A Generic Replicated Data Type for Strong Eventual Consistency

Kevin De Porre
kdeporre@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Florian Myter
Vrije Universiteit Brussel
Brussels, Belgium

Christophe De Troyer
Vrije Universiteit Brussel
Brussels, Belgium

Christophe Scholliers
Ghent University
Gent, Belgium

Wolfgang De Meuter
Vrije Universiteit Brussel
Brussels, Belgium

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT

Conflict-free replicated data types (CRDTs) [7] aid programmers develop highly available and scalable distributed systems. However, CRDTs require operations to commute which is not practical. This means that programmers cannot replicate regular objects without worrying about concurrency. In this paper, we introduce strong eventually consistent replicated objects (SECROs), a generic data type that is highly available and guarantees strong eventual consistency (SEC) without imposing restrictions on its operations.

CCS CONCEPTS

• **Software and its engineering → Distributed systems organizing principles**; **Consistency**;

KEYWORDS

distribution, replicated data types, strong eventual consistency

ACM Reference Format:
Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. 2019. A Generic Replicated Data Type for Strong Eventual Consistency. In *PaPoC '19: Workshop on Principles and Practice of Consistency for Distributed Data, March 25, 2019, Dresden, Germany.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

During the past decade we observed a shift from strongly consistent (CP) systems to highly available (AP) systems. A state-of-the-art approach towards high availability are conflict-free replicated data types (CRDTs) [7]. CRDTs rely on commutative operations to guarantee strong eventual consistency (SEC), a variation on eventual consistency that avoids the need for synchronisation, yielding high availability and low latency. However, this forces programmers to completely rethink data structures such that all operations commute. If the operations cannot be made commutative, programmers need to manually implement conflict resolution which is error-prone and results in brittle systems [1, 4, 7].

Other work proposed a hybrid approach that consists of synchronising non-commutative operations as they could lead to conflicts [2, 3]. This implies giving up on availability in favour of strong consistency when necessary.

In this work, we explore a different approach towards high availability which does not require operations to commute. The key idea is to find a conflict-free ordering for concurrent - potentially conflicting - operations. To this end, we rely on application-level *invariants*. Similarly to [8], we adhere to the idea that conflict detection and resolution depends on the semantics of the application.

2 OUR APPROACH

We introduce *strong eventually consistent replicated objects* (SECROs), a novel replicated data type that guarantees SEC and can be used to build AP systems. Like regular objects, SECROs contain state in the form of fields, and operations in the form of methods. In addition, SECROs also contain application-level invariants employed by the underlying replication protocol.

Since SECROs do not require operations to commute, replicas may diverge after concurrent operations. The only way to ensure convergence is to *totally* order the operations across all replicas. At first glance, this seems to require synchronisation. However, we show that it is possible to order the operations *asynchronously* as to remain highly available. This implies that replicas may need to reorder operations in the face of concurrent updates.

Naturally, randomly selecting a total order of operations is not sensible as it may leave the application in a state that violates application-level invariants. Instead, SECRO's replication protocol computes a total order of operations that upholds all invariants and respects causality.

From the programmer's perspective, developers can specify application-level invariants by means of *state validators*. A state validator is a declarative rule that is associated to an update operation and comes in two forms:

Preconditions enforce invariants prior to the execution of the associated update. Hence, preconditions approve or reject the state before applying the actual update. In case of a rejection, the operation is aborted and a different ordering of the operations will be tried.

Postconditions enforce invariants after the execution of the associated update. In contrast to preconditions, postconditions are checked after all *concurrent* operations complete. As

PaPoC '19, March 25, 2019, Dresden, Germany
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

such, postconditions approve or reject the state that results from a group of concurrent, potentially conflicting operations. In case of a rejection a different ordering is attempted.

To illustrate SECROs, we implement a collaborative text editor that represents text documents as a balanced tree of characters to provide efficient text manipulations. Both, insert and delete work on the granularity of individual characters. The code snippet below shows the core of this application in CScript [1], an extension of JavaScript with a prototype implementation of SECROs.

```
 1  class Document extends SECRO {
 2    constructor() {
 3      this._tree = new AvlTree((c1, c2) => c1.id - c2.id);
 4    }
 5    insertAfter(id, char) {
 6      const newId   = this.generateId(id),
 7            newChar = {char: char, id: newId};
 8      this._tree.add(newChar);
 9      return newChar;
10    }
11    pre insertAfter(doc, args) {
12      const [id, char] = args;
13      return id === null || doc.containsId(id);
14    }
15    post insertAfter(originalDoc, doc, args, newChar) {
16      const [id, char]  = args;
17      return (id === null && doc._tree.contains(newChar)) ||
18             doc.indexOf(id) < doc.indexOf(newChar.id);
19    }
20    delete(id) {
21      return this._tree.remove(id);
22    }
23    post delete(originalDoc, doc, args, res) {
24      const [id] = args;
25      return !doc.containsId(id);
26    }
27    indexOf(id) { /* ... */ }
28    containsId(id) { /* ... */ }
29    generateId(prev) { /* ... */ }
```

The above code snippet shows that characters are inserted by adding them to the underlying tree (lines 5 to 10), and deleted by removing them from the tree (lines 20 to 22). We use a third-party AVL tree implementation provided by the Closure library [2].

