The problem with embedded CRDT counters and a solution

Carlos Baquero
HASLab, INESC TEC &
Universidade do Minho
Braga, Portugal
cbm@di.uminho.pt

Paulo Sérgio Almeida
HASLab, INESC TEC &
Universidade do Minho
Braga, Portugal
psa@di.uminho.pt

Carl Lerche
Portland, Oregon
me@carllerche.com

ABSTRACT
Conflict-free Replicated Data Types (CRDTs) can simplify the design of deterministic eventual consistency. Considering the several CRDTs that have been deployed in production systems, counters are among the first. Counters are apparently simple, with a straightforward inc/dec/read API, but can require complex implementations and several variants have been specified and coded. Unlike sets and registers, that can be adapted to operate inside maps, current counter approaches exhibit anomalies when embedded in maps. Here, we illustrate the anomaly and propose a solution, based on a new counter model and implementation.

CCS Concepts
• Theory of computation → Distributed algorithms;

Keywords
Distributed Counting, Eventual Consistency, CRDTs.

1. INTRODUCTION
In order to support high-availability and low response latency in geo-replicated data storage systems, developers have successfully explored relaxed consistency models, such as eventual consistency [8, 1], and supporting frameworks, such as conflict-free replicated data types (CRDTs) [6, 7]. This trend towards fast querying and data manipulation at the edge, possibly under partitions, will likely become more prevalent with the growth of IoT deployments.

Complex CRDT deployments require mechanisms for composing together several base data types. A common strategy [5] is to define a replicated map data structure that maps keys to CRDT instances. In the Riak data store, maps can store sets, registers, flags, counters and even, recursively, other maps [3] (as they are CRDTs themselves). Maps need to support the addition and removal of entries (key bindings), and allow data type dependent updates over the stored CRDT instances.

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 the author(s) 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’16, April 18-21 2016, London, United Kingdom
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4296-4/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2911151.2911159

2. EMBEDDED COUNTERS ANOMALY

In order to provide a sound semantics for key removal, CRDT maps behave in a way such that a non-present key (e.g., after removal), when fetched returns the default initial state, i.e., bottom, of the embedded CRDT. Efficient implementations require the storable CRDT data types to provide a special reset operation that brings the instance back to bottom, which need not be stored in the map, while allowing the map meta-data to remember the reset state in an efficient way, without requiring any per-key metadata; i.e., no per-key tombstone. Technically this is done by keeping a global causal context for the whole map, that is common to all the recursively embedded CRDTs. This can be thought of as a observed-reset, in the sense that all operations that have been observed to be applied to the map when the reset is issued, should be equivalently reset on another instance which observes the reset upon a join. Bellow is an example of a correct reset over an add-wins set.

\[
\begin{align*}
\text{m1} & \text{["friend"]. add ("alice");} \\
\text{m2. join (m1);} \\
\text{m2. remove ("friend"); // m2: {}} \\
\text{m1["friend"]. add ("bob");} \\
\text{m1.join (m2); // m1: {"friend" -> {"bob"}}} \\
\end{align*}
\]

After m2.join(m1) both replicas hold a mapping to a AWSet with a single “alice” element. Once replica m2 removes the “friend” entry from the map, the set becomes implicitly empty. Concurrently, replica m1 adds a new element “bob” to the set. Later, after joining the two replicas, we see that the reset, implicitly called on the set when removing the entry, only undoes the add("alice") and not the add("bob").

