Memory-Efficient Embedded Counter CRDTs

Matthew Weidner[1] and Paulo Sérgio Almeida[2]

[1] *Carnegie Mellon University, maweidne@andrew.cmu.edu*
[2] *Universidade do Minho, psa@di.uminho.pt*

December 11, 2021

1 Introduction

Conflict-free Replicated Data Types (CRDTs) [6, 5] are highly available replicated data types used in distributed key-value stores [3] and collaborative web apps [4]. A counter CRDT is a simple CRDT with operations *increment* and *reset*, which add 1 and reset the value to 0, respectively.

Counters and other CRDTs are often embedded as values in CRDT maps, such as in the Riak data store [3]. When a key corresponding to a counter is removed, that counter is reset. Ideally, this reset operation has two properties:

1. It is an *observed-reset* operation, meaning that it undoes the effect of all prior increment operations, but concurrent increment operations are unaffected. This ensures that no increments are lost. Formally, the counter's value after a set of operations S is

$$\left|\{m = \texttt{inc} \in S \mid \nexists m' = \texttt{reset} \in S \text{ s.t. } m' \text{ sender-resets } m\}\right|, \qquad (*)$$

   where "m' sender-resets m" means that the sender of m' sent it after receiving m.

2. When a key's counter is fully reset (value 0), the map can remember it without storing any per-key metadata.

A simple design for such a counter stores every increment operation that has not yet been reset. When a reset operation is received, all affected increment operations are identified and removed. Unfortunately, this is inefficient: the state size grows in proportion to the number of not-yet-reset increments, which is unacceptable with possibly millions of increments.

Previous works addressed this inefficiency by abandoning the observed-reset semantics [3, 2] or by relying on causal stability [7]. The former yields undesirable semantics for some applications, while the latter only improves efficiency when all replicas send frequent messages—rarely the case for collaborative web apps.

In this paper, we present a novel counter CRDT that satisfies both properties above and whose state size is bounded by the number of replicas, independent of the number of increment operations. More specifically, each counter's state size is proportional to the number of replicas that have sent not-yet-reset increment operations. 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 even in an application).

2 Design

Our counter is an operation-based CRDT [6]. 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, inputs this message to the corresponding effector.

We assume that messages are delivered exactly once. We also assume FIFO delivery: for each replica j, all other replicas receive messages from j in the order they were sent, across all counter instances sharing the same version vector. We do not require causally-ordered delivery, although the semantics are clearer when that is the case—in particular, they then agree with the observed-reset semantics (Proposition 3.2).

Algorithm 1 gives the complete counter CRDT. It 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 map $M : I \to (p, n)$, where I is the set of replica identifiers and the values p, n are nonnegative integers. Its value is $\sum\{p - n \mid (p, n) \in M\}$. 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. That violates requirement 2 above.

2. Delete Fully-Reset Entries To fix the issue, we can delete any entries $M[i]$ such that $M[i].p = M[i].n$. 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 i sends an increment message p concurrently to a reset message. A replica that receives the reset message first will delete $M[i]$. Then after receiving the increment message, it will set $M[i] = (p, 0)$. But the correct entry—what it would have been in the original PN-counter—is $M[i] = (p, n)$, where n is whatever $M[i].n$ was before deletion.

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

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

Algorithm 1 Counter algorithm.

```
I set of replica identifiers
algorithm for replica i

global replica state (VV, total incs, shared by all CRDT instances):
  C : I -> int
  // assuming C[j] = 0 for unmapped keys

per instance CRDT state (payload):
  M : I -> (p: int, n: int, c: int)
  // assuming M[j] = (0, 0, 0) for unmapped keys

query value() : int
  return sum { p - n | (p, n, c) in M }

update inc

  generator ()
    if i not in dom(M)
      return (inc, i, C[i] + 1, true)
    else
      return (inc, i, M[i].p + 1, false)

  effector (inc, j, p, start)
    let c = C[j] + 1
    if start or j not in dom(M)
      M[j] <- max(M[j], (p, p - 1, c)) // max is taken entry-wise
    else
      M[j] <- max(M[j], (p, 0, c))
    if M[j].p = M[j].n and M[j].c = c
      M.remove(j)
    C[j] <- c

update reset

  generator ()
    return (reset, {(j, p, c) | (j, (p, n, c)) in M })

  effector (reset, R)
    for (j, p, c) in R
      if j not in dom(M)
        if c > C[j]
          M[j] <- (p, p, c)
      else
        M[j] <- max(M[j], (p, p, c))
        if M[j].p = M[j].n and M[j].c <= C[j]
          M.remove(j)
```

