Improving Replicated Sequences Performances

Matthieu Nicolas, Gérald Oster, and Olivier Perrin

Université de Lorraine, CNRS, Inria, LORIA, F-54500, France

1 Introduction

2 Replicated Sequences

2.1 Sequence

- Abstract Data Type (ADT) which allows to represent a list of values
- Provide two operations to update the sequence, *insert* and *remove*
- $insert(S, index, elt) = S'$ inserts the value *elt* at the index *index* into the sequence S and returns the updated sequence S'
- $remove(S, index) = S'$ removes the value at the position *index* in the sequence S and returns the updated sequence S'

2.2 Conflict-free Replicated Data Types (CRDTs) [1, 2]

- Optimistically replicated data structure
- Updates performed without coordination between nodes
- Ensure Strong Eventual Consistency (SEC) [2]

2.3 Logoot [3]

- Element-wise Sequence CRDT
- Key insight is to replace changing and unreliable *indexes* with universal and immutable *positions*
  - *indexes* are adapted for sequential executions
  - They are closely tied to the current state of the sequence, as updating the later provoke a shift to them
  - However, they can not be used in a distributed setting, in which the order of the execution of the updates can be different from one node to another, to embody the user's intention
  - Conversely, Logoot's *positions* are designed to be independent of the state
  - This allows the definition of commutative operations, providing a *Sequence* suited for distributed settings

  TODO: Insérer un exemple de manipulation concurrente d'une séquence (ins/remove en parallèle), sans mécanisme de résolution de conflits, pour illustrer la divergence du résultat si on ne se base que sur des index – Matthieu

- When inserting an element into the sequence, a fitting *position* is computed by the node and associated to the element

- These *positions* perform several roles
  - They identify uniquely an element
  - They order the elements relatively to each other
- To be able to perform these roles, *positions* need to comply with several constraints
  - Be globally unique: several nodes should not able to compute the same *position* concurrently
  - Be temporarily unique: a node should not be able to associate the same *position* to different elements during the lifetime of the sequence
  - Be totally ordered: a order relation must exist over *positions* so a node can order two elements given their respective *position*
  - Belong to a dense set: a node should always be able to generate a new *position* between two others
- To comply with these constraints, the *positions* are composed of one or several of the following tuples:

  $$< priority, id_{site}, seq_{site} >$$

  where:

  - *priority* allows to determine the location of this position relatively to others
  - id_{site} refers to the node's identifier, assumed to be unique
  - seq_{site} refers to the node's logical clock, which increases monotonically with local updates

  TODO: Reprendre l'exemple précédent, mais en remplaçant les index par des positions Logoot, pour illustrer la convergence – Matthieu

- It is worth to notice that: *J'illustre dans les prochains points comment la composition d'une position permet d'assurer l'ensemble des contraintes stipulées précédemment, mais l'explication concernant l'espace dense se révèle être particulièrement complexe et nécessiter de rentrer dans des détails. À garder ? – Matthieu*
  - The couple $< id_{site}, seq_{site} >$ of the last tuple of the position ensures its uniqueness as
    - No other node can generate a position using the same id_{site} as it is unique
    - No other position can be generated by the same node using the same seq_{site} as it is increasing monotonically with local updates
  - Every part of positions can be used to define a total order based on the lexicographical order
  - To ensure that positions form a dense set, we have to reserve exclusively the usage of the minimal value of *priority* to a specific tuple, *minTuple*. By doing this we have, for every other tuple t, $minTuple < t$. *minTuple* is then used only as an intermediary tuple, the default value when a node has to generate a tuple less than another one, but is not able to. Thus, given two positions $p1$ and $p2$ such as $p1 < p2$, any node is always able to generate a new position $p3$ such as $p1 < p3 < p2$ by reusing the tuples of $p1$, appending *minTuple* as many times as required and finally adding a new tuple t.

2.4 LogootSplit [4]

- Block-wise Sequence CRDTs
- Goal is to improve further the performances of Logoot
- Generating and associating a position to each element of the sequence is expensive
- André et al. [4] proposes to aggregate dynamically elements into blocks

- It allows to reduce the metadata per element as we only need to store a *position interval* for a block independently of how many elements it contains
- To do so, add a new component to positions tuples: the $offset$ *Préciser que $offset$ doit appartenir à un ensemble énumérable, de façon à ce que l'on puisse déterminer son prédecesseur, successeur ou si une valeur est manquante ? – Matthieu*
- LogootSplit *positions* are thus composed of one or several of the following tuples:

$$< priority, id_{site}, seq_{site}, offset >$$

- Define positions as aggregable if all but the $offsets$ of their last tuple are identical and their respective $offsets$ are consecutive
- Define *position interval* as the following couple:

$$< posBegin, end >$$

  where:

