Borrowing an Identity for a Distributed Counter

[Work in progress report]

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

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

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

João Leitão
NOVA LINCS, FCT
Universidade NOVA de Lisbon
Lisbon, Portugal

ABSTRACT

Conflict-free Replicated Data Types (CRDTs) are data abstractions (registers, counters, sets, maps, among others) that provide a relaxed consistency model called Eventual Consistency. Current designs for CRDT counters do not scale, having a size linear with the number of both active and retired nodes (i.e., nodes that leave the system permanently after previously manipulating the value of the counter). In this paper we present a new counter design called *Borrow-Counter*, that provides a mechanism for the retirement of transient nodes, keeping the size of the counter linear with the number of active nodes.

CCS CONCEPTS

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

KEYWORDS

Distributed Counting, Eventual Consistency, CRDTs.

ACM Reference format:
Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and João Leitão. 2017. Borrowing an Identity for a Distributed Counter. In *Proceedings of PaPoC'17, Belgrade, Serbia, April 23, 2017,* 3 pages.
DOI: http://dx.doi.org/10.1145/3064889.3064894

*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.
‡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.

PaPoC'17, Belgrade, Serbia

DOI: http://dx.doi.org/10.1145/3064889.3064894

1 INTRODUCTION

Counting events in a large-scale distributed system where messages can be duplicated and dropped is difficult: unreliable networks often lead to over- and under-counting [7]. CRDTs [6] that emulate the behaviour of counters, such as the GCounter, overcome this problem by storing the number of events per node that manipulates the value of the counter.

A GCounter is a *grow-only counter* that only supports the *increment* operation. While the discussion on this paper could be generalized to a counter supporting *increment* and *decrement* operations, we purposely opted to focus on this simpler case, discussing the relevant aspects of the design of CRDT counters. A GCounter can be specified as follows:

$$\begin{aligned} \text{GCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \\ \bot &= \{\} \\ \text{inc}_i(m) &= m\{i \mapsto m(i) + 1\} \\ \text{value}(m) &= \sum_{j \in \text{dom}(m)} m(j) \\ m \sqcup m' &= \{j \mapsto \max(m(j), m'(j)) \mid j \in \text{dom}(m) \cup \text{dom}(m')\} \end{aligned}$$

Each node has an unique identifier $i \in \mathbb{I}$, and increments its entry in the map, which stores the number of increments performed by the node associated with that entry. The value of the counter can be computed by simply summing all entries in the map. This design is immune to replays by calculating the maximum known value for each entry when synchronizing two replicas. Moreover, since the counter state grows monotonically, messages propagating the counter state among nodes can be dropped without compromising the (eventual) correctness of the counter value.

However, correctness comes with a price: scalability. The number of entries in a GCounter is linear with the number of all nodes that ever manipulated the value of the counter, including the retired ones (transient nodes that have permanently left the system or stopped replicating the counter). Almost all counter CRDT designs suffer this problem, one notable exception being *Handoff Counters* [1], which overcome the scalability problem but are considerably more complex than a typical CRDT.

In this paper we propose the *Borrow-Counter*, a more simple, and effective, solution for scaling distributed counters, also more

$$\text{Causal}\langle T : \text{DotStore}\rangle = T \times \text{CausalContext}$$
$$\textbf{where } T : \text{DotFun}\langle\_\rangle$$
$$(m, c) \sqcup (m', c') = (\{k \mapsto m(k) \sqcup m'(k) \mid k \in \text{dom}(m) \cap \text{dom}(m')\} \cup \{(d, v) \in m \mid d \notin c'\} \cup \{(d, v) \in m' \mid d \notin c\}, c \cup c')$$
$$\textbf{where } T : \text{DotMap}\langle\_, \_\rangle$$
$$(m, c) \sqcup (m', c') = (\{k \mapsto v(k) \mid k \in \text{dom}(m) \cup \text{dom}(m') \wedge v(k) \neq \bot\}, c \cup c') \qquad \textbf{where } v(k) = \text{fst}((m(k), c) \sqcup (m'(k), c'))$$

Figure 1: Join-semilattice for Causal CRDTs

in the spirit of typical CRDTs, the design being itself a *Causal CRDT* [3]. Instead of the generic hierarchical design of Handoff Counters, here we propose a simple two layer design, distinguishing only *permanent* nodes (e.g., datacenter nodes) and *transient* nodes (e.g., end-clients). *Borrow-Counter* makes use of the Causal CRDT concept to achieve the transfer of increments from transient to persistent nodes in an elegant way, allowing node retirement without incurring a permanent impact on state growth.

The remainder of this paper is organized as follows: Section 2 discusses fundamental concepts that are essential to the understanding of the contribution of the paper; Section 3 presents the *Borrow-Counter* design, and finally Section 4 concludes this paper with some final remarks.

2 CAUSAL CRDTS

This section introduces the fundamental concepts related with the design of Causal CRDTs, which compose the underlying building blocks for the design of the *Borrow-Counter* presented in the following section.

