*Technical Report*

UCAM-CL-TR-969
ISSN 1476-2986

Number 969

UNIVERSITY OF CAMBRIDGE

**Computer Laboratory**

# Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode

Martin Kleppmann

May 2022

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*https://www.cl.cam.ac.uk/*

© 2022 Martin Kleppmann

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

*https://www.cl.cam.ac.uk/techreports/*

ISSN 1476-2986

# Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode

Martin Kleppmann

**Abstract**

Designing algorithms for distributed systems has a reputation of being a difficult and error-prone task, but this difficulty is rarely measured or quantified in any way. This report tells the story of one informal experiment, in which users on Twitter were invited to identify the bug in an incorrect CRDT algorithm. Over the following 11 hours, at least 16 people (many of whom are professional software engineers) made attempts to find the bug, but most were unsuccessful. The two people who did identify the bug were both PhD students specialising in CRDTs. This result may serve as evidence of the difficulty of designing correct CRDT algorithms.

## 1 Introduction

In the autumn of 2020, I was writing new lecture notes for an undergraduate course on distributed systems.[1] One topic I wanted to cover was *Conflict-free Replicated Data Types* (CRDTs), a family of algorithms that allow different replicas to concurrently update some replicated data, and which ensure that those replicas converge to the same state as they communicate [7]. CRDTs work by making operations commutative, so different replicas can see operations in the same order and still converge.

To make the lecture notes more interesting, I wanted to include an example algorithm that would demonstrate some interesting features of CRDTs, while still being short enough to fit on one slide. I did not know of a published algorithm that provided the combination of features that I wanted, so I wrote out a new algorithm that I believed to be "obviously correct". I have been designing CRDT algorithms for years [2, 3, 4], which gave me confidence that I knew what I was doing.

I then added an exercise to the lecture notes asking students to prove the correctness of the algorithm, and I started writing my own proof for the solution notes. The proof turned out to be more difficult than expected. Only after a few hours of fruitlessly trying to prove the algorithm correct, I realised that the algorithm was in fact wrong: there were certain combinations of concurrent operations under which replicas would not converge. Frustrated, I posted the following on Twitter:

> **Martin Kleppmann** at 2020-11-12 22:48 GMT
>
> `https://twitter.com/martinkl/status/1327020435419041792`
>
> Today in "distributed systems are hard": I wrote down a simple CRDT algorithm that I thought was "obviously correct" for a course I'm teaching. Only 10 lines or so long. Found a fatal bug only after spending hours trying to prove the algorithm correct. 😭

[1] `https://www.cl.cam.ac.uk/teaching/2122/ConcDisSys/materials.html`

"10 lines" was an exaggeration; in fact the pseudocode was 24 lines long. After posting the tweet it occurred to me that I could also post the incorrect pseudocode, and ask my followers whether they could spot the bug. I posted it (see Section 2) and went to bed. The next morning I woke up to several suggestions, but nobody had found the bug yet. In the end, it took 11 hours before an example exhibiting the bug was posted by Sreeja Nair.

Fast forward 18 months, I have found out that my tweet has been cited by at least two major papers as evidence of CRDTs being difficult to get right: by Cheung et al. at CIDR 2021 [1, Figure 4] and by Soundarapandian et al. at PLDI 2022 [8, Reference 19]. However, a tweet is not a very good reference: it lacks context, and it is not very permanent (for example, if a Twitter user deletes their old tweets or closes their account, their replies to my tweet disappear from the comment thread).

This report provides a permanent archive of this Twitter episode, as well as some context and analysis. Although it is not a controlled study of the difficulty in finding bugs in a distributed algorithm, I do believe that it sheds some light on the challenges of CRDT design, highlighting that even experienced software engineers have difficulty understanding concurrent algorithms. While the understandability of consensus algorithms was studied in a lab setting by Ongaro and Ousterhout [6], I am not aware of a comparable study on the understandability of CRDT algorithms. The informal Twitter experiment in this report is a first step towards such a study.