  - *posBegin* refers to the first position of the interval
  - *end* refers to the value of the offset of the last position of the interval

- Given these two values, we are able to recreate all positions from the interval if needed
- It is worth to notice that
  - The uniqueness of a LogootSplit position is ensured by the triple $< id_{site}, seq_{site}, offset >$
  - As such, an $offset$ can not be used twice for the same block
  - It is then necessary to prevent such a case from happening when a node prepend or append a new element to an existing block
  - It can be achieved by disabling flags allowing to preprend/append when the first/last element of a block is removed

2.5 Limits

Besoin d'une section (ou autre) entre les limites de LogootSplit et le renommage pour justement introduire l'approche choisie (renommage) et les raisons (diminution de la taille des identifiants, aggrégation des blocs existants) – Matthieu

3 Renaming in a (de)centralized setting

3.1 System Model

3.2 Intuition

3.3 Renaming locally

3.4 Dealing with concurrent operations

3.5 Applying a remotely generated renaming operation

3.6 Garbage collection

3.7 Limits

4 Renaming in a fully distributed setting

4.1 System Model

4.2 Intuition

4.3 Strategy to determine leading epoch in case of concurrency

4.4 Transitioning from a losing epoch to the leading one

4.5 Garbage collection

5 Evaluation

6 Discussion

6.1 Offloading on disk unused renaming rules

- As stated previously, nodes have to keep renaming rules as long as another nodes may issue operations which would require to be transformed to be applied
- Thus nodes need to keep track of the progress of others to determine if such operations can still be issued or if it is safe to garbage collect the renaming rules
- In a fully distributed setting, this requirement is difficult to reach as a node may join the collaboration, perform few operations and then disconnect
- From the point of view of other nodes, they are not able to determine if this node disconnected temporarily or if it left definitely the collaboration
- However, as the disconnected node stopped progressing, it holds back the whole system and keeps the current active nodes from garbage collecting old renaming rules
- To limit the impact of stale nodes on active ones, we propose that nodes offload unused renaming rules by storing them on disk

Présenter une méthode pour déterminer les règles de renommage non-utilisées (conserver uniquement les règles utilisées pour traiter les x dernières opérations ?) – Matthieu

6.2 Alternative strategy to determine leading epoch

6.3 Postponing transition between epochs in case of instability

- May reach a situation in which several nodes keep generating concurrent renaming operations on different epoch branches
- In such case, switching repeatedly between these concurrent branches may prove wasteful *"costly" plutôt? – Matthieu*
- However, as long as nodes possess the required renaming rules, they are able to rewrite operations from the other side and to integrate them into their copy, even if they are not on the latest epoch of their branch
  - At the cost of an overhead per operation
- Thus not moving to the new current epoch does not impede the liveness of the system
- Nodes can wait until one branch arise as the leading one then move to this epoch
- To speed up the emergence of such a branch, communications can be increased between nodes in such case to ease synchronisation

6.4 Compressing the renaming operation

- Propagating the renaming operation consists in broadcasting the list of blocks on which the renaming was performed, so that other nodes are able to compute the same rewriting rules
- This could prove costly, as the state before renaming can be composed of many blocks, each using long positions
- We propose an approach to compress this operation to reduce its bandwidth consumption at the cost of additional computations to process it
- Despite the variable length of positions, the parts required to identify an position uniquely are fixed
  - We only need the *siteId* and the *seq* of the last tuple of the position to do so
- Instead of broadcasting the list of whole positions, the node which performs the renaming can just broadcast the list of tuples $< siteId, seq >$
- On reception of a compressed renaming operation, a node needs first to regenerate the list of renamed blocks to be able to apply it
- To achieve so, it can browse its current state looking for positions with corresponding tuples $< siteId, seq >$
- If some positions are missing from the state, it means that they were deleted concurrently
- The node can thus browse the concurrent remove operations to the renaming one to find the missing blocks
- Once all positions has been retrieved and the list of blocks computed, the renaming operation can be processed normally

6.5 Operational Transformation

Ajouter une section sur OT pour expliquer que gérer les opérations concurrentes aux renommages consiste en finalité à transformer ces opérations, mais qu'on a décidé de ne pas présenter l'approche comme étant de l'OT dans ce papier pour des raisons de simplicité ? – Matthieu

7 Conclusion

References

[1] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: `https://hal.inria.fr/inria-00555588`.

[2] 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. 2011, pp. 386–400. DOI: `10.1007/978-3-642-24550-3_29`.

[3] 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.* Montreal, QC, Canada: IEEE Computer Society, June 2009, pp. 404–412. DOI: `10.1109/ICDCS.2009.75`. URL: `http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.75`.

[4] 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.* Austin, TX, USA: IEEE Computer Society, Oct. 2013, pp. 50–59. DOI: `10.4108/icst.collaboratecom.2013.254123`.