We now change the example to illustrate the ideal (sound) semantics when counters are embedded. Initially we increment by 2 the “friend” entry and then we concurrently remove it and increment by 3. Ideally, removing the entry should undo the “increment by 2”, which when merged to m1 would leave the “increment by 3” as the only remaining operation, and counter value of 3.

\[
\begin{align*}
\text{m1["friend"]. inc (2);} \\
\text{m2. join (m1);} \\
\text{m2. remove ("friend");} \\
\text{m1["friend"]. inc (3);} \\
\text{m1.join (m2); // m1: {"friend" -> 3}} \\
\end{align*}
\]

The desired counter evolution would be:

\[
\begin{align*}
\text{m1} & 0 \quad \text{inc 2} \quad \text{inc 3} \quad 5 \quad \text{-------------------} \quad 3 \\
\text{m2} & 0 \quad \text{-------------------} \quad \text{inc 2} \quad \text{reset} \quad 0 \\
\end{align*}
\]


However, embedding a simple CRDT counter implementation, and namely the behaviour of Riak DT Counters, exhibit an anomaly, leading to the following outcome:

```
ml["friend"] . inc (2);
m2. join (ml); m2. remove ("friend");  
ml["friend"] . inc (3); 
ml. join (m2); // ml: {"friend" -> 3}
```

Leading to the counter evolution:

```
0 2 → join
0 1 3 ← inc 3
0 5
```

In the counter presented below, if no fresh() calls are made the counter scalability is \(O(r \log o)\), as usual, where \(r\) is the number of replicas that issued operations and \(o\) is the number of operations done. If we consider an arbitrary number of fresh calls \(f \geq r\), the scalability becomes \(O(r \log o)\), when fresh calls are made. Therefore, if meta-data size is a concern then fresh() should be called sparingly. The good news is that, upon resets, meta-data comes back to \(O(r \log o)\), so only the number of observed fresh() calls until a reset is relevant. Moreover, to obtain the ideal semantics, it is enough to perform a fresh() only after shipping the state to other replicas. A possible direction towards obtaining state-scalability could involve a combination of the use of fresh, together with periodic resets through some replica coordination.

```
0 2 ↘ join
0 1 3 ↗ inc 3
```

4. A SEMANTIC TRADE-OFF WITH STATE

Although we fixed the observed-reset anomaly, some applications might have a need for an add wins counter where reset only affects the (observed) past operations and leaves concurrent ones unaffected.

This behaviour is simple to obtain, but at a meta-data cost. The idea is to provide a fresh() operation which has the effect of protecting subsequent updates from being affected by resets concurrent to it. The following example shows that, by calling fresh(), the inc(3) operation is not affected by the concurrent reset, and the outcome is 3, as desired.
counter instance when the corresponding key is removed. Since CRDTs embedded in maps all share a common causal context, removing a map entry effectively removes all the state associated to the counter instance.

Finally, the joint function will: look for entries in common among the two maps \(m, m'\) and join the corresponding values by taking the pairwise maximum of the two \(\text{positive-negative}\) values; and for entries that are present only in one map, only those whose dot was never seen in the other causal context. The joined causal context is simply obtained, as usual for version vectors, by coordinate-wise maximum between maps \(c\) and \(c'\).

### 6. FINAL REMARKS

The new counter design we propose in this paper addresses the problem that prevented counters from being embedded in a map and still provide a \textit{reset} that would correctly remove all past operations, to be used when removing an entry from the map. Even when not embedded, current counters still have that problem if applications require a reset operation.

The solution we propose is efficient, in terms of meta-data cost, under a \textit{remove-wins} semantics. The alternative, \textit{add-wins}, that protects operations from being cancelled by concurrent resets, has considerable meta-data cost. This cost can be reduced by a system design that only creates \textit{fresh} entries after the counter state is sent to other replicas, possibly accepting the trade-off of a low rate of dissemination and less overall recency. Further research is needed, to evaluate this cost and to attempt garbage collection of entries possibly through \textit{reset} together with some coordination.

Reference implementations for the various counters, including the Riak DT counter (CCounter) and the proposed counter (RWCounter), are publicly available in GitHub [2] for C++. RUST implementations are under development.

### 7. ACKNOWLEDGMENTS

We thank the following funding sources: Project 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). Funding from the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement 609551, SyncFree project.

### 8. REFERENCES