## 2 The Algorithm

Shortly after posting the aforementioned tweet I followed up with a screenshot of the incorrect algorithm, which is reproduced here as Algorithm 1 on page 5. I tweeted:

> **Martin Kleppmann** at 2020-11-12 23:10 GMT
> `https://twitter.com/martinkl/status/1327025979454263297`
> Here's the algorithm (an op-based map CRDT with LWW semantics per key). See if you can figure out the bug.
>
> [Image showing Algorithm 1]

The algorithm essentially implements a replicated key-value store in which clients can perform three operations: read the value for a key $k$, set the value $v$ for a key $k$, or delete the key-value mapping for a key $k$. In my tweet, *op-based map CRDT* means that it is a key-value mapping, and that the algorithm is operation-based (as opposed to state-based), which means that one replica's updates are propagated to other replicas by sending a message describing the update that occurred.

Each update is associated with a timestamp, such as a Lamport timestamp [5], which is assumed to be globally unique. When two replicas concurrently update the same key, the update with the greater timestamp takes precedence over the one with the lower timestamp; this behaviour is known as *last write wins* (abbreviated as *LWW* in the tweet).

The basic correctness requirement for the algorithm is *convergence*: when two replicas have seen the same set of updates, they must have the same mapping of keys to values. The bug in Algorithm 1 is that it is possible to create a set of updates after which the replicas fail to converge.

**Algorithm 1** The buggy algorithm that I tweeted first.

**on** initialisation **do**
  $values := \{\}$
**end on**

**on** request to read value for key $k$ **do**
  **if** $\exists t, v.\ (t, k, v) \in values$ **then return** $v$ **else return** null
**end on**

**on** request to set key $k$ to value $v$ **do**
  $t :=$ newTimestamp() ▷ globally unique, e.g. Lamport timestamp
  **broadcast** $(\mathsf{set}, t, k, v)$ by causal broadcast (including to self)
**end on**

**on** delivering $(\mathsf{set}, t, k, v)$ by causal broadcast **do**
  $previous := \{(t', k', v') \in values \mid k' = k\}$
  **if** $previous = \{\} \ \vee\ \forall (t', k', v') \in previous.\ t' < t$ **then**
    $values := (values \setminus previous) \cup \{(t, k, v)\}$
  **end if**
**end on**

**on** request to delete key $k$ **do**
  **if** $\exists t, v.\ (t, k, v) \in values$ **then**
    **broadcast** $(\mathsf{delete}, t)$ by causal broadcast (including to self)
  **end if**
**end on**

**on** delivering $(\mathsf{delete}, t)$ by causal broadcast **do**
  $values := \{(t', k', v') \in values \mid t' \neq t\}$
**end on**

**Algorithm 2** The fixed version of the algorithm tweeted later.

**on** initialisation **do**
  $values := \{\}$
**end on**

**on** request to read value for key $k$ **do**
  $T := \{t \mid \exists v.\ (t, k, v) \in values\}$
  **if** $T = \{\}$ **then**
    **return** null
  **else**
    **return** the unique $v$ such that $(\max(T), k, v) \in values$
  **end if**
**end on**

**on** request to set key $k$ to value $v$ **do**
  $T := \{t \mid \exists v'.\ (t, k, v') \in values\}$
  $t :=$ newTimestamp() ▷ globally unique, e.g. Lamport timestamp
  **broadcast** $(\mathsf{set}, T, t, k, v)$ by causal broadcast (including to self)
**end on**

**on** delivering $(\mathsf{set}, T, t, k, v)$ by causal broadcast **do**
  $values := \{(t', k', v') \in values \mid t' \notin T\} \cup \{(t, k, v)\}$
**end on**

**on** request to delete key $k$ **do**
  $T := \{t \mid \exists v.\ (t, k, v) \in values\}$
  **if** $T \neq \{\}$ **then**
    **broadcast** $(\mathsf{delete}, T)$ by causal broadcast (including to self)
  **end if**
**end on**

**on** delivering $(\mathsf{delete}, T)$ by causal broadcast **do**
  $values := \{(t', k', v') \in values \mid t' \notin T\}$
**end on**

Over the following hours, several people volunteered ideas about the nature of the bug (see Section 3), but nobody was able to precisely identify the circumstances in which the bug would be triggered. The next morning I followed up:

> **Martin Kleppmann** at 2020-11-13 09:17 GMT
> `https://twitter.com/martinkl/status/1327178651209764865`
> Well, this is fun. @tim_1729, @steveloughran, @encthenet, @simpuleguy, @KevinDP55 got close, but nobody has yet identified the precise circumstances in which the bug occurs. The bug is divergence (two replicas have processed the same messages, but are not in the same state).