Causal CRDTs, introduced in [3], generalize the techniques presented in [2, 5], for efficient use of meta-data state. The state of Causal CRDTs is formed by a *dot store* and a *causal context*.

2.1 Causal Context

A *causal context* is a set of *dots* $\mathcal{P}(\mathcal{D})$, where each dot $d \in \mathcal{D}$ represents a unique event by a pair $\mathbb{I} \times \mathbb{N}$ of node-identifier and local sequence number.

$$\text{CausalContext} = \mathcal{P}(\mathcal{D})$$
$$\text{max}_i(c) = \text{max}(\{n \mid (i, n) \in c\} \cup \{0\})$$
$$\text{next}_i(c) = (i, \text{max}_i(c) + 1)$$

Function next_i can be used to generate a new dot.

2.2 Dot Store

A *dot store* contains data type specific information, tagged with event identifiers in the form of dots. In [3], three different dot stores are presented:

- DotSet : DotStore = $\mathcal{P}(\mathcal{D})$, a set of dots
- DotFun$\langle V$: Lattice$\rangle$: DotStore = $\mathcal{D} \hookrightarrow V$, a map from dots to some join-semilattice V
- DotMap$\langle K, V$: DotStore$\rangle$: DotStore = $K \hookrightarrow V$, a map from keys in some set K to a *dot store* V

2.3 Causal CRDTs

The state of a Causal CRDT is a pair, where the first component is a *dot store* and the second component is a *causal context*, as illustrated in Figure 1, where the lattice *join* is also defined for the two kinds of dot store we will use.

3 BORROW-COUNTERS

In this section we present our main contribution, a new design for counter CRDTs called *Borrow-Counter*. We start by discussing the underlying system model we assume when designing this new variant of CRDTs. We then present the *Borrow-Counter* design and finally present a brief discussion on the relationship of our design with that of a recent proposal [4].

3.1 System Model

When designing *Borrow-Counter* we consider systems composed of multiple interconnected nodes that communicate through the exchange of messages and replicate state among them by having a local copy of that state encapsulated in CRDTs. We further consider two different types of nodes: *permanent* and *transient* nodes. Permanent nodes are nodes containing replicas of the system state and whose life is entwined with the total system life. Transient nodes, in contrast, exist in the system only temporarily. While in the system, transient nodes also replicate fractions of the system state. We assume nodes (both permanent and transient) can fail, but eventually recover. When a node fails, it loses all transient state (which includes messages received from other nodes but not yet processed), and becomes unable to receive or transmit messages. However, stable storage, where the state of CRDT replicas is stored, can be recovered when the node itself recovers. We also assume an asynchronous system model, where there is no time bound for either computation or communication steps.

3.2 The Borrow-Counter Design

A *Borrow-Counter* is a Causal CRDT, where the *dot store* is a DotMap from node identifiers $\mathbb{I}$ to another dot store $\mathcal{F} = \text{DotFun}\langle\mathbb{B} \times \mathbb{N}\rangle$.

As discussed previously, in *Borrow-Counter* nodes can either act as permanent or as transient. A node i can increment the counter with mutator inc_i, if its entry in the BCounter map has at least one *active* dot: a dot mapped to a (False, _) pair. Dots are created by mutator create_i: when a node i calls $\text{create}_i(\_, i)$, it creates a dot for itself and becomes permanent; a permanent node i can also create a dot for a transient node j by calling $\text{create}_i(\_, j)$.

A transient node i can retire by invoking mutator retire_i, which makes all dots in its entry inactive. Mutator transfer_i allows permanent nodes to incorporate increments from transient nodes, that were registered in dots that have subsequently been made inactive, removing such inactive entries (i.e., effectively performing garbage collection of those entries from the *Borrow-Counter*). The transition from active to inactive is irreversible, as given by the False < True

lattice used in the pair; once a transient node makes a dot inactive, it surrenders the capability of issuing further increments to that entry, allowing a safe subsequent transfer to the permanent node that created it.

$$
\begin{aligned}
\mathcal{F} &= \mathsf{DotFun}\langle \mathbb{B} \times \mathbb{N} \rangle \\
\bot &= \{\} \\
\mathsf{create}_i(m,d) &= m\{d \mapsto (\mathsf{False}, 0)\} \\
\mathsf{inc}_i(m,d,n) &= m\{d \mapsto (\mathsf{False}, \mathsf{snd}(m(d)) + n)\} \\
\mathsf{freeze}_i(m,D) &= m\{d \mapsto (\mathsf{True}, n)) \mid d \in D \wedge (d,(\_,n)) \in m\} \\
\mathsf{active}(m) &= \{d \mid (d,(\mathsf{False},\_)) \in m\} \\
\mathsf{inactive}(m) &= \mathsf{dom}(m) \setminus \mathsf{active}(m) \\
\mathsf{value}(m) &= \sum_{p \in \mathsf{ran}(m)} \mathsf{snd}(p)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{BCounter} &= \mathsf{Causal}\langle \mathsf{DotMap}\langle \mathbb{I}, \mathcal{F} \rangle \rangle \\
\bot &= (\bot, \bot) \\
\mathsf{create}_i((m,c),j) &= (m\{j \mapsto \mathsf{create}_i(m(j),d)\}, c \cup \{d\}) \\
&\quad \textbf{where } d = \mathsf{next}_i(c) \\
\mathsf{inc}_i((m,c),n) &= (m\{i \mapsto \mathsf{inc}_i(m(i),d,n)\}, c) \\
&\quad \textbf{where } d = \mathsf{random}(\mathsf{active}(m(i))) \\
\mathsf{retire}_i((m,c)) &= (m\{i \mapsto \mathsf{freeze}_i(m(i), \mathsf{active}(m(i)))\}, c) \\
\mathsf{transfer}_i((m,c),j) &= (\bot, s) \sqcup \mathsf{inc}_i((m,c),n) \\
&\quad \textbf{where } s = \mathsf{inactive}(m(j)) \\
&\qquad\quad n = \sum_{d \in s} \mathsf{snd}(m(j)(d)) \\
\mathsf{value}((m,c)) &= \sum_{f \in \mathsf{ran}(m)} \mathsf{value}(f)
\end{aligned}
$$

