An Oblivious Observed-Reset Embeddable Replicated Counter

Matthew Weidner
Carnegie Mellon University
Pittsburgh, PA, USA
maweidne@andrew.cmu.edu

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

Abstract

Embedding CRDT counters has shown to be a challenging topic, since their introduction in Riak Maps. The desire for obliviousness, where all information about a counter is fully removed upon key removal, faces problems due to the possibility of concurrency between increments and key removals. Previous state-based proposals exhibit undesirable reset-wins semantics, which lead to losing increments, unsatisfactorily solved through manual generation management in the API. Previous operation-based approaches depend on causal stability, being prone to unbounded counter growth under network partitions. We introduce a novel embeddable operation-based CRDT counter which achieves both desirable observed-reset semantics and obliviousness upon resets. Moreover, it achieves this while merely requiring FIFO delivery, allowing a tradeoff between causal consistency and faster information propagation, being more robust under network partitions.

CCS Concepts: • **Theory of computation → Distributed algorithms**.

Keywords: CRDTs, Eventual Consistency, Distributed Counting

ACM Reference Format:
Matthew Weidner and Paulo Sérgio Almeida. 2022. An Oblivious Observed-Reset Embeddable Replicated Counter. In *Principles and Practice of Consistency for Distributed Data (PaPoC '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3517209.3524084

1 Introduction

Conflict-free Replicated Data Types (CRDTs) [6, 7] are highly available replicated data types used in distributed key-value stores [4] and collaborative web apps [5]. A basic counter CRDT is a simple CRDT with a query function *value*, to obtain the counter value, and an update operation *increment*, which adds 1 to the counter.

Frequently many counters may be needed, with patterns such as dynamic counter creation and removal. Therefore, it is desirable to be able to embed counters (or other CRDTs) as values in replicated maps. One of the first popular usages was in the Riak data store [4], which supported a map CRDT, that could store sets, registers, flags, counters and even, recursively, other maps.

Embedding CRDTs in maps posed the problem of what to do for a key removal. Simply removing the map entry would (for state-based CRDTs, such as the Riak map) cause the removed value to resurface when merging with other replicas using the semilattice join. A tentative solution was to use tombstones, but it is not satisfactory as it effectively makes the map state grow with the number of keys ever used, even if only a small number is currently present.

To allow maps with embedded CRDTs to apply a key removal in a uniform way motivated the introduction of a *reset* operation in CRDTs. A removal can simply invoke the reset for the embedded value, which desirably would lead to the entry being removed, if it becomes equivalent to the default value (e.g., a zero counter or an empty set) which is assumed for non-mapped keys, and does not need to be explicitly stored.

A reset at a given replica should cancel the effect of all operations that have been observed (applied) at that replica when the reset is issued. As such, this can be said to have *observed-reset* semantics. When the reset is applied at other replicas, it should cancel the same operations at those replicas. An observed-reset has in general desirable semantics. For counters, it has the nice property of not cancelling concurrent increments, allowing a usage pattern in which a given node periodically samples-and-resets the counter, to obtain a series of increments over time.

Delta CRDTs [1] defined a framework and taxonomy in which *Causal CRDTs* can be embedded in maps in a way that allows a key removal to fully remove the map entry, only requiring a top-level global *causal context*, shared by all entries, instead of an ever growing number of map entries serving as tombstones. The causal context is basically a version vector, which compactly describes the causal history of observed operations (that have been applied at the replica). Several

PaPoC '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9256-3/22/04.
https://doi.org/10.1145/3517209.3524084

CRDTs such as flags, registers, sets, or maps themselves can be described as causal CRDTs.

For state-based CRDTs, the problem with embbeding counters is that, contrary to most CRDTs, that can afford to assign a *dot* (unique event identifier as a pair replica-id and sequence number) per operation, for counters we can have millions of increments. Counter CRDTs cannot afford to tag each individual increment and as such are not causal CRDTs. One approach [2] to embedding makes counters be causal CRDTs by grouping increments in generations, but abandons the desirable observed-reset semantics, and allows a reset to cancel concurrent increments.

For operation-based CRDTs, a solution to embedding counters was devised [8], but it relies on causal stability [3]. As it keeps individual not-yet-causally-stable increments, it can lead to unacceptably large state growth, given a high increment rate, under network partitions.

