Efficient Renaming in Conflict-free Replicated Data Types (CRDTs)

Matthieu Nicolas
matthieu.nicolas@loria.fr
Université de Lorraine, CNRS, Inria, LORIA, F-54500
Nancy, France

Gérald Oster
gerald.oster@loria.fr
Université de Lorraine, CNRS, Inria, LORIA, F-54500
Nancy, France

Olivier Perrin
olivier.perrin@loria.fr
Université de Lorraine, CNRS, Inria, LORIA, F-54500
Nancy, France

Abstract

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

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

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

Keywords CRDT, real-time collaborative editing, eventual consistency, memory-wise optimisation

ACM Reference format:
Matthieu Nicolas, Gérald Oster, and Olivier Perrin. 2020. Efficient Renaming in CRDTs. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

- Real-time collaborative text editing
- Operational Transform (OT)
- CRDT

2 Background

To solve conflicts deterministically and ensure the convergence of all nodes, CRDTs relies on additional metadata. In the context of Sequence CRDTs, two different approaches were proposed, each trying to minimise the overhead introduced. The first one affixes constant-sized identifiers to each value in the sequence and uses them to represent the sequence as a linked list. The downside of this approach is an evergrowing overhead, as it needs to keep removed values to deal with potential concurrent updates, effectively turning them into tombstones. The second one avoids the need of tombstones by instead attaching densely-ordered identifiers to values. It is then able to order values into the sequence by comparing their respective identifiers. However this approach also suffers from an ever-increasing overhead, as the size of such densely-ordered identifiers is variable and grows over time.

In the context of this paper, we focus on the later approach.

2.1 LogootSplit

- Attach dense identifiers to elements
- Group elements with adjacent identifiers into blocks to reduce overhead

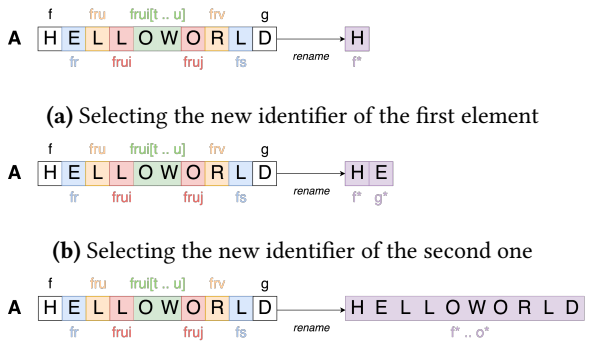
2.2 Limits

- Growth of identifiers
- Increasing number of blocks

3 Proposed approach

We propose a new Sequence CRDT relying on dense identifiers to order elements : *RenamableLogootSplit*.

To address the limitations of LogootSplit, we embed in this data structure a renaming mechanism. The purpose of this mechanism is to reassign shorter identifiers to elements to reduce the metadata of the whole sequence.

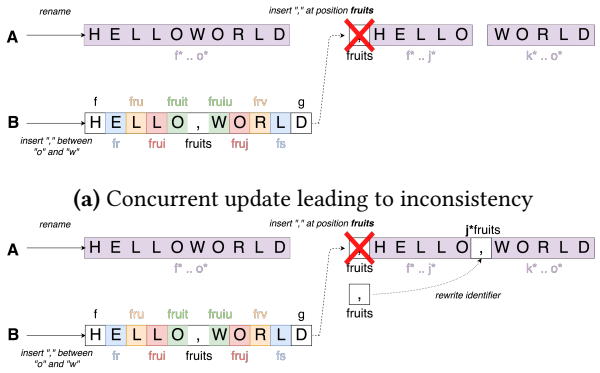


(a) Selecting the new identifier of the first element

(b) Selecting the new identifier of the second one

(c) Final state obtained

Figure 1. Renaming the sequence



(a) Concurrent update leading to inconsistency

(b) Fixed scenario

Figure 2. Dealing with concurrent updates

3.1 System Model

The system is composed of a dynamic set of nodes, as nodes join and leave dynamically the collaboration during its lifetime. Nodes collaborate to build and maintain a sequence using RenamableLogootSplit. Each node owns a copy of the sequence and edit it without any kind of coordination with others.

