Compact Resettable Counters through Causal Stability

Georges Younes∗
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

Conflict-free Data Types (CRDTs) were designed to automatically resolve conflicts in eventually consistent systems. Different CRDTs were designed in both operation-based and state-based flavors such as Counters, Sets, Registers, Maps, etc. In a previous paper [2], Baquero et al. presented the problem with embedded CRDT counters and a solution, covering state-based counters that can be embedded in maps, but needing an ad-hoc extension to the standard counter API. Here, we present a resettable operation-based counter design, with the standard simple API and small state, through a causal-stability-based state compaction.

CCS CONCEPTS

• Theory of computation → Distributed algorithms;

KEYWORDS

CRDTs; Eventual Consistency; Distributed Counting

ACM Reference format:
DOI: http://dx.doi.org/10.1145/3064889.3064892

1 INTRODUCTION

The need for high-responsiveness and high-availability in georeplicated systems pushed researchers and developers to further explore relaxed consistency models such as eventual consistency [1, 6]. As a result of that, many frameworks have been introduced such as Conflict-free Replicated Data Types (CRDTs) [5]. Many of those data types where implemented such as counters, sets, registers, flags, etc.

1 Project "Coral - Sustainable Ocean Exploitation: Tools and Sensors"/NORTE-01-0145-FEDER-000036” 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).
2 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PaPoC’17, Belgrade, Serbia
© 2017 ACM. 978-1-4503-4933-8/17/04 . $15.00
DOI: http://dx.doi.org/10.1145/3064889.3064892

To satisfy user requirements, developers must be able to compose complex data types together. A common strategy [4] is to define a replicated map data structure that maps keys to CRDT instances and others maps as well. For that, maps need to support adding and removing entries, and allow data type-dependent updates on the embedded CRDT instances.

In [2], Baquero et al. explained how previous counter CRDT designs do not allow them to be used as embedded counters inside maps. The main reason is that, contrary to container-like CRDTs like sets, where each element kept is individually tagged with a causal identifier, for counters we cannot afford to individually track each of the possibly millions of increments; therefore, these designs do not allow a reset operation that applies to a given subset of increments. Also, in the same paper, they presented a new state-based embedded counter design as a solution. However, the design has by default an undesired reset-wins semantics, and requires a special fresh operation to protect increments from concurrent resets.

Our aim in this paper, is to revisit the problem and propose an operation-based design of a resettable counter while keeping the standard API; i.e., with no need for special operations, such as fresh above. In Section 2 we introduce the standard pure op-based counter and the issues which prevent it from being resettable. In Section 3, we show a specification of a trivial resettable counter design and point to the meta-data trade-off of such design. In Section 4, we explain how causal stability, that is already a part of the pure op-based framework [3], can be used to remove unnecessary meta-data leading to a more compact design. We conclude, in section 5, with some final remarks.

2 THE STANDARD OP-BASED COUNTER

\[ \Sigma = \mathbb{N} \quad \sigma^0 = 0 \]

\[ \text{prepare}(o, \sigma) = 0 \]

\[ \text{effect}(\text{inc}, t, n) = n + 1 \]

\[ \text{eval}(\text{value}, n) = n \]

Figure 1: Pure G-counter

In the pure op-based model, each operation is tagged at the source with a unique logical timestamp \( t \) and delivered to all replicas by reliable causal broadcast. On delivery it is incorporated in the state by a effect function that receives the operation, source timestamp and local state to mutate. A GCounter (Figure 1) is identical to the purely sequential data type, given its commutative behavior, and exploiting the exactly-once delivery: the state (\( \Sigma \)) is simply an integer (\( \in \mathbb{N} \)), the inc operation increments it; and the eval query returns it.
By not keeping track of each individual increment, such an implementation is very efficient, but not suitable for a reset operation, as we cannot select a subset of the increment operations to discard. For instance, if reset was implemented as setting the integer to zero, this would lead to divergent states when such a reset was concurrent with an inc operation. Alternatively, if the reset was implemented as decrementing by the local counter value, this would lead to an incorrect outcome (decrement twice) if two reset operations were concurrently issued. These anomalies are caused by the non-commutative nature of a reset, when trying to implement it in the simple commutative, sequential data type above.

3 A NAIVE RESETTABLE COUNTER

A trivial, but naive, solution for a resettable counter is the design in Figure 3. The state is a POLog (Partially-Ordered Log), mapping order comparable unique timestamps (∈ T) to corresponding operations (∈ O). Each inc operation is tagged with a timestamp (by the Tagged Reliable Causal Broadcast middleware of the pure op-based model) and added to the POLog. The value query returns the POLog size, which corresponds to the number of inc operations. The reset operation, also tagged with a timestamp, discards all inc operations in the POLog that are in its causal past, matching its natural specification. In Figure 2, we show an example of a run between two replicas. This counter design is unusable in practice, as the number of entries in the POLog grows linearly with the number of increments.

4 COMPACTING THE COUNTER

The pure op-based model envisages the use of two mechanisms for compacting the POLog, causal redundancy and causal stability. These are not needed for the simple GCounter (Figure 1), but we now show that the second will allow obtaining a POLog-based compact and resettable counter, if we change the POLog definition from a set to a multiset.
4.3 Concrete Implementation

Finally, for an actual implementation, we observe that: for grow-only counters, a single kind of operation inc is in the POLog, and therefore, we do not need to store the operation itself; we can store an integer \( n \) that represents the multiplicity of stable operations; all non-stable timestamps have multiplicity one, which means we can store them in a set. This means that a concrete implementation can be as simple as Figure 6. When a timestamp is stable, it is discarded and \( n \) is incremented. A reset, sets \( n \) to 0 and discards timestamps in its causal past. The value query returns \( n \) plus the size of the set of non-stable operations.

\[
\Sigma = \mathbb{N} \times \mathcal{P}(T) \quad \sigma^0 = (0, \{\})
\]

\[
\text{prepare}(o, (n, s)) = o \quad \text{(with } o \text{ either inc or reset)}
\]

\[
\text{effect}(\text{inc}, t, (n, s)) = (n, s \cup \{t\})
\]

\[
\text{effect}(\text{reset}, t, (n, s)) = (0, s \setminus \{t' \in s \mid t' < t\})
\]

\[
\text{stabilize}(t, (n, s)) = (n + 1, s \setminus \{t\})
\]

\[
\text{eval}(\text{value}, (n, s)) = n + |s|
\]

Figure 6: Concrete Resettable Counter Implementation

5 FINAL REMARKS

In the specifications for both counters in Figures 3 and 4, we use what we consider the more intuitive semantics for the reset: a reset operation cancels all operations in its causal past, without affecting concurrent operations. Nevertheless, it is possible to support an alternative reset semantics, in which a reset also cancels concurrent operations, with some simple modifications: the reset is added to the POLog, the value query ignores inc operations with concurrent resets in the POLog; resets are removed once they become stable. To be able to apply causal stability, making a POLog a multiset was an essential ingredient: using the standard POLog definition as a set, means that applying stability would incur loss of increments, as they would be merged into a single element. It might be useful in the future to define the POLog in the pure op-based model as being a multiset (instead of a set) and thus have a more generic framework.

REFERENCES