A Low-Cost Set CRDT Based on Causal Lengths

Weihai Yu and Sigbjørn Rostad
UIT - The Arctic University of Norway
Tromsø, Norway
weihai.yu@uit.no

Abstract

CRDTs, or Conflict-free Replicated Data Types, are data abstractions that guarantee convergence for replicated data. Set is one of the most fundamental and widely used data types. Existing general-purpose set CRDTs associate every element in the set with causal contexts as meta data. Manipulation of causal contexts can be complicated and costly. We present a new set CRDT, CLSet (causal-length set), where the meta data associated with an element is simply a natural number (called causal length). We compare CLSet with existing general purpose CRDTs in terms of semantics and performance.

CCS Concepts: • **Theory of computation** → Distributed computing models; • **Computing methodologies** → Concurrent algorithms; • **Information systems** → Data replication tools.

Keywords: state-based CRDT, replication, eventual consistency, availability

ACM Reference Format:
Weihai Yu and Sigbjørn Rostad. 2020. A Low-Cost Set CRDT Based on Causal Lengths. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3380787.3393678

1 Introduction

CRDTs, or Conflict-free Replicated Data Types, are abstractions for data replicated at different sites [10]. CRDT data are guaranteed to be strongly eventually consistent [10]. A site queries and updates its local replica without coordination with other sites. When any two sites have applied the same set of updates, they reach the same state, regardless of the order in which the updates are applied.

Set is a fundamental and widely used data type. There exist a number of general-purpose set CRDTs that allow for concurrent addition and removal of elements. Common to these set CRDTs, every element is associated with some causal contexts as meta data. Manipulation of causal contexts could be complicated. It could also be costly, for instance, when there are many sites involved or the sites are dynamic.

We present a new general-purpose set CRDT, causal-length set CLSet, based on an abstraction called causal length. For every element, the associated meta data is simply a natural number, namely the causal length of the element.

We discuss the semantics of different set CRDTs and run some benchmarks to compare their performance.

PaPoC '20, April 27, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-7524-5/20/04.
https://doi.org/10.1145/3380787.3393678

2 CRDT Preliminary

There are two families of CRDT approaches, namely state-based and operation-based [10]. We focus on state-based CRDTs. The possible states of a state-based CRDT must form a join-semilattice [6], which is a sufficient condition for convergence. Briefly, the states form a join-semilattice if they are partially ordered with $\sqsubseteq$ and a join $\sqcup$ of any two states always exists ($s_1 \sqcup s_2$ gives the least upper bound of s_1 and s_2). State updates must be inflationary. That is, the new state supersedes the old one in $\sqsubseteq$. The merge of two states is the result of a join.

$$\begin{aligned}
\mathsf{GSet}(E) &\stackrel{\text{def}}{=} \mathcal{P}(E)\\
\mathsf{add}(s,e) &\stackrel{\text{def}}{=} \{e\} \cup s\\
\mathsf{add}^\delta(s,e) &\stackrel{\text{def}}{=} \begin{cases}\{e\} & \text{if } e \notin s\\ \{\} & \text{otherwise}\end{cases}\\
s \sqcup s' &\stackrel{\text{def}}{=} s \cup s'\\
\mathsf{in}(s,e) &\stackrel{\text{def}}{=} e \in s\\
\mathsf{all}(s) &\stackrel{\text{def}}{=} s
\end{aligned}$$

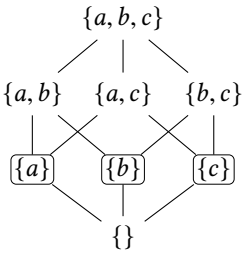

Figure 1. GSet CRDT and Hasse diagram of states

Figure 1 (left) shows GSet, a state-based CRDT for grow-only sets, where E is a set of possible elements, $\sqsubseteq \stackrel{\text{def}}{=} \subseteq$, $\sqcup \stackrel{\text{def}}{=} \cup$, add is a mutator (update operation), and in and all are queries. Obviously, an update through $\mathsf{add}(s,e)$ is an inflation, because $s \subseteq \{e\} \cup s$. Figure 1 (right) shows the Hasse diagram of the states in a GSet. A Hasse diagram shows only the "direct links" between states.