In this paper, we present a novel embeddable counter CRDT that simultaneously satisfies two desirable properties:

1. observed-reset semantics: a reset cancels all increments that have been observed (applied) at the replica where the reset is issued;
2. oblivious resets: a fully reset counter allows key removal from the map where it is embedded, i.e., allows discarding all per-counter metadata.

The state size is bounded by the number of replicas, independent of the number of increments. More specifically, each counter's state size is proportional to the number of replicas that have sent outstanding increments—either not yet reset, or reset but not yet received. The only other storage space required is a single global version vector, containing one entry per replica, that can be shared by all counters in a map (or recursively, by all counters embedded in all maps … embedded in a map).

The novel design is an op-based CRDT which draws ingredients from both op-based and state-based designs:

- Although op-based, the CRDT state is based on state-based designs, notably causal CRDTs, with per-node entries both in per-key metadata and the global causal context.
- It draws inspiration from grouping increments in generations, but contrary to the previous design [2], not only it does not require any manual generation management in the API, but it also ensures observed-reset semantics; one key difference from the previous design is that here the global version vector has a finer accounting granularity, while in that design it had generation granularity.
- It exploits the exactly-once delivery guarantee (present in the causal delivery mechanisms used in op-based designs) but it does not need causal delivery, requiring only the much simpler and cheaper FIFO order.

Not requiring causal delivery has the advantage of providing faster increment and reset propagation, and makes the algorithm more robust to network partitions. While for complex data types causal consistency is the norm of what to aim for in highly available systems, for counting it may not always be needed, and it may be beneficial to be able to tradeoff between freshness and causal consistency.

Depending only on FIFO order posed subtle problems, that were solved in the novel design, such as a reset arriving at a replica where some of its cancelled increments have not yet been delivered, while allowing oblivousness when the pending increments arrive later, if possible.

2 Observed-Reset Counters

For state-based CRDTs or op-based CRDTs when causal delivery is used, defining the acceptable semantics for what is an observed-reset CRDT is simple: a reset cancels the increments from the causal past.

However, for op-based CRDTs aiming for more general designs with less messaging requirements, namely not requiring causal delivery, gaining freshness but forgoing causal consistency, defining acceptable semantics requires some care.

In general, knowledge about both increments and resets issued can travel independently, carried by either resets or increments that are delivered. Given the incremental nature of op-based designs, full transitive knowledge propagation is unrealistic, specially when increments are delivered, but for resets (assuming that they are a more rarely issued operation) more knowledge propagation is realistic.

Regardless of slower or faster propagation, any given algorithm step will have to respect constraints, according to what the algorithm did in the past, that led to the current replica state. To define acceptable semantics, first we define two sets, as a function of the set of operations S applied at the replica (i.e., its state):

- $\mathsf{increments}(S)$: the increments propagated by S;
- $\mathsf{cancelled}(S)$: the cancellations propagated by S.

The increments that are being observed, i.e., that reached the replica and were not yet cancelled, is the observed set:

$$\mathsf{observed}(S) = \mathsf{increments}(S) \setminus \mathsf{cancelled}(S),$$

and the counter value is just the size of this set:

$$\mathsf{value}(S) = |\mathsf{observed}(S)|\,.$$

When a reset is issued it must cancel this set, no less and no more. However, both increments and resets, when delivered (applied) to a replica, may propagate information about both $\mathsf{increments}(S)$ and $\mathsf{cancelled}(S)$ for state S where the operation is issued.

We define $\mathsf{cancels}(O)$ as the set of cancellations that are propagated by an operation O, to be joined to the cancelled

increments when delivered to replica state S:

$$\mathsf{cancelled}(S \cup \{O\}) = \mathsf{cancelled}(S) \cup \mathsf{cancels}(O).$$

The possible boundaries, given an operation issued at replica state S, for an increment operation I:

$$\emptyset \subseteq \mathsf{cancels}(I) \subseteq \mathsf{cancelled}(S),$$

i.e., an increment does not need to propagate cancellations at all, but may propagate all known cancellations. For a reset operation R:

$$\mathsf{observed}(S) \subseteq \mathsf{cancels}(R) \subseteq \mathsf{increments}(S),$$

i.e., a reset needs to cancel at least the observed set, resulting in a zero counter when it is applied at the origin, but may propagate more information to other replicas, limited to the locally known increments. This prevents resetting not yet know increments, avoiding increment loss.