4. Extend to FIFO Delivery Finally, we need to handle non-causally-ordered delivery. Specifically, the algorithm breaks if we receive a reset message, delete $M[i]$ because of it, then receive a causally prior increment message sent by i. We fix this by waiting to delete $M[i]$ until after we have received all such increment messages. Algorithm 1 tracks that condition using a third entry, c, in $M[i] = (p, n, c)$. That entry tells us to wait to delete $M[i]$ until we have received the first c messages from i.

3 Correctness

Let $<$ denote the causal order. As in $(*)$, we say that a reset message m' *sender-resets* an increment message m if the sender of m' sent it after receiving m.[1] Let us also say that a reset message m' *directly resets* an increment message m if it sender-resets m and it is not causally greater than any other message that sender-resets m. In other words, the messages that directly reset m are the minimal elements (under $<$) of the set of messages that sender-reset m.

With these definitions, our correctness claim is as follows (proved later in this section):

Theorem 3.1. *In an execution of Algorithm 1, suppose a replica has received a set of messages S. Let I be the set of increment messages in S. Define:*

- *$D_{\max}$ is the set of increment messages m such that the causal history of S contains a message that sender-resets m, i.e., there is a message m' that sender-resets m and a message $m'' \in S$ such that $m' \leq m''$.*
- *$D_{\min}$ is the set of increment messages m such that S contains a message that directly resets m.*

Then there is a set D with $D_{\min} \subset D \subset D_{\max}$, depending only on S, such that the replica's value is $|I \setminus D|$.

Eventual consistency follows because D depends only on S, not on the particular replica or the order in which it received messages.

We deliberately underspecify D because its exact form is hard to describe. However, we believe that any such D gives a reasonable interpretation of the observed-reset semantics. $D = D_{\max}$ corresponds to an ideal, maximally reactive semantics, in which reset messages take effect as soon as they enter the causal history of S, and they affect all increment messages that they sender-reset. $D = D_{\min}$ corresponds to a minimally reactive semantics, in which we only account for reset messages actually in S, and those only affect increment messages that they directly reset.

Although not ideal, $D_{\min}$ and any intermediate D are still reasonable because they all eventually converge to the ideal semantics:

Proposition 3.2. *In the setting of Theorem 3.1, suppose S is downwards-closed under the causal order, i.e., if $m \in S$ and $m' < m$, then $m' \in S$. (This is always the case if we assume causally-ordered delivery.) Then $D_{\min} = D_{\max}$, and Theorem 3.1 implies that the value is given by the observed-reset semantics $(*)$.*

[1] We count a replica's own sent messages among its received messages.

Proof. Let $m \in D_{\max}$. We need to show that $m \in D_{\min}$. By definition, there exists a message m' that sender-resets m and a message $m'' \in S$ such that $m' \leq m''$. Since S is downwards-closed under the causal order, $m' \in S$.

If m' directly resets m, then we are done. Otherwise, m' is causally greater than a message m'_2 that sender-resets m. Since S is downwards-closed, $m'_2 \in S$. We can then repeat this argument with m'_2 in place of m', obtaining a series of messages $m'_3 > m'_4 > \cdots \in S$ that sender-reset m, eventually terminating with a message that directly resets m. Hence $m \in D_{\min}$.