As originally presented in [10], a message for an update is the data state of the replica in its entirety. This could be costly in practice. Delta-state CRDTs address this issue by

only sending join-irreducible states [2, 5]. Basically, join-irreducible states are elementary states and every state in the join-semilattice can be represented as a join of some join-irreducible state(s). In Figure 1, add^δ is a delta-mutator that returns join-irreducible states, which are singleton sets (boxed in the Hasse diagram).

GSet is an example of an *anonymous* CRDT, since the definitions of its mutators are not specific to the sites that perform the updates. Two concurrent executions of the same mutation, such as $\text{add}(\{\}, a)$, fulfill the same purpose.

A CRDT for general-purpose sets with both addition and removal operations can be designed as causal CRDTs [2] such as ORSet (observed-remove set [4, 8, 9]). Report [9] presented ORSet as an operation-based CRDT. Figure 2 is a state-based ORSet based on [8]. Figure 5 shows the states of a single element in ORSet. We describe the figure in more detail in Section 5 where we compare ORSet with CLSet.

$$
\begin{aligned}
\text{ORSet} &\stackrel{\text{def}}{=} s : E \hookrightarrow \mathcal{P}(dots) \times \mathcal{P}(dots) \\
\text{add}_i(s, e) &\stackrel{\text{def}}{=} s\{e \mapsto \langle \text{fst}(s(e)) \cup \{\text{next}_i\}, \text{snd}(s(e))\rangle\} \\
\text{add}_i^\delta(s, e) &\stackrel{\text{def}}{=} \{e \mapsto \langle \{\text{next}_i\}, \{\}\rangle\} \\
\text{remove}_i(s, e) &\stackrel{\text{def}}{=} s\{e \mapsto \langle \text{fst}(s(e)), \text{snd}(s(e)) \cup \text{fst}(s(e))\rangle\} \\
\text{remove}_i^\delta(s, e) &\stackrel{\text{def}}{=} \{e \mapsto \langle \{\}, \text{fst}(s(e))\rangle\} \\
s \sqcup s' &\stackrel{\text{def}}{=} \{(e \mapsto \langle \text{fst}(s(e)) \cup \text{fst}(s'(e)), \\
&\qquad\qquad \text{snd}(s(e)) \cup \text{snd}(s'(e))\rangle \\
&\qquad \mid e \in \text{dom}(s) \cup \text{dom}(s')\} \\
\text{in}(s, e) &\stackrel{\text{def}}{=} \text{fst}(s(e)) \supset \text{snd}(s(e)) \\
\text{all}(s) &\stackrel{\text{def}}{=} \{e \mid e \in \text{dom}(s) : \text{fst}(s(e)) \supset \text{snd}(s(e))\}
\end{aligned}
$$

Figure 2. ORSet CRDT

Basically, every element is associated with two causal contexts, in terms of a partial function[1]. A causal context is a set of event identifiers, also known as *dots*. A *dot* is typically represented as a pair of a site identifier and a site-specific sequence number [1]. next_i generates a new dot at site i. An addition or removal is achieved with inflationary updates of the associated causal contexts. Using causal contexts, we are able to tell explicitly which additions of an element have been later removed. However, maintaining causal contexts for every element can be costly, even though it is possible to compress causal contexts into version vectors, especially under causal consistency.

[1]Given a (total) function $f : \text{dom}(f) \to Y$ where $\text{dom}(f) \subseteq X$. A *partial function* $f : X \hookrightarrow Y$ maps x to $\bot_Y$ if $x \notin \text{dom}(f)$, where $\bot_Y$ is the *bottom element* of Y. For natural numbers $\mathbb{N}$, $\bot_\mathbb{N} = 0$. For $\mathcal{P}(S)$ ordered with $\subseteq$, $\bot_{\mathcal{P}(S)} = \{\}$. Using partial function conveniently simplifies the specification of some mutators and the join operation.

In the following, we design a new general-purpose set CRDT. It is anonymous and is based on the abstraction of causal length. Note that all causal CRDTs are *named*, i.e. not anonymous.

