

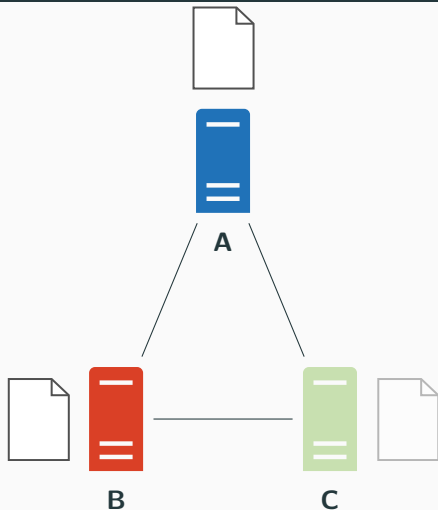
Efficient renaming in Conflict-free Replicated Data Types (CRDTs)

Case Study of a Sequence CRDT : LogootSplit

Matthieu Nicolas (matthieu.nicolas@loria.fr), Gérald Oster, Olivier Perrin
COAST team

December 12, 2021

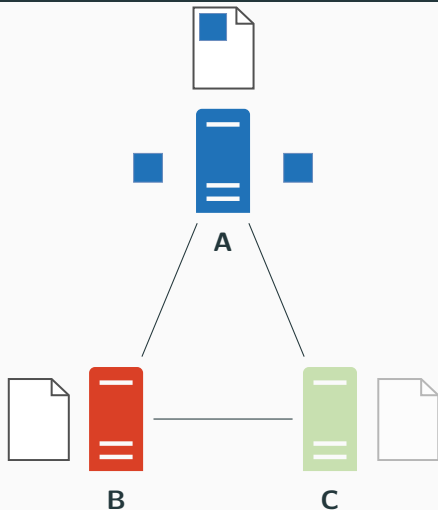
Conflict-free Replicated Data Types (CRDTs)^[1]



- Replicated data structure

^[1]Marc Shapiro et al. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400, 2011. DOI: 10.1007/978-3-642-24550-3_29 .

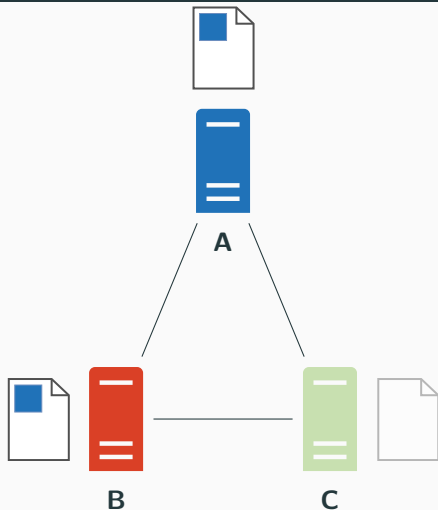
Conflict-free Replicated Data Types (CRDTs)^[1]



- Replicated data structure
- Updates performed without coordination

^[1]Marc Shapiro et al. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400, 2011. DOI: 10.1007/978-3-642-24550-3_29 .

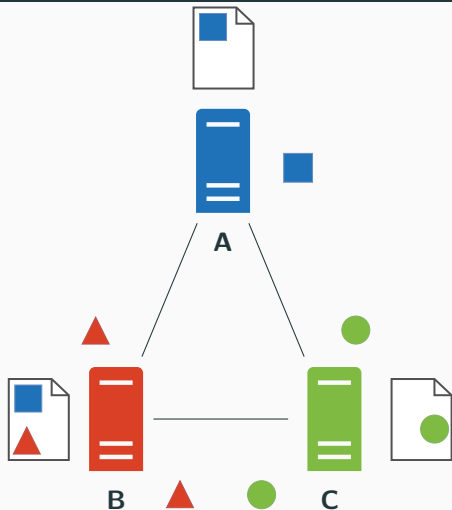
Conflict-free Replicated Data Types (CRDTs)^[1]



- Replicated data structure
- Updates performed without coordination

^[1]Marc Shapiro et al. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400, 2011. DOI: 10.1007/978-3-642-24550-3_29 .

