Efficient Renaming in Sequence 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. 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, unlike traditional data types, they are designed to natively support 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 belonging to a dense order. These constraints may hinder the identifiers' size from being bounded. As the system progresses, identifiers tend to grow. This inflation deepens the overhead of the CRDT over time, leading to performance issues.

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. Experimental results demonstrate that this mechanism decreases the overhead of the replicated data structure and eventually limits it.

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

1 Introduction

In order to serve an ever-growing number of users and provide an increasing volume of data, large-scale systems such as data stores or collaborative editing tools adopt the *eventual consistency* model [1]. This model 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 model, every node owns a copy of the data, can modify it and propagate the updates to others. Therefore, replicas can temporarily diverge. To ensure that nodes converge eventually despite concurrently generated updates, a conflict resolution mechanism is required.

Several approaches were introduced to design efficient conflict resolution mechanisms. The one we consider proposes to use Conflict-free Replicated Data Types (CRDTs) [2]. CRDTs are new specifications of abstract data types, e.g. the Set or the Sequence. However, when compared to former ones, they are designed to support natively concurrent modifications. To this end, they embed a conflict-resolution mechanism directly in their specification.

CRDTs appear as a keystone technology of a new paradigm of applications: Local-First Software [3]. They also have been proven a suitable approach to build distributed real-time collaborative editors [4]. Still, they exhibits some limitations. Especially in the context of real-time collaborative editing, the internal conflict resolution mechanism accumulates a large amount of metadata over time.

To address this particular issue, we propose a new CRDT for Sequence which embeds a renaming mechanism. It enables to minimise the overhead of the data structure by discarding accumulated metadata eventually.

This paper is organised as follows. Section 2 provides the background of our approach. In section 3, we introduce *RenamableLogootSplit*, our new CRDT for Sequence. Section 4 presents the benchmarks we performed to evaluate our proposition and the obtained results. In section 5, we compare our approach to existing ones. Finally, section 6 concludes and introduces possible 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 [5–7] 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 [8–10] avoids the need of tombstones by attaching identifiers from a dense totally ordered set to elements. Elements are ordered into the sequence by comparing their respective identifiers. However this approach suffers from an ever-increasing overhead, as the size of such dense totally ordered identifiers is variable and grows over time. In the context of this paper, we focus on the later approach.

2.1 LogootSplit

Proposed by André et al. [10], LogootSplit (LS) is the state of the art of the variable-size identifiers approach of Sequence

CRDT. As explained previously, it uses identifiers from a dense totally ordered set to position elements into the replicated sequence.

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

Instead of storing an identifier for each element of the sequence, the main insight of LogootSplit is to aggregate dynamically elements into blocks. Grouping elements into blocks enables LogootSplit to assign logically an identifier to each element 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. Figure 1 illustrates such a case: in Figure 1a, the element identifiers form a chain of contiguous identifiers. LogootSplit is then able to group them into one block to minimise the metadata stored, as shown in Figure 1b.

(a) Elements with their corresponding identifiers

(b) Elements grouped into a block

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

This feature allows to shift the root of metadata growth from the number of elements to the number of blocks. As blocks can contain an arbitrary number of elements, it enables to reduce significantly the memory overhead of the data structure.

2.2 Limits

As stated previously, the size of identifiers from a dense totally ordered set is variable. When nodes insert new elements between two others with the same *position* value, LogootSplit has no other option than to increase the size of the resulting identifiers. Figure 2 illustrates such cases. In this example, since the node inserts a new element between contiguous identifiers, LogootSplit is not able to generate a fitting 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.

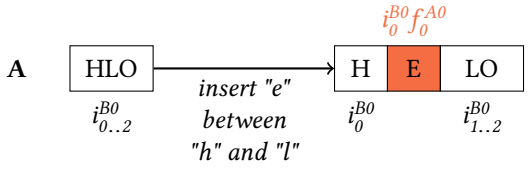


Figure 2. Insertion leading to longer identifiers

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

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

In our benchmarks, we measure that the content eventually represents less than 1% of the data structure size, the remaining 99% being metadata. It is thus necessary to address the previously highlighted issues.