3 Causal length

The key issue that a general-purpose set CRDT must address is how to identify the causality between the different addition and removal updates. We achieve this with the abstraction of causal length, which is based on two observations.

First, the additions and removals of a given element occur in turns, one causally dependent on the other. A removal is an inversion of the last addition it sees. Similarly, an addition is an inversion of the last removal it sees (or none, if the element has never been added).

Second, two concurrent executions of the same mutation of an anonymous CRDT fulfill the same purpose and therefore are regarded as the same update. Seeing one means seeing both (such as the concurrent additions of the same element in GSet). Two concurrent reversions of the same update are also regarded as the same one.

Figure 3 shows a scenario where three sites A, B and C concurrently add and remove element a. When sites A and B concurrently add a for the first time, with updates a_A^1 and a_B^1, they achieve the same effect. Seeing either one of the updates is the same as seeing both. Consequently, states s_A^1, s_A^2, s_B^1 and s_C^1 are equivalent as far as the addition of a is concerned.

Following the same logic, the concurrent removals on these equivalent states (with respect to the addition of a) are also regarded as achieving the same effect. Seeing one

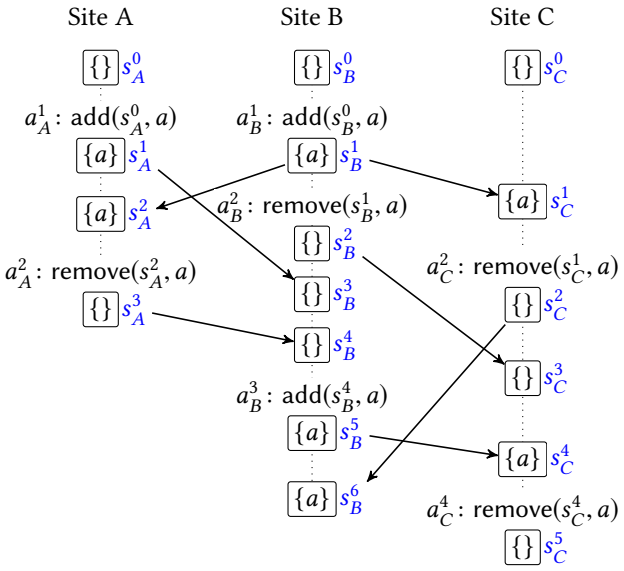

Figure 3. A scenario of concurrent set updates

Table 1. States of set element a

	states as equivalence classes	s_{cl}	all(s_{cl})
s_A^0	$\{\}$	$\{\}$	$\{\}$
s_A^1	$\{\{a_A^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_A^2	$\{\{a_A^1, a_B^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_A^3	$\{\{a_A^1, a_B^1\}, \{a_A^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^0	$\{\}$	$\{\}$	$\{\}$
s_B^1	$\{\{a_B^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_B^2	$\{\{a_B^1\}, \{a_B^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^3	$\{\{a_A^1, a_B^1\}, \{a_B^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^4	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^5	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2\}, \{a_B^3\}\}$	$\{\langle a,3\rangle\}$	$\{a\}$
s_B^6	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}\}$	$\{\langle a,3\rangle\}$	$\{a\}$
s_C^0	$\{\}$	$\{\}$	$\{\}$
s_C^1	$\{\{a_B^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_C^2	$\{\{a_B^1\}, \{a_C^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_C^3	$\{\{a_B^1\}, \{a_B^2, a_C^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_C^4	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}\}$	$\{\langle a,3\rangle\}$	$\{a\}$
s_C^5	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}, \{a_C^4\}\}$	$\{\langle a,4\rangle\}$	$\{\}$

of them is the same as seeing all. Therefore, states s_A^3, s_B^2, s_B^3, s_B^4, s_C^2 and s_C^3 are equivalent with regard to the removal of a.

Now we present the states of element a as the equivalence classes of the updates, as shown in Table 1. The concurrent updates that see equivalent states and achieves the same effect are in the same equivalent classes. For example, updates a_A^1 and a_B^1 are in the same equivalent class because they see equivalent states s_A^0 and s_B^0 and achieve the same effect, i.e. adding element a into the set. In [11], we made a more rigorous description of the equivalence classes in the context of support for concurrent undo.