I then posted a revised version of the algorithm that did not suffer from the bug, without explicitly revealing the nature of the bug. This algorithm is reproduced here as Algorithm 2 on page 6.

> **Martin Kleppmann** at 2020-11-13 09:19 GMT
> `https://twitter.com/martinkl/status/1327179176521166853`
> Several people have suggested that tombstones are needed. They are one possible approach, but not the only. Here is a variant of the algorithm that is correct (I believe), but does not use tombstones.
>
> [Image showing Algorithm 2]

Finally, after another hour, Sreeja Nair posted a correct answer, and I retweeted it:

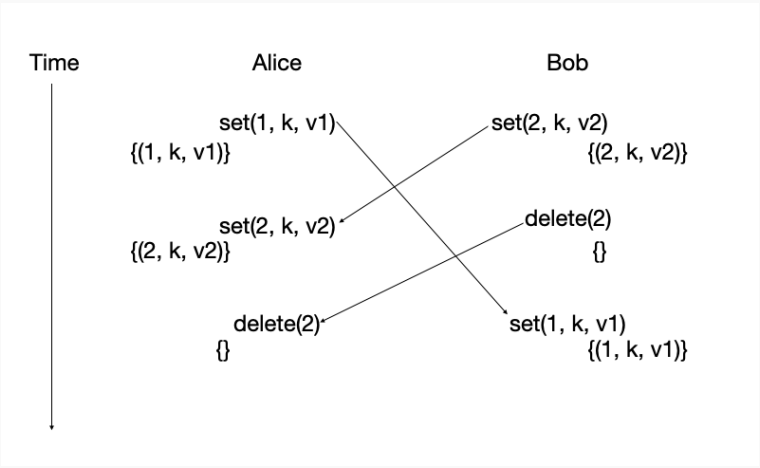
> **Martin Kleppmann** at 2020-11-13 11:20 GMT
> `https://twitter.com/martinkl/status/1327209597011189761`
> And here's the resolution. @sreejas found the bug, congratulations! 😄 🎉
>
> > **Sreeja Nair** at 2020-11-13 10:09 GMT
> > `https://twitter.com/sreejas/status/1327191953474203648`
> > Is this the issue?
>
> 
> 

Table 1: Summary of attempts to find the bug

| Name | Occupation | Answer |
|---|---|---|
| Taylor Blau | Software engineer at GitHub | none |
| Osama Khan | Cryptocurrency company founder | none |
| Werner Schuster | JavaScript/Clojure developer | definition of delete op |
| @digi_noise | unknown | definition of delete op |
| M.P. Korstanje | Software engineer | tombstones |
| Steve Loughran | Software engineer at Cloudera | tombstones |
| John-Mark Gurney | FreeBSD developer/consultant | set/delete ordering |
| @spudwaff1e | unknown | causal broadcast ordering |
| Ryan Doenges | PhD student at Cornell | old values reappear |
| Avais | Engineer | broadcast reliability |
| Stephen Spalding | Distributed systems engineer at Netflix | tombstones |
| Rajat Kanti Bhattacharjee | Engineer at Gojek | set/delete ordering |
| Matthew Sackman | Creator of a distributed database | tombstones/timestamp ordering |
| @signof | Senior software engineer | causal broadcast ordering |
| Kevin De Porre | PhD student at Vrije Universiteit Brussel | almost correct answer |
| Sreeja Nair | PhD student at Sorbonne Université | correct answer |

My final tweet on the topic was a comment on the composition of CRDTs:

> **Martin Kleppmann** at 2020-11-13 11:52 GMT
> `https://twitter.com/martinkl/status/1327217743960084480`
> The interesting thing about this bug is that it comes about only from the interaction of two features. A LWW map by itself is fine. A set in which you can insert and delete elements (but not update them) is fine. The problem arises only when delete and update interact.