Finally, we need to prove that for $D = D_{\max} = D_{\min}$, $|I \setminus D|$ equals the value given by the observed-reset semantics. Since S is downwards-closed, $D_{\max}$ is the set of increment messages m such that S contains a message that sender-resets m. Thus $|I \setminus D_{\max}|$ is identical to $(*)$. □

We begin the proof of Theorem 3.1 by considering the alternate algorithm in Algorithm 2. Its differences from the original algorithm are:

- C, M are renamed to C', M', for clarity in the discussion below.
- The lines `M.remove(j)` and their `if` statements are removed (lines 29–30 and 45–46).
- The condition `if i not in dom(M)` (line 18) is replaced with `if i not in dom(M') or M'[i].p = M'[i].n`.
- The condition `if start or j not in dom(M)` on line 25 is simplified to `if start`.
- The reset generator (line 36) is changed to `return (reset, {(j, p, c) | (j, (p, n, c)) in M' and not (p = n and c <= C'[j]) })`.
- In the reset effector, only the second case is present (line 44).

Our plan is to show that Algorithm 2 has the same behavior as the original algorithm, then prove Theorem 3.1 for Algorithm 2, since it is simpler than the original algorithm. We start with some basic facts about Algorithm 2.

Lemma 3.3. *In an execution of Algorithm 2, define the* C-value *of an increment message m sent by a replica j to be the count of m among all increment messages sent by j, across all CRDT instances, starting at 1. Then whenever a replica receives m, line 29 of its effector sets $C'[j]$ to the C-value of m.*

Proof. Trivial, since we assume FIFO delivery across all CRDT instances. □

Lemma 3.4. *In an execution of Algorithm 2, suppose a replica has received a set of messages S, and let the replica's state be (C', M'). Then for each replica j:*

(a) The values p appearing in increment messages sent by j in the causal history of S increase monotonically.

(b) $M'[j].p$ is less than or equal to the value of p in the most recent increment message sent by j in the causal history of S, and greater than or equal to the value of p in the most recent increment message sent by j in S (0 if there is no such message).

(c) $M'[j].n \leq M'[j].p$.

Algorithm 2 Alternate algorithm used in the proof of Theorem 3.1.

```
I set of replica identifiers
algorithm for replica i

global replica state (VV, total incs, shared by all CRDT instances):
  C' : I -> int
  // assuming C'[j] = 0 for unmapped keys

per instance CRDT state (payload):
  M' : I -> (p: int, n: int, c: int)
  // assuming M'[j] = (0, 0, 0) for unmapped keys

query value() : int
  return sum { p - n | (p, n, c) in M' }

update inc

  generator ()
    if i not in dom(M') or M'[i].p = M'[i].n
      return (inc, i, C'[i] + 1, true)
    else
      return (inc, i, M'[i].p + 1, false)

  effector (inc, j, p, start)
    let c = C'[j] + 1
    if start
      M'[j] <- max(M'[j], (p, p - 1, c)) // max is taken entry-wise
    else
      M'[j] <- max(M'[j], (p, 0, c))
    C'[j] <- c

update reset

  generator ()
    return (reset, {(j, p, c) | (j, (p, n, c)) in M' and not (p = n and c <= C'[j]) })

  effector (reset, R)
    for (j, p, c) in R
      M'[j] <- max(M'[j], (p, p, c))
```

(d) $M'[j].c$ is the C-value of the unique increment message in the causal history of S that was sent by j and has $p = M'[j].p$ (0 if there is no such message).

Proof. These facts are proved by mutual induction over the messages in the causal history of S. Formally, the induction proceeds in causal order, and its inductive claim applies to all replicas.

(a): Let m be an increment message sent by replica j. If j used the first case in the increment generator (line 19), then the claim follows because $C'[i] \geq M'[i].p$ always, as one can prove inductively. If j used the second case (line 21), then by (b) used inductively, $M'[j].p$ equalled the p value in j's previous increment message, and $M'[j].p + 1$ is greater.

(b): Essentially, increment messages sent by j are the only "source" of increase for $M'[j].p$, although our replica may learn of these increases indirectly via reset messages. Hence $M'[j].p$ is less than or equal to the maximum value of p in an increment message sent by j in the causal history of S. By (a) used inductively, this maximum equals the p value of the most recent such message. The lower bound holds due to lines 26/28 and the fact that $M'[j].p$ never decreases.