Given this representation, we can observe the following:

- Performing a new local update adds a new equivalence class that contains only the new local update.
- Merging two states is the same as the union of the equivalent classes.
- A site determines whether an element is in the set by counting the number of equivalence classes that the site currently observes, rather than the specific updates contained in the classes.

Due to the last observation, we can represent the state of an element with a single number, the number of equivalence classes. We call that number the *causal length* of the element. The s_{cl} column of Table 1 lists the states of element a in terms of causal lengths.

$$\mathsf{CLSet}(E) \stackrel{\text{def}}{=} E \hookrightarrow \mathbb{N}$$

$$\mathsf{add}(s,e) \stackrel{\text{def}}{=} \begin{cases} s\{e \mapsto s(e)+1\} & \text{if even}\big(s(e)\big) \\ s & \text{if odd}\big(s(e)\big) \end{cases}$$

$$\mathsf{add}^\delta(s,e) \stackrel{\text{def}}{=} \begin{cases} \{e \mapsto s(e)+1\} & \text{if even}\big(s(e)\big) \\ \{\} & \text{if odd}\big(s(e)\big) \end{cases}$$

$$\mathsf{remove}(s,e) \stackrel{\text{def}}{=} \begin{cases} s & \text{if even}\big(s(e)\big) \\ s\{e \mapsto s(e)+1\} & \text{if odd}\big(s(e)\big) \end{cases}$$

$$\mathsf{remove}^\delta(s,e) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if even}\big(s(e)\big) \\ \{e \mapsto s(e)+1\} & \text{if odd}\big(s(e)\big) \end{cases}$$

$$(s \sqcup s')(e) \stackrel{\text{def}}{=} \max\big(s(e), s'(e)\big)$$

$$\mathsf{in}(s,e) \stackrel{\text{def}}{=} \mathsf{odd}\big(s(e)\big)$$

$$\mathsf{all}(s) \stackrel{\text{def}}{=} \{e \mid \mathsf{odd}\big(s(e)\big)\}$$

Figure 4. CLSet CRDT

4 CLSet CRDT

Figure 4 shows the CLSet CRDT. Notice that the state s is a partial function: $s(e) = \bot_{\mathbb{N}} = 0$ when an element e has never been added and thus not in the domain of s.

An element e is in the set when its causal length is an odd number. A local addition has effect only when the element is not in the set. Similarly, a local removal has effect only when the element is actually in the set. A local addition or removal simply increments the causal length of the element by one. For every element e in s and/or s', the new causal length of e after merging s and s' is the maximum of the causal lengths of e in s and s'.

5 Comparison with existing set CRDTs

CLSet is a direct application of our earlier work on undo support for CRDTs [11]. It is obvious that addition and removal are inverse (i.e. undo) updates of one another. One reason for us to exercise this particular application to set here is that set is such a fundamental and versatile data type. Another reason is that we would like to make comparison to existing general-purpose set CRDTs in some detail.

Figure 5 shows the states of a single element in ORSet (described in Section 2 and Figure 2). In the figure, $1_A, 2_A, \ldots$ are the dots corresponding to the addition instances originated at site A. The states in the same shaded area correspond to the states with the same causal length.

ORSet and CLSet handle the states in red color in Figure 5 differently. For the concurrent addition and removal of the same element in these states, ORSet applies the add-wins semantics [3], which is different from CLSet. An alternative semantics of set CRDTs is remove-wins. For the blue states

