshot mechanisms. The snapshot mechanisms could even benefit from the renaming mechanism since the transferred states will be compressed.

Impact on performance of the frequency of* rename *operations To evaluate the impact on performance of the frequency of *rename* operations, we conducted another benchmark. This benchmark consists in replaying operation logs from simulations using diverse settings of CRDT : either LogootSplit, RenamableLogootSplit with *rename* operations issued every 30k operations or RenamableLogootSplit with *rename* operations issued every 7.5k operations. As the benchmark replays the operation log, it measures the integration times of operations and their sizes. Results are displayed in Table 2.

Regarding integration times, we observe that more frequently issuing *rename* operations improves integration times of *insert* and *remove* operations. This confirms our expected results since the *rename* operation reduces integration times of these operations by reducing the size of identifiers and the number of blocks in the sequence.

We also note that the frequency has no significant impact on the performance of the *rename* operation itself. This was also an expected result since the time complexity of the implemented *rename* operation depends on the number of elements, which is not impacted by *rename* operations.

Regarding the size of operations, we observe that *insert* and *remove* operations of RenamableLogootSplit are initially bigger than their counterpart in LogootSplit since they embed their epoch of generation as additional data. But while LogootSplit's operations grow indefinitely, the frequency of *rename* operations sets an upper-bound to the size of RenamableLogootSplit's ones. It enables RenamableLogootSplit to have a smaller footprint per operation.

On the other hand, we observe that *rename* operations are much bigger than other types of operation. It is explained by the fact that this operation embeds the *former state*, i.e. the list of all blocks at its time of generation. However, we observe the same phenomenon regarding *rename* operations as for other operations : the frequency of *rename* operations sets a limit to the size of the *rename* operation itself. One can therefore choose to frequently issue *rename* operations to limit their individual size. However, this induces additional computation for each *rename* operation in the current implementation. Another solution, which we present in Section 8.3, is to implement a compression mechanism to send only the needed components to identify each block of the *former state*.

7.3 Complexity analysis of the epochs garbage collection mechanism

In previous sections, we evaluated in an experimental manner our implementation of RenamableLogootSplit and compared its performance to LogootSplit's one. However, our implementation of RenamableLogootSplit does not provide an implementation of the garbage collection mechanism of epochs. To complete our evaluation, we propose a complexity analysis of it.

The algorithm of the garbage collection mechanism consists of the following steps. First, it establishes the version vector of causally stable operations. To this end, each node has to maintain a matrix of version vectors of every node. The algorithm computes the version vector of causally stable operations by keeping the minimal value for each entry of the vectors. This step consists thus in merging n version vectors of n entries, it is thus executed in $\mathcal{O}(n^2)$.

The second step consists in browsing the epoch tree in the reverse order of the one set by the *priority* relation until it finds the maximal epoch with the corresponding *rename* operation being causally stable. For each epoch browsed, the garbage collection mechanism computes and stores its path to the root of the epoch tree. This step is thus executed in $\mathcal{O}(e \cdot h)$, with e being the number of epochs in the tree and h the height of the tree.

From these paths, the mechanism computes the LCA. To do so, the algorithm computes in a successive manner the last intersection between the path from the root to the current LCA and the paths computed previously. The LCA is the last epoch composing the resulting path. This step is executed in $\mathcal{O}(e \cdot h)$ too.

The algorithm is now able to compute the set of *required epochs*. To this end, it browses the paths computed during the second step. For each path, it adds epochs that are after the LCA to the set of *required epochs*. Again, this step is executed in $\mathcal{O}(e \cdot h)$.

Having determined the *required epochs*, the mechanism can now remove obsolete epochs. It browses the epoch tree and deletes every epoch that is not contained in the set. This final step is executed in $\mathcal{O}(e)$.

Thus, we obtain that the time complexity of the garbage collection mechanism of epochs is $\mathcal{O}(n^2 + 3(e \cdot h) + e)$. We summarise this result in Table 3.

Despite its time complexity, the garbage collection mechanism should have a limited impact on the performance of the application. Indeed, this mechanism does not belong to the critical path of the application, i.e. the integration of updates. It may be triggered rarely, in the background and we may even target specific windows to do so, e.g. during idle times. As such, we thus did not delve on this part of RenamableLogootSplit in this work.