## 3 Solution attempts

At the time, I had approximately 29,400 followers on Twitter.[2] Of these, 16 people responded with tweets suggesting that they had attempted to find the bug. Those tweets are collected in this section. Replies that did not suggest active engagement with the problem are given in Section 4.

Table 1 summarises the solution attempts, along with the occupation of the respondent (if apparent from their social profile). I personally know five of the 16 people listed, and know that they have deep technical expertise. Therefore, even though this is a crowdsourced experiment, the "crowd" in question contains some highly qualified people; the fact that they were not able to identify the bug therefore indicates that it is non-obvious.

Some people said they had tried to find the bug but did not succeed:

> **Taylor Blau** at 2020-11-12 23:14 GMT
> `https://twitter.com/ttaylorr_b/status/1327027054290890759`
> I wish that I could see it; seems right to me.

[2] According to `https://web.archive.org/web/20201115151909/https://twitter.com/martinkl`

> **Osama Khan** at 2020-11-12 23:16 GMT
> `https://twitter.com/osamakhn/status/1327027488870961152`
> Can't see it right now but there goes my evening 🤔

Two users queried the construction of the deletion operation:

> **Werner Schuster** at 2020-11-12 23:38 GMT
> `https://twitter.com/murphee/status/1327032940379987968`
> What's the reason for using (delete, t) and not (delete, k)? Wouldn't that delete all entries for the same timestamp? ... Although I guess if timestamps are globally unique it might have the same effect as deleting k.

to which I replied:

> **Martin Kleppmann** at 2022-11-13 08:56 GMT
> `https://twitter.com/martinkl/status/1327173562952781824`
> It deletes only the entry with a particular timestamp, rather than all entries for a key. This is what we want, because if the delete is concurrent with a set operation, the intended end result is that the set operation takes precedence.

and another user replied:

> **Ergo Sum** at 2020-11-13 07:24
> `https://twitter.com/digi_noise/status/1327150244799320073`
> I had the same thoughts, but t is globally unique, so given t will always be found only for key k.

M.P. Korstanje suggested that deleted elements should be stored explicitly (this is known as a *tombstone*):

> **M.P. Korstanje** at 2020-11-12 23:35 GMT
> `https://twitter.com/LogAteWhale/status/1327032338207924224`
> Values aren't stored/marked as deleted so that's bound to give problems when the delete over takes a write that inserted the value.
>
> That's the direction I'd look in for bugs. But I don't understand enough to say that it is the bug.

This answer is not correct: causal broadcast ensures that when a value is deleted, all nodes process the insertion of the deleted value before processing the deletion. Tombstones are not necessary in this algorithm.

Steve Loughran also suggested tombstones:

> **Steve Loughran** at 2020-11-12 23:30 GMT
> `https://twitter.com/steveloughran/status/1327031059209461760`
> at a guess, deletion
>
> if someone issues a delete (k) to a node which hasn't yet received/processed an update from others then the delete won't be broadcast.

**Steve Loughran** at 2020-11-12 23:36 GMT

https://twitter.com/steveloughran/status/1327032633583415303

maybe also if problems you receive a delete (t) ahead of the set(t, k, v). better to just broadcast a set(t, k, tombstone) and assuming recipients process received events in order (?) then they could do the cleanup

**John-Mark Gurney** at 2020-11-13 00:56 GMT

https://twitter.com/encthenet/status/1327052580640284672

Looks like other people saw the same issue that if a set and delete are delivered in the wrong order, a set that was after a delete could persist though it should not.

John-Mark Gurney's suggestion is perhaps on the right track, but it does not contain enough detail to identify the bug.

**Spudwaffle** at 2020-11-12 23:37 GMT

https://twitter.com/spudwaff1e/status/1327032827267846144

Let's say we have three peers, Alice, Bob, and Carol:

Everyone starts with values = {(1, 'a', '1')}.