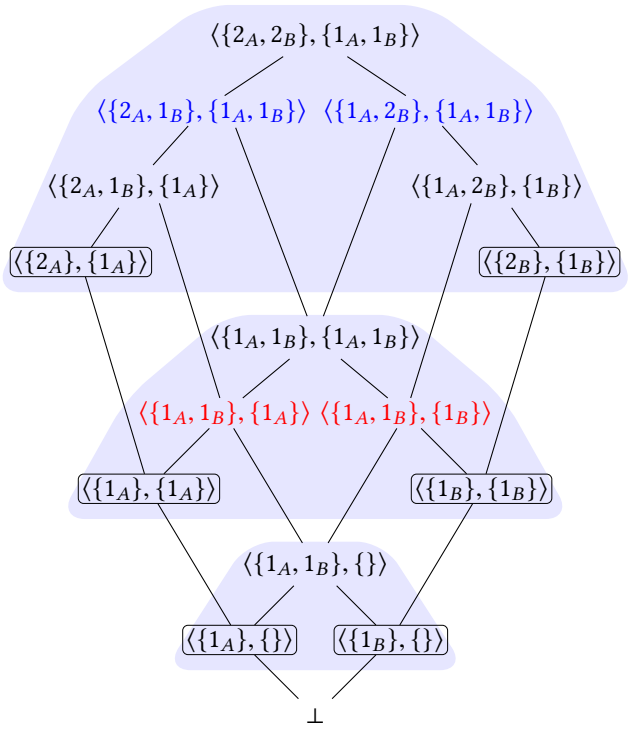

Figure 5. States of a single element in ORSet

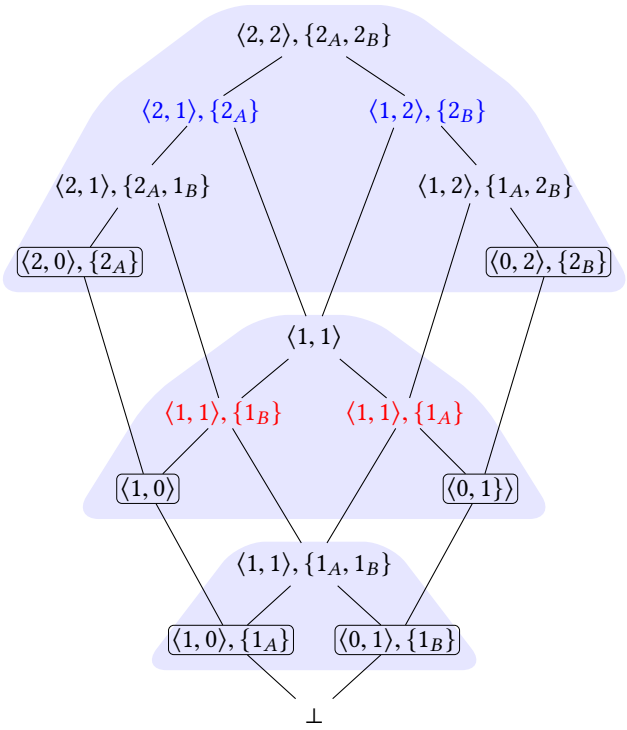

Figure 6. States of a single element in TFAWSet

in Figure 5, a remove-wins set has different effects from both an add-wins set and a CLSet.

Add-wins sets and remove-wins sets handle addition and removal updates in an asymmetric way (as their names indicate). In an add-wins set, a remove operation regards every individual addition update as distinct and only cancels the effects of the addition updates it sees at the time of the removal. On the other hand, an add operation handles the set of concurrent removal updates as indistinguishable and cancels all their effects. For example, the addition update represented with the state $\langle 2_A, 1_A \rangle$ in Figure 5 cancels the effects of removal updates represented with the states $\langle \{1_A\}, \{1_A\} \rangle$, $\langle \{1_B\}, \{1_B\} \rangle$, $\langle \{1_A, 1_B\}, \{1_A, 1_B\} \rangle$, and eventually also future removal updates such as (not shown in the figure) $\langle \{3_B\}, \{3_B\} \rangle$ etc.

Although different semantics may all be acceptable in a concurrent system, we argue that the CLSet semantic is more appropriate, as it "neutralizes" add-wins and remove-wins semantics and handles add and removal operations in a symmetric manner.