3 Proposed approach

We propose a new Sequence CRDT belonging to the variable-size identifiers approach: *RenamableLogootSplit* (RLS) [11].

To address the limitations of LogootSplit, we embed in the data structure a renaming mechanism. The purpose of this mechanism is to reassign shorter identifiers to elements and to group them into blocks to minimise the memory overhead of the whole sequence.

To avoid costly and blocking consensus algorithms, we instead adopt the *optimistic replication* [12] approach for our mechanism. Nodes perform renaming without any coordination. However, this operation is not intrinsically commutative with others. If conflicts arise, we use *operational transformations* [13, 14] to enable nodes to resolve them deterministically.

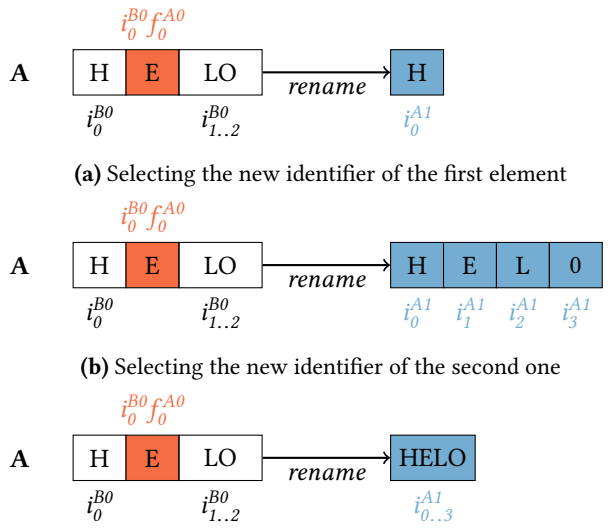
3.1 System Model

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 coordination.

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

Our *rename* operation helps RenamableLogootSplit to reduce the overhead of nodes replica. This operation reassigns arbitrary identifiers to elements.



(a) Selecting the new identifier of the first element

(b) Selecting the new identifier of the second one

(c) Final state obtained

Figure 3. Renaming the sequence on node A

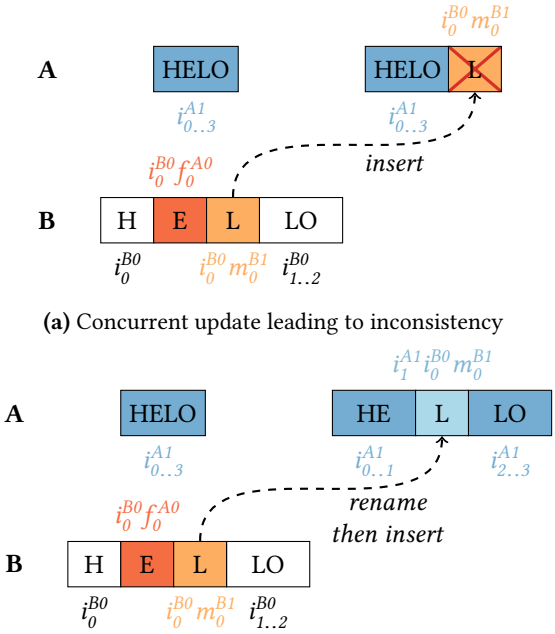
Its behaviour is illustrated in Figure 3. In this example, node A initiates a *rename* operation on its local state. First, node A reuses the id of the first element of the sequence (i_0^{B0}) but modifies it with its own node id (**A**) and current sequence number (*1*). Also the offset is set to 0. Node A reassigns the resulting id (i_0^{A1}) to the first element of the sequence as described in Figure 3a. Then, node A derivates 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.

To eventually converge, other nodes have to rename their state identically. However, they can not simply replace their current state with the new renamed one. Indeed, they may have performed concurrent updates on their states. To not discard these updates, nodes have to process the *rename* operation themselves. To this end, the node issuing the *rename* operation broadcasts its former state to others. Using the former state, other nodes compute the new identifier of each renamed identifier. As for concurrently inserted identifiers, we will explain in subsection 3.3 how nodes rename them in a deterministic way.