(c): All lines that set $M'[j].n$ (lines 26, 28, and 38) ensure this.

(d): The increment message is indeed unique by (a) used inductively. Using (d) inductively, one checks that whenever $M'[j]$ is set to a value $\max(M'[j], (p, n, c))$, c is the C-value of the unique increment message in the causal history of S that was sent by j and has p-value p. It remains to prove that in the max, either both p and c supersede the previous $M'[j]$, or neither of them do. That follows by using (d) and (a) inductively, together with the fact that j's C-values increase monotonically. □

Proposition 3.5. *In any execution, a replica using Algorithm 2 will have the same behavior as the same replica using the original algorithm, i.e., it will output the same messages and return the same query values.*

Proof. 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:

- $C' = C$.
- 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, it is easy to check that in every case, both algorithms' generators output the same messages, and they return the same query values. Note that to check that both algorithms always get the same truth values for their line 18s, we need to use the fact that $M'[i].c \leq C[i]$ by Lemma 3.4(d), hence $M'[i].p = M'[i].n$ implies that M omits i.

We now prove the correspondence claim by induction on the number of received messages. It is trivially true initially. Now assume the claim is true after receiving a set of messages S, and let m be a newly received message; we must prove the correspondence claim for the state after effecting m.

Case $m = (\texttt{inc}, j, p, \texttt{start})$**:** Except for lines 29–30 of Algorithm 1, which are missing in Algorithm 2, the only scenario when the two executions are different is when $\texttt{start} = \texttt{false}$ and Algorithm 1 does not have an entry $M[j]$.

Suppose that is the case. Since $\texttt{start} = \texttt{false}$, j must have previously sent an increment message, and FIFO delivery implies that we have already received it. Hence $M'[j]$ exists. By the correspondence, we must have $M'[j].p = M'[j].n$. Also, $M'[j].p = p - 1$: by FIFO delivery, we must have received the previous increment message from replica j; by Lemma 3.4(b) applied to replica j, that message had $p - 1$ for its third argument; and by Lemma 3.4(b) applied to our replica, $M'[j].p = p - 1$. Hence line 28 of Algorithm 2 sets $M'[j] \leftarrow (p, p - 1, c)$ (using Lemma 3.4(c)(d) to argue that $p - 1 \geq M'[j].n$ and $c \geq M'[j].c$), just like line 26 of the original algorithm.

Finally, lines 29–30 of the original algorithm ensure that at the end of the effector, M omits entry j if and only if $M'[j].p = M'[j].n$ and $M'[j].c \leq C[j]$, as required for the correspondence. Note that the condition $M[j].c = C[j]$ is equivalent to $M[j] \leq C[j]$ because lines 26/28 ensure $M[j].c \geq C[j]$.

Case $m = (\texttt{reset}, R)$**:** We need to verify that for each $(j, n, c) \in R$, $M[j]$ is set in correspondence with $M'[j]$. Entries $M[j]$ and $M'[j]$ for which j does not appear in R are unchanged by either algorithms' effectors, and C and C' are not changed, so those remain in correspondence.

So, let $(j, n, c) \in R$. If $M[j] = M'[j]$ (possibly both not present), then both algorithms set $M'[j] \leftarrow \max(M'[j], (p, p, c))$, except that in the end, the original algorithm deletes entry j from M if and only if $M'[j].p = M'[j].n$ and $M[j].c \leq C[j]$. This preserves the correspondence.

Otherwise, M omits entry j, $M'[j].p = M'[j].n$, and $M'[j].c \leq C[j]$. If $c \leq C[j]$, then at the end of Algorithm 1, M still omits entry j. Meanwhile, at the end of Algorithm 2, M' satisfies $M'[j].p = M'[j].n$ and $M'[j].c \leq C[j]$: $M'[j].p$ and $M'[j].n$ either both remain unchanged or both get set to p (whichever is greater); and $M'[j].c$ either remains unchanged or gets set to c (whichever is greater), in either case at most $C[j]$.