Figure 2 shows an example with two nodes $a, b \in \mathbb{I}$: node a acts as permanent and b as transient. Node a starts by creating dot a_1 for itself, and later on dot a_2 for node b; node a increments the counter by 9, and b by 8; node b disables its dot and node a transfers node b increments to its entry in the BCounter. (Here we are denoting inactive dots by bold numbers, and representing the *causal context* by its maximal entries, i.e. $\{a_1, a_2, c_1\} \equiv \{a \mapsto 2, c \mapsto 1\}$).

3.3 Relation with other CRDT counter Designs

While the proposed design is new, we note that there is a relevant particularity with Bounded Counters, a CRDT counter introduced in [4]. In Bounded Counters, a design based on escrow is employed, which enables replicas, holding a copy of a counter, to synchronize outside the critical path of user operations in order to exchange *fractions* of the counter among them. This design allows to enforce constraints over the (global) value of the counter while enabling replicas to perform operations locally (provided they locally hold a large enough *fraction* of the counter to execute that operation).

We note however that this design contrary to ours suffers from the same linear growth in state controlled by the total number of nodes that have manipulated the value of the counter, since operations performed by individual replicas have to be kept (explicitly) as part of the counter state, whereas our design allows to garbage collect any state associated with transient nodes that

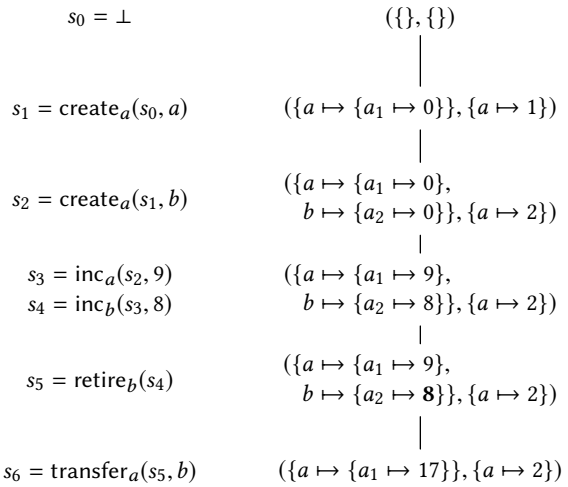

Figure 2: BCounter **example with** $a, b \in \mathbb{I}$

explicitly informed the system that they are no longer replicating the counter.

4 FINAL REMARKS

In this paper we have proposed an alternative design for CRDT counters based on Causal CRDTs called *Borrow-Counter*. While we focused our presentation on increment-only counters, we argue that this design can be easily extended for general purpose counters. The design presented in this paper allows to perform an efficient management of state, by enabling garbage collection of entries associated with nodes that no longer are part of the system. This design can be an interesting starting point for a new class of CRDT designs suited for systems with large number of replicas, particularly systems that enable replication of state at the edge of the network (e.g., directly at the client [8, 9]) and systems supporting partial replication.

REFERENCES

[1] P. S. Almeida and C. Baquero. Scalable Eventually Consistent Counters over Unreliable Networks. *CoRR*, abs/1307.3207, 2013.

[2] P. S. Almeida, C. Baquero, R. Gonçalves, N. M. Preguiça, and V. Fonte. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. In *Distributed Applications and Interoperable Systems, DAIS 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 67–81, 2014.

[3] P. S. Almeida, A. Shoker, and C. Baquero. Delta State Replicated Data Types. *CoRR*, abs/1603.01529, 2016.

[4] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. M. Preguiça. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015, Montreal, QC, Canada, September 28 - October 1, 2015*, pages 31–36, 2015.

[5] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.

[6] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-Free Replicated Data Types,. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011*, 2011.

[7] Sylvain Lebresne. Add a proper retry mechanism for counters in case of failed request. https://issues.apache.org/jira/browse/CASSANDRA-2495, 2011.

[8] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*, Perth, Australia, Apr. 2017.

[9] M. Zawirski, N. M. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, pages 75–87, 2015.