To limit bandwidth consumption of *rename* operations, we propose a compression technique to broadcast only necessary components to uniquely identify blocks instead of whole identifiers. It reduces the data to send to a fixed amount per block. 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.

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 naively concurrent updates would result in inconsistencies as illustrated in Figure 4a. It is thus necessary to handle concurrent operations to *rename* operations in a particular manner.



(a) Concurrent update leading to inconsistency

(b) Fixed scenario

Figure 4. Dealing with concurrent updates

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 operations, they tag them with their current epoch. 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.

Algorithm 1 enables nodes to transform *insert* or *remove* operations against the *rename* one. 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 4b.

Algorithm 1 Rename concurrently generated identifier

```
function RENAMEID(id, renamedBlocks, nId, nSeq)
▷ id is the identifier to rename
▷ renamedBlocks is the former state
▷ nId is node id of the node which issued the rename op
▷ nSeq is node seq of the node which issued the rename op

    length ← renamedBlocks.length
    firstId ← idBegin(renamedBlocks[0])
    lastId ← idEnd(renamedBlocks[length − 1])

    pos ← position(firstId)
    newFirstId ← new Id(pos, nId, nSeq, 0)
    newLastId ← new Id(pos, nId, nSeq, length − 1)

    if firstId < id and id < lastId then
        pred ← findPredecessor(id, renamedBlocks)
        indexOfPred ← findIndex(pred, renamedBlocks)
        newPred ← new Id(pos, nId, nSeq, indexOfPred)
        return concat(newPredecessor, id)
    else if lastId < id and id < newLastId then
        return concat(newLastId, id)
    else if newFirstId < id and id < firstId then
        predOfNewFirstId ← new Id(pos, nId, nSeq, −1)
        return concat(predOfNewFirstId, id)
    else
        return id ▷ Return the identifier unchanged as it
does not conflict with the rename op
    end if
end function
```

As explained in subsection 3.2, some nodes may have applied concurrent *insert* operations to their state before receiving a given remote *rename* operation. Algorithm 1 also solves this case. It allows them to eventually converge with nodes which processed the *rename* operations before the concurrent *insert* operations.

Since nodes rely on the former state to transform concurrent operations to a *rename* one, they have to store it. Nodes need it until each of them can no longer issue concurrent operations to the corresponding *rename* operation. In other words, nodes can safely garbage collect the former state once the *rename* operation became causally stable [15]. Meanwhile, nodes can offload it onto the disk as it is only required to handle concurrent operations.

4 Evaluation

4.1 Simulations and benchmarks

To validate the proposed renaming mechanism, we performed an experimental evaluation to measure its performance on several aspects: 1. the size of the data structure 2. the integration times 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 real-time collaborative editing sessions, we ran simulations to generate traces to evaluate our data structure. The scenario is as follows: several authors collaboratively write an article. Initially, they mainly insert elements into the document. A few *remove* operations are still issued to simulate spelling mistakes. Once the document reaches a critical length, collaborators switch to a 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 all of them received all operations. We take snapshots of the document at given steps of the collaboration to measure the evolution of the document.

Simulations have been run with 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. 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), bots switch to the second phase and set both probabilities to 50%. Generated operations are broadcast to other nodes 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.

Code, benchmarks and results are available at: https://github.com/coast-team/mute-bot-random/.

4.2 Results

Memory overhead Using the snapshots generated, we compare the evolution of the size of the data structure. Results are depicted in Figure 5 where the blue line corresponds to the size of the content while the dash-dotted red one exhibits the growth of the LogootSplit data structure.

