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 shift 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, Logoot *positions* are composed of one or several of the following tuples:

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

  where:

  NOTE: on utilise actuellement des entiers pour représenter les éléments des tuples à l'heure actuelle mais pourrait utiliser n'importe quel type disposant d'une relation d'ordre, donc je ne sais pas quel type indiquer ici – Matthieu

  - *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 its own new tuple t.

2.4 LogootSplit [4]

- Block-wise Sequence CRDTs
- Goal is to improve further the performances of Logoot
- It is expensive to generate and associate a position to each element of the sequence
- 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 components but the $offsets$ of their last tuple are identical and if 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 keeping track of used $offset$ per block
    - To be more precise, only need to keep track of the minimum and maximum $offsets$ ever used for this block

2.5 Limits

- As shown previously, LogootSplit positions' size is not bounded in order to comply with the dense set constraint
- Positions will become longer as the collaboration progresses, downgrading the performances of the application *NOTE: trouver un autre terme que "collaboration"? – Matthieu*
  - Since nodes have to broadcast positions, store them but also compare them to determine their order

- *Ici j'aimerais aussi ajouter quelques mots pour expliquer que le nombre de blocs influe négativement sur les performances de l'application (augmente le temps de parcours de la liste, et donc le temps d'exécution d'une insertion ou d'une suppression; doit conserver un position interval pour chaque bloc). Une séquence est donc optimale si elle composée d'un unique bloc. Mais en raison des contraintes définissant les positions aggrégeables (insertions à des index consécutifs par un même noeud, sans suppression intermédiaire) et de l'absence de mécanisme pour fusionner à posteriori les blocs existants, les séquences issues d'une collaboration sont généralement morcelées en de nombreux blocs. – Matthieu*

3 Approach

3.1 Overview

- We build up on top of LogootSplit
- To address its limitations, we introduce a renaming mechanism
- The purpose of this mechanism is to reassign shorter positions to each element such as all of them can be aggregated into one unique block
- This allows to reduce the metadata per element, the computation time required to apply next updates and also the bandwidth used to broadcast future updates
- *TODO: Expliquer que toutefois le mécanisme de renommage a un coût et qu'on cherche à le minimiser. Mais j'ai du mal à voir comment introduire la notion de cet overhead sans rentrer dans les détails qui en sont la source (règles de réécriture, arborescence des epochs, transformation des opérations concurrentes) – Matthieu*
- For simplicity purposes, will first present the renaming mechanism in the context of a centralised system, with only one node able to trigger the renaming mechanism
  - Will describe the functioning of this initial version and discuss its limitations
- Will then present a more elaborated version of the renaming mechanism, overcoming the limitations of the centralised version and allowing its use in a distributed setting

3.2 Renaming in a centralised setting

3.2.1 System Model

NOTE: Ça me paraît correct mais confusant de parler d'un système centralisé pour un système P2P où seulement une fonctionnalité (le renommage) n'est disponible que pour un noeud particulier. Voir pour mieux présenter cet aspect – Matthieu

- Peer-to-peer network
  - Nodes join and leave dynamically
- Nodes build and maintain a sequence collaboratively using LogootSplit
  - Each node owns a copy of the sequence and can edit it without any kind of coordination with others
- The network is unreliable
  - Messages can be lost, re-ordered and delivered multiple times
- To overcome the faults of the network, a message-passing layer is used to deliver messages to the application exactly-once, in the correct order
  - Removal of an element happens only after its insertion

- An anti-entropy mechanism is used by nodes to synchronise in a pairwise manner to detect and re-exchange lost messages
- One node is arbitrarily designed as the leader
- This node only is able to trigger renamings

3.2.2 Intuition

- Want to rename all positions from the current state
  - i.e. reassign a new, shorter position to each element such as all elements can be aggregated into one block as well
- Have to be able to deal with concurrent operations
  - As only the leader can perform renaming, can not have a concurrent renaming operation...
  - ... but can still receive concurrent insertions and removals
  - Applying them as such would lead to inconsistencies as the renaming may interfere
    - e.g. removing a position which has been since renamed
    - *TODO: Ajouter le schéma correspondant? – Matthieu*
  - It is required to transform these operations against the renaming to ensure consistency
- Have to broadcast the renaming operation so other nodes can also update their own state
- *TODO: Revenir sur la notion d'overhead du mécanisme de renommage. Expliquer qu'il génère des métadonnées supplémentaires qu'on GC à un moment donné pour des raisons d'efficacité – Matthieu*

3.2.3 Renaming locally

3.2.4 Dealing with concurrent operations

3.2.5 Applying a remotely generated renaming operation

3.2.6 Garbage collection

3.2.7 Limits

- The system is vulnerable to failures, as only one particular node is able to trigger renamings
  - A failure of this node would prevent the renaming mechanism from being triggered ever again
  - But other nodes would still be able to continue their collaboration in such scenario
  - The failure of the renaming mechanism does not impede the liveness of the system
- To address this fault-tolerance issue, can set up a consensus-based system
  - Require nodes to perform a consensus to trigger a renaming
  - But consensus algorithms are expensive and not suited for dynamic systems
  - Can adapt the idea introduced in [5]
  - In this paper, authors propose to divide a distributed system into two tiers: the *Core*, a small set of controlled and stable nodes, and the *Nebula*, an uncontrolled set of nodes
    - Only nodes from the *Core* would participate in the consensus leading to a renaming
  - Provide a trade-off between the cost of performing a renaming and the resilience of the system
- But this approach is not suited for all kind of applications
- In fully distributed systems, there is no central authority to provide a set of stable nodes acting as the *Core*

3.3 Renaming in a fully distributed setting

3.3.1 System Model

3.3.2 Intuition

3.3.3 Strategy to determine leading epoch in case of concurrency

3.3.4 Transitioning from a losing epoch to the leading one

3.3.5 Garbage collection

4 Evaluation

5 Discussion

5.1 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 nodes may join the collaboration, perform some operations and then disconnect
- Other nodes, from their point of view, are not able to determine if they disconnected temporarily or if it left definitely the collaboration
- However, as the disconnected nodes stopped progressing, they hold back the whole system and keep 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

5.2 Alternative strategy to determine leading epoch

5.3 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

5.4 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

5.5 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 et formaliser l'approche comme étant de l'OT dans ce papier pour des raisons de simplicité ? – Matthieu

6 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.

[5] 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*. Vol. 44. Operating Systems Review 2. Big Sky, MT, United States: Assoc. for Computing Machinery, Oct. 2009, pp. 29–34. DOI: 10.1145/1773912.1773921. URL: https://hal.inria.fr/hal-01248270.