LWW-Element-Set[2] [9] is another general-purpose set CRDT that allows concurrent addition and removal of elements[3]. It associates every element with two timestamps, one for addition and one for removal. The updates of a single-element state are inflationary on the timestamps. The (add or remove) operation with a greater timestamp wins. Similar to CLSet, LWW-Element-Set is an anonymous CRDT and the size of the meta data associated with each element is constant. The semantics of set operations depend on the semantics of the timestamps. For example, with hybrid logic clock [7], if event e_1 happens before event e_2, their corresponding clock values t_1 and t_2 have the property $t_1 < t_2$. Thereby, a removal update cancels the effects of all the addition updates it sees (similar to add-wins) together with a few more concurrent addition updates with smaller clock values. Similarly, an addition update cancels the effects of all the removal updates it sees (similar to remove-wins) together with a few more concurrent removal updates with smaller clock values. Apparently nodes with faster clocks tend to have a higher chance to win the competition. LWW-Element-Set with hybrid logic clock "mixes" in a sense the semantics of add-wins and remove-wins.

Tombstones are the metadata associated with the elements that have been removed from the set. Report [4] presented a tombstone-free set CRDT. It is based on the causality between a removal and the additions it observed. Such causality can be captured with a set-wise (i.e. shared by all elements) version vector. More specifically, an addition of an element is considered to be removed if the element is absent in the set but the addition instance is covered by the version vector. Figure 6 shows the Hasse diagram of the states of an element in a tombstone-free add-wins set (TWAWSet). Here a state is

[2]LWW stands for Last Writer Wins.
[3]AWLWWSet and RWLWWSet in [2] are similar variations.

$$\text{TFAWSet} \stackrel{\text{def}}{=} (E \hookrightarrow \mathcal{P}(dots)) \times \mathcal{P}(dots)$$

$$\text{add}_i^\delta(\langle m, c\rangle, e) \stackrel{\text{def}}{=} \langle\{e \mapsto d\}, d\rangle \text{ where } d = \{\text{next}_i(c)\}$$

$$\text{remove}_i^\delta(\langle m, c\rangle, e) \stackrel{\text{def}}{=} \langle\{\}, m(e)\rangle$$

$$\langle m, c\rangle \sqcup \langle m', c'\rangle \stackrel{\text{def}}{=} \langle\{e \mapsto d'' \mid e \in \text{dom}(m) \cup \text{dom}(m') \wedge d'' \neq \{\}\}, c \cup c'\rangle$$

$$\text{where } d = m(e), d' = m'(e) \text{ and } d'' = (d \cap d') \cup (d - c') \cup (d' - c)$$

$$\text{in}(\langle m, c\rangle, e) \stackrel{\text{def}}{=} e \in \text{dom}(m)$$

$$\text{all}(\langle m, c\rangle) \stackrel{\text{def}}{=} \text{dom}(m)$$

Figure 7. TFAWSet delta-state CRDT

represented as a pair of a set-wise version vector and a set of dots for the addition instances that have not been removed. The shape of the Hasse diagram is exactly the same as that of the ORSet CRDT (Figure 5).

The report [4] adopted a mixed operation-based and state-based approach. Figure 7 shows TFAWSet presented in [2] (where it is named AWSet). The states of a TFAWSet is represented as a pair of a partial function and a dot set (known as a causal context). For two TFAWSet states $(m(e), c_e)$ and $(m'(e), c'_e)$ concerning element e, the partial order is defined as $(m(e), c_e) \sqsubset (m'(e), c'_e) \stackrel{\text{def}}{=} (c_e \subset c'_e) \vee (c_e = c'_e \wedge m(e) \supset m'(e))$. This is somewhat counter-intuitive: the partial order $\sqsubset$ is defined with the $\supset$ rather than the $\subset$ relation on the dot sets of addition instances. This ordering is enforced by the join operation, which removes the dots of the addition instances observed by subsequent removal updates. In Figure 6, the $\sqsubset$ order between the states with same version vector value $\langle 1, 1\rangle$ are decided by the $\supset$, not $\subset$, relation of the dots of the addition instances of the same element.

When the system enforces causal message delivery, the causal contexts can be compressed into version vectors. The CRDT is thereby tombstone-free.

Compared to CLSet, TFAWSet requires causal delivery for tombstone elimination, which is a stronger requirement. It could outperform CLSet if the vast majority of elements are removed. The elements that remain in the set are associated with more metadata than CLSet. The actual amount depends on the number of additions that have not been removed.