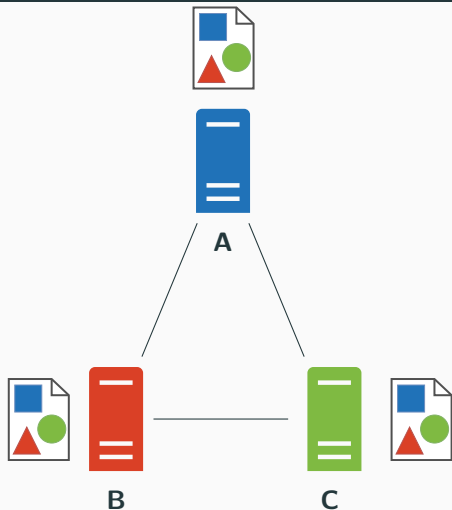
Conflict-free Replicated Data Types (CRDTs)^[1]



- Replicated data structure
- Updates performed without coordination

^[1]Marc Shapiro et al. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2011*, pages 386–400, 2011. DOI: 10.1007/978-3-642-24550-3_29 .

Conflict-free Replicated Data Types (CRDTs)^[1]



- Replicated data structure
- Updates performed without coordination
- Strong Eventual Consistency

^[1]Marc Shapiro et al. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2011*, pages 386–400, 2011. DOI: 10.1007/978-3-642-24550-3_29 .

- State of the art of *Sequence CRDTs*
- Elements are ordered by their identifier, noted here with the following formalism: $position_{\text{offset}}^{node_id \ node_seq}$

^[2]Luc André et al. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2013*, pages 50–59, Austin, TX, USA. IEEE Computer Society, October 2013. DOI: 10.4108/icst.collaboratecom.2013.254123 .

- State of the art of *Sequence CRDTs*
- Elements are ordered by their identifier, noted here with the following formalism: $position_{offset}^{node_id \ node_seq}$

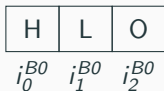


Figure 1: State of a sequence which contains the elements "hlo" and their corresponding identifiers

^[2]Luc André et al. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2013*, pages 50–59, Austin, TX, USA. IEEE Computer Society, October 2013. DOI: 10.4108/icst.collaboratecom.2013.254123 .

- State of the art of *Sequence CRDTs*
- Elements are ordered by their identifier, noted here with the following formalism: $position_{offset}^{node_id \ node_seq}$

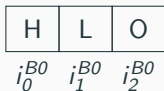


Figure 1: State of a sequence which contains the elements "hlo" and their corresponding identifiers



Figure 2: State of a sequence which contains the block "hlo"

^[2]Luc André et al. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2013*, pages 50–59, Austin, TX, USA. IEEE Computer Society, October 2013. DOI: 10.4108/icst.collaboratecom.2013.254123 .

- State of the art of *Sequence CRDTs*
- Elements are ordered by their identifier, noted here with the following formalism: $position_{\substack{node_id \\ offset}}^{node_seq}$

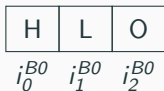


Figure 1: State of a sequence which contains the elements "hlo" and their corresponding identifiers



Figure 2: State of a sequence which contains the block "hlo"

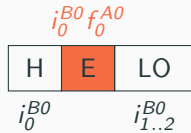
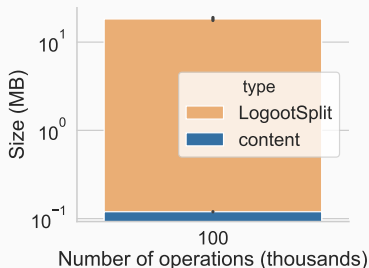


Figure 3: State of a sequence which contains the elements "hlo" and their corresponding identifiers

^[2]Luc André et al. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2013*, pages 50–59, Austin, TX, USA. IEEE Computer Society, October 2013. DOI: 10.4108/icst.collaboratecom.2013.254123 .

- **Evergrowing overhead:** impacts memory, bandwidth and CPU



- **Operation count:** 100k
- **Size of content:** 100KB
- **Size of data structure:** 20MB

Figure 4: Memory footprint of the data structure

How to reduce the overhead introduced by the data structure?

