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.

PaPOC'16, April 18-21 2016, London, United Kingdom

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 a 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*.

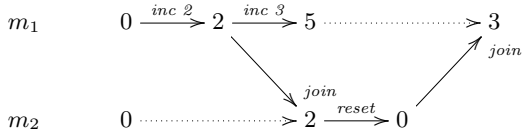
```
m1["friend"].add("alice");
m2.join(m1);
m2.remove("friend"); // m2: {}
m1["friend"].add("bob");
m1.join(m2); // m1: {"friend" -> {"bob"}}
```

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")`.

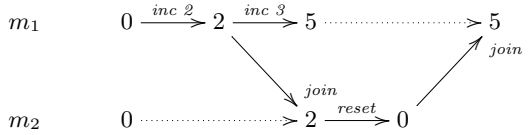
We now change the example to illustrate the ideal (sound) semantics when counters are embedded. Initially we increment by 2 the "friend" entry and then we concurrently remove it and increment by 3. Ideally, removing the entry should undo the "increment by 2", which when merged to `m1`, would leave the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

```
m1["friend"].inc(2);
m2.join(m1); m2.remove("friend");
m1["friend"].inc(3);
m1.join(m2); // m1: {"friend" -> 3}
```

The desired counter evolution would be:



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



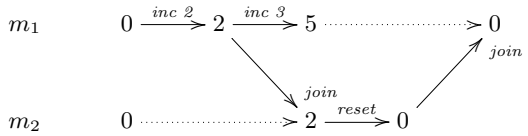
The problem is that the reset does not undo all observed operations (here the initial increment by 2) when merging to other replica, if such replica concurrently updates the counter. This limitation is known in Riak DT and it was an open problem to find an alternative solution [4].

3. A NEW EMBEDDED COUNTER

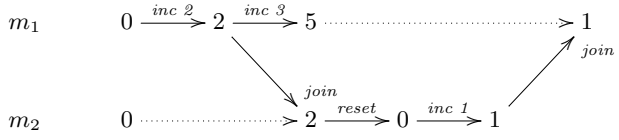
Our approach to address the problem is to try to obtain the desired observed-reset behaviour without compromising the scalability of the underlying meta-data. We found that this can be obtained in a *remove wins counter* design. In this counter all increments (and decrements) that are observed in a replica are correctly reset upon entry removal. Moreover, any concurrent operations are also affected by the reset, thus the remove wins behaviour. Lets illustrate this in our example.

```
m1["friend"].inc(2);
m2.join(m1); m2.remove("friend");
m1["friend"].inc(3);
m1.join(m2); // m1: {"friend" -> 0}
```

Leading to the counter evolution:



Notice that although concurrent operations are affected (both increments and decrements), any operations that causally follow the reset are not affected. Thus if we had done `m2["friend"].inc(1)` after `m2.remove("friend")` the outcome would have been 1:



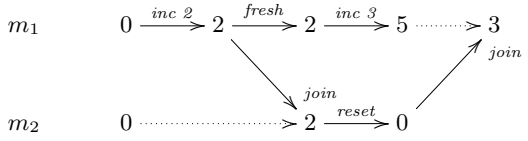
4. A SEMANTIC TRADE-OFF WITH STATE

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.

```
m1["friend"].inc(2);
m2.join(m1); m2.remove("friend");
m1["friend"].fresh(); m1["friend"].inc(3);
m1.join(m2); // m1: {"friend" -> 3}
```

Leading to the counter evolution:



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 $O(f \log o)$, 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.

5. COUNTER SPECIFICATION

Figure 1 shows a mathematical specification of the proposed counter. The state is a pair (m, c) formed by a map m from replica generated ids to a pair of integers, and by a map c (referred to as causal context) from replica ids to integers that compactly encodes causality (essentially a version vector). The first map is what we call a dot store; each key, called a dot, serves as globally unique id, being formed by a pair of replica id and a monotonically increasing counter; the value is a pair of integers that contain the positive and negative partial counts registered under that key.