6 Performance

We have run some experiments to study the performance of three set CRDTs, namely CLSet, ORSet and TFAWSet. We have implemented CLSet and ORSet in Elixir, and adapted an open source implementation for TFAWSet.[4]

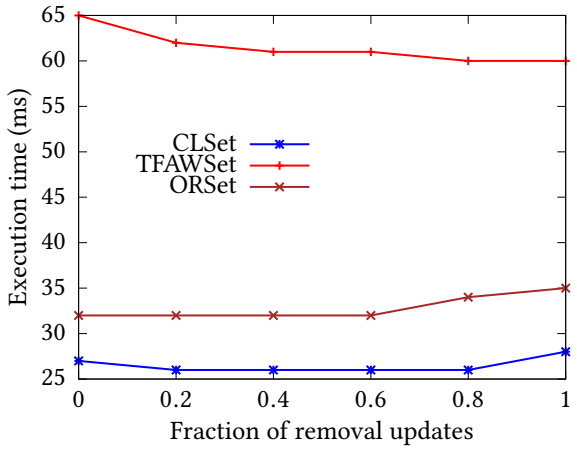

Figure 8. Time for concurrent updates and merges

We ran the benchmarks using the Benchee[5] library with Elixir 10.1 (OTP 22.2) on Ubuntu Linux 18.04. The computer has an Intel Xeon CPU E3-1245 v5 at 3.50GHz and 32GB Ram. Since we ran all the benchmarks in a single Erlang process (thread), the number of CPU cores does not play any role.

We first study how well the three CRDTs perform updates and merges. For each CRDT, we set up 10 instances that are initiated with 1000 elements. The sets may have up to 2000 elements during each execution (i.e, there are initially 1000 empty "slots"). For each execution, we update the CRDTs in iterations. In every iteration, we perform concurrently 2 to 5 random updates locally at 2 to 5 randomly chosen CRDT instances. Then all instances merge with these updates. The next iteration starts as soon as the current one finishes. The execution finishes after 500 updates. We vary the fraction of removal updates.

To make the comparison fair, we do not allow existing elements to be added into an ORSet or a TFAWSet (which we believe is more appropriate than the original design in Figures 2 and 7).

Figure 8 shows the average time spent to finish the benchmark executions. To our surprise, TFAWSet took longer time to finish the executions than ORSet in all of the situations. It turns out that computing d'' in Figure 7 contributed to the longer execution time, at least with this current implementation. Notice that the number of updates applied on a single element is typically very low. The sizes of the dot sets in ORSet are therefore typically very small. On the other hand, the sizes of the causal contexts in TFAWSet depends on the number of CRDT instances (or nodes), which is typically

[4]We removed the "map" part of the AWLWWMap CRDT available at https://github.com/derekkraan/delta_crdt_ex.
[5]https://github.com/bencheeorg/benchee

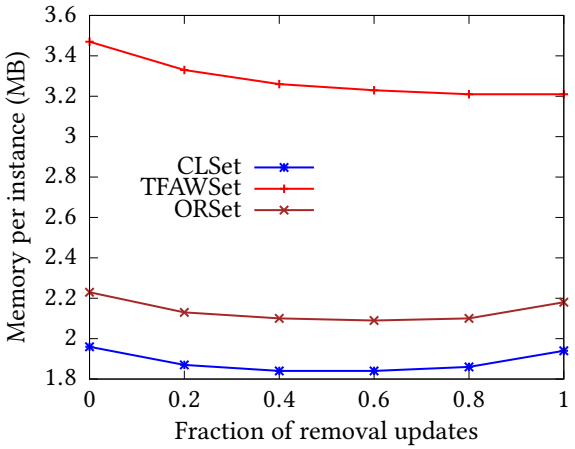

Figure 9. Memory consumption per CRDT instance

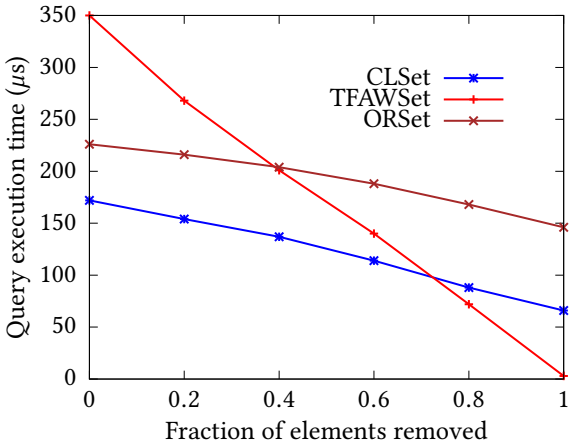

Figure 10. Time for performing the all query

higher than the number of updates that have been applied on a single element.