8 Discussion

8.1 Issuing *rename* operations

As stated in Section 3.2, *rename* operations are system operations. It is thus up to the designers of the system to

Parameters		Integration time (μs)					Size (B)				
Type	CRDT	Mean	Median	IQR	1^{rst} Quant.	99th Quant.	Mean	Median	IQR	1^{rst} Quant.	99th Quant.
insert	LS	471	460	130	224	768	593	584	184	216	1136
	RLS - 30k	397	323	67	171	587	442	378	92	314	958
	RLS - 7.5k	393	265	55	133	381	389	378	0	314	590
remove	LS	280	270	71	140	435	632	618	184	250	1170
	RLS - 30k	247	181	39	98	308	434	412	0	320	900
	RLS - 7.5k	296	151	35	75	214	401	412	0	320	596
Parameters		Integration time (ms)					Size (KB)				
Type	CRDT	Mean	Median	IQR	1^{rst} Quant.	99th Quant.	Mean	Median	IQR	1^{rst} Quant.	99th Quant.
rename	RLS - 30k	1022	1188	425	540	1276	1366	1258	514	635	3373
	RLS - 7.5k	861	974	669	123	1445	273	302	132	159	542

TABLE 2: Integration times and sizes of operations per type and frequency of *rename* operations

TABLE 3: Time complexity of the garbage collection mechanism of epochs

Step	Time
computing stable version vector	n^2
computing paths from root to potential current epochs	$e \cdot h$
identifying LCA	$e \cdot h$
computing required epochs	$e \cdot h$
deleting obsolete epochs	e
total	$n^2 + 3(e \cdot h) + e$

n: number of nodes of the system, e: number of epochs in the *epoch tree*, h: height of the *epoch tree*

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 11, 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 times 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 negatively impact the user experience. Indeed, nodes cannot 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 frequently issue *rename* operations to keep their integration times below the perceptible limit.

Finally, the last aspect to consider is the amount of concurrent *rename* operations. Figure 11 shows that concurrent *rename* operations decrease RenamableLogootSplit's performance. The proposed strategy must then aim to minimise the number of concurrent *rename* operations issued. However, it should avoid relying 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.

To recap, we can propose strategies aiming to individually minimise each of these parameters. And while some of these goals converge (to minimise the size of the data structure and to minimise the integration times of *rename* operations), others conflict (to issue a *rename* operation as soon as a threshold is reached vs. to minimise the amount of concurrent *rename* operations). System designers have thus to propose a trade-off between parameters according to the constraints of the system (real-time application or not, nodes' hardware limitations...). It is hence necessary to deploy the system to evaluate its performance on each aspect, its usage and to find the right trade-off between all the parameters of the renaming strategy. For instance, in the context of real-time collaborative editing systems, [39] have shown that delay decreases the quality of the collaboration. In such systems, we would thus aim to keep the integration times of operations (including *rename* ones) below the limit corresponding to their perception by users.

8.2 Offloading on disk unused *former states*

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 recently have 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 preemptively retrieve them 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*...

8.3 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.

9 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.

9.1 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 revertRenameId. It also simplifies the implementation of the garbage collection mechanism of epochs and *former states*.

9.2 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 periodically reducing 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 strategies fitted to different editing pattern. Nodes randomly pick 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 periodically reset the sequence's metadata using *rename* operations while LSEQ's identifier allocation strategies would reduce their growth in-between. This would also enable reducing the frequency of *rename* operations, decreasing the system computations overall.

10 Conclusions and future work

Conflict-free Replicated Data Types (CRDTs) enable the design of highly available large-scale distributed systems while ensuring the Strong Eventual Consistency (SEC) model. Still, some of these data structures, notably Sequence CRDTs, suffer from a continuous growth of their metadata.

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 automatically triggered by nodes when deemed necessary. Since the proposed *rename* operation is not commutative with *insert* and *remove* operations,

we use Operational Transformation (OT) techniques to solve resulting conflicts.

Experiments show that the renaming mechanism enables nodes to reduce the size of their data structure by several hundred times compared to previous work. RenamableLogootSplit achieves this result even in the case where several concurrent *rename* operations are issued, since the introduced overhead is but temporary and is eventually removed.

In future work, we plan to propose other strategies than the lexicographical one to define *priority*, the strict total order relation on the set of epochs. These new strategies could for example favour the branch of the epoch tree with the most nodes and work. The use of these strategies would then reduce the overall computations of the system.

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, Feb 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6127847/

[2] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys*, 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.