Alice does (set, 2, 'a', '2')
Bob receives (set, 2, 'a', 2) from Alice, has {(2, 'a', '2')}
Bob does (delete, 2)
Carol receives (delete, 2) from Bob, has {(1, 'a', '1')}
Carol receives (set, 2, 'a', '2') from Alice, has {(2, 'a', '2')}
Alice receives (delete, 2) from Bob, has {}

Alice and Bob end with {}, while Carol ends with {(2, 'a', '2')}
We could fix this by creating a new timestamp on delete, and setting v to null.
(Tombstones)

This answer is incorrect: in this example, causal broadcast will ensure that Carol processes Alice's (set, 2) before Bob's (delete, 2), because (set, 2) causally precedes (delete, 2).

**Ryan Doenges** at 2020-11-12 23:33 GMT

https://twitter.com/hackedy/status/1327031870895353858

delete could make old values reappear, is that the problem? You want a deletion to be delete(k,t) and remove all bindings (t',k,v) where t' <= t ?

to which I replied: "Old values reappearing is not the problem. The algorithm preserves the invariant that there is at most one entry in the set of values for a given key, which is why it's safe for a delete message to contain only one timestamp."

**Avais** at 2020-11-13 01:37 GMT

https://twitter.com/gramaester/status/1327062888020787200

Is the issue that the source of the boardcast may itself fail after a successful broadcast?

to which I replied: "We can assume that the broadcast protocol will take care of resending any missing messages (for example, a node that is not the original source of a broadcast may resend it)."

> **Stephen Spalding** at 2020-11-13 02:10 GMT
> https://twitter.com/stephenspalding/status/1327071247054696448
> What is dead may never die! (due to lack of tombstone)

The lack of tombstones is not the problem.

> **Rajat Kanti Bhattacharjee** at 2020-11-13 03:56 GMT
> https://twitter.com/simpuleguy/status/1327098070115385344
> I am trying to figure the puzzle. What I can see is set us following the logic of no old writes based on time stamp. But delete has a sense of time stamp based deletion.(time as the key) Would it cause a discrepancy when doing set and delete on same key. Delayed set -> read err

This suggestion indicates thinking in the right direction, but there is not enough detail.

> **Matthew Sackman** at 2020-11-13 08:42 GMT
> https://twitter.com/hylomorphism/status/1327169931419541504
> Tombstones on delete maybe, as others have suggested. But also you don't require that your newTimestamp has a > relationship with any locally-existing t yet you rely on < elsewhere.

Matthew Sackman correctly points out that new timestamps need to be strictly greater than any existing local timestamp, but Lamport timestamps ensure this property, so this is not the bug.

> **Signof** at 2020-11-13 09:39 GMT
> https://twitter.com/signof/status/1327184197975420930
> My guess is that Lamport timestamps are not totally ordered and that causal broadcast isn't strong enough to order independent concurrent writes.

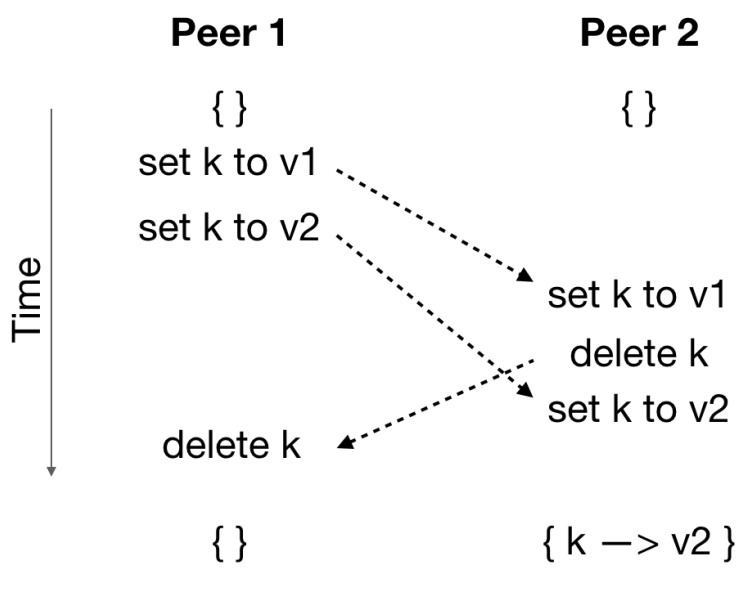
Lamport timestamps are in fact totally ordered, and the ordering of causal broadcast is not the problem here.

