The Single-Writer Principle in CRDT Composition

Vitor Enes*
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

Paulo Sérgio Almeida†
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

Carlos Baquero†
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

ABSTRACT

Multi-master replication in a distributed system setting allows each node holding a replica to update and query the local replica, and disseminate updates to other nodes. Obtaining high availability typically entails allowing replicas to diverge and requires a background mechanism for re-establishing consistency. Conflict-free Replicated Data Types (CRDTs) extend standard sequential data-types with appropriate merge functions, and often can be composed together to create more complex ones. In this work we add a generic CRDT composition approach that explores the single-writer principle. By carefully controlling which part of the composition can be updated by each replica, we can derive efficient designs that cover new use-cases. After introducing the new construction we exemplify some uses, including how to emulate a simple Doodle functionality for selecting a common meeting schedule among different participants.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**;

KEYWORDS

Single-Writer Principle, Eventual Consistency, CRDTs.

ACM Reference format:
Vitor Enes, Paulo Sérgio Almeida, and Carlos Baquero. 2017. The Single-Writer Principle in CRDT Composition. In *Proceedings of PMLDC'17, Barcelona, Spain, June 20, 2017,* 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Conflict-free Replicated Data Types [5] were designed to handle concurrent updates and resolve possible conflicts in a predictable and meaningful way. Sequential data-types with non-commutative operations lead to conflicts when these operations occur concurrently, e.g., adding and removing the same element from a set. In this case, CRDTs resolve the conflict by allowing the element to be in the set (*add-wins*) or not be in the set (*remove-wins*). This decision is application-specific, but these conflicting operations may not occur in all applications.

Obtaining more powerful CRDTs by composition of more basic ones has been addressed in various ways, one example being the *causal CRDTs* [1] such as maps, in which the values are themselves CRDTs. In such generic compositions, each individual component must be designed in a way that it may be updated by multiple participants. However, there are usage scenarios where each component is semantically tied to a given participant, which is the sole updater. One example is Doodle[1], an online scheduling tool for meetings, used by a group of people to decide on a date. Each participant selects a set of desirable dates and Doodle aggregates the responses from all participants, finding the dates that will work best for everyone. In this scenario, each participant changes its own value in the scheduling page, adding or removing dates. Using current general CRDT designs (e.g., a map from participants to sets) is unnecessarily complex, as it does not exploit this single-writer scenario, where conflicting operations over each component will never occur.

The *single-writer principle* in concurrent programming, where each register is only written by a single process, is a powerful concept, leading to simplifications or elegant designs, as the classic Bakery algorithm [4]. In this paper we present a new design for CRDT composition, exploiting the single-writer principle, to be used in scenarios where each participant's action is tracked individually by single-writer versioned objects, and conflicting updates will never occur by design.

In Section 2 we start by showing how we can build a join-semilattice given a set of values of any sequential data-type; in Section 3 we use this technique to build a single-writer versioned object that stores the updates of a single participant; Section 4 defines a generic collection of single-writer versioned objects, to be used by several participants, and presents two concrete examples; Section 5 concludes the paper with some final remarks.

2 FROM SEQUENTIAL DATA-TYPES TO JOIN-SEMILATTICES

Given any sequential data-type with values in S_μ, where $\mu \in S_\mu$ is the default value, we form a join-semilattice $S_\mu^\top$ (called S_μ "sinked") by taking an element $\top \notin S_\mu$ and defining $\sqsubseteq$ on $S_\mu^\top = S_\mu \cup \{\top\}$ such that:

$$s \sqsubseteq s' \equiv s = s' \vee s' = \top$$

*Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020" is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).
†The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505, project LightKone.

PMLDC'17, June 20, 2017, Barcelona, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://doodle.com

All the elements in S_μ are unordered, forming an antichain [3]. Two elements in $S_\mu^\top$ are ordered if they are the same element or if one of the elements is $\top$.