Our approach

Reassign shorter identifiers and aggregate them into blocks in a fully distributed manner

RenamableLogootSplit

- Propose *RenamableLogootSplit*, *LogootSplit* with a *rename* operation
- Can be performed without coordination

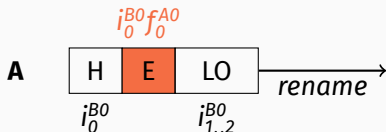


Figure 5: Example of renaming

RenamableLogootSplit

- Propose *RenamableLogootSplit*, *LogootSplit* with a *rename* operation
- Can be performed without coordination

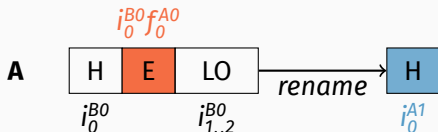


Figure 5: Example of renaming

- Generates a new identifier for the first element, based on its previous identifier

RenamableLogootSplit

- Propose *RenamableLogootSplit*, *LogootSplit* with a *rename* operation
- Can be performed without coordination

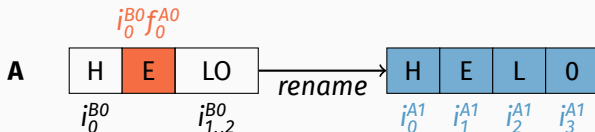


Figure 5: Example of renaming

- Generates a new identifier for the first element, based on its previous identifier
- Then generates contiguous identifiers for all following elements

RenamableLogootSplit

- Propose *RenamableLogootSplit*, *LogootSplit* with a *rename* operation
- Can be performed without coordination

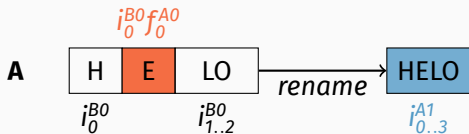


Figure 5: Example of renaming

- Generates a new identifier for the first element, based on its previous identifier
- Then generates contiguous identifiers for all following elements

Handling concurrent operations

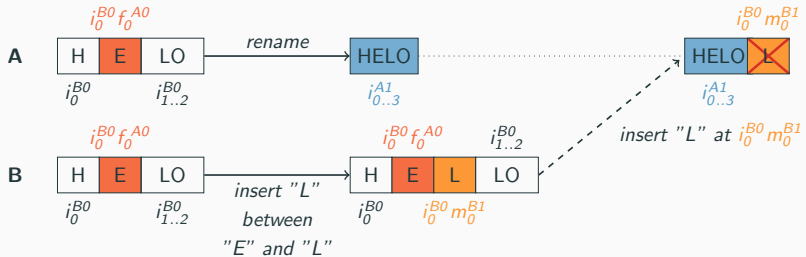


Figure 6: Example of concurrent update

- Can issue operations concurrently to *rename*
- Produce inconsistencies if applied naively

Handling concurrent operations

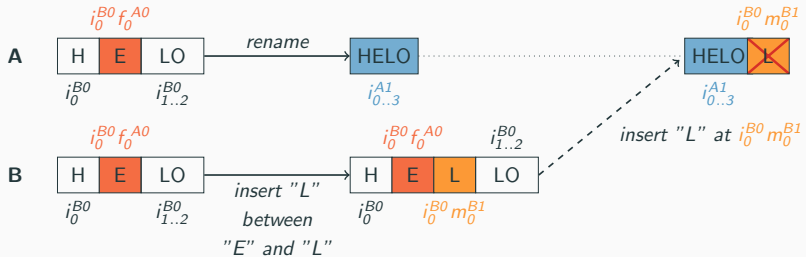


Figure 7: Example of concurrent update

- Can issue operations concurrently to *rename*
- Produce inconsistencies if applied naively

Handling concurrent operations

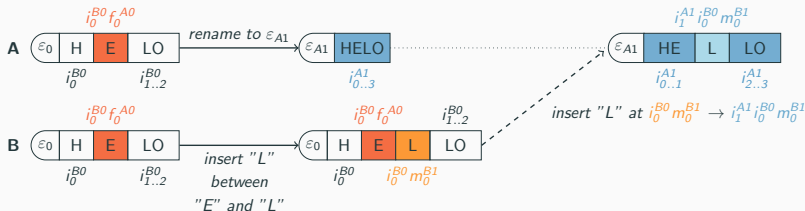


Figure 8: Example of concurrent update

- Use *epoch-based* system to track concurrent operations
- Use transform operations against *rename* ones (*OT*)

What about concurrent rename operations ?

What about concurrent rename operations ?

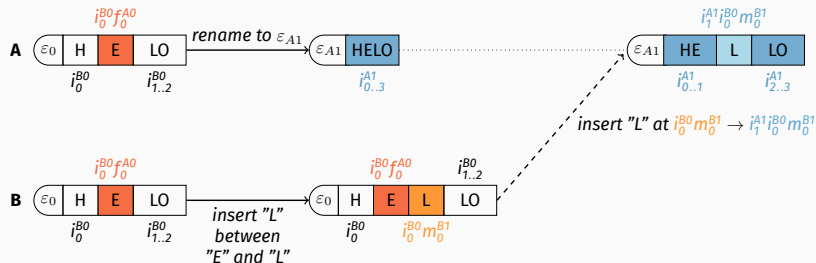


Figure 9: Concurrent *rename* operations leading to divergent states

What about concurrent rename operations ?