[4] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near real-time peer-to-peer shared editing on extensible data types," in *Proceedings of the 19th International Conference on Supporting Group Work*, ser. GROUP '16. New York, NY, USA: ACM, Nov. 2016, p. 39–49. [Online]. Available: https://doi.org/10.1145/2957276.2957310

[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://ieeexplore.ieee.org/document/7909007/

[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: ACM, Oct. 2019, p. 154–178. [Online]. Available: https://dl.acm.org/doi/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: ACM, 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 *Proceedings of the 11th ACM Symposium on Document Engineering*, ser. DocEng '11. New York, NY, USA: ACM Press, 2011, p. 103–112. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2034691.2034717

[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: ACM Press, 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," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 10, pp. 1454 – 1470, Oct. 2009. [Online]. Available: https://doi.org/10.1109/TPDS.2008.240

[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," in *Web Information Systems Engineering – WISE 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 503–512. [Online]. Available: https://doi.org/10.1007/978-3-540-76993-4_42

[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, Mar. 2011. [Online]. Available: https://doi.org/10.1016/j.jpdc.2010.12.006

[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*. IEEE, Jun. 2009, pp. 395–403. [Online]. Available: https://doi.org/10.1109/ICDCS.2009.20

[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. Mostefaoui, 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 '13. New York, NY, USA: ACM, Sep. 2013, p. 37–46. [Online]. Available: https://doi.org/10.1145/2494266.2494278

[31] B. Nédelec, P. Molli, and A. Mostéfaoui, "A scalable sequence encoding for collaborative editing," *Concurrency and Computation: Practice and Experience*, p. e4108, 2017. [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. Software Eng.*, vol. SE-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*, ser. Lecture Notes in Computer Science, K. Magoutis and P. Pietzuch, Eds., vol. 8460. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 126–140. [Online]. Available: http://link.springer.com/10.1007/978-3-662-43352-2_11

[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. [Online]. Available: https://doi.org/10.1109/DSN.2002.1028914

[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, Jun. 2018, pp. 22–25. [Online]. Available: https://ieeexplore.ieee.org/document/8416202/

[38] C. Baquero, P. S. Almeida, and A. Shoker, "Pure operation-based replicated data types," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1710.04469

[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," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, p. 172–182, Dec. 1995. [Online]. Available: https://dl.acm.org/doi/10.1145/224057.224070

Matthieu Nicolas received a diplôme d'ingénieur from TELECOM Nancy in 2014. He worked several years as a R&D software engineer in the Inria Coast project-team. He is now a PhD student at the University of Lorraine. His research interests include Conflict-free Replicated Data Types (CRDTs) and distributed collaborative systems.

Gérald Oster is an Associate Professor at LORIA, Inria Nancy - Grand Est, University of Lorraine. His research work explores distributed collaborative systems with a focus on optimistic content replication mechanisms and their applicability. He is one of the pioneers of the CRDT approach as he participated in the design of the WOOT algorithm that initiated researches on these distinctive data structures. He is currently investigating the applicability in diverse domains of these novel replicated data structures.

Olivier Perrin is full professor at University of Lorraine, and works at LORIA in the Coast project. His research interests always been related to data and service management, and include various topics about collaborative systems, services composition, and distributed systems. He has published several articles in international journals and conferences, and he was involved in many international and European projects. His current work deals with trust and privacy in distributed collaborative applications.

