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:
Georges Younes, Paulo Sérgio Almeida, and Carlos Baquero. 2016. Compact Resettable Counters through Causal Stability. In *Proceedings of PaPoC'17, Belgrade, Serbia, April 23, 2017,* 3 pages.
DOI: http://dx.doi.org/10.1145/3064889.3064892

1 INTRODUCTION

The need for high-responsiveness and high-availability in geo-replicated 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. 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

$$
\begin{aligned}
\Sigma &= \mathbb{N} \qquad \sigma^0 = 0\\
\mathsf{prepare}(o,\sigma) &= o\\
\mathsf{effect}(\mathsf{inc},t,n) &= n+1\\
\mathsf{eval}(\mathsf{value},n) &= n
\end{aligned}
$$

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 (Σ) is simply an integer ($\in \mathbb{N}$); the *inc* operation increments it; and the *eval* query returns it.

*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).
†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.
‡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).

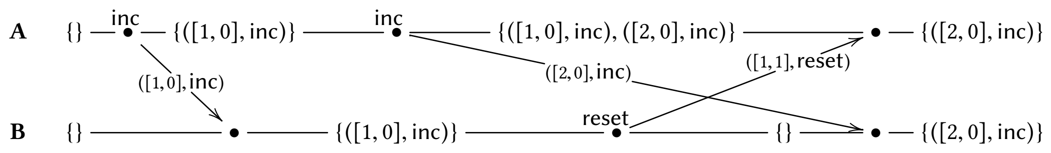

Figure 2: Example of a Naive Resettable Counter

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 ($\in T$) to corresponding operations ($\in O$). 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.

$$
\begin{array}{rcl}
\Sigma = T \hookrightarrow O & & \sigma^0 = \{\} \\
\mathsf{prepare}(o, s) & = & o \quad \text{(with } o \text{ either } \mathsf{inc} \text{ or } \mathsf{reset}) \\
\mathsf{effect}(\mathsf{inc}, t, s) & = & s \cup \{(t, \mathsf{inc})\} \\
\mathsf{effect}(\mathsf{reset}, t, s) & = & s \setminus \{(t', \mathsf{inc}) \in s \mid t' < t\} \\
\mathsf{eval}(\mathsf{value}, s) & = & |s|
\end{array}
$$

Figure 3: Naive Resettable Counter

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.

4.1 Causal Stability

A timestamp t, and corresponding message, is causally stable at node i when all messages subsequently delivered at i will have timestamp $t' > t$. Stability can be locally detected by tracking in each node the last timestamps received from each other node. The pure op-based model uses causal stability, to discard timestamp information of operations in the POLog once they become causally stable.

4.2 Compact POLog-based Resettable Counter

We propose a new specification, in Figure 4, for a compact resettable counter that is based on the naive counter, with two modifications:

- Causal stability is used, through stabilize, to discard timestamps, replacing them by a single bottom value.
- The POLog is a multiset (several instances of the same base element are allowed, i.e., each base element has a given *multiplicity*).

$$
\begin{array}{rcl}
\Sigma = \mathbb{N}^{T \times O} & & \sigma_i^0 = \{\} \\
\mathsf{prepare}(o, s) & = & o \quad \text{(with } o \text{ either } \mathsf{inc} \text{ or } \mathsf{reset}) \\
\mathsf{effect}(\mathsf{inc}, t, s) & = & s \uplus \{(t, \mathsf{inc})\} \\
\mathsf{effect}(\mathsf{reset}, t, s) & = & s \setminus \{(t', \mathsf{inc}) \in s \mid t' < t\} \\
\mathsf{stabilize}(t, s) & = & s[(\bot, \mathsf{inc})/(t, \mathsf{inc})] \\
\mathsf{eval}(\mathsf{value}, s) & = & |s|
\end{array}
$$

Figure 4: Resettable POLog-based Counter using causal stability

We illustrate stabilization with an example in Figure 5: once an operation with a timestamp t_a is stable its timestamp is replaced by $\bot$, resulting in one more operation of the form $(\bot, \mathsf{inc})$. Over time, all but a small number of not-yet-stable increments will have migrated to the multiplicity (denoted in subscript brackets $[\mathbb{N}]$) of the $(\bot, \mathsf{inc})$ element, keeping the size of the base set small.

$$
\begin{array}{rcl}
s_0 & = & \{(\bot, \mathsf{inc})_{[4]}, (t_a, \mathsf{inc})_{[1]}, (t_b, \mathsf{inc})_{[1]}, \ldots, (t_z, \mathsf{inc})_{[1]}\} \\
s_1 & = & \mathsf{stabilize}(t_a, s_0) \\
 & = & \{(\bot, \mathsf{inc})_{[5]}, (t_b, \mathsf{inc})_{[1]}, \ldots, (t_z, \mathsf{inc})_{[1]}\}
\end{array}
$$

Figure 5: stabilize **Example**

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.

$$\begin{aligned}
\Sigma = \mathbb{N} \times \mathcal{P}(T) & \qquad \sigma^0 = (0, \{\}) \\
\mathsf{prepare}(o, (n, s)) &= o \qquad \text{(with } o \text{ either } \mathsf{inc} \text{ or } \mathsf{reset}\text{)} \\
\mathsf{effect}(\mathsf{inc}, t, (n, s)) &= (n, s \cup \{t\}) \\
\mathsf{effect}(\mathsf{reset}, t, (n, s)) &= (0, s \setminus \{t' \in s \mid t' < t\}) \\
\mathsf{stabilize}(t, (n, s)) &= (n + 1, s \setminus \{t\}) \\
\mathsf{eval}(\mathsf{value}, (n, s)) &= n + |s|
\end{aligned}$$

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

[1] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Commun. ACM* 56, 5 (May 2013), 55–63. DOI:https://doi.org/10.1145/2447976.2447992

[2] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. 2016. The Problem with Embedded CRDT Counters and a Solution. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '16)*. ACM, New York, NY, USA, Article 10, 3 pages. DOI:https://doi.org/10.1145/2911151.2911159

[3] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings.* 126–140. DOI:https://doi.org/10.1007/978-3-662-43352-2_11

[4] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. 2014. Riak DT Map: A Composable, Convergent Replicated Dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency (PaPEC '14)*. ACM, New York, NY, USA, Article 1, 1 pages. DOI:https://doi.org/10.1145/2596631.2596633

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Technical Report. 50 pages. http://hal.upmc.fr/inria-00555588/

[6] Werner Vogels. 2008. Eventually Consistent. *Queue* 6, 6 (oct 2008), 14. DOI:https://doi.org/10.1145/1466443.1466448