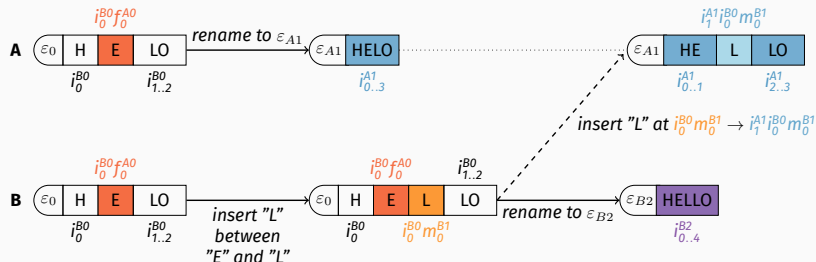


Figure 9: Concurrent *rename* operations leading to divergent states

What about concurrent rename operations ?

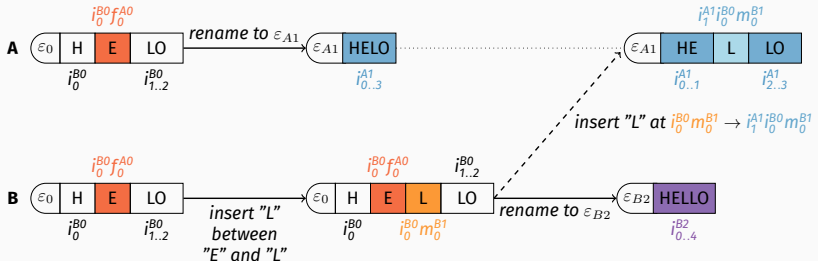


Figure 9: Concurrent *rename* operations leading to divergent states

- *Rename* operations are system operations

What about concurrent rename operations ?

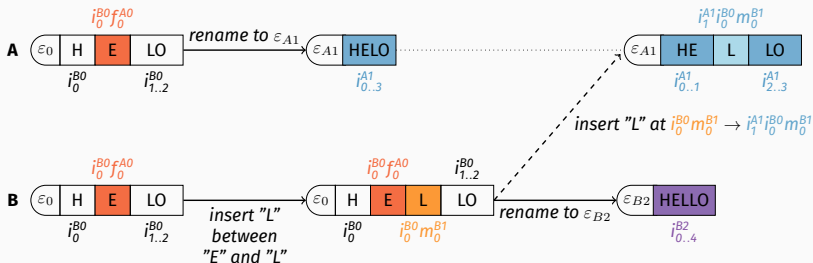


Figure 9: Concurrent *rename* operations leading to divergent states

- *Rename* operations are system operations
- Can resolve conflict by only applying one of them

How to do so ?

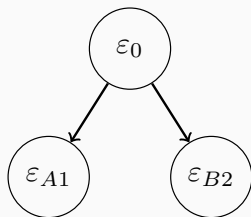


Figure 10: *Epoch tree* corresponding to previous scenario

How to do so ?

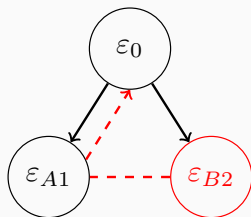


Figure 10: *Epoch tree* corresponding to previous scenario

- Define total order on epochs to select **target epoch**

How to do so ?

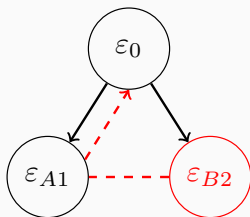


Figure 10: *Epoch tree* corresponding to previous scenario

- Define total order on epochs to select **target epoch**
- Design transformation function to revert *rename* operation