Similarly, we define $\mathsf{adds}(O)$ as the set of increments that are propagated by an operation O, to be joined to $\mathsf{increments}(S)$ when delivered to replica state S:

$$\mathsf{increments}(S \cup \{O\}) = \mathsf{increments}(S) \cup \mathsf{adds}(O).$$

The possible boundaries, given an operation issued at replica state S, for an increment operation I:

$$\{I\} \subseteq \mathsf{adds}(I) \subseteq \mathsf{increments}(S) \cup \{I\},$$

i.e., an increment needs to propagate itself, but may propagate all known increments. For a reset operation R:

$$\emptyset \subseteq \mathsf{adds}(R) \subseteq \mathsf{increments}(S),$$

i.e., a reset may propagate from nothing up to the full set of known increments.

As a further constraint, a reset does not need to propagate increments at all, but if it chooses to do so, given that all those increments were already or are being cancelled at the origin, it should propagate at least those also as cancellations, i.e., the constraint:

$$\mathsf{adds}(R) \subseteq \mathsf{cancels}(R).$$

3 Design

Our counter is an operation-based CRDT [7]. Each operation has a *generator* and an *effector*. The generator is called by the acting replica and returns a message to be broadcast to the other replicas. Each replica, including the sender, applies the operation by inputting this message to the corresponding effector.

We assume that operations are applied exactly once. We also assume FIFO order: for each replica j, all other replicas apply operations from j in the order they were sent, across all counter instances sharing the same global version vector. We do not require causal delivery.

Algorithm 1 gives the complete counter CRDT. We use i to denote the local replica, so in the generators, i is the sender, while in the effectors, i is the receiver and j is the sender. Each CRDT and replica has its own map M, while the map C is shared by all CRDTs on a given replica.

```
1  types:
2    𝕀, set of replica identifiers
3  global replica state:
4    C : 𝕀 ⇀ ℕ
5    // assuming C[j] = 0 for unmapped keys
6  per-instance CRDT state:
7    M : 𝕀 ⇀ (p : ℕ, n : ℕ, c : ℕ)
8    // assuming M[j] = (0, 0, 0) for unmapped keys
9  query value() : ℕ
10   return ∑{p − n | (j, (p, n, c)) ∈ M}
11 update inc()
12   generator ()
13     if i ∉ dom(M) then
14       return (inc, i, C[i] + 1, true)
15     else
16       return (inc, i, M[i].p + 1, false)
17   effector (inc, j, p, start)
18     let c = C[j] + 1
19     if start ∨ j ∉ dom(M) then
20       M[j] ← max(M[j], (p, p − 1, c))
21       // max is taken entry-wise
22     else
23       M[j] ← max(M[j], (p, 0, c))
24     if M[j].p = M[j].n ∧ M[j].c = c then
25       M. remove(j)
26     C[j] ← c
27 update reset()
28   generator ()
29     return (reset, {(j, p, c) | (j, (p, n, c)) ∈ M})
30   effector (reset, R)
31     for (j, p, c) in R do
32       if j ∉ dom(M) then
33         if c > C[j] then
34           M[j] ← (p, p, c)
35       else
36         M[j] ← max(M[j], (p, p, c))
37         if M[j].p = M[j].n ∧ M[j].c ≤ C[j] then
38           M. remove(j)
```

Algorithm 1: Counter algorithm for replica i.

Algorithm 1 is a modification of the delta state-based PN-counter CRDT [1], as we now explain in four steps.

1. Start with Delta State-Based PN-Counter. The state of a delta state-based PN-counter CRDT is a partial map $M : \mathbb{I} \rightharpoonup (p, n)$, where $\mathbb{I}$ is the set of replica identifiers and the values p, n are nonnegative integers. Its value is

$\sum\{p - n \mid (j, (p, n)) \in M\}$. To increment the counter, replica i broadcasts $M[i].p + 1$; each replica then takes the max of this with its own $M[i].p$ entry. Ordinarily, the n entries are used in an analogous way for decrements, but we can instead use them for resets: to reset the counter, replica i broadcasts $\{(j, p) \mid (j, (p, n)) \in M\}$; then for each (j, p), each replica sets $M[j].n \leftarrow \max(M[j].n, p)$.