If instead $c > C[j]$, then at the end of the original Algorithm 1, $M[j] \leftarrow (p, p, c)$ (line 42). Meanwhile, the same happens to M' in Algorithm 2, as follows. We have $c > C[j] \geq M'[j].c$, so the max in line 38 sets $M'[j].c \leftarrow c$. Also, $p > M'[j].p$: since $c > M'[j].c$, p comes from an increment message sent by j after sending the message with p-value $M'[j].p$ (using Lemma 3.4(d) applied to m's sender's state at the time they sent m), so it is greater by Lemma 3.4(a). Hence the max in line 38 also sets $M'[j].p \leftarrow p$. Finally, we likewise have $p \geq M'[j].n = M'[j].p$, so $M'[j].n \leftarrow p$. □

Proof of Theorem 3.1. Thanks to Proposition 3.5, it suffices to prove the theorem for Algorithm 2. It is easy to check that after receiving a set of messages S, the state of Algorithm 2 has the following properties:

- For each j, $M'[j].p$ is the maximum of (ai) the value p in the most recent message $(\texttt{inc}, j, p, \texttt{start}) \in S$, and (aii) the maximum across all values p such that (j, p, c) appears in a reset message in S. (If there are no such messages, $M'[j]$ is not present.)
- For each j, $M'[j].n$ is the maximum of (bi) the value $p - 1$ corresponding to the most recent message $(\texttt{inc}, j, p, \texttt{true}) \in S$, and (bii) the maximum across all values p such that (j, p, c) appears in a reset message in S. (If there are no such messages, $M'[j]$ is not present, since j's first increment message must have $\texttt{start} = \texttt{true}$.)

We need to prove that the query value `sum { p - n | (p, n, c) in M' }` resulting from the above state obeys the desired semantics. Let I be the set of increment messages in S. Define D to be the set of increment messages $m = (\texttt{inc}, j, p, \texttt{start})$ such that:

(1) There is a message $m' = (\texttt{reset}, R) \in S$ such that $(j, q, c) \in R$ for some $q \geq p$.

(2) Or, there is a message $m'' = (\texttt{inc}, j, q, \texttt{true}) \in S$ with $q > p$.

Query value: It suffices to prove that for each entry $M'[j]$, $M'[j].p - M'[j].n$ equals the number of increment messages sent by j in $I \setminus D$.

Suppose $M'[j].p$ is given by (aii). Then $M'[j].n$ must be given by (bii), so $M'[j].p - M'[j].n = 0$. Meanwhile, every increment message sent by I is in D due to case (1), so there are also 0 increment messages sent by j in $I \setminus D$, as claimed.

Now suppose $M'[j].p$ is given by (ai). Note that for each replica j, the p values in increment messages sent by j increase by one each time, except in increment messages with $\texttt{start} = \texttt{true}$. Thus $M'[j].p$ equals the value $p-1$ corresponding to the most recent message $(\texttt{inc}, j, p, \texttt{true}) \in S$ plus the number of increments after or including that message. This proves the claim in case $M'[j].n$ is given by (bi). If $M'[j].n$ is instead given by (bii), then $M'[j].p - M'[j].n$ equals the number of increments sent after the increment corresponding to (bii)'s maximal (j, p, c), again proving the claim.

Proof that $D_{\min} \subset D$: If a message $m' \in S$ directly resets m, then it must contain an entry (j, q, c) with $q \geq p$, since the sender must have had $M'[j].p > M'[j].n$. So $m' \in D$ by case (1).

Proof that $D \subset D_{\max}$: Let $m \in D$. Suppose $m \in D$ due to case (1), and let $m' \in S$ be as in that case. Since m' has an entry $(j, q, c) \in R$ with $q \geq p$, before sending m', its sender must have either received m, or a reset message m'_2 that likewise contains (j, q, c). In the former case, m' sender-resets m, hence $m \in D_{\max}$. In the latter case, note that $m'_2 < m'$. Repeating the argument with m'_2 in place of m, etc., we eventually find a chain of messages $m'_n < m'_{n-1} < \cdots < m'_2 < m'$ such that m'_n sender-resets m, hence again $m \in D_{\max}$.