Applying concurrent rename operations

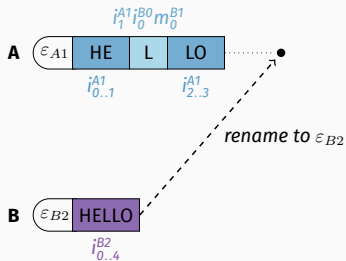


Figure 11: Applying a concurrent *rename* operation

Applying concurrent rename operations

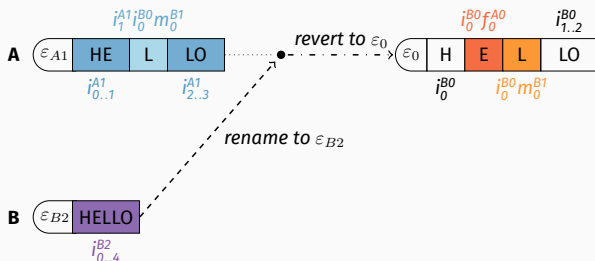


Figure 11: Applying a concurrent *rename* operation

- Revert state to equivalent one at LCA epoch

Applying concurrent rename operations

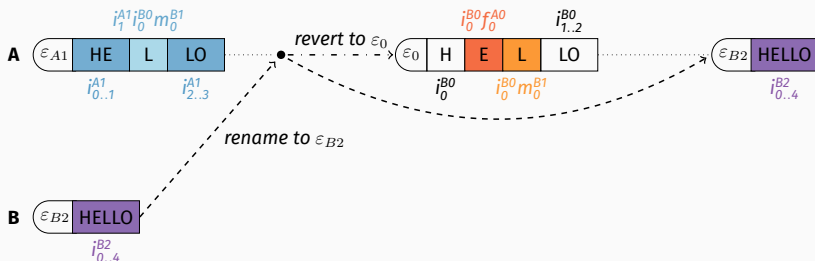


Figure 11: Applying a concurrent *rename* operation

- Revert state to equivalent one at LCA epoch
- Apply then *rename* operations leading to target epoch

Need to store former state until no more concurrent operations

- Can garbage collect it once the *rename* operation is causally stable^[3]
- Can offload it to the disk meanwhile

Need to propagate former state to other nodes

- Can compress the operation to minimise bandwidth consumption
- Can trigger *rename* operations at a given number of blocks

^[3]Carlos Baquero et al. Making operation-based crdts operation-based. In Kostas Magoutis et al., editors, *Distributed Applications and Interoperable Systems*, pages 126–140, Berlin, Heidelberg. Springer Berlin Heidelberg, 2014 .

Evaluation

Ran simulations to compare performance of
RenamableLogootSplit to LogootSplit one

Results - Convergence

- Compared final content of nodes per sessions
- Did not observe any divergence
- Empirical result, not a proof...
- ... but represents first step towards the validation

Results - Memory footprint

- **Phase 1 (content generation):** 80/20% of *insert/remove*
- **Phase 2 (editing):** 50/50% of *insert/remove*
- Nodes switch to phase 2 when document reaches critical size (15 pages - 60k elements)

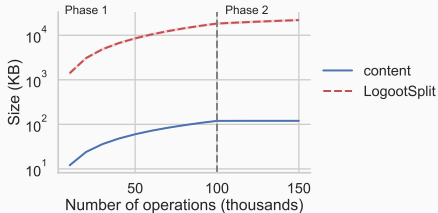


Figure 12: Evolution of the size of the document

Results - Memory footprint

- **Phase 1 (content generation):** 80/20% of *insert/remove*
- **Phase 2 (editing):** 50/50% of *insert/remove*
- Nodes switch to phase 2 when document reaches critical size (15 pages - 60k elements)

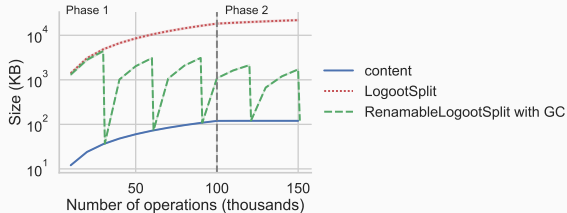


Figure 12: Evolution of the size of the document

- *Rename* resets the overhead of the CRDT, if can garbage collect

Results - Memory footprint

- **Phase 1 (content generation):** 80/20% of *insert/remove*
- **Phase 2 (editing):** 50/50% of *insert/remove*
- Nodes switch to phase 2 when document reaches critical size (15 pages - 60k elements)