This modified PN-counter implements the observed-reset semantics and has a small state. However, once such a counter is used, its state is always nontrivial: even when reset, it stores the largest p value sent out by each replica. Thus it is not oblivious.

2. Delete Fully-Reset Entries. To fix the issue, we can delete any entries $M[j]$ such that $M[j].p = M[j].n$. This does not affect the counter's value, and it means that fully-reset counters have empty M. Hence fully-reset counters in a map can be stored without any per-key metadata. However, the resulting design is broken: it has the wrong semantics, and it is not even a CRDT.

3. Fix Semantic Problems. The first problem comes when replica j sends an increment p concurrently to a reset. A replica that receives the reset first will delete $M[j]$. Then after receiving the increment, it will set $M[j] = (p, 0)$. But the correct entry—what it would have been in the original PN-counter—is $M[j] = (p, n)$, where n is whatever $M[j].n$ was before deletion.

To fix this, we observe: assuming FIFO order, *the old value of n must be $p - 1$*. Indeed, the new increment p is the first increment sent by j that is not cancelled by the reset. Thus the reset must have cancelled all of j's prior increments, which went up to $p - 1$. By this observation, when a replica receives the increment p, it should set $M[j] = (p, p - 1)$ instead of $M[j] = (p, 0)$ (line 20).

The second problem comes when replica j wants to send an increment just after receiving a reset. It is supposed to send $p + 1$, where p is whatever $M[j].p$ was before the reset, but $M[j]$ may have been deleted. However, the PN-counter still works if j sends a value $p' \geq p + 1$, so long as they also instruct recipients to set $M[j].n = p' - 1$ (using a *start* flag to signal whether a new *generation* of consecutive p values is starting). One such value is $p' = C[j] + 1$, where $C[j]$ is a *global* value, shared by all CRDT instances, that counts the total number of increments issued by replica j.

4. Extend to FIFO Order. Finally, we need to handle non-causally-ordered delivery. Specifically, the algorithm breaks if we receive a reset, delete $M[j]$ because of it, then receive a causally prior increment sent by j. We fix this by waiting to delete $M[j]$ until after we have received all such increments. Algorithm 1 tracks that condition using a third entry, c, in $M[j] = (p, n, c)$. That entry tells us to wait to delete $M[j]$ until we have received the first c increments from j (across all CRDT instances). In order to track this condition, each replica i stores not just its own C entry $C[i]$, but also an entry $C[j]$ for each replica j, counting the total number of increments received from replica j.

4 Correctness

Theorem 4.1. *Algorithm 1 is an observed-reset counter. That is, there are functions* cancels(O) *and* adds(O)*, satisfying the constraints in Section 2, such that in any execution of Algorithm 1 with FIFO order, a replica that has applied the set of operations S has value* value(S).

In particular, the algorithm exhibits strong eventual consistency: two replicas that have applied the same sets of operations have the same value.

We prove Theorem 4.1 indirectly using Algorithm 2. Algorithm 2 is like Algorithm 1, except it does not delete fully-reset entries. This makes it easier to analyze. We will prove that the two algorithms are equivalent, then prove that Algorithm 2 is an observed-reset counter, from which Theorem 4.1 follows.

The specific changes in Algorithm 2 relative to Algorithm 1 are:

- C and M are renamed to C' and M', for clarity in the discussion below.
- The lines $M.\text{remove}(j)$ and their **if** statements are removed (lines 24–25 and 37–38).
- The condition **if** $i \notin \text{dom}(M)$ (line 13) is replaced with **if** $i \notin \text{dom}(M') \vee M'[i].\text{p} = M'[i].\text{n}$.
- The condition **if** $start \vee j \notin \text{dom}(M)$ on line 19 is simplified to **if** $start$.
- The reset generator (line 29) is changed to

  return $(\text{reset}, \{(j, p, c) \mid (j, (p, n, c)) \in M' \wedge \neg(p = n \wedge c \leq C'[j])\})$
- In the reset effector, only the second case is present (line 36).

Proposition 4.2. In any execution, a replica using Algorithm 2 will have the same behavior as the same replica using Algorithm 1, i.e., it will output the same messages and return the same query values.

Proof sketch. We claim that in any execution, using both algorithms side-by-side identically, the original state (C, M) and the alternate state (C', M') always satisfy the following correspondence:

> $C' = C$, and M is the same as M' except that M omits any entries j such that $M'[j].p = M'[j].n$ and $M'[j].c \leq C'[j]$.