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 the failures of the network, nodes rely on a message-passing layer. As RenamableLogootSplit is built on top of LogootSplit, it shares the same requirements for the operation delivery. This layer is thus used to deliver messages to the application exactly-once. The layer also ensures that *remove* operations are delivered after corresponding *insert* operations. Nodes use an anti-entropy mechanism to synchronise in a pairwise manner, by detecting and re-exchanging lost operations.

3.2 *rename* operation

RenamableLogootSplit enables nodes to reduce the overhead of their replica by the means of a new operation : the *rename* operation. This operation reassigns arbitrary identifiers to elements.

Its behavior is illustrated in Figure 1 and can be described as follow : 1. It reuses the id of the first element of the sequence, but modified with the node's own id and sequence number (Figure 1a) 2. It generates contiguous identifiers for all following elements (Figure 1b). As we assign contiguous identifiers to all elements of the sequence, we can eventually group them into one block as illustrated in Figure 1c. It allows nodes to benefit the most from the block feature and to minimise the overhead of the resulting state.

In order for the system to eventually converge, other nodes have to rename their state identically. To achieve this, the node issuing the *rename* operation broadcasts its former state to others. Using the former state, others compute the new identifier of each renamed identifier. However, other nodes' states may contain concurrently inserted identifiers. We will explain in subsection 3.3 how to rename them deterministically.

Broadcasting the *rename* operation embedding the former state may be quite bandwidth consuming since the size of identifiers and the number of blocks are not bounded. To partially adress this issue, we propose a compression mechanism which sends only the necessary components to identify uniquely a block instead of whole identifiers. This compression mechanism allows to reduce to a fixed amount per block the metadata to broadcast.

As stated previously, nodes can issue operations without any coordination. However, the designed *rename* operation is not intrinsically commutative. Additional steps are thus needed to handle scenarios with concurrent *rename* operations. For the sake of simplicity and brevity, we focus in this paper only on scenarios without any concurrent ones. We leave the presentation and evaluation of mentioned additional steps to a future work.

3.3 Dealing with concurrent updates

As *rename* operations can be issued without any kind of coordination, it is possible for other nodes to perform updates concurrently. Since identifiers are modified by the *renaming* mechanism, applying concurrent updates as they are would result in inconsistencies as illustrated in Figure 2a. It is thus necessary to handle concurrent operations to *rename* operations in a particular manner.

To detect them, we use an *epoch-based* system. We add an *epoch* to the sequence as a property. Each time a *rename* operation is applied, the sequence progresses to a new epoch. When nodes issue operation, they tag them with their current epoch. Upon the reception of an operation, nodes compare their current epoch to the operation's one. If they differ, nodes have to transform the operation before applying it.

To transform an operation, nodes use the algorithm described in Algorithm 1. This algorithm enables nodes to transform identifiers against the *rename* operation. The main idea of this algorithm is to use the predecessor of the given identifier to do so. The algorithm consists mainly in 1. finding the predecessor of the given id in the former state 2. computing its counterpart in the renamed state 3. prepending it to the given id to generate the renamed id. An example of its usage is illustrated in Figure 2b.

Nodes applying remote *rename* operations use the same algorithm to rename identifiers from their state which do not appear in the propagated state, i.e. identifiers which has been inserted concurrently to the renaming.

Since nodes rely on the former state to transform concurrent operations to a *rename* operation to preserve their semantics, nodes has to store it. Nodes need it until each of them can not longer issue concurrent operations to the corresponding *rename* operation. In other words, nodes can safely garbage collect the former state once the *rename* operation became causally stable. Meanwhile, nodes can offload it onto the disk as it is only required to handle concurrent operations.

Algorithm 1 Rename position

```
function renamePos(pos : P, newId : 𝕀, newSeq : ℕ, renamedBlocks : {b_i ∈ B}_{i∈ℕ}): P
    firstPos ← posBegin(renamedBlocks[0])
    lastPos ← posEnd(renamedBlocks[renamedBlocks.length−1]

    newPriority ← priority(firstPos)
    newFirstPos ← new P(newPriority, newId, newSeq, 0)
    newLastPos ← new P(newPriority, newId, newSeq, length)

    if firstPos < pos and pos < lastPos then
        predecessor ← findPredecessor(pos, renamedBlocks)
        indexOfPredecessor ← findIndex(predecessor, renamedBlocks)
        newPredecessor ← new P(newPriority, newId, newSeq, indexOfPredecessor)
        return concat(newPredecessor, pos)
    else if lastPos < pos and pos < newLastPos then
        return concat(newLastPos, pos)
    else if newFirstPos < pos and pos < firstPos then
        predecessorOfNewFirstPos ← new P(newPriority, newId, newSeq, −1)
        return concat(predecessorOfNewFirstPos, pos)
    else
        return pos ▷ Return the position unchanged as it does not interfere with the renaming
    end if
end function
```