Initially both the dot store and the causal context are empty, and the reported count value, by query function value_i, returns 0. In order to increment or decrement the counter at replica i an active entry for i must be found, or created, in the dot store m. Mutator functions inc_i and dec_i invoke an auxiliary mutator upd_i with a pair (either $(1,0)$ or $(0,1)$) containing the number of increments and decrements to be applied to the partial count on an active entry for replica i. Thus, function upd_i before updating must check if an active entry is already available for replica i in the dot store m or create a new one by calling fresh_i; if an active entry exists in m its key corresponds to the more recent dot known in the causal context in i, i.e., dot $(i, c(i))$.

The fresh_i mutator will always create a new entry in m by creating a new (globally unique) dot, with replica id i and the next sequence number, $c(i) + 1$, and map it to the $(0,0)$ *positive-negative* partial count. It leaves other entries untouched and, therefore, does not change the counter value. The query function value_i simply sums up all the positive values in the active map and subtracts the corresponding negative ones. By calling reset_i all mappings are removed from the dot store and only the causal context is preserved; thus, the reported value will be again 0. When counter CRDTs are embedded inside maps, a *reset* is called on the

$$
\begin{aligned}
\mathsf{Counter} &= (\mathbb{I} \times \mathbb{N} \hookrightarrow \mathbb{N} \times \mathbb{N}) \times (\mathbb{I} \hookrightarrow \mathbb{N}) \\
\bot &= (\{\}, \{\}) \\
\mathsf{inc}_i(s) &= \mathsf{upd}_i(s, (1, 0)) \\
\mathsf{dec}_i(s) &= \mathsf{upd}_i(s, (0, 1)) \\
\mathsf{upd}_i((m, c), u) &= (m'\{d \mapsto m'(d) + u\}, c') \text{ where } d = (i, c'(i)), \\
&\quad (m', c') = \begin{cases} \mathsf{fresh}_i((m, c)) & \text{if } (i, c(i)) \notin \mathsf{dom}\, m \\ (m, c) & \text{otherwise} \end{cases} \\
\mathsf{fresh}_i((m, c)) &= (m\{(i, c(i) + 1) \mapsto (0, 0)\}, c\{i \mapsto c(i) + 1\}) \\
\mathsf{reset}_i((m, c)) &= (\{\}, c) \\
\mathsf{value}_i((m, c)) &= \sum_{d \in \mathsf{dom}\, m} \mathsf{fst}\, m(d) - \mathsf{snd}\, m(d) \\
(m, c) \sqcup (m', c') &= (\{d \mapsto m(d) \sqcup m'(d) \mid d \in \mathsf{dom}\, m' \cap \mathsf{dom}\, m\} \cup \\
&\quad \{((j, n), v) \in m \mid n > c'(j)\} \cup \{((j, n), v) \in m' \mid n > c(j)\}, \\
&\quad c \sqcup c')
\end{aligned}
$$

Figure 1: Resettable Counter, replica i.

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 join 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 *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 are preserved. This is done by checking if the number n in the dot (j, n) is strictly higher than the highest entry known for j 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 to be embedded in a map and still provide a *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 *remove-wins* semantics. The alternative, *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 *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 *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

[1] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, Mar. 2013.

[2] C. Baquero. Delta-enabled-crdts. URL http://github.com/CBaquero/delta-enabled-crdts, Retrieved 22-dec-2015.

[3] Basho. Riak datatypes. URL http://github.com/basho, Retrieved 22-dec-2015.

[4] R. Brown. Personal Communication, Jan 2016.

[5] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak dt map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 1:1–1:1, New York, NY, USA, 2014. ACM.

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, INRIA, Rocquencourt, France, Jan. 2011.

[7] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, Oct. 2011. Springer-Verlag.

[8] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, Oct. 2008.