Using this correspondence claim, one can check that in every case, the two algorithms output the same messages and return the same query values.

To prove the correspondence claim, we need to show that if it is true before effecting a message m, then it is also true afterwards.

types:
  $\mathbb{I}$, set of replica identifiers
global replica state:
  $C' : \mathbb{I} \rightharpoonup \mathbb{N}$
  // assuming $C'[j] = 0$ for unmapped keys
per-instance CRDT state:
  $M' : \mathbb{I} \rightharpoonup (\mathsf{p} : \mathbb{N}, \mathsf{n} : \mathbb{N}, \mathsf{c} : \mathbb{N})$
  // assuming $M'[j] = (0, 0, 0)$ for unmapped keys
query value() : $\mathbb{N}$
  return $\sum\{p - n \mid (j, (p, n, c)) \in M'\}$
update inc()
  generator ()
    if $i \notin \mathsf{dom}(M') \vee M'[i].\mathsf{p} = M'[i].\mathsf{n}$ **then**
      return $(\mathsf{inc}, i, C'[i] + 1, \mathsf{true})$
    else
      return $(\mathsf{inc}, i, M'[i].\mathsf{p} + 1, \mathsf{false})$
  effector $(\mathsf{inc}, j, p, start)$
    let $c = C'[j] + 1$
    if $start$ **then**
      $M'[j] \leftarrow \mathsf{max}(M'[j], (p, p - 1, c))$
      // max is taken entry-wise
    else
      $M'[j] \leftarrow \mathsf{max}(M'[j], (p, 0, c))$
    $C'[j] \leftarrow c$
update reset()
  generator ()
    return $(\mathsf{reset}, \{(j, p, c) \mid (j, (p, n, c)) \in M' \wedge \neg(p = n \wedge c \leq C'[j])\})$
  effector (reset, R)
    for (j, p, c) **in** R **do**
      $M'[j] \leftarrow \mathsf{max}(M'[j], (p, p, c))$

Algorithm 2: Alternate algorithm for replica i, used in the proof of Theorem 4.1.

This is a case analysis. The only interesting case is when $m = (\mathsf{inc}, j, p, \mathsf{false})$ is an increment that does not start a new generation, but Algorithm 1 does not have an entry $M[j]$. That can only happen if we previously received a concurrent reset that caused us to delete entry $M[j]$. Thus we are in the setting of Step 3 in Section 3. From the argument there, it follows that Algorithm 2 ends the effector with $M'[j].n = p - 1$, just like in Algorithm 1. □

Proof sketch for Theorem 4.1. By Proposition 4.2, it suffices to prove that Algorithm 2 is an observed-reset counter.

It is easy to check that after applying the set of operations S, the state of Algorithm 2 satisfies:

- For each j, $M'[j].p$ is the maximum of:
  - the value p in the most recent increment message $(\mathsf{inc}, j, p, start) \in S$, and
  - the maximum across all values p such that (j, p, c) appears in a reset message in S.

  (If there are no such messages, then $M'[j]$ is not present.)
- For each j, $M'[j].n$ is the maximum of:
  - the value $p - 1$ corresponding to the most recent increment message of the form $(\mathsf{inc}, j, p, \mathsf{true}) \in S$, and
  - the maximum across all values p such that (j, p, c) appears in a reset message in S.

  (If there are no such messages, then $M'[j]$ is not present.)

We can translate this point-by-point into definitions for $\mathsf{adds}(O)$ and $\mathsf{cancels}(O)$:

- adds:
  - For an increment operation I, $\mathsf{adds}(I) = \{I\}$.
  - For a reset operation R, $\mathsf{adds}(R)$ is the set of increments $I = (\mathsf{inc}, j, p', start)$ such that there exists $(j, p, c) \in R$ with $p' \leq p$.
- cancels:
  - For an increment operation $I = (\mathsf{inc}, j, p, start)$, if $start = \mathsf{true}$, then $\mathsf{cancels}(I)$ is the set of prior increments sent by replica j; else $\mathsf{cancels}(I) = \emptyset$.
  - For a reset operation R, $\mathsf{cancels}(R)$ is the same as $\mathsf{adds}(R)$.