4 Evaluation

4.1 Simulations and benchmarks

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

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

We ran these simulations with the following experimental settings : we deployed 10 bots as separate Docker containers on a single workstation. Each container corresponds to a single mono-threaded Node.js process (version 13.1.0) simulating an author. The bots share and edit collaboratively the document using either LogootSplit or RenamableLogootSplit according to the session. In both cases, each bot performs an *insert* or a *remove* operation locally every 200 ± 50ms. During the first phase, the probabilities for each operation of being an *insert* or a *remove* are respectively of 80% and 20%. Once the document reaches 60k characters (around 15 pages), the probabilities are both set to 50%. The generated operation is then broadcast to others using a P2P full mesh network. After issuing an operation, there are 5% of chances that the bot moves its cursor to another position in the document. Each bot performs 15k operations. Snapshots are taken every 10k operations overall. Additionally, in the case of RenamableLogootSplit, one bot is arbitrarily designated as the master. It performs *rename* operations every 30k operations overall.

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

Meanwhile, our implementation of LogootSplit and RenamableLogootSplit are available at https://github.com/coast-team/mute-structs. Both implementations use an AVL

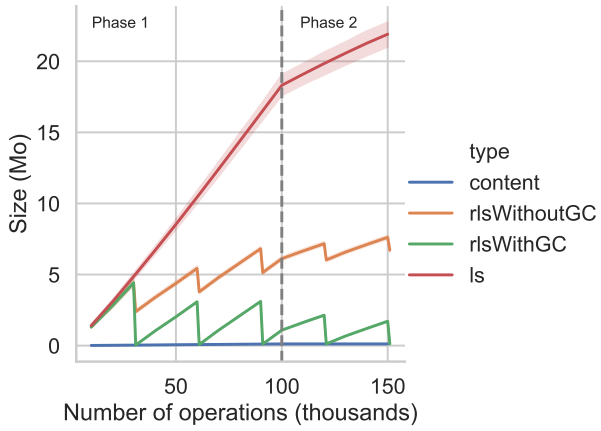


Figure 3. Evolution of the size of the document

Tree, a self-balancing binary search tree, to represent the sequence. This data structure enables us to achieve *insert* and *remove* operations in logarithmic time.

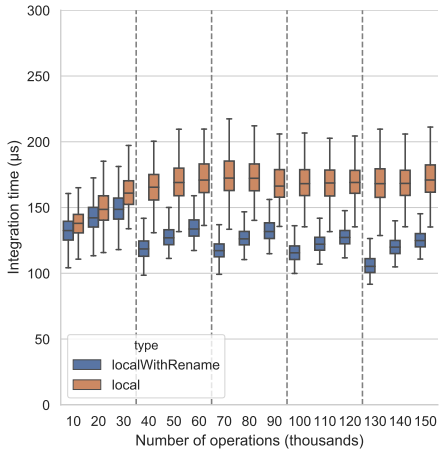
4.2 Results

Memory overhead Using the snapshots generated, we compare the evolution of the size of the data structure in collaborative editing session. The results are displayed in Figure 3. On this plot, the blue line corresponds to the size of the content while the red one exhibits the growth of the LogootSplit data structure.