This technique allow us to define a join-semillatice given a set of values of any sequential data-type, as depicted in Figure 1.

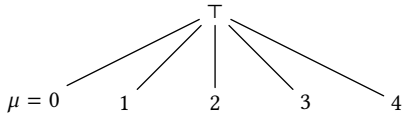

Figure 1: Example of $S_\mu^\top$ with $S_\mu = \{0, 1, 2, 3, 4\}_0$

The binary join operator $\sqcup$ on $S_\mu^\top$ is defined as:

$$s \sqcup s' = \begin{cases} s & \textbf{if } s = s' \\ \top & \textbf{otherwise} \end{cases}$$

3 SINGLE-WRITER VERSIONED OBJECT

State-based CRDTs model data evolution by inflation under a join-semilattice. For instance, we can evolve a grow-only set $\{x, y\}$ by adding a new element $\{z\}$ and get $\{x, y, z\}$. The corresponding semilattice evolves by set union, and is ordered by set inclusion. Since the evolution order is fixed, it is not possible to go back to a smaller state from a larger one. Sets that can grow and shrink, two-phase-sets, end up being modeled, with limitations, by a pair of grow-only semilattices, one for added elements and another for removed ones. A more general way to capture semilattice state that can be safely inflated and deflated is to use it in the context of a lexicographic product.

The lexicographic product $A \boxtimes B$ of a total order A and a join-semilattice B is a join-semilattice [1, 2], with the join of two lexicographic pairs $(a, b), (a', b') \in A \boxtimes B$ defined as:

$$(a, b) \sqcup (a', b') = \begin{cases} (a, b) & \textbf{if } a \sqsupset a' \\ (a', b') & \textbf{if } a \sqsubset a' \\ (a, b \sqcup b') & \textbf{if } a = a' \end{cases}$$

A single-writer versioned object with values in S_μ can be defined as $\mathbb{N} \boxtimes S_\mu^\top$: the first component of the lexicographic pair is a natural (that will serve as a version number) and the second component is the current value of the object. The versioned object can be queried or updated by

$$\mathsf{get}((c, v)) = v$$
$$\mathsf{update}(o, (c, v)) = (c + 1, o(v))$$

which return the value (get), and update the object by incrementing the version number and applying an operation o from the sequential data-type to the current value. Used as single-writer, i.e., with only one participant performing updates (increasing the version number on each update), whenever the version $c = c'$, also the value $v = v'$, and the $\top$ value will never be produced in a merge.

4 COLLECTION OF NAMED SINGLE-WRITER VERSIONED OBJECTS

A collection of named single-writer versioned objects can be defined as a map that stores, for each writer $i \in \mathbb{I}$, a versioned object in S_μ. Each partipicant can update the collection by passing an operation from the sequential data-type to an apply operation, which will update the map entry correponding to that partipipant. Each entry is single-writer by design: for a map m, entry $m(i)$ can only be updated by participant i.

Notation. $m\{k \mapsto v\}$ updates map m, mapping k to v; $m(k)$ retrieves the value in map m associated with key k (if that entry is not present in m, $(0, \mu)$ is returned); the domain of map m is given by $\mathsf{dom}(m)$.

$$\begin{aligned} \mathsf{Collection}\langle S_\mu \rangle &= \mathbb{I} \hookrightarrow (\mathbb{N} \boxtimes S_\mu^\top) \\ \bot &= \{\} \\ \mathsf{apply}_i(o, m) &= m\{i \mapsto \mathsf{update}(o, m(i))\} \\ m \sqcup m' &= \{j \mapsto m(j) \sqcup m'(j) \mid j \in \mathsf{dom}(m) \cup \mathsf{dom}(m')\} \end{aligned}$$

Figure 2: Generic collection of named single-writer versioned objects - participant i.

A generic collection is defined (Figure 2) by taking a sequential data-type S_μ as parameter and by defining the apply_i operation, which takes a data-type operation o from the sequential data-type as parameter. The join uses the generic definition for lexicographic pairs, being data-type independent.