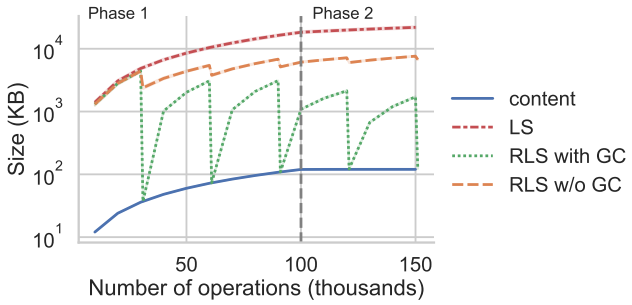


Figure 5. Evolution of the size of the document

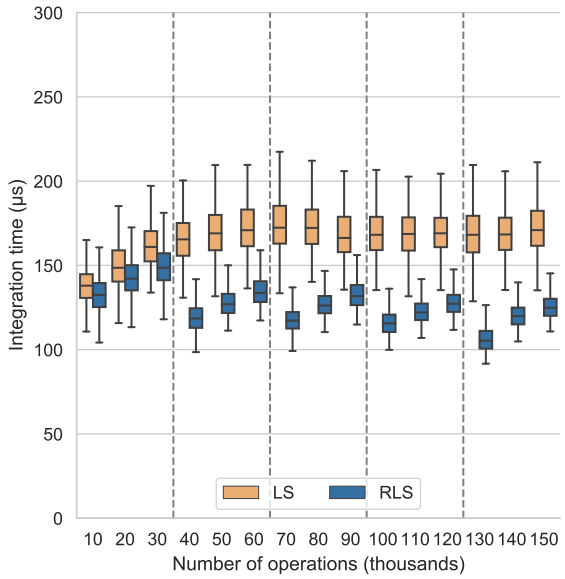
The dotted green line illustrates the growth of the RenamableLogootSplit document in its best-case scenario where *rename* operations become stable as soon as they are issued. Former states can then be garbage collected safely, maximising the benefits of the *renaming* mechanism. In this case, we observe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

The dashed orange line represents RenamableLogootSplit worst-case scenario. Here, we assume that *rename* operations never become causally stable and that nodes have to store former states forever. However, obtained results show that RenamableLogootSplit still outperforms LogootSplit and reduces by 66% the overhead. This outcome is due to the change of data structure used to represent the state that takes place when applying a *rename* operation. To perform updates efficiently, our implementation uses an AVL, a self-balancing tree, to represent the sequence. However, we no longer update the former state once it has been renamed but only query it to transform concurrent operations. We thus store it as an array, a more efficient memory-wise data structure, to save space.

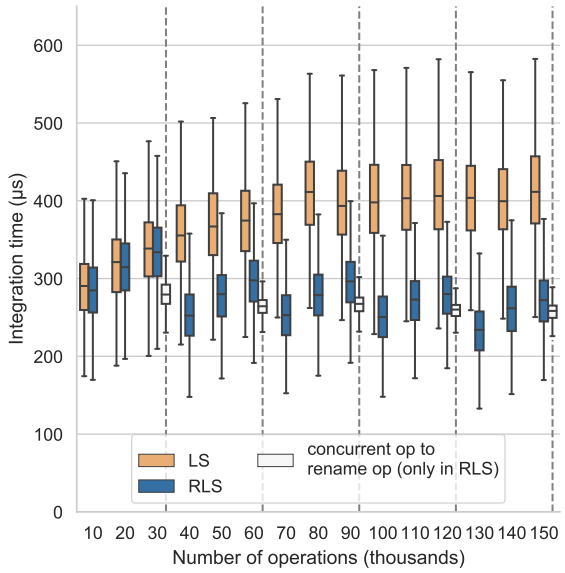
Integration times of standard operations We also measure the impact of the renaming mechanism on the integration times of *insert* and *remove* operations.

Figure 6a displays the integration times of local operations while Figure 6b exhibits remote ones. In both cases, the light orange boxplots correspond to integration times of LogootSplit while dark blue ones to the integration times of RenamableLogootSplit. The results show that the *renaming* mechanism allows to reduce the integration times of future operations.

In Figure 6b, the white boxplots display the integration times of concurrent operations to a *rename* one. As illustrated



(a) Local operations



(b) Remote operations

Figure 6. Integration time of standard operations

in subsection 3.3, these operations require to be transformed before being applied to the renamed state. The results presented here show that the whole process is actually faster than applying them directly on the former state.

Integration time of* rename *operation Finally, we measured the integration times of *rename* operations according to the size of the document. Results are displayed in Figure 7 where the blue line corresponds to the integration times of local *rename* operations while the dashed orange one corresponds to the integration times of remote ones.