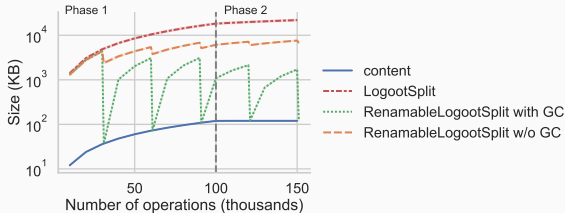
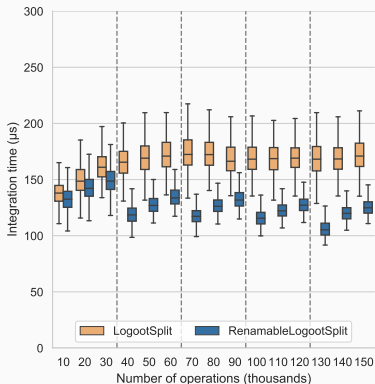


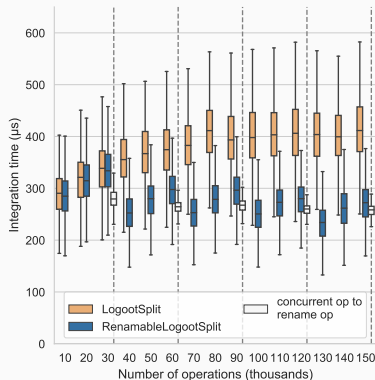
Figure 12: Evolution of the size of the document

- *Rename* resets the overhead of the CRDT, if can garbage collect
- *Rename* still reduces by 66% the size otherwise

Results - Integration time of insert operations



(a) Local operations



(b) Remote operations

Figure 13: Evolution of the integration time of *insert* operations

- *Rename* resets integration times of future operations
- Transforming concurrent operations is actually faster than applying them on former state

Results - Integration time of rename operations

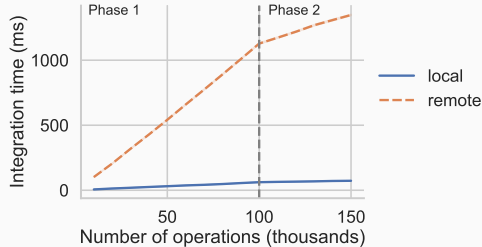


Figure 14: Evolution of the integration time of *rename* operations

- Noticeable by users if delayed too much

Research trail: propose strategies to retain acceptable integration time for *rename* operations

Conclusion

Done

- Designed a *rename* operation for LogootSplit
- Compared its performance to one of LogootSplit

Conclusion

Done

- Designed a *rename* operation for LogootSplit
- Compared its performance to one of LogootSplit

Work in progress

- Publishing it
- Writing the manuscript

Conclusion

Done

- Designed a *rename* operation for LogootSplit
- Compared its performance to one of LogootSplit

Work in progress

- Publishing it
- Writing the manuscript

To do

- Prove formally the correctness of the mechanism
- Design better strategies to select the target epoch
- Improve performance of *rename* operations

Thanks for your attention, any questions?



LogootSplit identifiers

- To comply with these constraints, LogootSplit proposes identifiers composed of quadruplets of integers of the following form:

$$position_{offset}^{node_id \ node_seq}$$

- *position* allows to determine the position of this identifier compared to others
- *node_id* refers to the node's identifier, assumed to be unique
- *node_seq* refers to the node's logical clock, which increases monotonically with local operations
- *offset* refers to the element position in its original block

Identifier constraints

- To fulfill their role, identifiers have to comply to several constraints:

Globally unique

- Identifiers should never be generated twice, neither by different users nor by the same one at different times

Totally ordered

- We should always be able to compare and order two elements using their identifiers

Dense set

- We should always be able to add a new element, and thus a new identifier, between two others

- Core-nebula approach^[4]
 - Reassigns shorter identifiers to elements. . .
 - . . . but requires consensus
- LSEQ^[5]
 - Set of strategies to reduce the growth of identifiers . . .
 - . . . but overhead still proportional to number of elements

^[4]Marek Zawirski et al. 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> .

^[5]Brice Nédelec et al. A scalable sequence encoding for collaborative editing. *Concurrency and Computation: Practice and Experience*:e4108. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4108>.

Propose a strategy to avoid conflicting rename operations

- How to minimise likelihood of concurrent *rename* operations without coordinating?

Propose a smarter strategy to choose the "winning" renaming

- How to minimise the overall computations?

Experimental settings

- Use Node.js version 13.1.0
- Obtained documents sizes using our fork of *object-sizeof* ^[6]
- Ran benchmarks on a workstation equipped of a Intel Xeon CPU E5-1620 (10MB Cache, 3.50 GHz) with 16GB of RAM running Fedora 31
- Measured times using `process.hrtime.bigint()`

^[6]<https://www.npmjs.com/package/object-sizeof>