Kevin De Porre mentions tombstones, which are not needed, but also includes a diagram that *almost* identifies the scenario in which the bug occurs:

**Kevin De Porre** at 2020-11-13 08:11 GMT

https://twitter.com/KevinDP55/status/1327162097604714496

Deleting is always tricky with CRDTs.
That's why many CRDTs mimick deletions by marking elements as deleted rather than really deleting them.



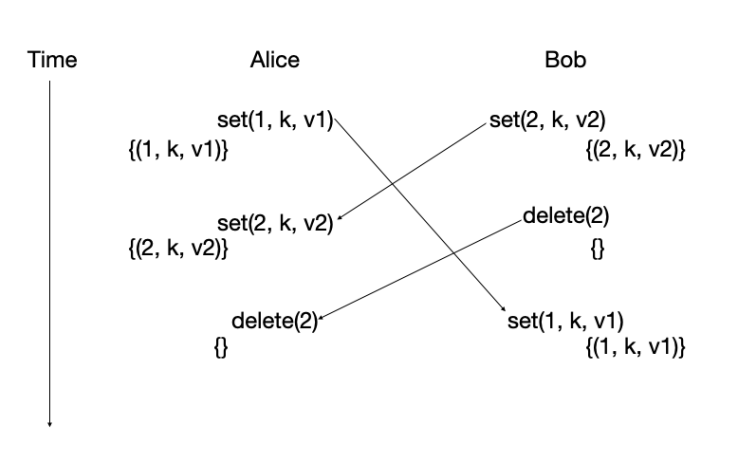

. . . to which I replied: "Close. Can you be specific about the conditions on the timestamps in each operation in order for the bug to be triggered? Also, tombstones are one potential solution, but not necessarily required here."

Finally, Sreeja Nair posted a full example of the bug occurring, including the specific timestamps on the messages that trigger the bug:

**Sreeja Nair** at 2020-11-13 10:09 GMT

https://twitter.com/sreejas/status/1327191953474203648

Is this the issue?




I accepted this tweet as the correct answer.

# 4 Other replies

In this section I have collected replies to my tweets that did not indicate an attempt to find the bug, but which provide some further context.

> **Yik San Chan** at 2020-11-12 22:57 GMT
> https://twitter.com/yiksanchan/status/1327022607405772800
> Do you use Isabelle for the proof?

to which I replied: "No, just pen and paper in this case. Though it would have been quite easily doable in Isabelle, I think."

> **Dimanne** at 2020-11-13 07:30 GMT
> https://twitter.com/DimanNe/status/1327151787573452801
> I think it depends on what this algorithm was supposed to do :)
>
> By the way, have you tried using TLA framework for formal description and verification? Can it help?

to which I replied: "A minimum expectation of CRDTs is that all replicas converge to the same state when they have delivered the same set of messages (possibly in different orders). The bug in this algorithm is a situation in which replicas remain inconsistent. I've not used TLA+ myself very much, but I have read quite a few TLA+ specifications and I hope to use it more in the future. I've done more work on correctness proofs with Isabelle/HOL."

There were also a couple of comments and questions:

> **Chris Batt** at 2020-11-13 04:03 GMT
> https://twitter.com/Ti3eInc/status/1327099729163980802
> It's ok Martin. DS will never be 'solved'. It's the CS equivalent of evolution solving the Cambrian Explosion!

> **Nacho del Valle** at 2020-11-13 07:10 GMT
> https://twitter.com/idelvall/status/1327146815024721921
> Consistency is broken?

> **Mateusz Górski** at 2020-11-13 09:04 GMT
> https://twitter.com/gorski_mt/status/1327175538767777793
> What tools do you use to prove the algorithm? TLA+? Isabel? F*?

> **Jonas Bonér** at 2020-11-13 09:19 GMT
> https://twitter.com/jboner/status/1327179366808346624
> I'm grateful. Your (and other CS researchers) hard work battling with these issues makes my life (a little bit) easier, slowly, step by step... :-)

> **Rajat Kanti Bhattacharjee** at 2020-11-13 09:19 GMT
> https://twitter.com/simpuleguy/status/1327179227645370369
> 👀 Ouch double write. and missing update problem at once .... 😐 Woaw , tough bug. .....