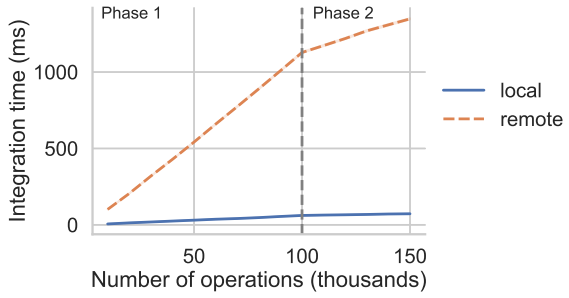


Figure 7. Integration time of rename operations

The main result of this benchmark is that the unit of time used when applying *rename* operations is in hundreds of milliseconds. It is necessary to take this into account when designing the strategy used to trigger *rename* operations in order to minimise the impact of integration times on the user experience.

5 Related work

Several approaches have been proposed to reduce the growth of variable-size identifiers. However, to the best of our knowledge, no work has been presented to decrease the number of blocks generated.

5.1 The core-nebula approach

In [16, 17], authors design for Treedoc [8] a renaming mechanism to reassign shorter identifiers to elements. Nodes rely on a consensus mechanism to trigger the renaming and a catch-up protocol to handle concurrent updates. Since consensus algorithms are costly in large-scale distributed systems with churn, Letia et al. [16] introduce a two-tier architecture. Nodes are splitted between the *core*, a set of stable and highly connected nodes, and the *nebula* made of the remaining ones. Only members of the *core* participate in the consensus leading to the renaming.

While quite similar, our approach differs on several aspects. First, in the *core-nebula* approach, nodes from the *core* are unable to integrate concurrent operations to the *rename* one. Nodes from the *nebula* have to use the catch-up protocol to transform their concurrent operations and to re-broadcast them. It requires additional communications and introduces some delay in the collaboration. In our approach, every node is able to transform and to integrate operations from past epochs directly.

Second, we designed RenamableLogootSplit with fully distributed systems in mind. Nodes can issue *rename* operations without coordination and use *operational transformation* to resolve conflicts. However, to simplify the system, it is possible to adopt the *core-nebula* approach to prevent nodes from issuing concurrent *rename* operations.

5.2 The LSEQ approach

In [4, 18], Nédelec et al. introduce another approach to address the identifiers growth issue: LSEQ. Its insight consists in combining several strategies to generate new identifiers. It enables LSEQ to reduce the growth of identifiers from a linear progression to a polylogarithmic one.

However, LSEQ does not prevent each inserted element to introduce its own overhead. The document continues to inflate with each insertion. On the other hand, our approach allows the metadata of the data structure to be periodically reset, regardless of the number of elements.

As with the *core-nebula* approach, it is possible to combine the LSEQ approach with ours. The identifier allocation strategies of LSEQ would allow to reduce the growth of identifiers between *rename* operations. It would enable to reduce the frequency of the expensive *rename* operation without deteriorating the performance of the data structure.

6 Conclusions and future work

In this paper, we introduced a novel Sequence CRDT belonging to the variable-size identifiers approach: RenamableLogootSplit. This new data structure embeds a renaming mechanism in its specification. It enables nodes to reassign shorter identifiers to elements and group them into one block to minimise the metadata. Experiments results showed that RenamableLogootSplit reduces the overhead of the data structure in both best-case (by hundreds times) and worst-case scenarios.

Regarding future work, we now have to design an efficient strategy used to trigger *rename* operations while minimising their impact on the user experience. User behaviour studies, inspired by [19, 20], could be led in the context of real-time collaborative writing to set an acceptable upper-bound to their integration times.

Finally, we designed RenamableLogootSplit with fully distributed systems in mind. However, for the sake of brevity, we presented it in this paper under the assumption that no concurrent *rename* operations were issued. This scenario is actually akin to systems in which nodes synchronise to trigger the renaming mechanism. In a future work, we will introduce and evaluate the additional steps required to use RenamableLogootSplit in its original settings.