To convert the AVL tree into a SECRO, we associate preconditions and postconditions to the operations. The precondition on `insertAfter` (lines 11 to 14) ensures that the reference character after which to insert the new character exists. The postcondition on `insertAfter` (lines 15 to 19) ensures that the newly added character occurs at the correct position in the resulting tree, i.e. after the reference character identified by `id`. According to this postcondition any interleaving of concurrently inserted characters is valid. The postcondition on `delete` (lines 23 to 26) ensures that the deleted character no longer occurs in the document.

The aforementioned SECRO can be freely replicated and ensures that the three invariants are respected in the advent of concurrent operations, and that all replicas eventually converge.

3 SECRO'S REPLICATION PROTOCOL

SECROs propagate update operations to all replicas by means of an asynchronous broadcasting mechanism. Replicas are tuples (s_i, v_i, h) that maintain some state s_i, a version number v_i, and a sequence of operations called the *operation history h*. Incoming operations are inserted in the replica's operation history. This may require reordering part of the history.

To reorder its history the replica searches for a total order of the operations that meets two requirements. First, the order must reflect the causal relations between the operations. Second, applying the operations in the given order must uphold all application-level invariants. This second condition can be checked using the state validators defined by the programmer. An ordering which respects the aforementioned conditions denotes a *valid execution*.

Formally, given a set of update operations U, the happens-before relation $\prec$ induces a partial order between the updates [5]. We can further restrict this partial order such that the operations do not violate application-level invariants. This results in a partial order of the updates $O = (U, <)$ where $u_1 < u_2 \iff u_1 \prec u_2 \lor (u_1 \parallel u_2 \land respects\_invariants(apply(state, [u_1, u_2])))$ [3].

Finding a valid execution thus boils down to computing a linear extension O' of O. Such a serialisation is a total order O' that respects the partial order O. Our replication protocol guarantees that all replicas converge towards the same valid execution (*eventual consistency*), and that replicas that received the same operations have identical operation histories (*strong convergence*).

Unfortunately, finding a valid execution can be costly as it requires executing the history of updates (possibly more than once) and validating the resulting state. This is problematic when replicas have big operation histories. To tackle this performance problem, we introduce a *commit* operation which allows replicas to clear their history periodically. The commit operation commits the replica's internal state $s_i \longrightarrow s_{i+1}$. To this end, the replica applies the updates from the history and takes on this new state $s_{i+1} = apply(s_i, h)$. In addition, commit also increments the replica's version number and clears the history: $commit((s_i, v_i, h)) = (s_{i+1}, v_{i+1}, [])$. We designed the commit operation to commute. As such it does not require synchronising the replicas. Instead, commits are propagated in the background. Performance benchmarks involving commit can be found in [6].

4 ONGOING WORK

Although commit improves performance, it restricts the size of operation histories and thus puts an upper bound on concurrency. As an example, limiting operation histories to n operations implies that maximum n concurrent operations are supported.

We are currently working on improving the computation of valid executions. A possibility is to discover conflict patterns using static analysis, similar to I-offender sets in [2]. Using these patterns we could try to infer conflict-free orderings for the different types of conflicts. This means that at runtime replicas can order concurrent operations rather then exhaustively searching for a valid ordering. This may also omit the need for commit.

ACKNOWLEDGMENTS

Kevin De Porre is funded by an SB Fellowship of the Research Foundation - Flanders. Project number: 1S98519N.

[1]https://gitlab.com/iot-thesis/framework/tree/master
[2]https://developers.google.com/closure/library/

[3]$u_1 \parallel u_2$ denotes two concurrent updates u_1 and u_2.

REFERENCES

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient State-based CRDTs by Delta-Mutation. In *Int. Conference on Networked Systems*, Ahmed Bouajjani and Hugues Fauconnier (Eds.). Springer-Verslag, Agadir, Morocco, 62–76.

[2] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *10th European Conference on Computer Systems (EuroSys '15)*. Article 6, 16 pages.

[3] Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, Joao Leitao, Nuno Preguiça, et al. 2016. Geo-Replication: Fast If Possible, Consistent If Necessary. *IEEE Data Engineering Bulletin* 39, 1 (2016), 12.

[4] Martin Kleppmann and Alastair R Beresford. [n. d.]. A Conflict-Free Replicated JSON Datatype. *IEEE Trans. on Parallel and Distributed Systems* ([n. d.]).

[5] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[6] Kevin De Porre. 2018. *CScript: A Distributed Programming Language for Available and Consistent Sharing of Objects*. Master's thesis. Vrije Universiteit Brussel, Belgium.

[7] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer-Verslag, Grenoble, France, 386–400.

[8] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15th ACM Symp. on Operating Systems Principles (SOSP '95)*, M. B. Jones (Ed.). 172–182.