4.1 Collection of Named Sets

This collection stores per participant a set of values. When participant $i \in \mathbb{I}$ adds an element $e \in E$ to the set, we use the generic apply_i defined in Figure 2, passing an anonymous function $\lambda s \cdot s \cup \{e\}$ as a parameter: this function takes the current set value $s \in \mathcal{P}(E)$ and adds the element e. If participant i is writing to the collection for the first time, the default value $\mu = \{\}$ is passed to the anonymous function (i.e., $s = \{\}$). A remove operation is defined in a similar way. This collection also defines two query functions: union, to find elements that are in some set in the collection; and intersection, to find elements that are in all sets in the collection.

$$\begin{aligned} \mathsf{CollectionSet}\langle E \rangle &= \mathsf{Collection}\langle \mathcal{P}(E)_{\{\}} \rangle \\ \mathsf{add}_i(e, m) &= \mathsf{apply}_i(\lambda s \cdot s \cup \{e\}, m) \\ \mathsf{rmv}_i(e, m) &= \mathsf{apply}_i(\lambda s \cdot s \setminus \{e\}, m) \\ \mathsf{union}(m) &= \cup\{\mathsf{get}(m(j)) \mid j \in \mathsf{dom}(m)\} \\ \mathsf{intersection}(m) &= \cap\{\mathsf{get}(m(j)) \mid j \in \mathsf{dom}(m)\} \end{aligned}$$

In Doodle, intersection could be used to find the dates where all the participants are available to meet. More sophisticated queries can be defined, for example, a query that returns a list of elements, sorted by number of occurrences in the collection (which would be useful for an application like Doodle to find the best dates for a meeting, even if some participants cannot attend).

4.2 Collection of Named Flags

This collections keeps a versioned flag per participant, allowing each participant to enable and disable its flag. It also provides two query functions to perform the logical and/or of the flags.

$$\begin{aligned}
\mathsf{CollectionFlag} &= \mathsf{Collection}\langle \mathbb{B}_{\mathsf{False}} \rangle \\
\mathsf{enable}_i(m) &= \mathsf{apply}_i(\lambda s \cdot \mathsf{True}, m) \\
\mathsf{disable}_i(m) &= \mathsf{apply}_i(\lambda s \cdot \mathsf{False}, m) \\
\mathsf{some}(m) &= \vee\{\mathsf{get}(m(j)) \mid j \in \mathsf{dom}(m)\} \\
\mathsf{all}(m) &= \wedge\{\mathsf{get}(m(j)) \mid j \in \mathsf{dom}(m)\}
\end{aligned}$$

The concept of task can be modelled with this collection: a task can be considered completed if all the participants completed the task; or alternatively, if at least one of the participants completed the task.

5 FINAL REMARKS

In this paper we have shown how the single-writer principle can be used to construct a new class of CRDTs in which conflicting updates never occur by design. We have also defined a generic collection of single-writer versioned objects along with two concrete collections.

While this construction is classic in many systems and can be already found in the design of some flavours of CRDT counters, it was hidden inside specific implementations (*e.g.* counters defined under section 7.3. in [1]). By making this pattern an explicit composition construct, and since it can be used to make CRDTs from any sequential data-type, we aim to provide a new tool for safe construction of complex CRDT compositions.

REFERENCES

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2016. Delta State Replicated Data Types. *CoRR* abs/1603.01529 (2016).

[2] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. 2015. Composition of State-based CRDTs. (2015).

[3] Brian A. Davey and Hilary A. Priestley. 2002. *Introduction to Lattices and Order*. Cambridge University Press.

[4] Leslie Lamport. 1974. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM* 17, 8 (1974). https://doi.org/10.1145/361082.361093

[5] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types,. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium*. https://doi.org/10.1007/978-3-642-24550-3_29