The memory consumption of the CRDTs (Figure 9) shows a similar pattern as the execution time. TFAWSets consume more memory because it generates significant amount of intermediate data while merging the updates. Since the benchmarks are run intensively within a single Erlang process, the intermediate data have not got the chance to be garbage collected. Garbage collection may reduce the memory footprint of TFAWSets at the cost of additional CPU cycles.

We have also run the benchmarks to see how well the CRDTs perform the all query. We set up the CRDTs by first adding 1000 elements and then removing a fraction of them. We run the query benchmarks with these CRDTs.

Figure 10 shows the average time to perform the queries. For all CRDTs, the execution time decreases with the increase of the fraction of the elements that are removed. This is due to the decreased sizes of the query results. As the consequence of tombstone elimination, the execution time on TFAWSets decreases much faster. Still, CLSet out-performs TFAWSet when up to two thirds of the elements remain in the set.

7 Conclusion

We have presented CLSet, a general-purpose state-based set CRDT. The only metadata associated with a set element is a single natural number called causal length, which captures the causality of concurrent set updates. CLSet has low runtime overhead compared to existing general-purpose set CRDTs.

Acknowledgments

The first author thanks the members of the COAST team at INRIA-LORIA in France, in particular Claudia-Lavinia Ignat and Victorien Elvinger, for inspiring discussions. The authors also thank the anonymous reviewers for insightful comments that help us make improving revisions.

References

[1] Almeida, P. S., Baquero, C., Gonçalves, R., Preguiça, N. M., and Fonte, V. Scalable and accurate causality tracking for eventually consistent stores. In *14th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)* (2014), LNCS 8460, Springer, pp. 67–81.

[2] Almeida, P. S., Shoker, A., and Baquero, C. Delta state replicated data types. *J. Parallel Distrib. Comput. 111* (2018), 162–173.

[3] Bieniusa, A., Zawirski, M., Preguiça, N. M., Shapiro, M., Baquero, C., Balegas, V., and Duarte, S. Brief announcement: Semantics of eventually consistent replicated sets. In *26th International Symposium on Distributed Computing (DISC)* (2012), LNCS 7611, Springer, pp. 441–442.

[4] Bieniusa, A., Zawirski, M., Preguiça, N. M., Shapiro, M., Baquero, C., Balegas, V., and Duarte, S. A optimized conflict-free replicated set. *Rapport de recherche 8083*, INRIA, (October 2012).

[5] Enes, V., Almeida, P. S., Baquero, C., and Leitão, J. Efficient Synchronization of State-based CRDTs. In *IEEE 35th International Conference on Data Engineering (ICDE)* (April 2019).

[6] Garg, V. K. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.

[7] Kulkarni, S. S., Demirbas, M., Madappa, D., Avva, B., and Leone, M. Logical physical clocks. In *Principles of Distributed Systems (OPODIS)* (2014), LNCS 8878, Springer, pp. 17–32.

[8] Meiklejohn, C., and Van Roy, P. Lasp: a language for distributed, coordination-free programming. In *the 17th International Symposium on Principles and Practice of Declarative Programming* (2015), pp. 184–195.

[9] Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. A comprehensive study of convergent and commutative replicated data types. *Rapport de recherche 7506*, INRIA, (January 2011).

[10] Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS)* (2011), pp. 386–400.

[11] Yu, W., Elvinger, V., and Ignat, C.-L. A generic undo support for state-based CRDTs. In *23rd International Conference on Principles of Distributed Systems (OPODIS2019)* (2020), vol. 153 of *LIPIcs*, pp. 14:1–14:17.