Now suppose $m \in D$ due to case (2). Since $\texttt{start} = \texttt{true}$ in m'' and $m'' > m$ is not the first increment message that j sent, replica j must have had $p \leq M'[i].p = M'[i].n$ when it sent m'' (see line 18 of Algorithm 2). Since $p \leq M'[i].n$, replica j must have previously received a message $m''' = (\texttt{reset}, R)$ such that $(j, q, c) \in R$ for some $q \geq p$. We then proceed as in case (1) with m''' in place of m', noting that $m''' < m'' \in S$. □

4 Efficiency

4.1 Storage Space

Theorem 4.1. *In an execution of Algorithm 1, let S, I, and D be as in Theorem 3.1. Suppose S is downwards-closed under the causal order. Then the number of entries in the replica's state M is equal to the number of distinct senders of messages in $I \setminus D$. In particular, if the replica's value is 0, then M is empty.*

Proof. Consider the analogous execution of Algorithm 2. We have $M'[j].c \leq C'[j]$ for all j: Lemma 3.4(d) says that $M'[j].c$ is the C-value of an increment message m in the causal history

of S; and since S is downwards-closed, we already received m, so $C'[j]$ is at least as large as its C-value. Hence by the correspondence in the proof of Proposition 3.5, M is the same as M' except that M omits any entries j such that $M'[j].p = M'[j].n$. Since $M'[j].n \leq M'[j].p$ by Lemma 3.4(c), it follows that each entry $M[j]$ must have $M[j].n < M[j].p$, contributing a positive amount to the replica's value. This proves the claim, since $M[j]$ contributes an amount equal to the number of messages in $I \setminus D$ sent by replica j. □

When M is empty, the counter's state can be removed from memory entirely, except for the global version vector C that is shared by all counters. If the counter is needed later (e.g., to process a newly received message), it can be re-initialized with empty M, as if it was being created for the first time. Also, enforcing FIFO delivery only requires the global version vector C, not any instance-specific metadata. Thus fully-reset counters take up no space in memory. In particular, a counter-valued map does not need to store any per-key metadata for fully-reset values.

Even when S is not downwards-closed, the number of entries is always upper bounded by the number of replicas.

4.2 Network Traffic

Increment messages have constant size. This holds even when counting metadata for FIFO delivery, which is just the single number $C[i] + 1$ and the sender's id i.

Reset messages have the same asymptotic size as the sender's state space. In particular, when the sender's previously-received messages are downwards-closed under the causal order, Theorem 4.1 implies that the reset message's size is proportional to the number of distinct senders of messages being reset. Even when this is not the case, it size is always bounded by the number of replicas.

5 Future Work

One direction for future work could be to relax the FIFO delivery requirement to just exactly-once delivery, without any ordering requirements.

Another could be to define a merge function, so that the counter can be used as a state-based or delta state-based CRDT. However, the obvious merge function (entrywise max) does not work, due to deleted entries.

In the case of non-causally-ordered delivery, one could pursue different semantics from our design. In particular, it would be desirable to have a simple and precise guarantee, instead of allowing the range $D_{\min} \subset D \subset D_{\max}$. Or, one could use an even more reactive interpretation of the observed-wins semantics than $D_{\max}$: an increment message is reset by any *causally greater* reset message, not just a message that sender-resets it.

Acknowledgments

Matthew Weidner is supported by an NDSEG Fellowship sponsored by the US Office of Naval Research.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. 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, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Basho. Riak datatypes, 2015. `http://github.com/basho`.

[4] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 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*, Onward! 2019, page 154–178, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. *Conflict-Free Replicated Data Types CRDTs*, pages 1–10. Springer International Publishing, Cham, 2018.

[6] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[7] Georges Younes, Paulo Sérgio Almeida, and Carlos Baquero. Compact resettable counters through causal stability. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, pages 2:1–2:3, New York, NY, USA, 2017. ACM.