References

[1] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5): 172–182, December 1995. ISSN 0163-5980. doi: 10.1145/224057.224070. URL https://doi.org/10.1145/224057.224070.

[2] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400, 2011. doi: 10.1007/978-3-642-24550-3_29.

[3] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark 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*, Onward! 2019, page 154–178, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369954. doi: 10.1145/3359591.3359737. URL https://doi.org/10.1145/3359591.3359737.

[4] Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. A scalable sequence encoding for collaborative editing. *Concurrency and Computation: Practice and Experience*, n/a(n/a):e4108. doi: 10.1002/cpe.4108. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4108. e4108 cpe.4108.

[5] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data Consistency for P2P Collaborative Editing. In *ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, pages 259 – 268, Banff, Alberta, Canada, November 2006. ACM Press. URL https://hal.inria.fr/inria-00108523. http://portal.acm.org/.

[6] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2010.12.006. URL http://www.sciencedirect.com/science/article/pii/S0743731510002716.

[7] Loïck Briot, Pascal Urso, and Marc Shapiro. High Responsiveness for Group Editing CRDTs. In *ACM International Conference on Supporting Group Work*, Sanibel Island, FL, United States, November 2016. doi: 10.1145/2957276.2957300. URL https://hal.inria.fr/hal-01343941.

[8] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, June 2009. doi: 10.1109/ICDCS.2009.20.

[9] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot : A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, pages 404–412, Montreal, QC, Canada, June 2009. IEEE Computer Society. doi: 10.1109/ICDCS.2009.75. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.75.

[10] Luc André, Stéphane Martin, Gérald Oster, and Claudia-Lavinia Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2013*, pages 50–59, Austin, TX, USA, October 2013. IEEE Computer Society. doi: 10.4108/icst.collaboratecom.2013.254123.

[11] Matthieu Nicolas. Efficient renaming in CRDTs. In *Middleware 2018 - 19th ACM/IFIP International Middleware Conference (Doctoral Symposium)*, Rennes, France, December 2018. URL https://hal.inria.fr/hal-01932552.

[12] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005. ISSN 0360-0300. doi: 10.1145/1057977.1057980. URL https://doi.org/10.1145/1057977.1057980.

[13] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, page 59–68, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130090. doi: 10.1145/289444.289469. URL https://doi.org/10.1145/289444.289469.

[14] D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, Oct 2009. ISSN 2161-9883. doi: 10.1109/TPDS.2008.240.

[15] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, pages 126–140, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[16] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. In *LADIS 2009 - 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, volume 44 of *Operating Systems Review*, pages 29–34, Big Sky, MT, United States, October 2009. Assoc. for Computing Machinery. doi: 10.1145/1773912.1773921. URL https://hal.inria.fr/hal-01248270.

[17] Marek Zawirski, Marc Shapiro, and Nuno Preguiça. Asynchronous rebalancing of a replicated tree. In *Conférence Française en Systèmes d'Exploitation (CFSE)*, page 12, Saint-Malo, France, May 2011. URL https://hal.inria.fr/hal-01248197.

[18] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng 2013, pages 37–46, September 2013. doi: 10.1145/2494266.2494278.

[19] Claudia-Lavinia Ignat, Gérald Oster, Meagan Newman, Valerie Shalin, and François 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*, Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, pages 191 – 198, Seattle, WA, United States, September 2014. doi: 10.1007/978-3-319-10831-5_29. URL https://hal.archives-ouvertes.fr/hal-01088815.

[20] Claudia-Lavinia Ignat, Gérald Oster, Olivia Fox, François Charoy, and Valerie Shalin. How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking. In Nina Boulus-Rodje, Gunnar Ellingsen, Tone Bratteteig, Margunn Aanestad, and Pernille Bjorn, editors, *European Conference on Computer Supported Cooperative Work 2015*, Proceedings of the 14th European Conference on Computer Supported Cooperative Work, pages 223–242, Oslo, Norway, September 2015. Springer International Publishing. doi: 10.1007/978-3-319-20499-4_12. URL https://hal.inria.fr/hal-01238831.