APPENDIX A
HELPER FUNCTIONS

- POSITION : Retrieves the value of the position of the first tuple of the given id. For example, POSITION($i_1^{A1}\, i_0^{B0}\, m_0^{B1}$) returns i.
- FINDINDEX : Retrieves the index of the given id in the given list of ids renamedIds.
- FINDINDEXOFPRED : Retrieves the index of the predecessor of the given id in the given list of ids renamedIds.
- CONCAT : Returns a new id made of the tuples of given ids. For example, CONCAT(i_1^{A1}, $i_0^{B0}\, m_0^{B1}$) returns $i_1^{A1}\, i_0^{B0}\, m_0^{B1}$.
- ISRENAMEDID : Indicates if the given id is part of the block resulting of the *rename* operation, i.e. id is of the form $pos_{offset}^{nId\ nSeq}$ with *pos* corresponding to the position of the first renamed id, *nId* to the node id of the author of the *rename* operation and *nSeq* to its sequence number at the time.
- GETFIRSTOFFSET : Retrieves the value of the offset of the first tuple of the given id. For example, GETFIRSTOFFSET($i_1^{A1}\, i_0^{B0}\, m_0^{B1}$) returns 1.
- GETTAIL : Returns a new id obtained by removing the n first tuples of the given id. For example, GETTAIL($i_1^{A1}\, i_0^{B0}\, m_0^{B1}$, 1) returns $i_0^{B0}\, m_0^{B1}$.
- GETLASTOFFSET : Retrieves the value of the offset of the last tuple of the given id. For example, GETLASTOFFSET($i_1^{A1}\, i_0^{B0}\, m_0^{B1}$) returns 0.
- CREATEIDFROMBASE : Returns a new id by copying the given id and replacing its last offset with the given one. For example, CREATEIDFROMBASE($i_1^{A1}\, i_0^{B0}\, m_0^{B1}$, 5) returns $i_1^{A1}\, i_0^{B0}\, m_5^{B1}$.
- ISPREFIX : Indicates if the given id is a prefix of the other given id. For example, ISPREFIX($i_1^{A1}\, i_0^{B0}$, $i_1^{A1}\, i_0^{B0}\, m_0^{B1}$) returns true.

APPENDIX B
REMAINING FUNCTIONS OF RENAMEID

In Algorithm 3, we present the remaining functions composing RENAMEID. RENIDLESSTHANFIRSTID (resp. RENIDGREATERTHANLASTID) enables to rename identifiers with id such as $id < firstId$ (resp. $lastId < id$).

```
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 functions to rename an identifier

The main insight of these functions is that it is not necessary to modify identifiers out of the range of renamed identifiers and of resulting identifiers ($[min(firstId, predNewFirstId), max(lastId, succNewLastId)]$)

In other cases, like RENIDFROMPREDID, these functions prepend a suitable prefix to the given id to preserve the intended order (the predecessor of $newFirstId$ for RENIDLESSTHANFIRSTID, $newLastId$ for RENIDGREATERTHANLASTID).

APPENDIX C
REMAINING FUNCTIONS OF REVERTRENAMEID

Algorithm 4 features the remaining functions composing REVERTRENAMEID. REVRENIDLESSTHANNEWFIRSTID (resp. REVRENIDGREATERTHANNEWLASTID) enables to revert the effects of a previously applied *rename* operation on identifiers with id such as $id < newFirstId$ (resp. $newLastId < id$).

```
function REVRENIDLESSTHANNEWFIRSTID(id, firstId, newFirstId)
    predNewFirstId ← createIdFromBase(newFirstId, -1)
    if isPrefix(predNewFirstId, id) then
        tail ← getTail(id, 1)
        if tail < firstId then
            return tail
        else
                    ▷ id has been inserted causally after the rename op
            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
                    ▷ id has been inserted causally after the rename op
        return concat(lastId, MIN_TUPLE, id)
    else if isPrefix(newLastId, id) then
        tail ← getTail(id, 1)
        if tail < lastId then
                    ▷ id has been inserted causally after the rename op
            return concat(lastId, MIN_TUPLE, tail)
        else if tail < newLastId then
            return tail
        else
                    ▷ id has been inserted causally after the rename op
            return id
        end if
    else
        return id
    end if
end function
```

Alg 4: Remaining functions to revert an identifier renaming

Like RENIDLESSTHANFIRSTID and RENIDGREATERTHANFIRSTID, these functions does not modify identifiers out of the range of renamed identifiers and of resulting identifiers.

To process other cases, the functions study the pattern of the given identifier. If REVRENIDLESSTHANNEWFIRSTID (resp. REVRENIDGREATERTHANNEWLASTID) infers from the value of the identifier that it may come from a concurrent *insert* operation to the *rename*

one, REVRENIDLESSTHANNEWFIRSTID (resp. REVRENIDGREATERTHANNEWLASTID) simply removes the prefix added by RENIDLESSTHANFIRSTID (resp. RENIDGREATERTHANLASTID).

If the value of an identifier instead indicates that it was inserted causally after the *rename* operation, REVRENIDLESSTHANNEWFIRSTID (resp. REVRENIDGREATERTHANNEWLASTID) generates a corresponding identifier at the parent epoch preserving the intended order using $predFirstId$ and MAX_TUPLE (resp. $lastId$ and MIN_TUPLE).