> **Shaazahm** at 2020-11-13 10:27 GMT
> https://twitter.com/shaaza12/status/1327196346575687680
> Tangent: is the "on ... do" syntax just pseudocode or a specific language? Could this be automatically converted to CSP/TLA?

to which I replied: "It's just pseudocode, but the intention is that it could be translated quite easily into real code (as event handlers) or something like TLA+ (as actions)."

> **Daniel Marbach** at 2020-11-13 11:29 GMT
> https://twitter.com/danielmarbach/status/1327211987634429954
> That experience is probably the greatest lessons to teach

The "experience" in this tweet presumably refers to my comment that I had "found a fatal bug only after spending hours trying to prove the algorithm correct".

> **Eduard Popescu** at 2020-11-13 12:26 GMT
> https://twitter.com/zyrconium13/status/1327226224159191041
> TLA+ or http://alloytools.org might help. Model checkers. Just sayin'

> **Siddharth Goel** at 2020-11-15 06:30 GMT
> https://twitter.com/siddharthgoel88/status/1327861565496848385
> Somehow the font in which you wrote the algorithm is the one we see in books as well. So the mind perceives that it cannot be wrong :-D

to which I replied, "It's just the default font in LaTeX 😆"

## 5 Discussion and conclusions

The bug in Algorithm 1, as illustrated in Sreeja Nair's tweet, is due to the fact that the algorithm only stores a single value and timestamp for a given key, and that mapping is removed when the key is deleted. Thus, when there are concurrent updates to the same key, and a deletion of an update with a greater timestamp, an update with a lower timestamp that is concurrent to the deletion can cause divergence.

One possible solution would be to never actually delete items, but to keep the latest timestamp for a given key as a tombstone, and to update that timestamp when a deletion is requested. However, tombstones cause problems with unbounded storage growth. Algorithm 2 is a tombstone-free alternative: it works by storing several values and timestamps for the same key when there are concurrent updates, and only returning the one with the highest timestamp at read time. Every update or delete operation includes the set of timestamps that it overwrites. Causal delivery ensures that the overwritten operation is applied before the overwriting operation. This approach incurs only slightly greater storage and message overhead than Algorithm 1.

The respondents who found the bug (Sreeja Nair) or came close (Kevin De Porre) were both, at the time, PhD students specialising in CRDT algorithms. In fact, Nair's PhD supervisor was Marc Shapiro, one of the researchers who initially defined the concept of CRDTs [7]. Thanks to this background, they were attuned to the types of issues that tend to appear in CRDTs.

However, relying on people with PhDs in this very specific niche of research is not a scalable strategy for developing correct algorithms. The fact that all of the experienced software engineers who replied were not able to spot the bug should be seen as a call to improve the situation. We need to make it possible for software engineers to develop correct replication algorithms without requiring a PhD.

Presumably the answer is better tool support. But what sort of tools would be best? Do we need better programming languages, better compilers, or better formal reasoning tools? Various research efforts in these directions are already underway, and I am excited to see where they lead.

### Acknowledgements

I gratefully acknowledge the support of the Leverhulme Trust, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, Prisma, RelationalAI, SoftwareMill, and Adam Wiggins.

## References

[1] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Matthew Milano. New directions in cloud programming. In *11th Annual Conference on Innovative Data Systems Research*, CIDR 2021, 2021. URL: `http://www.cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf`.

[2] Martin Kleppmann. Moving elements in list CRDTs. In *7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2020. ACM, April 2020. `doi:10.1145/3380787.3393677`.

[3] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, April 2017. `doi:10.1109/tpds.2017.2697382`.

[4] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1711–1724, October 2021. `doi:10.1109/tpds.2021.3118603`.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. `doi:10.1145/359545.359563`.

[6] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, ATC. USENIX, June 2014. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[7] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400. Springer, 2011. `doi:10.1007/978-3-642-24550-3_29`.

[8] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. Certified mergeable replicated data types. In *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2022, June 2022. `doi:10.1145/3519939.3523735`.