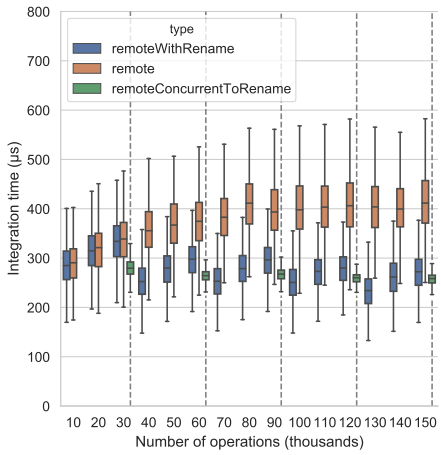
The green line illustrates the growth of the RenamableLogootSplit document in its best-case scenario. In this scenario, *rename* operations become stable as soon as they are issued. Former states can then be garbage collected safely, maximising the benefits of the *renaming* mechanism. In this case, we observe that *rename* operations reset the overhead of the data structure and eventually reduce by hundred times the document size compared to LogootSplit equivalent one.

As for the orange line, it represents RenamableLogootSplit worst-case scenario. Here, we assume that *rename* operations never become stable and that nodes has to store former states forever. However, obtained results show that RenamableLogootSplit outperforms LogootSplit and reduce by 66% the size of the data structure, even in this case. This outcome is explained by the fact that the AVL does not only store the content and blocks corresponding to the sequence. Some metadata is actually added to the state to browse the sequence more efficiently when performing updates. When a *rename* operation is applied, nodes only keep the sequence of blocks from the former state as an array to be able to transform concurrent operations. Other metadata is scrapped, which results in this memory gain.

Integration times of standard operations We set up benchmarks to measure the impact of the renaming mechanism on



(a) Local operations



(b) Remote operations

Figure 4. Evolution of the integration time of standard operations

the integration times of *insert* and *remove* operations. The obtained results are presented in Figure 4.

Figure 4a displays the integration times of local operations while Figure 4b exhibits remote ones. In both cases, the orange boxplots correspond to LogootSplit's integration times while blue ones to RenamableLogootSplit's ones. The results show that the *renaming* mechanism allows to reduce the integration times of future operations.

In Figure 4b, the green boxplots display the integration times of concurrent operations to a *rename* one. As illustrated in subsection 3.3, these operations require to be transformed before being applied to the renamed state. The results presented here show that this is actually faster than applying them directly on the former state.

Integration time of* rename *operation Finally, we measured the integration time of the *rename* operation according to the size of the document. Results are displayed in Figure 5.

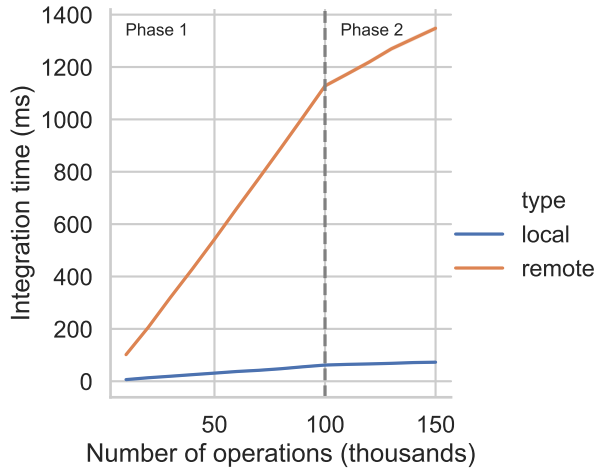


Figure 5. Evolution of the integration time of rename operations

In this figure, the blue line corresponds to the integration time of a *local* rename operation while the orange one corresponds to the integration time of a remote one.

The main result of this benchmark is that the unit of time used when applying *rename* operations is in hundreds of milliseconds. However other operations can not be integrated during the processing of *rename* operations : remote operations won't be displayed to the user while local ones won't be propagated to others. *Rename* operations can thus be perceived as spikes of latency by users and degrade their experience if they are too long to process. It is necessary to take this concern into account when designing the strategy used to trigger *rename* operations to avoid such cases.

5 Related works

- Core and Nebula
- LSEQ
- Specification and Complexity of Collaborative Text Editing
- OT

6 Conclusions and future work

- Designed new dense identifier-based Sequence CRDT embedding a renaming mechanism : *RenamableLogoot-Split*
- Achieved better performances memory-wise than state of the art, even in worst-case scenario...
- ... at the cost of expensive but infrequent *rename* operations
- Study user behavior to set a limit to integration time of *rename* operations
- Present and validate mechanism to handle concurrent *rename* operations
- Design strategies to limit likelyhood of concurrent *rename* operations
- Design strategies to limit overall workload in case of concurrent *rename* operations