With these definitions, the query value

$$\sum\{p - n \mid (j, (p, n, c)) \in M'\}$$

equals $\mathsf{value}(S)$. Finally, one can check that $\mathsf{adds}(O)$ and $\mathsf{cancels}(O)$ satisfy the constraints in Section 2. □

5 Efficiency

5.1 State Size

The per-instance state for a counter using Algorithm 1 is just the map M. Thus the per-instance state size is proportional to the number of entries in M. There is also the global version vector C, shared by all CRDT instances, which has one entry per replica.

The number of entries in M is always upper bounded by the number of replicas. However, we can get a more precise guarantee:

Proposition 5.1. In any execution of Algorithm 1 with FIFO order, a replica that has applied the set of operations S has an entry $M[j]$ for replica j if and only if either:

(a) $\mathsf{observed}(S)$ contains an increment sent by j, i.e., j has sent a not-yet-reset increment; or
(b) there is an increment I sent by j such that

$$I \in \mathsf{cancelled}(S) \setminus S,$$

i.e., the increment has been cancelled but not yet received, due to a reset applied out-of-order.

Proof sketch. By the correspondence in the proof of Proposition 4.2, $M[j]$ exists if and only if in the corresponding execution of Algorithm 2, $M'[j]$ exists and either $M'[j].p \neq M'[j].n$ or $M'[j].c > C[j]$. The characterization of M' in the proof of Theorem 4.1 shows that $M'[j].p \neq M'[j].n$ if and only if case (a) holds. Meanwhile, one can check that $M'[j].c > C[j]$ if and only if case (b) holds. □

In particular, if a replica has received all increments cancelled by its received resets, then its state size is proportional to the number of replicas that contribute to the current value. In this case, if the counter is fully reset (value 0), then its state is trivial and can be discarded. Thus the counter is oblivious.

5.2 Message Size

Increment messages have constant size (ignoring logarithmic factors, i.e., using fixed sized integers as usually done in practice). This holds even when counting metadata for enforcing exactly-once delivery and the FIFO order, namely, the sender's id i and a sequence number.

Reset messages have the same asymptotic size as the sender's state. In particular, if the sender had received all increments cancelled by its received resets, then the reset message's size is proportional to the number of replicas whose increments are being reset. In any case, the size is upper bounded by the number of replicas.

6 Conclusions and Future Work

Due to being unrealistic to track individual increments in state-based counter CRDTs, and given their gossip-based state propagation, achieving both goals of observed-reset semantics and obliviousness remains an unsolved problem for state-based embeddable counter CRDTs.

By combining state-designs from state-based CRDTs and the generator-effector execution model, we have designed an operation-based embeddable CRDT counter which achieves both goals of observed-reset and obliviousness. Moreover, not only it does not rely on causal stability, but it also does not depend on causal delivery, requiring only cheap FIFO messaging, with the added benefit of allowing faster information propagation under network problems.

One direction for future work could be to relax the FIFO order requirement to just exactly-once delivery, but we suspect that FIFO is the essential ingredient, and given the obliviousness goal we cannot "implement FIFO" in the CRDT. A conjecture, to be proved (or refuted) is then that FIFO is the most relaxed messaging order that can be used to achieve both goals.

Acknowledgments

Matthew Weidner is supported by an NDSEG Fellowship sponsored by the US Office of Naval Research. Paulo Sérgio Almeida is supported by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173. https://doi.org/10.1016/j.jpdc.2017.08.003

[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* (London, United Kingdom) *(PaPoC '16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 3 pages. https://doi.org/10.1145/2911151.2911159

[3] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Proceedings of the 14th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 8460*. Springer-Verlag, Berlin, Heidelberg, 126–140. https://doi.org/10.1007/978-3-662-43352-2_11

[4] Basho. 2015. Riak datatypes. http://github.com/basho.

[5] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 154–178. https://doi.org/10.1145/3359591.3359737

[6] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. 2018. *Conflict-Free Replicated Data Types CRDTs*. Springer International Publishing, Cham, 1–10. https://doi.org/10.1007/978-3-319-63962-8_185-1

[7] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[8] Georges Younes, Paulo Sérgio Almeida, and Carlos Baquero. 2017. Compact Resettable Counters Through Causal Stability. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data* (Belgrade, Serbia) *(PaPoC '17)*. ACM, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/3064889.3064892